CHAPTER 2

BACKGROUND

In this chapter, we present a overview of the polyhedral model, and introduce notation used throughout the dissertation. The mathematical background on linear algebra and linear programming required to understand the theoretical aspects of this dissertation is fully covered in this chapter. A few fundamental concepts and definitions relating to cones, polyhedra, and linear inequalities have been omitted and they can be found in [Wil93] and [Sch86]. Detailed background on traditional loop transformations can be found in [Wol95, Ban93]. Overall, I expect the reader to find the background presented here self-contained. [Bas04b, Gri04] are among other sources for introduction to the polyhedral model.

All row vectors will be typeset in bold lowercase, while regular vectors are typeset with an overhead arrow. The set of all real numbers, the set of rational numbers, and the set of integers are represented by $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{Z}$, respectively.

2.1 Hyperplanes and Polyhedra

Definition 1 (**Linear**)**.** A k-dimensional function f is linear iff it can expressed in the following form:

$$linear \quad function \quad f(\vec{v}) = M_f\vec{v} \tag{2.1}$$

where $\vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}$ and $M_f \in \mathbb{R}^{k\times d}$ is a matrix with k rows and d columns. In our context, M_f is an integer matrix, i.e., $M_f \in \mathbb{Z}^{k\times d}$

Definition 2 (**Affine**)**.** A k-dimensional function f is affine iff it can expressed in the following form:

$$affine \quad function \quad f(\vec{v}) = M_f\vec{v} + \vec{f_0} \tag{2.2}$$

where $\vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}$ and $M_f \in \mathbb{R}^{k\times d}$ is a matrix with k rows and d columns, $f_0 \in \mathbb{R}^k$ is a k-dimensional vector. In all cases, we deal with affine functions with $M_f \in \mathbb{Z}^{k\times d}$ and $f_0 \in \mathbb{Z}^k$. The domain is also a set of integers: $\vec{v} \in \mathbb{Z}^d$.

Definition 3 (**Null space**)**.** The *null space* of an affine function $f(\vec{v}) = M_f\vec{v} + \vec{f_0}$ is $\left\{\vec{x} \mid f(\vec{x}) = \vec{0}\right\}$.

f is a one-to-one mapping iff M_f has full column rank, i.e., if it has as many linearly independent rows (and columns) as the number of its columns. In such a case, the null space is 0-dimensional, i.e., trivially the vector $\vec{0}$.

Definition 4 (**Affine spaces**)**.** A set of vectors is an affine space iff it is closed under affine combination, i.e., if $\vec{x}, \vec{y}$ are in the space, all points lying on the line joining $\vec{x}$ and $\vec{y}$ belong to the space.

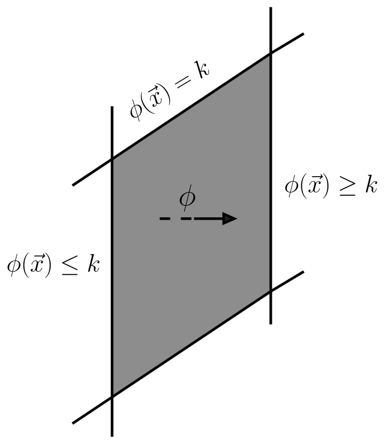
A line in a vector space of any dimensionality is a one-dimensional affine space. In 3-d space, any 2-d plane is an example of a 2-d affine sub-space. Note that 'affine function' as defined in (2.2) should not be confused with 'affine combination', though several researchers use the term affine combination in place of an affine function.

Definition 5 (**Affine hyperplane**)**.** An affine hyperplane is an $n-1$ dimensional affine sub-space of an n dimensional space.

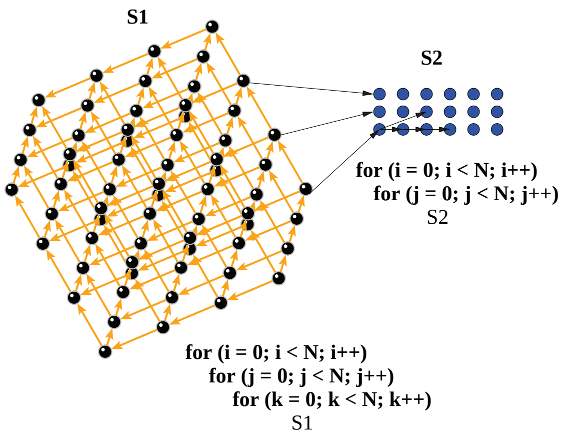
In our context, the set of all vectors $v \in \mathbb{Z}^n$ such that $\mathbf{h}.\vec{v} = k$, for $k \in \mathbb{Z}$, forms an affine hyperplane. The set of parallel *hyperplane instances* correspond to different values of k with the row vector $\mathbf{h}$ normal to the hyperplane. Two vectors $\vec{v_1}$ and $\vec{v_2}$ lie in the same hyperplane if $\mathbf{h}.\vec{v_1} = \mathbf{h}.\vec{v_2}$. An affine hyperplane can also be viewed as a one-dimensional affine function that maps an n-dimensional space onto a one-dimensional space, or partitions an n-dimensional space into $n-1$ dimensional slices. Hence, as a function, it can be written as:

$$\phi(\vec{v}) = \mathbf{h}.\vec{v} + c \tag{2.3}$$

Figure 2.1(a) shows a hyperplane geometrically. Throughout the dissertation, the hyperplane is often referred to by the row vector, $\mathbf{h}$, the vector normal to the hyperplane. A hyperplane $\mathbf{h}.\vec{v} = k$ divides the space into two *half-spaces*, the positive half-space,



(a) An affine hyperplane



```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    S2
```

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      S1
```

(b) Polyhedra (courtesy: Loopo display tool)

Figure 2.1: Hyperplane and Polyhedron

$h.\vec{v} \geq k$, and a negative half-space, $\mathbf{h}.\vec{v} \leq k$. Each half-space can be represented by an affine inequality.

Definition 6 (Polyhedron, Polytope). A *polyhedron* is an intersection of a finite number of half-spaces. A *polytope* is a bounded polyhedron.

Each of the half-spaces provides a face to the polyhedron. Hence, the set of affine inequalities, each representing a face, can be used to compactly represent the polyhedron. If there are m inequalities, then the polyhedron is

$$\left\{\vec{x} \in \mathbb{R}^n \mid A\vec{x} + \vec{b} \geq \vec{0}\right\}$$

where $A \in \mathbb{R}^{m \times n}$ and $\vec{b} \in \mathbb{R}^m$.

A polyhedron also has an alternate dual representation in terms of vertices, rays, and lines, and algorithms like the Chernikova algorithm [LeV92] exist to move from the face representation to the vertex one. Polylib [Pol] and PPL [BHZ] are two libraries that provide a range of functions to perform various operations on polyhedra and use the dual representation internally.

In our context, we are always interested in the integer points inside a polyhedron since loop iterators typically have integer data types and traverse an integer space. The matrix A and $\vec{b}$ for problems we will deal with also comprise only integers. So, we always have:

$$\left\{\vec{x} \in \mathbb{Z}^n \mid A\vec{x} + \vec{b} \geq \vec{0}\right\} \tag{2.4}$$

where $A \in \mathbb{Z}^{m\times n}$ and $\vec{b} \in \mathbb{Z}^m$.

Lemma 1 (**Affine form of the Farkas lemma**). *Let $\mathcal{D}$ be a non-empty polyhedron defined by p affine inequalities or faces*

$$\mathbf{a_k}.\vec{x} + b_k \geq 0, \quad k = 1, p$$

then, an affine form ψ is non-negative everywhere in $\mathcal{D}$ iff it is a non-negative linear combination of the faces:

$$\psi(\vec{x}) \equiv \lambda_0 + \sum_{k=1}^{p} \lambda_k \left(\mathbf{a_k}\vec{x} + b_k\right), \quad \lambda_0, \lambda_1, \ldots, \lambda_p \geq 0 \tag{2.5}$$

The non-negative constants λ_k are referred to as the Farkas multipliers. Proof of the *if* part is obvious. For the *only if* part, see Schrijver [Sch86]. We provide the main idea here roughly. The polyhedron $\mathcal{D}$ lies in the non-negative half-space of the hyperplane $\psi(\vec{x})$. This makes sure that λ_0 has to be non-negative if the hyperplane is pushed close enough to the polytope so that it touches a vertex of the polyhedron first without cutting the polyhedron. If a hyperplane passes through a vertex of the polyhedron and with the entire polyhedron in its non-negative half-space, the fact that it can be expressed as a non-negative linear combination of the faces of the polyhedron directly follows from the Fundamental Theorem of Linear Inequalities [Sch86].

Definition 7 (Perfect loop nest, Imperfect loop nest). A set of nested loops is called a *perfect loop nest* iff all statements appearing in the nest appear inside the body of the innermost loop. Otherwise, the loop nest is called an *imperfect loop nest*. Figure 2.6 shows an imperfect loop nest.

Definition 8 (Affine loop nest). *Affine loop nests* are sequences of imperfectly nested loops with loop bounds and array accesses that are affine functions of outer loop variables and program parameters.

Program parameters or structure parameters are symbolic constants that appear in loop bounds or access functions. They very often represent the problem size. N is a program parameter in Figure 2.1(b), while in Figure 2.2, N and β are the program parameters.

2.2 The Polyhedral Model

The polyhedral model is a geometrical as well as a linear algebraic framework for capturing the execution of a program in a compact form for analysis and transformation. The compact representation is primarily of the dynamic instances of statements of a program surrounded by loops in a program, the dependences between such statements, and transformations.

Definition 9 (Iteration vector). The *iteration vector* of a statement is the vector consisting of values of the indices of all loops surrounding the statement.

Let S be a statement of a program. The iteration vector is denoted by $\vec{i}_S$. An iteration vector represents a dynamic instance of a statement appearing in a loop nest that may be nested perfectly or imperfectly.

Definition 10 (Domain, Index set). The set of all iteration vectors for a given statement is the *domain* or the *index set* of the statement.

A program comprises a sequence of statements, each statement surrounded by loops in a given order. We denote the domain of a statement S by $\mathcal{D}^S$. When the loop bounds and data accesses are affine functions of outer loop indices and other program parameters, and all conditionals are statically predictable, the domain of every statement is a polyhedron as defined in (2.4). Again, conditionals that are affine functions of outer loop indices and program parameters are statically predictable. Affine loop nests with static control are also called static control programs or regular programs. These programs are readily accepted in the polyhedral model. Several of the restrictions for the polyhedral model can be overcome with tricks or conservative assumptions while still making all analysis and transformation meaningful. However, many pose a challenging problem requiring extensions to the model. Techniques developed and implemented in this thesis apply to all programs for which a polyhedral representation can be extracted. All codes used for experimental evaluation are regular programs with static control.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    S1: A[i,j] = A[i,j]+u1[i]*v1[j] + u2[i]*v2[j];

for (k=0; k<N; k++)
  for (l=0; l<N; l++)
    S2: x[k] = x[k]+beta*A[l,k]*y[l];
```

Figure 2.2: A portion of the GEMVER kernel

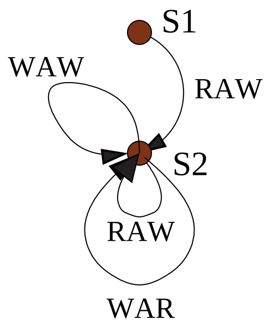


Figure 2.3: The data dependence graph

Each dynamic instance of a statement S, in a program, is identified by its iteration vector $\vec{i_S}$ which contains values for the indices of the loops surrounding S, from outermost to innermost. A statement S is associated with a polytope $\mathcal{D}^S$ of dimensionality m_S. Each point in the polytope is an m_S-dimensional iteration vector, and

$$\begin{array}{rcl} i & \geq & 0 \\ j & \geq & 0 \\ -i+N-1 & \geq & 0 \\ -j+N-1 & \geq & 0 \end{array} \qquad \mathcal{D}^{S_1} : \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq 0$$

Figure 2.4: Domain for statement S1 from Figure 2.2

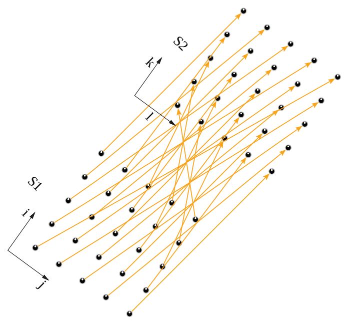
the polytope is characterized by a set of bounding hyperplanes. This is true when the loop bounds are linear combinations (with a constant) of outer loop indices and program parameters (typically, symbolic constants representing the problem size).

2.3 Polyhedral Dependences

Dependences. Two iterations are said to be dependent if they access the same memory location and one of them is a write. A true dependence exists if the source iteration's access is a write and the target's is a read. These dependences are also called read-after-write or RAW dependences, or flow dependences. Similarly, if a write precedes a read to the same location, the dependence is called a WAR dependence or an anti-dependence. WAW dependences are also called output dependences. Read-after-read or RAR dependences are not actually dependences, but they still could be important in characterizing reuse. RAR dependences are also called input dependences.

Dependences are an important concept while studying execution reordering since a reordering will only be legal if does not violate the dependences, i.e., one is allowed to change the order in which operations are performed as long as the transformed program has the same execution order with respect to the dependent iterations.

Data Dependence Graph. The Data Dependence Graph (DDG) $G = (V, E)$ is a directed multi-graph with each vertex representing a statement, i.e., $V = \mathbf{S}$. An edge, $e \in E$, from node S_i to S_j represents a dependence with the source and target



$$\left(\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 \\ \hline 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \end{array}\right) \left(\begin{array}{c} i \\ j \\ k \\ l \\ N \\ 1 \end{array}\right) \begin{array}{c} \\ \\ \geq \\ \\ = \\ \\ \end{array} \left(\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ \hline 0 \\ 0 \end{array}\right)$$

Figure 2.5: Flow dependence from A[i][j] of S1 to A[l][k] of S2 and its dependence polyhedron (courtesy: Loopo display tool)

conflicting accesses in S_i and S_j respectively. Figure 2.3 shows the DDG for the code in Figure 2.2.

2.3.1 Dependence polyhedron.

For an edge e, the relation between the dynamic instances of S_i and S_j that are dependent is captured by the *dependence polyhedron*, $\mathcal{P}_e$. The dependence polyhedron is in the sum of the dimensionalities of the source and target statement's polyhedra along with dimensions for program parameters. If $\vec{s}$ and $\vec{t}$ are the source and target iterations that are dependent, we can express:

$$\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e \iff \vec{s} \in \mathcal{D}^{S_i}, \vec{t} \in \mathcal{D}^{S_j} \text{ are dependent through edge } e \in E \qquad (2.6)$$

The ability to capture the exact conditions on when a dependence exists through linear equalities and inequalities rests on the fact that there exists a affine relation between the iterations and the accessed data for regular programs. Equalities can be replaced by two inequalities ('≥ 0' and '≤ 0') and everything can be converted to inequalities in the ≥ 0 form, i.e., the polyhedron can be expressed as the intersection of a finite number of non-negative half-spaces. Let the i^{th} inequality after conversion to such a form be denoted by $\mathcal{P}_e^i$. For the code shown in Figure 2.2, consider the dependence between the write at A[i][j] from S1 and the read A[l][k] in S2. The dependence polyhedron for this edge is shown in Figure 2.5.

In the next chapter, we see that the dependence polyhedra is the most important structure around which the problem of finding legal and good transformations centers. In particular, the Farkas lemma (Sec. 2.1) is applied for the dependence polyhedron. A minor point to note here is that the dependence polyhedra we see are often integral polyhedra, i.e., polyhedra that have integer vertices. Hence, the application of Farkas lemma for it is exact and not conservative. Even when the dependence polyhedron is not integral, i.e., when its integer hull is a proper subset of the polyhedron, the difference between applying it to the integer hull and the entire polyhedron is highly unlikely to matter in practice. If need be, one can construct the integer hull of the polyhedron and apply the Farkas lemma on it.

2.3.2 Strengths of dependence polyhedra

The dependence polyhedra are a very general and accurate representation of instance-wise dependences which subsume several traditional notions like distance vectors (also called uniform dependences), dependence levels, and direction vectors. Though a similar notion of exact dependences was presented by Feautrier [Fea91] for value-based array dataflow analysis, this notion of dependence polyhedra has only been sharpened in the past few years by researchers [CGP+05, VBGC06, PBCV07] and is not restricted to programs in single assignment form nor does it require conversion to single-assignment form. Dependence abstractions like direction vectors or distance vectors are tied to a particular syntactic nesting unlike dependence polyhedra which is more abstract and captures the relation between integer points of polyhedra. One could obtain weaker dependence representations from a dependence polyhedra.

```
for (t=0; t<tmax; t++) {
  for (j=0; j<ny; j++)
    ey [0][ j ] = t;
    for (i=1; i<nx; i++)
      for (j=0; j<ny; j++)
        ey[i ][ j ] = ey[i ][ j ] − 0.5∗(hz[i ][ j]−hz[i−1][j ]);
    for (i=0; i<nx; i++)
      for (j=1; j<ny; j++)
        ex[i ][ j ] = ex[i ][ j ] − 0.5∗(hz[i ][ j]−hz[i ][ j−1]);
    for (i=0; i<nx; i++)
      for (j=0; j<ny; j++)
        hz[i ][ j]=hz[i[ j ] −
          0.7∗(ex[i ][ j+1]−ex[i][j]+ey[i+1][j]−ey[i ][ j ]);
}
```

Figure 2.6: An imperfect loop nest

$$\begin{aligned}
0 &\le t \le T-1\\
0 &\le t' \le T-1\\
0 &\le i \le N-1\\
0 &\le j \le N-1\\
0 &\le i' \le N-1\\
0 &\le j' \le N-1\\
t &= t'-1\\
i &= i'-1\\
j &= j'
\end{aligned}$$

Figure 2.7: Dependence polyhedron: $S4(hz[i][j]) \rightarrow S2(hz[i-1][j])$

Another example. For the code shown in Figure 2.6, consider the flow dependence between S4 and S2 from the write at hz[i][j] to the read at hz[i-1][j] (later time steps). Let $\vec{s} \in \mathcal{D}^{S4}$, $\vec{t} \in \mathcal{D}^{S2}$, $\vec{s} = (t, i, j)$, $\vec{t} = (t', i', j')$; then, $\mathcal{P}_e$ for this edge is shown in Figure 2.7.

Uniform and Non-uniform dependences. Uniform dependences traditionally make sense for a statement in perfectly nested loop nest or two statements which are in the same perfectly nested loop body. In such cases a uniform dependence is a dependence where the source and target iteration in question are a constant vector distance apart. Such a dependence is also called a constant dependence and represented as a distance vector [Wol95].

For detailed information on polyhedral dependence analysis and a good survey of older techniques in the literature including non-polyhedral ones, the reader can refer to [VBGC06].

2.4 Polyhedral Transformations

A one-dimensional affine transform for statement S is an affine function defined by:

$$\begin{aligned} \phi_S(\vec{i}) &= \begin{pmatrix} c_1^S & c_2^S & \dots & c_{m_S}^S \end{pmatrix} \begin{pmatrix} \vec{i}_S \end{pmatrix} + c_0^S \\ &= \begin{pmatrix} c_1^S & c_2^S & \dots & c_{m_S}^S & c_0^S \end{pmatrix} \begin{pmatrix} \vec{i}_S \\ 1 \end{pmatrix} \end{aligned} \tag{2.7}$$

where $c_0, c_1, c_2, \dots, c_{m_S} \in \mathbb{Z}, \quad \vec{i} \in \mathbb{Z}^{m_S}$ Hence, a one-dimensional affine transform for each statement can be interpreted as a partitioning hyperplane with normal $(c_1, \dots, c_{m_S})$. A multi-dimensional affine transformation can be represented as a sequence of such ϕ's for each statement. We use a superscript to denote the hyperplane for each level. ϕ_S^k represents the hyperplane at level k for statement S. If $1 \le k \le d$, all the ϕ_S^k can be represented by a single d-dimensional affine function $\mathcal{T}_S$ given by:

$$\mathcal{T}_S \vec{i}_S = M_S \vec{i}_S + \vec{t}_S \tag{2.8}$$

where $M_S \in \mathbb{Z}^{d \times m_S}$, $\vec{t}_S \in \mathbb{Z}^d$.

$$\mathcal{T}_S(\vec{i}) = \begin{pmatrix} \phi_S^1(\vec{i}) \\ \phi_S^2(\vec{i}) \\ \vdots \\ \phi_S^d(\vec{i}) \end{pmatrix} = \begin{pmatrix} c_{11}^S & c_{12}^S & \dots & c_{1m_S}^S \\ c_{21}^S & c_{22}^S & \dots & c_{2m_S}^S \\ \vdots & \vdots & \vdots & \vdots \\ c_{d1}^S & c_{d2}^S & \dots & c_{dm_S}^S \end{pmatrix} \vec{i}_S + \begin{pmatrix} c_{10}^S \\ c_{20}^S \\ \vdots \\ c_{d0}^S \end{pmatrix} \tag{2.9}$$

Scalar dimensions. The dimensionality of $\mathcal{T}_S$, d, may be greater than m_S as some rows in $\mathcal{T}_S$ serve the purpose of representing partially fused or unfused dimensions at a level. Such a row has $(c_1, \dots, c_{m_S}) = \mathbf{0}$, and a particular constant for c_0. All statements with the same c_0 value are fused at that level and the unfused sets are placed in the increasing order of their c_0s. We call such a level a **scalar dimension**. Hence, a level is a scalar dimension if the ϕ's for all statements at that level are constant functions. Figure 2.8 shows a sequence of matrix-matrix multiplies and how a transformation captures a legal fusion: the transformation fuses ji of S1 with jk of S2; ϕ^3 is a scalar dimension.

Complete scanning order. The number of rows in M_S for each statements should be the same (d) to map all iterations to a global space of dimensionality d. To provide a complete scanning order for each statement, the number of linearly independent ϕ_S's for a statement should be the same as the dimensionality of the statement, m_S, i.e., $\mathcal{T}_S$ should have full column rank. Note that it is always possible to represent any transformed code (any nesting) with at most $2m_S^* + 1$ rows, where $m_S^* = \max_{S \in \mathbf{S}} m_S$.

Composition of simpler transformations. Multi-dimensional affine functions capture a sequence of simpler transformations that include permutation, skewing,

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
    S1: C[i,j] = C[i,j] + A[i,k] * B[k,j];
    }
  }
}
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
    S2: D[i,j] = D[i,j] + E[i,k] * C[k,j];
    }
  }
}
```

Original code

```
for (t0=0;t0<=N−1;t0++) {
  for (t1=0;t1<=N−1;t1++) {
    for (t3=0;t3<=N−1;t3++) {
      C[t1,t0]=A[t1,t3]*B[t3,t0]+C[t1,t0];
    }
    for (t3=0;t3<=N−1;t3++) {
      D[t3,t0]=E[t3,t1]*C[t1,t0]+D[t3,t0];
    }
  }
}
```

Transformed code

$$T_{S_1}(\vec{i}_{S_1}) = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$T_{S_2}(\vec{i}_{S_2}) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

i.e.,

$$\begin{array}{ll} \phi^1_{S_1} = j & \phi^1_{S_2} = j \\ \phi^2_{S_1} = i & \phi^2_{S_2} = k \\ \phi^3_{S_1} = 0 & \phi^3_{S_2} = 1 \\ \phi^4_{S_1} = k & \phi^4_{S_2} = i \end{array}$$

Figure 2.8: Polyhedral transformation: an example

reversal, fusion, fission (distribution), relative shifting, and tiling for fixed tile sizes. Note that tiling cannot be readily expressed as an affine function on the original iterators ($\vec{i_S}$), but can be once supernodes are introduced into the domain increasing its dimensionality: this is covered in detail in a general context in Chapter 5.

Traditional transformations like unimodular transformations [Ban93, Wol95] and non-unimodular [Ram92, LP94, Ram95] ones were applied to a single perfect loop nest in isolation. They are therefore subsumed. Due to the presence of scalar dimensions, polyhedral transformations can be used to represent or transform to any kind of nesting structure. Also, they map iterations of statements to a common multidimensional space providing the ability to interleave iterations of different statements as desired.

One can notice a one-to-one correspondence between the A, B, Γ representation used for URUK/WRAP-IT [GVB+06] and the one we described above, except that we have all coefficients in Γ set to zero, i.e., no parametric shifts. The motivation behind this will be clear in the next two chapters. The above representation for transformations was first proposed, though in different forms, by Feautrier [Fea92b] and Kelly et al. [Kel96], but used systematically by viewing it in terms of three components, A, B, and Γ only recently [CGP+05, GVB+06, Vas07].

The above notation for transformations directly fits with scattering functions that a code generation tool like CLooG [Bas04a, Clo] supports. It refers to $\mathcal{T}_S$ as a scattering function. On providing the original statement domains, $\mathcal{D}^S$, along with $\mathcal{T}_S$, Cloog can scan the domains in the global lexicographic ordering imposed by $T_S(\vec{i_S})$ across all $S \in \mathbf{S}$. The goal of automatic transformation is to find the unknown coefficients of $\mathcal{T}_S$, $\forall S \in \mathbf{S}$.

2.4.1 Why affine transformations?

Definition 11 (**Convex combination**)**.** A convex combination of vectors, $\vec{x}_1$, $\vec{x}_2$, $\dots$, $\vec{x}_n$, is of the form $\lambda_1\vec{x}_1 + \lambda_2\vec{x}_2 + \cdots + \lambda_n\vec{x}_n$, where $\lambda_1, \lambda_2, \dots, \lambda_n \geq 0$ and $\sum_{i=1}^{n} \lambda_i = 1$.

Informally, a convex combination of two points always lies on the line segment joining the two points. In the general case, a convex combination of any number of points lies inside the convex hull of those points.

The primary reason affine transformations are of interest is that affine transformations are the most general class of transformations that preserve the collinearity and convexity of points in space, besides the ratio of distances. An affine transformation transforms a polyhedron into another polyhedron and one stays in the polyhedral abstraction for further analyses and most importantly for code generation. Code generation is relatively easier and so has been studied extensively for affine transformations. We now quickly show that if $\mathcal{D}^S$ is convex, its image under the affine function $\mathcal{T}_S$ is also convex. Let the image be:

$$T(\mathcal{D}^S) = \left\{ \vec{z} \mid \vec{z} = \mathcal{T}_S(\vec{x}), \vec{x} \in \mathcal{D}^S \right\}$$

Consider the convex combination of any two points, $\mathcal{T}_S(\vec{x})$ and $\mathcal{T}_S(\vec{y})$, of $T(\mathcal{D}^S)$:

$$\lambda_1\mathcal{T}_S(\vec{x}) + \lambda_2\mathcal{T}_S(\vec{y}), \quad \lambda_1 + \lambda_2 = 1, \lambda_1 \geq 0, \lambda_2 \geq 0$$

Now,

$$\begin{aligned}
\lambda_1\mathcal{T}_S(\vec{x}) + \lambda_2\mathcal{T}_S(\vec{y}) &= \lambda_1 M_S(\vec{x}) + \lambda_1\vec{t}_S + \lambda_2 M_S(\vec{y}) + \lambda_2\vec{t}_S \\
&= M_S(\lambda_1\vec{x} + \lambda_2\vec{y}) + \vec{t}_S, \quad (\because \lambda_1 + \lambda_2 = 1) \\
&= \mathcal{T}_S(\lambda_1\vec{x} + \lambda_2\vec{y}) \qquad (2.10)
\end{aligned}$$

Since $\mathcal{D}^S$ is convex, $\lambda_1\vec{x} + \lambda_2\vec{y} \in \mathcal{D}^S$. Hence, from (2.10), we have:

$$\begin{aligned}
\lambda_1\mathcal{T}_S(\vec{x}) + \lambda_2\mathcal{T}_S(\vec{y}) &\in T(\mathcal{D}^S) \\
\Rightarrow T(\mathcal{D}^S) \text{ is convex}
\end{aligned}$$

If M_S (the linear part of $\mathcal{T}_S$) has full column rank, i.e., the rank of M_S is m^S, $\mathcal{T}_S$ is a one-to-one mapping from $\mathcal{D}^S$ to $T(\mathcal{D}^S)$. A point to note when looking at integer spaces instead of rational or real spaces is that not every integer point in the rational domain that encloses $T(\mathcal{D}^S)$ may have an integer pre-image in $\mathcal{D}^S$, for example, transformations that are non-unimodular may create sparse integer polyhedra. This is not a problem since a code generator like Cloog can scan such sparse polyhedra by inserting modulos. Note that just like convexity, affine transformations also preserve the ratio of distances between points. Since integer points in the original domain are equally spaced, they are so in the transformed space too. Techniques for removal of modulos also exist [Vas07]. Hence, no restrictions need be imposed on the affine function $\mathcal{T}_S$. Sparse integer polyhedra also correspond to code with non-unit strides. However, these can be represented with an additional dimension as long as the stride is a constant, for eg., as $\{0 \leq i \leq n-1,\ i = 2k\}$ for i going from 0 to $n-1$ with stride two. However, if one is interested, a more direct representation for integer points in a polyhedron can be used [S.P00, GR07]. The term $\mathcal{Z}$-polyhedron is associated with such a representation which is the image of a rational polyhedron under an affine integer lattice. Closure properties with such a representation under various operations including affine image and pre-image have been proved [GR07].

2.5 Putting the Notation Together

Let us put together the notation introduced so far. Let the statements of the program be S_1, S_2, …, S_m. Let $\mathbf{S}$ be the set of all statements. Let $\vec{n}$ be the vector of program parameters, i.e., typically loop bounds or symbols appearing in the loop bounds, access functions, or the statement body.

Let $G = (V, E)$ be the data dependence graph of the original program, i.e., $V = \mathbf{S}$ and E is the set of data dependence edges. $e^{S_i \rightarrow S_j} \in E$ denotes an edge from S_i to S_j, but we will often drop the superscript on e. For every edge $e \in E$ from S_i to S_j, let the dependence polyhedron be $\mathcal{P}_e$, the fact that a source iterations $\vec{s} \in \mathcal{D}^{S_i}$ and a target iteration $\vec{t} \in \mathcal{D}^{S_j}$ are dependent are known through the equalities and inequalities in the dependence polyhedron, and we express this fact by:

$$\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e \quad \Longleftrightarrow \quad \vec{s} \in \mathcal{D}^{S_i}, \vec{t} \in \mathcal{D}^{S_j} \text{ are dependent through edge } e^{S_i \rightarrow S_j} \in E \tag{2.11}$$

$\phi^k_{S_i}$ denotes the affine hyperplane or function for level k for S_i, $1 \leq k \leq d$. The set of all $\phi^k_{S_i}$, for $S_i \in \mathbf{S}$ represent the interleaving of all statement instances at level k. $\mathcal{T}_S$ is a d-dimensional affine function for each S as defined in (2.9). The subscript on ϕ^k is dropped when referring to the property of the function across all statements, since all statements instances are mapping to a target space-time with dimensions $\phi^1, \phi^2, \ldots, \phi^d$.

2.6 Legality and Parallelism

Dependence satisfaction. A dependence edge e with polyhedron $\mathcal{P}_e$ is *satisfied* at a level l iff l is the first level at which the following condition is met:

$$\forall k (1 \leq k \leq l-1) : \phi^k_{S_j}\left(\vec{t}\right) - \phi^k_{S_i}\left(\vec{s}\right) \geq 0 \quad \bigwedge \quad \phi^l_{S_j}\left(\vec{t}\right) - \phi^l_{S_i}\left(\vec{s}\right) \geq 1, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e$$

Legality. Statement-wise affine transformations ($\mathcal{T}_S$) as defined in (2.9) are legal iff

$$T_{S_j}(\vec{t}) - T_{S_i}(\vec{s}) \succ \vec{0}_d, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e, \forall e \in E \tag{2.12}$$

Definition 12 (**Permutable band**)**.** The ϕs at levels p, $p+1$, $\ldots$, $p+s-1$ form a permutable band of loops in the transformed space iff

$$\forall k \quad (p \leq k \leq p+s-1) : \quad \phi^k_{S_j}(\vec{t}) - \phi^k_{S_i}(\vec{s}) \geq 0, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e, e \in E_p \tag{2.13}$$

where E_p is the set of dependences not satisfied up to level $p-1$.

The above directly follows from (2.12). Loops within a permutable band can be freely interchanged or permuted among themselves. One can see that doing so will not violate (2.12) since dependence components for all unsatisfied dependences are non-negative at each of the dimensions in the band. We will later find the above definition a little conservative. Its refinement and associated intricacies will be discussed in Section 5.4.2 of Chapter 5.

Definition 13 (Outer parallel). A $\{\phi_{S1}, \phi_{S2}, \dots, \phi_{S_m}\}$ is an outer parallel hyperplane if and only if

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) = 0, \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e, \quad \forall e \in E$$

Outer parallelism is also often referred to as communication-free parallelism or synchronization-free parallelism.

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] = a[i][j-1] + 1;
  }
}
```

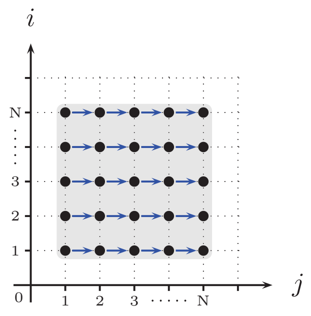


Figure 2.9: Outer parallel loop, i: hyperplane (1,0)

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        a[i][j] = a[i-1][N-1-j] + 1;
    }
}
```

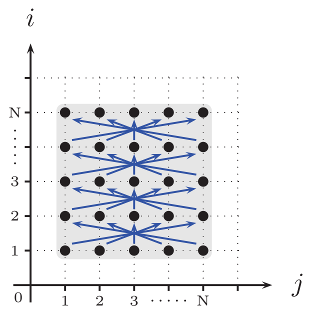


Figure 2.10: Inner parallel loop, j: hyperplane (0,1)

Definition 14 (Inner parallel). A $\{\phi^k_{S1}, \phi^k_{S2}, \dots, \phi^k_{S_m}\}$ is an inner parallel hyperplane if and only if $\phi^k_{S_i}(\vec{t}) - \phi^k_{S_i}(\vec{s}) = 0$, for every $\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e$, $e \in E_k$, where E_k is the set of dependences not satisfied up to level $k-1$.

It is illegal to move an inner parallel loop in the outer direction since the dependences satisfied at loops it has been moved across may be violated at the new position of the moved loop. However, it is always legal to move an inner parallel loop further inside.

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        a[i][j] = a[i][j-1] + a[i-1][j];
    }
}
```

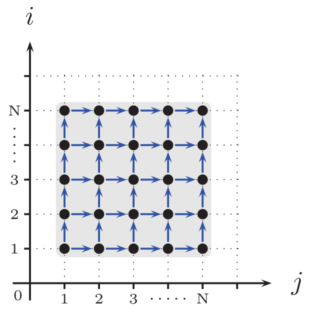


Figure 2.11: Pipelined parallel loop: i or j

Outer and inner parallelism is often referred to as doall parallelism. However, note that inner parallelism requires synchronization every iteration outer to the loop.

Pipelined parallelism. Two or more loops may have dependences that have dependence components along each of them can still be executed in parallel if one of them can be delayed with respect to the other by a fixed amount. If dependence components are non-negative along each of the dimensions in question, one just needs a delay of one. Figure 2.11 shows a code with dependences along both i and j. However, say along i, successive iterations can start with a delay of one and continue executing iterations for j's in sequence. Similarly, if there are n independent dimensions, at most $n-1$ of them can be pipelined while iterations along at least one will be executed in sequence. Pipelined parallelism is also often referred to as doacross parallelism. We will formalize conditions for this in a very general setting in Chapter 3 since it is goes together with tiling. Code generation for pipelined parallelism is discussed in Chapter 5.

Space-time mapping. Once properties of each row of $\mathcal{T}_S$ are known, some of them can be marked as space, i.e., a dimension along which iterations are executed by different processors, while others can be marked as time, i.e., a dimension that is executed sequentially by a single processor. Hence, $\mathcal{T}_S$ specifies a complete space-time mapping for S. Each of the d dimensions is either space or time. Since M_S is of full column rank, when an iteration executes and where it executes, is known. However, in reality, post-processing can be done to $\mathcal{T}_S$ before such a mapping is achieved.