OPTIMIZING DATA LOCALITY AND PARALLELISM FOR SCALABLE MULTIPROCESSORS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Ashish Narayan
B.E., Sri Jayachamarajendra College of Engineering, India, 1986
May 1994

ACKNOWLEDGMENTS

Acknowledgments

I wish to thank Dr. J. Ramanujam for giving me the opportunity to work with him in the field of Parallelizing Compilers. He was of tremendous inspiration in the entire course of research and was always filled with ideas.

I also would like to thank Prof. Subhash Kak and Dr. R. Vaidyanathan for being on my committee.

I wish to thank Dr. Coon and Dr. Advokat in the Department of Psychology, LSU for all their support during my study at LSU.

I would like to thank my parents for all the encouragement they have given to pursue my graduate studies and to my wife Padmashri for all the support she has given during the course of my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

ABSTRACT

As processor speeds continue to increase at a much higher rate than memory speeds, improving memory performance becomes critical in realizing high performance in parallel machines. The aim is to have most of the memory accesses to faster memory in the memory hierarchy resulting in high levels of data locality. In many cases, it is best left to an optimizing compiler to restructure the program to extract good memory performance. The programmer will otherwise have to write code to exploit the parallelism inherent in the input code and also to improve data locality. In scientific codes, since most of the CPU time is spent in loops, we study the effect of restructuring loops on the performance. Loop transformations to exploit parallelism and to improve data locality are the most important techniques used today. Data locality and parallelism are conflicting goals and there is always a trade-off between the two. Improving data locality is aimed at reducing communication between processors or between different levels in the memory hierarchy. An important aspect in good memory performance is the estimation of the amount of local memory needed for an array accessed in a nested loop. Data transfers between levels and processors has to optimized for the best performance.
This thesis presents loop transformations to automatically restructure code at compile time which aims in improving data locality. We also present an algorithm to obtain an optimum compound loop transformation for task allocation and data distribution on distributed memory machines which yields in optimal communication and efficient use of block transfers, resulting in message vectorization. Finally, we discuss a technique to estimate the number of distinct references to an array which facilitates efficient data transfers to local memory.

CHAPTER 1

INTRODUCTION

In this thesis, the problem of automating the process of transforming programs to optimize the utilization of the memory hierarchy is considered. Data locality is highly critical in achieving high performance on parallel machines. Non-local data accesses result in communication which greatly affects the performance. Improving data locality is achieved by bringing iterations that access the same data as close (in time) as possible while preserving dependences. The problem can be addressed by the application of compound transformations. Loop transformations such as interchange, skewing and reversal have been known to be useful in parallelism and efficient use of memory hierarchy. These transformations can be modeled as elementary matrix transformations. Compound transformations can be derived as products of these elementary transformations. Our work is based on this matrix representation of compound loop transformations.
Chapter 2 discusses the problem of compile time analysis and improvement of data locality. We use the term *reuse* to mean that the same data item is used in more than one iteration of the loop. A detailed analysis on the effect of unimodular transformations on data locality including the calculation of the maximum window size is discussed. The term *maximum window size* is introduced and quantitative expressions are derived to compute the window size. The smaller the value of the maximum window size, the higher the amount of data locality in the loop.
Chapter 3 deals with the problem of optimizing parallelism and locality on distributed memory machines. The mapping of iterations and the distribution of data on to processors are discussed. An algorithm is presented to derive loop transformations and generate code which map iterations on to processors and include block transfers where ever possible. The derived data distribution ensures that if references to non-local data become necessary, they are grouped together for block transfers, resulting in *message vectorization.*
Chapter 4 addresses the problem of estimating the amount of local memory needed for transfers of data in the memory hierarchy. The problem of estimating the region associated with a statement or the set of elements referenced by a statement during the execution of the entire set of nested loops is analyzed. A quantitative analysis of the number of elements referenced is presented; exact expressions for uniformly generated references and a close upper and lower bound for non-uniformly generated references are derived. Chapter 5 concludes with a summary and points to further research.

1.1 RELATED WORK

Our work on loop transformations for improving data locality bears most similarity to work by Gannon *et al.* [10, 9] and Eisenbeis *et al.* [6]. They define the notion of a reference window for each dependence arising from uniformly generated references.

Unlike our work, they do not use compound transformations – only interchange and reversal are considered. In addition, the use of a reference window and the resultant need to approximate the combination of these windows results in a loss of precision; tileability is not considered in [6]. Fang and Lu [7] present a method to reduce cache thrashing that requires expensive runtime calculation of which iterations are to be executed by the same processor in order to maximize locality. Ferrante *et al.* [8] present a formula that estimates the number of distinct references to array elements; their technique does not use dependence information. Kennedy and McKinley [12] discuss a strategy to decide the best ordering of loops for locality; their technique is widely applicable but they do not use compound transformations which can dramatically increase locality. Wolf and Lam [13, 19] develop an algorithm that estimates temporal and spatial reuse of data using *localized vector space.* Their algorithm combines reuses from multiple references. Their method does not use loop bounds and the estimates used are less precise than the ones presented here. Their method performs an exhaustive search of loop permutations that maximizes locality. Li and Pingali [14] discuss the completion of partial transformations derived from the data access matrix of a loop nest; the rows of the data access matrix are selected subscript functions for various array accesses (excluding constant offsets). While their technique exploits reuse arising from input and output dependences, it does not work well with flow or anti-dependences.

Of these, the work discussed in [6, 10, 14, 19] are most related to our work. Our work is restricted to uniformly generated references like that of [9, 10, 6], while providing a method to synthesize transformations that exploit locality. When applied to uniformly generated references, our work improves locality by a greater factor than [14, 19]. In addition, none of these works address the problem of improving single processor performance (improving local memory access cost), while reducing non-local accesses. These two problems can not be dealt with separately. The framework presented in Chapter 2 allows us to address both the problems in a unified manner.

Techniques for optimizing parallelism and locality have been presented by [14, 12]. Li and Pingali [14] use the concept of access normalization for restructuring loops to exploit data locality. They use the *basis matrix* and the *padding matrix* that are contructed from the rows of the *data access matrix* (which is a single matrix for the entire loop nest) to derive a transformation matrix and use an ordering of the rows. We use the concept of a *data reference matrix,* which is similar to the data access matrix, but is defined for each distinct reference to every array in the loop nest. We present an algorithm in Chapter 3, where we systematically derive a transformation matrix to obtain maximum data locality. The algorithm generates the best possible transformation which gives the optimum data distribution and block transfers where ever possible. Kennedy and McKinley [12] choose specific loop ordering for optimizing data locality and generating parallel code on shared memory machines. They do not look at compound loop transformations which can result in data distribution techniques which in turn yields optimum locality. Anderson and Lam [3] discuss optimizing parallelism and locality using the concept of data and computation decomposition onto the processors.

The estimation of the number of references to an array in order to predict cache effectiveness in hierarchical memory machines have been discussed by Ferrante *et al.* [8] and Gallivan *et al.* [9]. The image of the iteration space onto the array space to optimize global transfers have been discussed in [9]. A framework for estimating bounds for the number of elements accessed only was given. Ferrante *et al.* gave exact values for uniformly generated references but did not consider multiple references. Also, for non-uniformly generated references, arbitrary correction factors were given for arriving at lower and upper bounds for the number of distinct references. We present a technique in Chapter 4 which gives accurate results for most practical cases and very close bounds where ever necessary.

CHAPTER 2

LOOP TRANSFORMATIONS FOR IMPROVING DATA LOCALITY

Optimizations aimed at improving locality have become extremely important as memory bandwidth and latency are becoming the dominant limiting factors in achieving high performance. This chapter presents a technique for choosing transformations that enhance locality while enabling parallelization. We use the reference window size associated with a given loop structure as a measure of locality; the window associated with a loop nest is the set of data that is referenced up to some point in execution that will be referenced again later. The reference window represents reused data. Exploiting locality involves bringing computations that reuse data closer in time subject to data dependences. We show a method of estimating the window size as a function of an arbitrary unimodular transformation applied to the loop nest. The method does not involve symbolic manipulation of linear inequalities. The transformation that minimizes the window size subject to dependences is chosen. We show an example to compare the effectiveness of our technique with extant techniques aimed at locality enhancement.

In this chapter, we address the problem of compile-time analysis and improvement of data locality; we focus our attention on loops which are common to scientific codes. We use the term *reuse* to mean that the same data item is used in more than one iteration of a loop nest [19]. We are primarily interested in cases where the number of times a data item is reused is a function of the loop limits. Improving data locality is achieved by bringing iterations that access the same data as close (in time) as possible while preserving dependences and retaining parallelism; parallelism and data locality are conflicting goals. In Section 2.1, we present the problem framework and discuss related work, highlighting how our method differs from other published work in exploiting data locality. Section 2.2 motivates the concept through the use of examples, which illustrate how the techniques presented here improve upon extant techniques. In Section 2.3, we present detailed analysis of data locality and derive closed-form expressions for the impact of a unimodular transformation on locality including the calculation of maximum window size. Section 2.4 discusses locality issues for 3-nested loops.

2.1 BACKGROUND AND FRAMEWORK

2.1.1 TERMINOLOGY

The set of computations in a perfectly nested loop is called the *computation domain.* Here only the structural information of the algorithm (nested loop), *i.e.,* the computation domain and the dependence matrix D are needed. With the assumption on loop bound linearity, the sets of computations considered are finite convex polyhedra of some iteration

space Z^K, where K is the number of loops in the nest which is also the dimensionality of the iteration space. Each element, an iteration of the loop body S, is referred to by its iteration vector $\vec{J}$.
The iteration set $J \subset Z^K$ of any given algorithm is described as a set of integer points (vectors) whose convex hull $\mathcal{I}$ is a non-degenerate (full dimensional) polyhedron, i.e.,

$$\mathcal{I} = \left\{\vec{x} : P\vec{x} \leq \vec{q}, P \in Z^{a \times K}, \vec{q} \in Z^a, \vec{x} \in Z^K\right\}$$

where a is a positive integer and $J \subseteq \left\{\vec{j} : \vec{j} \in \mathcal{I} \wedge j \in Z^K\right\}$. Matrix P is called the *constraint matrix* of J; vector $\vec{q}$ is called the *size vector.* The above relation simply states that $\mathcal{I}$ is a polyhedron. Extreme points of polyhedron $\mathcal{I}$ are always integers and belong to the iteration space J because $\mathcal{I}$ is the convex hull of iteration space J. Different *size vectors* correspond to instances of the same algorithm that differ only in their sizes (but have the same shape of the computation domain). The following example illustrates the concepts introduced here.

for $i = 1$ to N do
  for $j = i + 1$ to M do
    $\cdots$
  endfor
endfor

The computation domain is given by:

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} -1 \\ N \\ -1 \\ M \end{bmatrix}$$

For convenience, we represent the computation domain as the matrix

$$\left[\begin{array}{cc|c} -1 & 0 & -1 \\ 1 & 0 & N \\ 1 & -1 & -1 \\ 0 & 1 & M \end{array}\right]$$

which is just the P matrix augmented with the size vector $\vec{q}$.

2.1.2 DATA DEPENDENCES AND LOOP TRANSFORMATIONS

Dependence analysis is critical to the success of parallelizing compilers [1, 4, 21, 23]. We deal with sets of perfectly nested loops, whose upper and lower bounds are all linear, enclosing a loop body with affine array references. That is, each subscript of an array variable index expression must be an affine expression over the scalar integer variables of the program. We assume that the loop body does not contain any goto's exiting one or many loops and that all statements are at the same nesting level. We assume familiarity with definitions of the types of dependences.

Dependence relations are often represented in *Iteration Space Graphs* (ISG's); for a K-nested loop with index set $(I_1, I_2, \ldots, I_K)$, the nodes of the ISG are points on a K-dimensional discrete Cartesian space and a directed edge exists between the iteration defined by $\vec{I_1}$ and the iteration defined by $\vec{I_2}$ whenever a dependence exists between statements in the loop constituting the iterations $\vec{I_1}$ and $\vec{I_2}$. Many dependences that occur in practice have a constant distance in each dimension of the iteration space. In such cases, the vector $\vec{d} = \vec{I_2} - \vec{I_1}$ is called the `distance vector`. An algorithm has a number of such dependence vectors; the dependence vectors of the algorithm are written collectively as a `dependence matrix` $D = [\vec{d_1}, \vec{d_2}, \ldots, \vec{d_n}]$. Methods to calculate data dependence vectors can be found in [1, 4, 20, 21, 23].

A dependence matrix a collection of dependence distance vectors for a nested loop – the distance vectors are the columns of the dependence matrix. For a K-nested loop with n dependence vectors $d_i, i = 1, \ldots, n$, the dependence matrix D is a $K \times n$ matrix. Let $i_1, i_2, \ldots, i_K$ be the loop index variables for the loop nest levels. The vector $(i_1, i_2, \ldots, i_K)$ is referred to as the *nest vector.*

Loop transformations are applied to the loop nest vector and the dependence matrix. Let $\vec{I} = (I_1, \ldots, I_K)$ and $\vec{J} = (J_1, \ldots, J_K)$ be two iterations of a nested loop such that $\vec{I} \prec \vec{J}$ (read $\vec{I}$ precedes $\vec{J}$ in sequential execution) and there is a dependence of constant distance $\vec{d_i}$

between them. Applying a linear transformation T to the iteration space (nest vector) also changes the dependence matrix since $T(\vec{J}) - T(\vec{I}) = T(\vec{J} - \vec{I}) = T\ \vec{d_i}$. All dependence vectors are *positive vectors,* i.e., the first non-zero component should be positive. We do not include *loop-independent* dependences which are zero vectors. A transformation is legal if the resulting dependence vectors are still *positive* vectors [4]. Thus, the linear algebraic view leads to a simpler notion of the legality of a transformation. For a K-nested sequential loop, the $K \times K$ identity matrix (I_K) denotes sequential execution order. Any unimodular transformation can be realized through reversal, interchange and skewing [5, 18].

2.1.3 Iteration Space Tiling

Loop tiling [2, 11, 15, 16, 18, 21, 22] is a transformation that decomposes a K-dimensional iteration space into K-dimensional "blocks" composed of a number of iterations where each block executes atomically; hence there should be no dependence cycles (deadlock) among the blocks. Tiling is extremely useful in distributed memory machines and hierarchical shared memory machines in exploiting locality of reference in the presence of dependences.

Tiles in K-dimensional (K-D) spaces are defined by K families of parallel hyperplanes (or planes) each of which is a $K - 1$ dimensional hyperplane. To form K-dimensional tiles from a K-dimensional iteration space (assuming that the dependence matrix D (a $K \times n$ matrix) whose columns are dependence vectors has a rank K), the vectors normal to the K tile boundaries should be linearly independent. Let H denote the $K \times K$ matrix whose rows are the vectors $\vec{h_1}, \ldots, \vec{h_K}$. Thus, the rank of H must be K in order to define K-D tiles. In matrix notation, let $D^+ = HD$. The tiling condition implies that, for tiles to be legal [11, 16],

$$D^+_{ij} \geq 0 \qquad i = 1, \ldots, K \qquad j = 1, \ldots, n. \tag{2.1}$$

All entries in D^+ are non-negative. Since, the rows of H are linearly independent,

H must be non-singular and H^{-1} exists. Thus, the dependence matrix D can written as

$$D = H^{-1}D^{+}, \quad \text{where } D^{+}_{ij} \geq 0. \tag{2.2}$$

2.1.4 ASSUMPTIONS

We restrict our analysis to affine array accesses in perfectly nested loops whose upper and lower bounds are affine. We primarily deal with loop nests with reuse *i.e.,* access to one dimensional arrays in doubly-nested loops, 1- or 2-dimensional arrays in triply nested loops *etc.* Data dependence is used for both locality and parallelism. We assume that all array references are *uniformly generated* [10, 6].

for $i_1 = l_1$ to u_1 do
  for $i_2 = l_2$ to u_2 do
    ...
      for $i_n = l_n$ to u_n do
        $\mathcal{R}_1[A_1(i_1, \ldots, i_n) + \vec{b_1}]$
        $\mathcal{R}_2[A_2(i_1, \ldots, i_n) + \vec{b_2}]$
        ...
        $\mathcal{R}_k[A_k(i_1, \ldots, i_n) + \vec{b_k}]$
      endfor
  endfor
endfor

where $\mathcal{R}_1, \ldots, \mathcal{R}_k$ are references to arrays not necessarily distinct. A_i are matrices of size $n \times d_i$ where d_i is the dimension of the array referenced in $\mathcal{R}_i$; $\vec{b_i}$ is a vector of size d_i. We assume that all array reference to an array X are uniformly generated, i.e., A_i for all references to X are equal. An example of a loop with a uniformly generated references is shown below:

for $i = 1$ to N_1 do
  for $j = 1$ to N_2 do
    $X[\boxed{2i + 3j}+2] = Y[\boxed{i + j}]$
    $Y[\boxed{i + j}+1] = X[\boxed{2i + 3j}+3]$
  endfor
endfor

Here, the two references to array X are of the form $\boxed{2i + 3j + \text{constant}}$ and both references to array Y are of the form $\boxed{i + j + \text{constant}}$. We use the notion of a reference window of an array in loop nest (which is different form the notion of the reference window of a dependence as used by [6, 10]) that allows us to deal with each distinct array as a whole and not on a per-reference-pair-to-the-array basis.

The reference window $W_X(\vec{I})$ (where $\vec{I} = (I_1, \ldots, I_n)$ is a point in an n-dimensional iteration space graph) is the set of all elements of array X that are referenced by any of the statements in all iterations $\vec{J_1} \preceq = \vec{I}$ (read $\vec{J_1}$ precedes or is equal to $\vec{I}$) that are also referenced in some (later) iteration $\vec{J}_2$ such that $\vec{J}_2 \succ \vec{I}$ (read $\vec{J}_2$ follows $\vec{I}$). This allows us to precisely the define those iterations which need a specific value in local memory. The size of the window $W_X(\vec{I})$ is the number of elements in that window. The maximum reference window size is given by

$$\max_{\vec{I}} \left| W_X(\vec{I}) \right|$$

and is defined over the entire iteration space. In the presence of multiple arrays $X_1, \ldots, X_K$, the maximum reference window size is:

$$\max_{\vec{I}} \sum_{j=1}^{K} \left| W_{X_j}\left(\vec{I}\right) \right|$$

The reference window is a dynamic entity, whose shape and size changes with execution. For nested loops with uniformly generated references, the maximum window size (MWS) is a function of the loop limits. The smaller the value of MWS, the higher the amount of data locality in the loop nest for the array. For simplicity of exposition, we assume that there are multiple uniformly generated references to a single array in a loop nest. The results derived here easily generalize to multiple arrays and higher levels of nesting [17].

2.2 Motivation and Examples

Consider the following example which is a minor variant of the example from [6]:

Example 1:

for $i = 1$ to 20 do
  for $j = 1$ to 30 do
    $\cdots X[2i - 3j] \cdots$
  endfor
endfor

Eisenbeis *et al.* [6] mention that the cost of the window (the same as MWS) for this loop is 89. They use only two transformations: loop interchange and reversal. On applying interchange, the MWS reduces to 41. On reversal applied to the original loop, the cost becomes 86 while reversing the interchanged loop reduces the cost to 36. Using the technique presented here, the cost or MWS for this loop can be reduced to 1, i.e., all iterations accessing any element of the array X can be made consecutive iterations of an inner loop. The only dependence in this example is the vector $(3, 2)$. We use the following legal transformation, T:

$$T = \begin{bmatrix} 2 & -3 \\ 1 & -1 \end{bmatrix}$$

Even though the technique in [14] can be used to derive this transformation, there are situations where the techniques presented here *improves* locality while that in [14] does not improve locality. Consider the loop shown in Example 2:

Example 2:

for $i = 1$ to 25 do
  for $j = 1$ to 10 do
    $X[2i + 5j + 1] = X[2i + 5j + 5]$
  endfor
endfor

The distance vectors for this loop are: $(3, -2), (2, 0), (5, -2)$; $(3, -2)$ is the flow dependence, $(2, 0)$ is an anti-dependence and $(5, -2)$ is the output dependence vector. These are the only direct dependences. Li and Pingali use transformation matrices whose first row is either $(2, 5)$ or $(-2, -5)$. Any transformation that uses $(2, 5)$ as its first row is illegal because of the distance vector $(3, -2)$; the first component of $(3, -2)$ after the transformation is $((2, 5) \cdot (3, -2)^T = -4$ is $< 0)$. Similarly any transformation that uses $(-2, -5)$ as its first row is illegal due to the distance vector $(2, 0)$ since $((-2, -5) \cdot (2, 0)^T = -4$ is $< 0)$. The maximum window size is 50. Li and Pingali's technique will not find any partial transformation that can be completed to a legal transformation. Where as, by applying techniques presented in the following sections, we can apply the legal transformation, T:

$$T = \begin{bmatrix} 2 & 3 \\ 1 & 1 \end{bmatrix}$$

Applying T reduces the maximum window size to 21. A combination of reversal and interchange does not change the maximum window size from 50.

2.3 Effect of Transformations on Locality

Consider a nested loop with r uniformly generated references to an array X of the form: $\lambda_1 i + \lambda_2 j + c_k$ $(k = 1, \ldots, r)$ as shown below:

Example 3:

for $i = 1$ to N_1 do
  for $j = 1$ to N_2 do
    $\cdots X[\lambda_1 i + \lambda_2 j + c_1] \cdots$
    $\cdots X[\lambda_1 i + \lambda_2 j + c_2] \cdots$
    $\cdots$
    $\cdots X[\lambda_1 i + \lambda_2 j + c_r] \cdots$
  endfor
endfor

Table 2.1: Table of outer loop limits on applying transformation T

Values of l_1 and u_1 on applying T			
No.	Condition	l_1	u_1
1.	$a > 0, b > 0$	$a + b$	$aN_1 + bN_2$
2.	$a > 0, b < 0$	$a + bN_2$	$aN_1 + b$
3.	$a < 0, b > 0$	$aN_1 + b$	$a + bN_2$
4.	$a < 0, b < 0$	$a + b$	$aN_1 + bN_2$

We need to compute the effect of a legal unimodular transformation, T:

$$T = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

on the maximum window size . In addition to legality, we require that the loop nest be *tileable* [11, 16, 19, 22]; this permits us to use block transfers in the case of shared memory machines and fewer (but larger) messages in message passing machines in order to reduce the effect of high message startup costs.
The optimum transformation thus satisfies two conditions:

1. legality condition for tiling
2. minimizes the maximum window size

The determinant of the unimodular transformation, $T = \Delta(T) = ad - bc = \pm 1$. If $\Delta = 1$, then

$$T^{-1} = \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

If $\Delta = -1$, then

$$T^{-1} = \begin{bmatrix} -d & b \\ c & -a \end{bmatrix}$$

Let x and y be the new loop outer and inner indices respectively on applying T. Let l_1 and u_1 be the lower and upper limits of the outer loop x; let l_2 and u_2 be the lower and upper limits of the inner loop y. The values of l_1 and u_1 are summarized in Table 2.1. Let the loop limits be given in the form of a set of inequalities: $P\vec{I} \leq \vec{q}$. On applying T, the new loop limits are given by the relation, $PT^{-1}\vec{I'} \leq \vec{q}$ since $\vec{I'} = T\vec{I}$. For $\Delta = 1$, the relations are:

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} -1 \\ N_1 \\ -1 \\ N_2 \end{bmatrix} \implies \begin{bmatrix} -d & b \\ d & -b \\ c & -a \\ -c & a \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} -1 \\ N_1 \\ -1 \\ N_2 \end{bmatrix}$$

Table 2.2: Table of inner loop limits on applying transformation T with $\Delta(T) = 1$

Values of l_2 and u_2 on applying T with $\Delta(T) = 1$			
No.	Condition	l_2	u_2
1.	$a > 0, b > 0$	$max\left(\left\lceil\frac{dx-N_1}{b}\right\rceil, \left\lceil\frac{1+cx}{a}\right\rceil\right)$	$min\left(\left\lfloor\frac{dx-1}{b}\right\rfloor, \left\lfloor\frac{N_2+cx}{a}\right\rfloor\right)$
2.	$a > 0, b < 0$	$max\left(\left\lceil\frac{dx-1}{b}\right\rceil, \left\lceil\frac{1+cx}{a}\right\rceil\right)$	$min\left(\left\lfloor\frac{dx-N_1}{b}\right\rfloor, \left\lfloor\frac{N_2+cx}{a}\right\rfloor\right)$
3.	$a < 0, b < 0$	$max\left(\left\lceil\frac{dx-1}{b}\right\rceil, \left\lceil\frac{N_2+cx}{a}\right\rceil\right)$	$min\left(\left\lfloor\frac{dx-N_1}{b}\right\rfloor, \left\lfloor\frac{1+cx}{a}\right\rfloor\right)$
4.	$a < 0, b > 0$	$max\left(\left\lceil\frac{dx-N_1}{b}\right\rceil, \left\lceil\frac{N_2+cx}{a}\right\rceil\right)$	$min\left(\left\lfloor\frac{dx-1}{b}\right\rfloor, \left\lfloor\frac{1+cx}{a}\right\rfloor\right)$

Table 2.3: Table of inner loop limits on applying transformation T with $\Delta(T) = -1$

Values of l_2 and u_2 on applying T with $\Delta(T) = -1$			
No.	Condition	l_2	u_2
1.	$a > 0, b > 0$	$max\left(\left\lceil\frac{dx+1}{b}\right\rceil, \left\lceil\frac{cx-N_2}{a}\right\rceil\right)$	$min\left(\left\lfloor\frac{dx+N_1}{b}\right\rfloor, \left\lfloor\frac{cx-1}{a}\right\rfloor\right)$
2.	$a > 0, b < 0$	$max\left(\left\lceil\frac{dx+N_1}{b}\right\rceil, \left\lceil\frac{cx-N_2}{a}\right\rceil\right)$	$min\left(\left\lfloor\frac{dx+1}{b}\right\rfloor, \left\lfloor\frac{cx-1}{a}\right\rfloor\right)$
3.	$a < 0, b < 0$	$max\left(\left\lceil\frac{dx+N_1}{b}\right\rceil, \left\lceil\frac{cx-1}{a}\right\rceil\right)$	$min\left(\left\lfloor\frac{dx+1}{b}\right\rfloor, \left\lfloor\frac{cx-N_2}{a}\right\rfloor\right)$
4.	$a < 0, b > 0$	$max\left(\left\lceil\frac{dx+1}{b}\right\rceil, \left\lceil\frac{cx-1}{a}\right\rceil\right)$	$min\left(\left\lfloor\frac{dx+N_1}{b}\right\rfloor, \left\lfloor\frac{cx-N_2}{a}\right\rfloor\right)$

If $b > 0$, then $\frac{dx-N_1}{b} \leq y \leq \frac{dx-1}{b}$; and if $a > 0$, then $\frac{1+cx}{a} \leq y \leq \frac{N_2+cx}{a}$; Thus the lower limit of the inner loop is:

$$max\left(\left\lceil\frac{dx - N_1}{b}\right\rceil, \left\lceil\frac{1 + cx}{a}\right\rceil\right)$$

and the upper limit is:

$$min\left(\left\lfloor\frac{dx - 1}{b}\right\rfloor, \left\lfloor\frac{N_2 + cx}{a}\right\rfloor\right)$$

For $\Delta = -1$, the loop limits on applying T are given by the set of inequalities:

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -d & b \\ c & -a \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} -1 \\ N_1 \\ -1 \\ N_2 \end{bmatrix} \Longrightarrow \begin{bmatrix} d & -b \\ -d & b \\ -c & a \\ c & -a \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} -1 \\ N_1 \\ -1 \\ N_2 \end{bmatrix}$$

The lower limits of the inner loop, namely l_2 and u_2 for $\Delta = 1$ are given in Table 2.2 and for $\Delta = -1$ are given in Table 2.3.

On applying T, the loop is transformed into:
Example 4:
for $x = l_1$ to u_1 do
  for $y = l_2$ to u_2 do
    $\cdots X[\lambda_1' x + \lambda_2' y + c_1] \cdots$
    $\cdots X[\lambda_1' x + \lambda_2' y + c_2] \cdots$
    $\cdots$
    $\cdots X[\lambda_1' x + \lambda_2' y + c_r] \cdots$
  endfor
endfor

where

$$\begin{bmatrix} \lambda_1' & \lambda_2' \end{bmatrix} = \begin{cases} \begin{bmatrix} \lambda_1 & \lambda_2 \end{bmatrix} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \begin{bmatrix} \lambda_1 d - \lambda_2 c & \lambda_2 a - \lambda_1 b \end{bmatrix} & \text{if } \Delta = 1 \\ \begin{bmatrix} \lambda_1 & \lambda_2 \end{bmatrix} \begin{bmatrix} -d & b \\ c & -a \end{bmatrix} = \begin{bmatrix} \lambda_2 c - \lambda_1 d & \lambda_1 b - \lambda_2 a \end{bmatrix} & \text{if } \Delta = -1 \end{cases}$$

The maximum window size (MWS) is a function of the maximum inner loop span, *i.e.,* the maximum trip count of the inner loop (difference between the upper and lower limits of the inner loop) over all outer loop iterations [6].

$$\text{MWS} = \text{maxspan} \times \Delta \times (\lambda_2 a - \lambda_1 b) \tag{2.3}$$

2.3.1 Calculation of Span

Consider the case of $\Delta = 1, a > 0$ and $b > 0$. Then the lower limit of the inner loop (l_2) is:

$$l_2 = max\left(\left\lceil \frac{dx - N_1}{b} \right\rceil, \left\lceil \frac{1 + cx}{a} \right\rceil\right)$$

and the upper limit (u_2) is:

$$u_2 = min\left(\left\lfloor \frac{dx - 1}{b} \right\rfloor, \left\lfloor \frac{N_2 + cx}{a} \right\rfloor\right)$$

We ignore the floors and ceilings in calculating a close approximation to the span. We calculate values of l_2 and u_2 as follows:

$$l_2 = \frac{dx - N_1}{b} \quad \text{if} \quad \frac{dx - N_1}{b} \geq \frac{1 + cx}{a},$$

i.e., $l_2 = \frac{dx-N_1}{b}$ if $x \geq aN_1 + b$. Thus,

$$l_2 = \begin{cases} \frac{dx-N_1}{b} & \text{if } x \geq aN_1 + b \\ \frac{1+cx}{a} & \text{if } x < aN_1 + b \end{cases}$$

Similarly, we derive the upper limit, u_2:

$$u_2 = \begin{cases} \frac{dx-1}{b} & \text{if } x \leq a + bN_2 \\ \frac{N_2+cx}{a} & \text{if } x > a + bN_2 \end{cases}$$

Now, we can calculate the span of the inner loop (y loop) based on whether $a + bN_2$ is greater than or equal to or less than $b + aN_1$. Recall that $\Delta = 1, a > 0$ and $b > 0$. If $b + aN_1 \geq a + bN_2$, *i.e.,* $a - b \leq aN_1 - bN_2$ (note that $bc - ad = -\Delta = -1$)

$$span(y) = \begin{cases} \frac{N_2+cx}{a} - \frac{dx-N_1}{b} + 1 = \frac{bN_2+aN_1-x}{ab} + 1 & \text{if } x \geq b + aN_1 \\ \frac{N_2+cx}{a} - \frac{cx+1}{a} + 1 = \frac{N_2-1}{a} + 1 & \text{if } a + bN_2 \leq x \leq b + aN_1 \\ \frac{dx-1}{b} - \frac{1+cx}{a} + 1 = \frac{x-a-b}{ab} + 1 & \text{if } x < a + bN_2 \end{cases}$$

Similarly if $b + aN_1 \leq a + bN_2$, *i.e.,* $a - b \geq aN_1 - bN_2$,

$$span(y) = \begin{cases} \frac{N_2+cx}{a} - \frac{dx-N_1}{b} + 1 = \frac{bN_2+aN_1-x}{ab} + 1 & \text{if } x \geq a + bN_2 \\ \frac{dx-1}{b} - \frac{dx-N_1}{b} + 1 = \frac{N_1-1}{b} + 1 & \text{if } b + aN_1 \leq x \leq a + bN_2 \\ \frac{dx-1}{b} - \frac{1+cx}{a} + 1 = \frac{x-a-b}{ab} + 1 & \text{if } x < b + aN_1 \end{cases}$$

Thus, the maximum span *maxspan* is:

$$maxspan(y) = \begin{cases} \frac{N_1-1}{b} + 1 & \text{if } a - b \geq aN_1 - bN_2 \\ \frac{N_2-1}{a} + 1 & \text{if } a - b \leq aN_1 - bN_2 \end{cases}$$

The maximum span is calculated in a similar fashion for other cases. In general, the maximum span is given by:

$$maxspan(y) = \begin{cases} \left|\frac{N_1-1}{b}\right| + 1 & \text{if } a - b \geq aN_1 - bN_2 \\ \left|\frac{N_2-1}{a}\right| + 1 & \text{if } a - b \leq aN_1 - bN_2 \end{cases}$$

2.3.2 Calculation of Maximum Window Size (MWS)

Based on equation (1) and results derived in the last section, the maximum window size (MWS) is given by:

$$MWS = \begin{cases} \left(\left|\frac{N_1-1}{b}\right| + 1\right) |\lambda_2 a - \lambda_1 b| & \text{if } a - b \geq aN_1 - bN_2 \\ \left(\left|\frac{N_2-1}{a}\right| + 1\right) |\lambda_2 a - \lambda_1 b| & \text{if } a - b \leq aN_1 - bN_2 \end{cases} \qquad (2.4)$$

Thus to minimize the maximum window size, the value of MWS from equation (2) should be minimized among all unimodular transformations T that are valid for tiling. In many cases, MWS is minimized when $|\lambda_2 a - \lambda_1 b|$ is minimized.

2.3.3 Legal Transformation

Let $\vec{d_i} = (d_{i,1}, d_{i,2})$ $(i = 1, \ldots, m)$ be a set of dependence distance vectors. With uniformly generated references, all the dependences in a nested loop are distance vectors. Given any two uniformly generated references $\lambda_1 i + \lambda_2 j + c_1$ and $\lambda_1 i + \lambda_2 j + c_2$, to test for a dependence from iteration (i_1, i_2) to iteration (j_1, j_2), we check for integer solutions within the loop range to the equation:

$$\lambda_1 i_1 + \lambda_2 i_2 + c_1 = \lambda_1 j_1 + \lambda_2 j_2 + c_2$$

i.e.,

$$\lambda_1(j_1 - i_1) + \lambda_2(j_2 - i_2) = c_1 - c_2.$$

We can write $x_1 = j_1 - i_1$ and $x_2 = j_2 - i_2$ where (x_1, x_2) is a distance vector. Since $\lambda_1, \lambda_2, c_1, c_2$ are constants, every solution gives a distance vector. The smallest lexicographically positive solution is the dependence vector of interest. In order for the transformation T to render the loop nest tileable, the following conditions must hold:

$$ad_{i,1} + bd_{i,2} \geq 0 \qquad i = 1, \cdots, m$$

$$cd_{i,1} + dd_{i,2} \geq 0 \qquad i = 1, \cdots, m$$

We illustrate the use of technique through Example 2. Consider the loop nest:

for $i = 1$ to 25 do
  for $j = 1$ to 10 do
    $X[2i + 5j + 1] = X[2i + 5j + 5]$
  endfor
endfor

The distance vectors for this loop are: $(3, -2), (2, 0), (5, -2)$. The problem here is to find a unimodular transformation T:

$$T = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

such that the loop is tileable, *i.e.,*

$$3a - 2b \geq 0 \qquad (2.5)$$
$$2a \geq 0 \qquad (2.6)$$
$$5a - 2b \geq 0 \qquad (2.7)$$
$$3c - 2d \geq 0 \qquad (2.8)$$
$$2c \geq 0 \qquad (2.9)$$

$$5c - 2d \geq 0 \tag{2.10}$$

and the maximum window size (MWS)

$$MWS = \begin{cases} \left(\left|\frac{24}{b}\right| + 1\right)|5a - 2b| & \text{if } a - b \geq 25a - 10b \quad \text{i.e., } 9b \geq 24a \\ \left(\left|\frac{9}{a}\right| + 1\right)|5a - 2b| & \text{if } a - b \leq 25a - 10b \quad \text{i.e., } 9b \leq 24a \end{cases}$$

is minimized. Given the set of inequalities that should be satisfied,

$$3a - 2b \geq 0 \Longrightarrow b \leq \frac{3a}{2} \Longrightarrow 9b \leq \frac{27a}{2}.$$

Since, $9b \leq \frac{27a}{2}$, the second condition applies, *i.e.,* $9b \leq 24a$. Therefore,

$$MWS = \left(\frac{9}{a} + 1\right)(5a - 2b) = 45 + (5a - 2b) - \frac{18b}{a}$$

needs to be minimized subject to inequalities (2.5–2.10). We use either a branch and bound technique (or general nonlinear programming techniques) to minimize this function; the number of variables is linear in the number of nested loops which is usually very small in practice (≤ 4) resulting in small solution times. Alternately, if we minimize $5a - 2b$ subject to constraints (2.5–2.10), we get very good solutions in practice. In the example loop nest, $a = 2, b = 3$ is an optimal solution, giving an minimum MWS estimate of 22 which is very close to the actual minimum MWS which is 21. In general, the system of inequalities arising legal tiling requirement are combined with either $a - b \leq aN_1 - bN_2$ or with $a - b \geq aN_1 - bN_2$ to form two groups of inequalities; if both the groups have valid solutions, we find the best of these. If only one group has valid solutions, the problem is a lot easier. For the solution $a = 2, b = 3$, the set of values for c and d which give rise to unimodular T while satisfying tiling legality condition is $c = 1, d = 1$. This was the transformation used in Section 2.2.

2.4 Maximum Window Size for 3-Nested Loops

The window size in 3-nested loops cannot be just derived using the coefficients of the access functions. They are estimated using the dependences, or, in other words the null space vector of the access matrices. The window is a function of the null space vector and the loop limits. The window size is estimated using the largest lexicographic dependence vector since it spans the maximum region in the iteration space. Consider the following loop:

for $i = 1$ to N_1 do
  for $j = 1$ to N_2 do
    for $k = 1$ to N_3 do
      $\cdots A[f_1(i,j,k), f_2(i,j,k)] \cdots$

```
    endfor
  endfor
endfor
```

Let (d_1, d_2, d_3) be the null space vector of the data reference matrix of A. The Maximum Window Size (MWS) is given by

$$MWS = \begin{cases} d_1(N_2 - |d_2|)(N_3 - |d_3|) + 1 & \text{if } d_2 \leq 0 \\ d_1(N_2 - |d_2|)(N_3 - |d_3|) + |d_2|(N_3 - |d_3|) + 1 & \text{if } d_2 > 0 \end{cases}$$

Note from the above result the maximum window size can be reduced if $d_2 = 0$ and further reduced if d_1 is made zero. In other words, we get the best locality if we can find transformation such that the dependences are carried by the inner levels.

Example 5:

for $i = 1$ to 10 do
  for $j = 1$ to 20 do
    for $k = 1$ to 30 do
      $\cdots A[3i + k, j + k] \cdots$
    endfor
  endfor
endfor

The data reference matrix is $\begin{pmatrix} 3 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$. The null space vector of the data reference matrix is $(1, 3, -3)$; the maximum window size is given by:

$$MWS = 1(30 - 3)(20 - 3) + 3(30 - 3) = 540.$$

As we can see from the above example, the maximum window size can be reduced if the dependences are carried by inner levels. So, legal transformations which make the inner loop carry dependences can be applied to reduce the window size and thus increase locality. In other words, if the data reference matrix is used as part of the transformation matrix, only the inner most level can be made to carry the dependence. Thus, with a transforamtion T, with the first two rows the same as the data reference matrix as shwon below,

$$T = \begin{pmatrix} 3 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix},$$

the maximum window size reduces to one.

2.5 Summary

This chapter presented a technique for choosing transformations that enhance locality while enabling parallelization in the case of uniformly generated array references. We used the reference window size associated with a given loop structure as a measure of reuse; transformations were derived in order to minimize the reference window size as a means of exploiting reuse to achieve locality. We showed a method of estimating the window size as a function of an arbitrary unimodular transformation applied to the loop nest. We showed two examples to illustrate how our technique works. Unlike the works of Eisenbeis *et al.* [6] and Li & Pingali [14] which focus only on input and output dependences, our technique exploits the entire dependence structure to derive locality optimizing transformations.

CHAPTER 3

INTEGRATING DATA DISTRIBUTION AND LOOP TRANSFORMATIONS

On a distributed memory machine, the access to local memory by a processor is much faster than accesses to non-local data. The access to non-local data results in communication. Inter-processor communication is a major factor in the performance of a parallel machine. When a number of non-local accesses are to be made between processors, it is preferable that one uses block transfers (also referred to as *message vectorization*, that is, send fewer but larger messages rather than smaller messages more frequently; this is because the message setup cost is usually large.

Given a program segment, our aim is to determine the computation and data mapping on the processors. Parallelism can be exploited by transforming the loop nest suitably and then distributing the iterations of the transformed outermost loop onto the processors. The distribution of data onto processors may then result in communication and synchronization which counters the advantages obtained by parallelism. This chapter presents an algorithm which results in the optimal performance while simultaneously considering the conflicting goals of parallelism and data locality.

While a programmer can manually write code to enhance data locality by specifying data distribution among processors, we present a technique where we can automatically derive data distribution given the program structure. We present a method by which the program is restructured such that when the outer loop iterations are mapped onto the processors, it results in the least communication. Wherever communication is imperative, we restructure the inner loop(s) so that data can be transferred using block transfers; such an approach is referred to as *message vectorization.*

In this chapter, we consider the cases where we allocate outer iterations to processors so that each outer iteration is done by a single processor. The data is then allocated so that there is minimum communication and all communication is done through block transfers. This chapter deals with an algorithm to restructure the program to enhance data locality while still enabling parallelism. We construct the entries in an invertible transformation matrix so that there is a one-to-one mapping from the original iteration space to the transformed iteration space. This transformation when applied to the original loop structure will do the following:

- Allows the outermost most loop iteration to be distributed over the processors *i.e.,* an entire outermost iteration is mapped on to a single processor.
- Determines the data distribution onto the processors. We consider blocked or wrapped distribution of an entire dimension of the array onto a processor.

- Allow blocks transfers outside the innermost loop so that all the necessary data are transferred to the respective local memories before the execution of the innermost loop.

The transformations are derived such that they are legal and unimodular.

3.1 FRAMEWORK

We construct a transformation matrix which is invertible so that there is a one-to-one mapping from the old iteration space to the transformed loop nest. The transformation matrix is derived from the *Data Reference Matrix* of the array references.

3.1.1 DATA REFERENCE MATRIX

Given a loop nest with indices $i_1, i_2, \cdots, i_n$ which is represented by a column vector $\vec{I}$, we define a *Data Reference Matrix*, A_D, for each array reference A (distinct or non-distinct) in a loop nest such that the array reference can be written in the form $A_D\vec{I} + \vec{b}$ where $\vec{b}$ is the offset vector.

Example 6:

for $i = 1$ to N_1 do
  for $j = 1$ to N_2 do
    for $k = 1$ to N_3 do
      $B[i+3, j-i] = B[i+3, j-i] + A[i+1, j+k]$
    endfor
  endfor
endfor

In the above example, we have the following data reference matrix for the array B (B_D):

$$B_D = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{pmatrix}$$

and the data reference matrix for array A (A_D) is:

$$A_D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

We note that there are two data reference matrices for the array B though they are identical and one for array A. In general, there are as many data reference matrices as there are arrays. For each array, we choose only the distinct data reference matrices.

3.1.2 EFFECT OF TRANSFORMATION

On applying a transformation T to a loop with index I, the transformed loop index becomes $\vec{I'} = T\vec{I}$ and the transformed data reference matrix becomes $A'_D = A_D T^{-1}$. The columns of T^{-1} determine the array subscripts of the references in the transformed loop. The entries in the inverse of the transformation matrix are derived starting from the data reference matrices.

3.2 MOTIVATION AND EXAMPLES

Consider Example 6 above which is reproduced below and is similar to the one in [14]:

Example 6:
for $i = 1$ to N_1 do
  for $j = 1$ to N_2 do
    for $k = 1$ to N_3 do
      $B[i+3, j-i] = B[i+3, j-i] + A[i+1, j+k]$
    endfor
  endfor
endfor

There are two references to the array B (though not distinct) and one reference to the array A. Unlike in [14] the loop can be distributed in such a way that there is no communication incurred. Both the arrays can be distributed by rows, *i.e.,* each processor can be assigned an entire row of array A and an entire row of array B. This makes the loop run without any communication. We notice that the first row in the data reference matrix for the arrays A and B are the same *i.e.,* $\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$. This allows the first dimension of both the arrays to be distributed (*i.e.,* by rows) over the processors so that there is no communication. In the next section, we derive an algorithm to construct a transformation matrix, which determines the distribution of data.

3.3 ALGORITHM

3.3.1 ASSUMPTIONS

We restrict our analysis to affine array references in loop nests whose upper and lower bounds are affine. We assume that the iterations of the outermost loop are distributed among processors. To exploit data locality and reduce communication among processors, we further look at transformations that facilitate block transfers so that the data elements which are referenced are brought to local memory in large chunks; this allows to amortize the high message start-up costs over large messages.
We assume that the data can be distributed along any one dimension of the array (wrapped or blocked). The results can be easily generalized where data is distributed along multiple dimensions and block transfers set up in outer iterations.

3.3.2 CRITERIA FOR CHOOSING THE ENTRIES IN THE TRANSFORMATION MATRIX

Let the array indices of the original loop be $i_1, i_2, \ldots, i_n$. Let the array indices of the transformed loop be $j_1, j_2, \ldots, j_n$. We look for transformations such that the LHS array has the outermost loop index as the only element in any one of the dimensions of the array, *e.g.* $C(*, *, \ldots, j_1, \ldots, *)$ where j_1 is in the d^{th} dimension. The LHS array can then be distributed along dimension d. This means that the data reference matrix of the transformed array reference C, *i.e.,* $C_D^{'}$ has at least one row which has the first entry as non-zero and the rest as zero. For all arrays that appear on the right hand side (RHS arrays):

- If a row in all the data reference matrices of an array is identical to a row in the reference matrix in the LHS array, then that array can be distributed in the same way as the LHS array. There is no communication due to that array, since they are always mapped onto the same processor. **Note:** If all the references of all the arrays have a row in the data reference matrix identical to that of the LHS array, then the entire loop can be distributed along that dimension and there is no communication.

- If the above condition does not hold, choose the entries in T^{-1} such that one dimension of the RHS has only the innermost loop index, *e.g.* $A(*, \ldots, j_n, \ldots, *)$ and all the other dimensions are independent of the innermost loop index (that is, "*" indicates a term independent of j_n). This allows a block transfer to the local memory before the execution of the innermost loop. This means that a row in the transformed data reference matrix $A_D^{'}$ has a row with all entries zero except in the last column, which is non-zero. Also, the last column of the $A_D^{'}$ has all remaining entries as zero.

The transformation should also satisfy the condition that the determinant is non-zero and must preserve the dependences in the program.

3.3.3 THE ALGORITHM

Consider the following loop where n is the loop nesting level and d the dimension of the arrays.

for $l_1 = 1$ to N_1 do
  for $l_2 = 1$ to N_2 do
    ...
    ...
      for $l_n = 1$ to N_n do
        $L(f_1(l_1, \ldots, l_n)) = R(f_2(l_1, \ldots, l_n))$
      endfor
  endfor
endfor

where f_1 and f_2 are mappings from $Z^n \longrightarrow Z^d$. Let the data reference matrix of the LHS array be:

$$\begin{pmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ c_{21} & c_{22} & \ldots & c_{2n} \\ \ddots & & & \\ c_{d1} & c_{d2} & \ldots & c_{dn} \end{pmatrix}$$

and that of RHS array be:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \ddots & & & \\ a_{d1} & a_{d2} & \dots & a_{dn} \end{pmatrix}$$

Let the inverse of the transformation matrix be:

$$T^{-1} = \begin{pmatrix} t_{11} & t_{12} & \dots & t_{1n} \\ t_{21} & t_{22} & \dots & t_{2n} \\ \ddots & & & \\ t_{n1} & t_{n2} & \dots & t_{nn} \end{pmatrix}$$

Let $\vec{c_i}$ be the i^{th} row of the reference matrix of the LHS array and $\vec{a_i^r}$ be the i^{th} row of the reference matrix of the r^{th} *distinct* RHS array. Let $\vec{t_j}$ be the j^{th} column of T^{-1}. The algorithm is shown below.

Step 0: If a row in the reference matrix of all the arrays are the same, then there is no communication involved. The data can be distributed along the respective dimension and all the data for computation will be in local memory. (Initialize $i \longleftarrow 1$).

Step 1: Distribute LHS array along dimension i, *i.e.,* make:

$$\vec{c_i}.[T^{-1}] = \begin{pmatrix} 1 & 0 & \cdots & 0 \end{pmatrix}$$

Step 2: Choose an RHS array which does not have a row in the reference matrix the same as that of LHS array.
For all j make:

$$\vec{a_j^r}.[T^{-1}] = \begin{pmatrix} 0 & 0 & \cdots & 0 & 1 \end{pmatrix}$$

for all references of that array and

$$\overline{a_{k \neq j}}.\vec{t_n} = 0$$

If a valid T^{-1} is found, check the determinant of T^{-1}. If non-zero block transfers are possible for that RHS array. [Break – Goto step 3]

If there are no valid T^{-1} or the determinant of T^{-1} is zero for all j, block transfers are not possible on that array with the given distribution of the LHS array. [Increment j – goto Step 2].

Step 3: Repeat step 2 for all the reference matrices of a particular array.

Step 4: Repeat step 2 for all distinct arrays on RHS. (Increment r)

Step 5: Check the number of arrays where block transfers are possible.

Step 6: Repeat Step 1 to Step 4 for LHS array distributed along all dimensions (Increment i).

Step 7: Compare the number of arrays that can have block transfers and distribute LHS array along the dimension which yields maximum number of block transfers for the arrays on the right hand side.

3.4 EXAMPLES

Example 7:

```
for i = 1 to N1 do
  for j = 1 to N2 do
    for k = 1 to N3 do
      B[i, j - i] = B[i, j - i] + A[i, j + k]
    endfor
  endfor
endfor
```

In this loop the reference matrix of the arrays are:

Reference matrix of B, B_D = $\begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{pmatrix}$

Reference matrix of A, A_D = $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$

From Step 0, we see that one row of each of the reference matrices are the same $\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$ and therefore this loop can be distributed that there is no communication. We distribute both the arrays A and B by rows and the entire loop can execute without any communication.

Example 8:

```
for i = 1 to N1 do
  for j = 1 to N2 do
    for k = 1 to N3 do
      C[i, j - i] = C[i, j - i] + A[j - i, j + k] * B[k, j - i]
    endfor
  endfor
endfor
```

In this loop the reference matrix of the arrays are:

Reference matrix of C, C_D = $\begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{pmatrix}$

Reference matrix of A, A_D = $\begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$

Reference matrix of B, B_D = $\begin{pmatrix} 0 & 0 & 1 \\ -1 & 1 & 0 \end{pmatrix}$

From Step 0, we see that one row of each of the reference matrices are the same $\begin{pmatrix} -1 & 1 & 0 \end{pmatrix}$ and therefore this loop can be distributed that there is no communication. We choose a transformation T such that $\begin{pmatrix} -1 & 1 & 0 \end{pmatrix}[T^{-1}] = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$. We distribute the array C and B by columns and A by rows and the entire loop can execute without any communication. A valid T^{-1} is given below:

$$T^{-1} = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and

$$T = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Example 9: Matrix Multiplication

for $i = 1$ to N do
  for $j = 1$ to N do
    for $k = 1$ to N do
      $C(i,j) = C(i,j) + A(i,k) * B(k,j)$
    endfor
  endfor
endfor

The reference matrices of the arrays are as follows:

Reference matrix of C, C_D = $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

Reference matrix of A, A_D = $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Reference matrix of B, B_D = $\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$

Dependence = $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$

Step 1: (C distributed along first dimension)

We have,

$\vec{c_1}.\vec{t_1} = 1$
$\vec{c_1}.\vec{t_2} = 0$
$\vec{c_1}.\vec{t_3} = 0$

Therefore we have,

$$t_{11} = 1; \quad t_{12} = 0; \quad t_{13} = 0$$

Step 1a: [Derive distribution of Array A] Since A has row 1 same as that of C distribute A same as C

Step 2.1: [Derive distribution for Array B]
$\vec{b_1}.\vec{t_1} = 0$
$\vec{b_1}.\vec{t_2} = 0$
$\vec{b_1}.\vec{t_3} = 1$
Therefore we have,

$$t_{31} = 0; \quad t_{32} = 0; \quad t_{33} = 1$$

and

$$\vec{b_2}.\vec{t_3} = 0 \Longrightarrow t_{23} = 0$$

and finally we have

$$T^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ t_{21} & t_{22} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

For unimodular transformation $t_{22} = \pm 1$
Note: Dependence is $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$ and there are no constraints on t_{21} . This results in the identity matrix – No transformation necessary.

Distribute A,C by rows and B by columns. The transformed loop looks as follows:

```
for u = 1 to N do
  for v = 1 to N do
    readB(*, v)
    for w = 1 to N
      C(u, v) = C(u, v) + A(u, w) * B(w, v)
    endfor
  endfor
endfor
```

The above gives the best we can get in terms of parallelism and locality. We go ahead and complete the algorithm by looking at distributing the LHS array in the next dimension.

Step 1.1: (C distributed along second dimension)
$\vec{c_2}.\vec{t_1} = 1$
$\vec{c_2}.\vec{t_2} = 0$
$\vec{c_2}.\vec{t_3} = 0$
Therefore we have,

$$t_{21} = 1; \quad t_{22} = 0; \quad t_{23} = 0$$

Step 1.1a: (Derive distribution for Array B)
Since B has row 2 same as that of C distribute B same as C.

Step 2.2: (Derive distribution for Array A)

$$\vec{a_1}.\vec{t_1} = 0$$

$$\vec{a_1}.\vec{t_2} = 0$$

$$\vec{a_1}.\vec{t_3} = 1$$

Therefore we have,

$$t_{11} = 0; \quad t_{12} = 0; \quad t_{13} = 1$$

and $\vec{a_2}.\vec{t_3} = 0 \Longrightarrow t_{33} = 0$
and finally we have

$$T^{-1} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ t_{31} & t_{32} & 0 \end{pmatrix}$$

For a unimodular transformation, $t_{32} = \pm 1$. Therefore,

$$T^{-1} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

and

$$T = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Distribute A,B,C by columns. The transformed loop is given below:

for $u = 1$ to N do
  for $v = 1$ to N do
    $readA(*, v)$
    for $w = 1$ to N do
      $C(w, u) = C(w, u) + A(w, v) * B(v, u)$
    endfor
  endfor
endfor

We see that the performance of the loop is similar in both the cases. Therefore the array C can either be distributed by columns with the above transformation or by rows with no transformation for the same performance with respect to communication.
Consider the SYR2K example with the original loop as follows:

Example 10: SYR2K

for $i = 1$ to N do
  for $j = i$ to $min(i + 2b - 2, N)$ do
    for $k = max(i - b + 1, j - b + 1, 1)$ to $min(i + b - 1, j + b - 1, N)$ do
      $C[i, j - i + 1] = C[i, j - i + 1] + A[k, i - k + b] * B[k, j - k + b]$
        $+A[k, j - k + b] * B[k, i - k + b]$
    endfor
  endfor
endfor

The reference matrices for the arrays are given below:

Reference matrix of C, C_D = $\begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{pmatrix}$

Reference matrix of A, A_D = $\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \end{pmatrix}$ and $\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix}$

The reference matrix for B is the same as A.

Step 1: C row distributed: Make:

$\vec{c_1}.\vec{t_1} = 1$
$\vec{c_1}.\vec{t_2} = 0$
$\vec{c_1}.\vec{t_3} = 0$

Therefore we have,

$$t_{11} = 1; \quad t_{12} = 0; \quad t_{13} = 0$$

Step 1.1: Make:

$\vec{a_1}.\vec{t_1} = 0$
$\vec{a_1}.\vec{t_2} = 0$
$\vec{a_1}.\vec{t_3} = 1$

and we have $\vec{a_2}.\vec{t_3} = 0$

Therefore,

$$t_{31} = 0; \quad t_{32} = 0; \quad t_{33} = 1$$

and

$$\vec{a_2}.\vec{t_3} = 0 \Longrightarrow t_{13} - t_{33} = 0$$

which is not true.

Step 1.2: Make

$\vec{a_2}.\vec{t_1} = 0$
$\vec{a_2}.\vec{t_2} = 0$
$\vec{a_2}.\vec{t_3} = 1$ and $\vec{a_1}.\vec{t_3} = 0$

$$t_{11} - t_{31} = 0 \Longrightarrow t_{13} = 1$$

$$t_{12} - t_{32} = 0 \Longrightarrow t_{32} = 0$$

$$t_{13} - t_{33} = 1 \Longrightarrow t_{33} = -1$$

and $t_{33} = 1$ which is not true.

Since the reference matrix for array A and B are the same, there can be no block transfers for B as well.

Step 2.0: C column distributed: Make: $\vec{c_2}.\vec{t_1} = 1$
$\vec{c_2}.\vec{t_2} = 0$
$\vec{c_2}.\vec{t_3} = 0$

$$-t_{11} + t_{21} = 1 \Longrightarrow t_{11} = t_{21} - 1$$

$$-t_{12} + t_{22} = 0 \Longrightarrow t_{12} = t_{22}$$

$$-t_{13} + t_{23} = 0 \Longrightarrow t_{13} = t_{23}i$$

Step 2.1a: (First reference of A): Make: $\vec{a_1}.\vec{t_1} = 0$
$\vec{a_1}.\vec{t_2} = 0$
$\vec{a_1}.\vec{t_3} = 1$

$$t_{31} = 0; \quad t_{32} = 0; \quad t_{33} = 1$$

and $\vec{a_2}.\vec{t_3} = 0 \Longrightarrow t_{13} - t_{33} = 0 \Longrightarrow t_{13} = 1 \Longrightarrow t_{23} = 1$

Step 2.1b: (Second reference of A): Make: $\vec{a_1}.\vec{t_1} = 0$
$\vec{a_1}.\vec{t_2} = 0$
$\vec{a_1}.\vec{t_3} = 1$

$$t_{31} = 0; \quad t_{32} = 0; \quad t_{33} = 1$$

and $a_2.t_3 = 0 \Longrightarrow t_{23} - t_{33} = 0 \Longrightarrow t_{23} = 1$

Hence we have the inverse of the transformation matrix as

$$T^{-1} = \begin{pmatrix} t_{11} & t_{12} & 1 \\ t_{21} & t_{22} & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Since the reference matrices for B is the same as that of A we distribute B as A is distributed. We choose the unknown values such that T is a legal unimodular transformation.

Possible $T^{-1} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$

and $T = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix}$

The transformed reference matrices are as follows:

Reference matrix of C, C'_D = $\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}$

Reference matrix of A, A'_D = $\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ and $\begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$

Using the above algorithm we distribute A,B,C by columns and the transformed code with block transfers is as follows:

```
for u = 1 to N do
  for v = u to min(u + 2b - 2, N) do
    read A(*,v+b); read A(*,u+v+b);
    read B(*,u+v+b); read B(*,v+b);
    for w = max(u - b + 1, v - b + 1, 1) to min(u + b - 1, v + b - 1, N) do
      C[v + w + 1, u] = C[v + w + 1, u] + A[w, v + b] * B[w, u + v + b]
              +A[w, u + v + b] * B[w, v + b]
    endfor
  endfor
endfor
```

3.5 SUMMARY

This chapter illustrated an algorithm which derives the terms in the transformation matrix which gives the best locality and minimum communication on distributed memory machines. We used the concept of *Data Reference Matrices* for individual array references. Using this as the starting point, we systematically derived the best set of transformation matrices which give both good locality while enabling parallelism. Unlike [14], where a *padding matrix* is used along with an arbitrary set of rows in the *basis matrix,* we generate a transformation matrix systematically. The algorithm also gives an optimal distribution of arrays on to the processors such that block transfers are enabled to reduce inter-processor communication. The input to the algorithm is the reference matrices of all the references to all the arrays. Here distribution of data only along one dimension is considered. However complex distributions, where the allocation of arrays with allocation functions having more than one index, is a simple extension of the above algorithm. Similarly the distribution function of the iterations over the processors can be more complex and loop nests can be tiled.

CHAPTER 4

ESTIMATING DATA ACCESS IN NESTED LOOPS

The set of elements referenced during the execution of nested loops is of interest. In hierarchical shared memory machines, when data transfers from global memory to local memory has to be optimized, the region associated with the references in a nested loop is important. We present a technique that estimates the number of distinct array accesses in loop nests. In distributed memory machines, the elements not owned by the node can be estimated and a technique to transfer those elements can be chosen.

In most practical cases, the number of distinct accesses is much smaller than the size of the iteration space (or, the total number of accesses). This is due to the repeated accesses to the same memory location in the course of the execution of the loop. This would result in a large overhead on the machine to transfer all the data from global to local memory. If we can find the elements which are accessed before hand, a copy of the elements to local memory can be made more efficiently. If a data element is accessed more than once in a program then we have a dependence from the first statement that referenced it to the second statement which references it. We use the term *reuse* for all subsequent accesses of a data element after its first access. We use the concept of reuse to calculate the number of distinct accesses (A_d) to an array. Since reuse is closely related to dependences, we can calculate the number of distinct accesses using dependences.

4.1 FRAMEWORK

4.1.1 DATA REFERENCE MATRIX

Given a loop nest with indices $i_1, i_2, \cdots, i_n$ which is represented by a column vector $\vec{I}$, we define a *data reference matrix*, A_D, for each array access A in a loop nest such that the array reference can be written in the form $A_D\vec{I} + \vec{b}$ where $\vec{b}$ is the offset vector. The data reference matrix is a $d \times n$ matrix. An array element which is referenced (read or written) more than once in a loop nest constitutes a *reuse.* The reuse count depends on the number of references to the array in the loop (r), the relative values of the loop nest levels and the dimensionality (d) of the array, and also the loop limits.

For an array where its reference matrix is a non-singular matrix (that is, if the dimension d of the array and the loop nesting n is the same) the number of times an element is referenced is at most r where r is the number of references to the array in a loop. Therefore the *reuse* for a data element is at most $r - 1$. For an array whose dimension is one less than the loop nest, the reuse of an element is along the direction of the null space of the access matrix and the amount of reuse due to an element depends on the loop bounds.

The maximum number of times an element is accessed in such cases is min($\lceil N_i/d_i \rceil$). It can be noted that in cases where the dimensionality is less than the loop nest level, for a

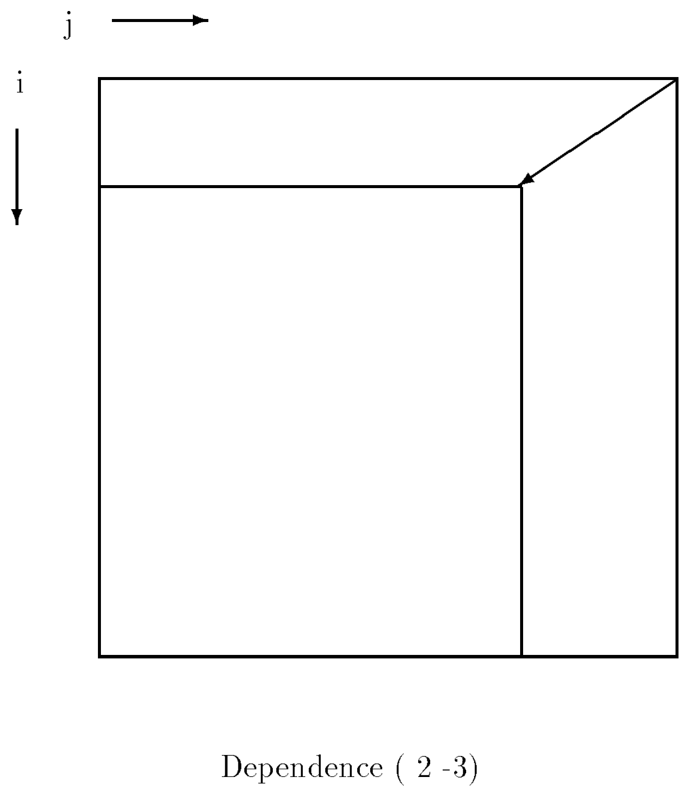


Figure 4.1: Iteration space for a 2-nested loop.

given reference of the array, there is reuse along the null space vector. Therefore, for r references there are at most $r \times \min(\lceil N_i/d_i \rceil)$.
The number of distinct references (A_d) can be found using dependences in the loop as shown in Figure 4.1. The n-dimensional cube (in the case of 2-nested loop - a square) formed by the dependence vectors as shown in Figure 4.1 represents the reused area in the iteration space.
Consider the following examples:

Example 10(a):
for $i = 1$ to 10 do
  for $j = 1$ to 10 do
    $\cdots A[i, j] \cdots$
    $\cdots A[i - 3, j + 2] \cdots$
  endfor
endfor

Example 10(b):
for $i = 1$ to 10 do
  for $j = 1$ to 10 do
    $\cdots A[2 * i + 3 * j] \cdots$

  endfor
endfor

In both Example 10(a) and 10(b), the dependence vector is $(3, -2)$. Note that in the first example, the dimensionality of the array is the same as that of the loop nest level, the number of references (r) is 2 and the reuse count is at most 1. (The number of times an element of the array is referenced is at most 2).
In the second example, the dimensionality is less than the loop nest level, the number of references (r) is 1, and the maximum reuse count for an element is $\lceil 10/3 \rceil$ = 4. The total reuse is the same in both the examples which is $(10-3) \times (10-2) = 56$. We consider the cases where the dimension of the array accessed within the loop is the same as the nest level and where the dimensionality is less than the loop nesting level.

4.2 LOOPS WITH ARRAY DIMENSION d = NESTING n

4.2.1 SINGLE REFERENCE

We will first consider a loop where the dimension of the array accessed d is the same as the loop nest n. Consider the loop:

for $i = 1$ to N_1
  for $j = 1$ to N_2
    $\ldots A[f_1(i,j), f_2(i,j)] \ldots$
  endfor
endfor

Note that f_1 is in general different from f_2. The above loop has only one access to the array A in the loop. and it does not have any reuse. During each iteration a distinct element of the array A is accessed. Therefore the reuse for such a loop is zero. Also notice that there are no dependences in the loop. Therefore the number of distinct elements of the array accessed is $N_1 \times N_2$.

Example 11:
for $i = 1$ to 10
  for $j = 1$ to 10
    $\ldots A[i,j] \ldots$
  endfor
endfor

In the above loop we have a distinct element accessed in each iteration and the number of distinct accesses equals the iteration space, that is, A_d = $N_1 \times N_2$ = 100. Also note that there is no dependence in the loop and hence, *reuse* is zero.

4.2.2 MULTIPLE REFERENCES

In general for r references in a loop where the array dimension is the same as the loop nesting level there are a total of $\frac{r(r-1)}{2}$ dependences. Note that there is at least one node in the dependence graph which is a sink to the dependence vectors from each of the remaining $r-1$ nodes. In other words there exists a statement with $r-1$ direction vectors directed from each of the remaining statements. The $r-1$ dependences due to all the other references to this reference gives the amount of reuse.
Consider the following code fragment:

for $i = 1$ to N_1
  for $j = 1$ to N_2
    $\ldots A[f_1(i,j)+c_1, f_2(i,j)+k_1]\ldots$
    $\ldots A[f_1(i,j)+c_2, f_2(i,j)+k_2]\ldots$
    $\ldots$
    $\ldots$
    $\ldots A[f_1(i,j)+c_r, f_2(i,j)+k_r]\ldots$
  endfor
endfor

Let the dependences on one reference due to all other references be

$$\begin{pmatrix} d_{11} & d_{21} & \cdots & d_{r-11} \\ d_{12} & d_{22} & \cdots & d_{r-12} \end{pmatrix}.$$

The amount of reuse for that array is:

$$\text{reuse} = \sum_{i=1}^{r-1}(N_1 - d_{i1})(N_2 - d_{i2})$$

and the number of distinct elements is given by

$$A_d = N_1 \times N_2 \times r - \text{reuse}.$$

Consider the following loop (in Example 12) where there are two uniformly generated references to the array A and the access matrix is non-singular.
Example 12:
for $i = 1$ to N_1
    for $j = 1$ to N_2
S_1: $\cdots A(i,j)\cdots$
S_2: $\cdots A(i-1, j+2)\cdots$
    endfor
endfor

Here there is a flow dependence $(1, -2)$ from statement S_1 to statement S_2. This dependence is used to calculate the amount of reuse for each element. The amount of reuse is:

$$\text{reuse} = (N_1 - 1)(N_2 - 2)$$

and the number of distinct accesses to the array A in the above loop is

$$A_d = N_1 \times N_2 \times 2 - \text{reuse}.$$

Example 13:

for $i = 1$ to 10
    for $j = 1$ to 10
S_1: $\cdots A(i, j) \cdots$
S_2: $\cdots A(i-1, j) \cdots$
S_3: $\cdots A(i, j-1) \cdots$
S_4: $\cdots A(i-1, j-1) \cdots$
    endfor
endfor

The dependences from statement S_1 to all other statements is $\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}$. The amount of reuse is calculated as follows:

$$\text{reuse} = (10-1)(10-0) + (10-0)(10-1) + (10-1)(10-1) = 90 + 90 + 81 = 261$$

and the the number of distinct accesses is:

$$A_d = 10 \times 10 \times 4 - 261 = 139.$$

Thus, we see that for cases where the loop nesting level is the same as the dimension of the array accesses in the loop there is only one dependence vector between a pair of statements and the maximum reuse for a particular element is at most $r - 1$. In other words, there are a maximum of r references to an array element.

4.3 Loops with Array Dimension $d = n - 1$

4.3.1 Single Reference

Now consider the case where the dimension of the array is at least one less than the loop nest. If $d = n - 1$ then there is reuse along the direction of the null space vector of the access matrix.

Example 14

for $i = 1$ to 20 do
  for $j = 1$ to 10 do
    $\cdots A(2i + 5j + 1) \cdots$
  endfor
endfor

Here the reuse direction is $(5, -2)$ which is the same as the dependence vector for the loop. We now look at the n dimensional cube formed by the dependence vector (in this case, a square) on the iteration space which represents the reused elements of the array. Note that all elements within the square formed by the vector is a sink to a direction vector which is a reused element by definition. Therefore, for the above example where there is a single statement, we can obtain the figure for the number of data elements reused in the array as:

$$\text{reuse} = (N_1 - d_{11})(N_2 - |d_{21}|)$$

or,

$$\text{reuse} = (20 - 5)(10 - 2) = 120$$

and the number of distinct accesses to the array is

$$A_d = N_1 \times N_2 - \text{reuse}$$

or,

$$A_d = 20 \times 10 - 120 = 80$$

4.3.2 Banerjee's Inequalities

We now show the close relation between the distinct access in array and the range of elements obtained by using Banerjee's inequalities to figure out the upper and lower bound on an array. For a 1-d array in a 2 nested loop the distinct elements can be calculated in terms of Banerjee's inequalities. A correction factor is given to the inequalities to get the exact count of the number of elements accessed.

In the above example, the lower bound and upper bound on the array is calculated using Banerjee's inequalities.

Lower Bound (LB) = $2 \times 1 + 5 \times 1 + 1 = 8$

Upper Bound (UB) = $2 \times 20 + 5 \times 10 + 1 = 91$

The range of elements accessed can be obtained using Banerjee's inequalities:

$$UB - LB + 1 = 91 - 8 + 1 = 84$$

which is offset by the exact value by 4. We derive a correction factor to the range obtained by Banerjee's inequalities to get the exact estimate of the number of elements accessed.

Let the coefficients array index be λ_1 and λ_2. We can now represent the dependence in terms of the array indices and the actual figure for reuse is:

$$N_1N_2 - (N_2 - \lambda_1)(N_1 - \lambda_2) = \lambda_1 N_1 + \lambda_2 N_2 - \lambda_1\lambda_2$$

We have the range obtained using Banerjee's inequalities

$$(\lambda_1 N_1 + \lambda_2 N_2) - (\lambda_1 1 + \lambda_2 1) + 1$$

Therefore, the figure obtained for the number of distinct elements using Banerjee's inequalities exceeds the actual value by $(\lambda_1 - 1)(\lambda_2 - 1)$
Note that if one of the coefficients (λ_1 or λ_2) is 1 which is the case in most scientific programs, the correction factor is *zero* and the range using Banerjee's inequalities gives the exact value of the number of accesses to the array. Now consider the case of a 2-dimensional array accessed in a three nested loop.

Example 15:
for $i = 1$ to 10 do
  for $j = 1$ to 20 do
    for $k = 1$ to 30 do
      $\cdots A(3i + k, j + k) \cdots$
    endfor
  endfor
endfor

Here the reuse direction is $\begin{pmatrix} 1 & 3 & -3 \end{pmatrix}$ and the reuse is calculated as:

$$\text{reuse} = (10 - 1)(20 - 3)(30 - 3) = 4131$$

and the distinct accesses,

$$A_d = 10 \times 20 \times 30 - 4131 = 1869$$

In each of the above cases the reused elements are in a n-dimensional cube formed by the dependence (distance) vector.

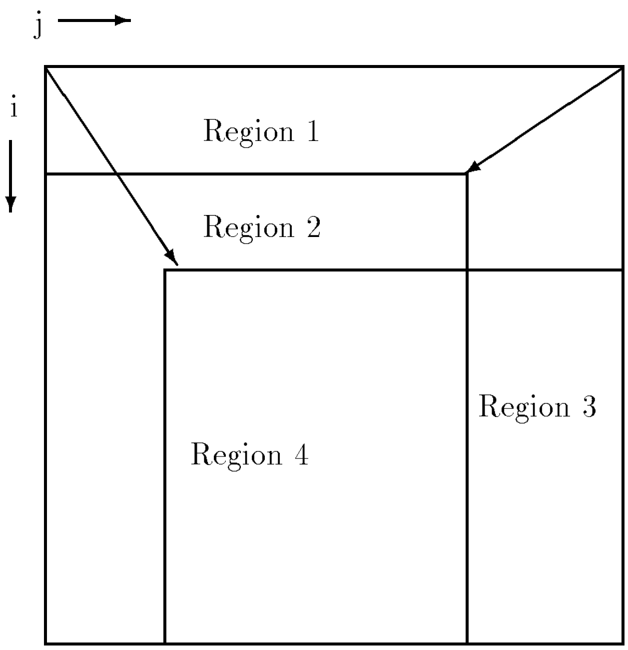
4.3.3 Multiple References

For multiple references of uniformly generated accesses, the reused elements are calculated using the dependence vectors. Consider an example where there are two references (in say two statements) and two dependence vectors as shown in Figure 4.2. There are four regions of interest (in the case where there are 2 dependences). In region 1, there are two distinct elements in every iteration point. There is one distinct element (and one reused element) in regions 2 and 3. Finally, all the elements in the iteration points of region 4 are reused.
Now, the total number of elements which are distinct can be calculated by taking the union of all the regions according to whether they are reused or not. *Algorithm 1* calculates the number of distinct elements for multiple uniformly generated references for 1-d array in a 2-nested loop. *Algorithm 2* calculates the number of distinct elements for multiple uniformly generated references for 2-d array in a 3-nested loop. *Algorithm 3* calculates the number of distinct elements in triangular loops.

4.4 Non-uniformly Generated References

The distinct elements in loops where there are non-uniformly generated references are more complex to compute. The dependence vectors for these loops are distance vectors and thus it is not possible to represent the exact reuse using the dependence vectors. Consider non uniformly generated references of the form:

for $i = 1$ to N_1 do



Dependence (2 -3) S2 to S1
Dependence (3 1) S1 to S2

Region 1: Distinct elements of both statements S1 and S2
Region 2: Distinct elements of S! and reused elements of S2
Region 3: Reused elements of S1 and distinct elements of S2
Region 4: Reused elements of both statements S1 and S2

Figure 4.2: Iteration space for multiple references

for $j = 1$ to N_2 do
$\cdots A[f_1(i,j)] \cdots$
$\cdots A[f_2(i,j)] \cdots$
endfor
endfor

Here we consider two cases

1. the coefficients are small compared to the loop limits
2. the coefficients are very much larger than the loop limits and there is no self dependence.

For the above cases a lower and upper bound can be obtained for the number of distinct accesses to the array.

- Case 1: Consider the general loop with n references to an array A.

  for $i = 1$ to N_1 do
  for $j = 1$ to N_2 do
  $(f_1) \cdots A(a_{11}i + a_{12}j + c_1) \cdots$
  $(f_2) \cdots A(a_{21}i + a_{22}j + c_2) \cdots$
  $\cdots$
  $\cdots$
  $(f_n) \cdots A(a_{n1}i + a_{n2}j + c_n) \cdots$
  endfor
  endfor

  where $a_{ij} < N_1, N_2$. We apply Banerjee's inequalities to obtain the upper and lower bound for the elements accessed in the array for each statements. We have the following:

  $$LB_1 \le f_1 \le UB_1$$
  $$LB_2 \le f_2 \le UB_2$$
  $$\cdots$$
  $$\cdots$$
  $$LB_n \le f_n \le UB_n$$

  Find the maximum of UB_i and the minimum of LB_i, that is, UB_{max} and LB_{min} and the corresponding access functions, say,

  $$a_{h1}i + a_{h2}j + c_h$$

  and

  $$a_{l1}i + a_{l2}j + c_l$$

Clearly, the upper bound for the number of distinct accesses is the maximum range of all the individual upper and lower bounds. In other words the upper bound on the number of references is given by:

$$UB_{max} - LB_{min} + 1$$

The lower bound on the number of references is:

$$(UB_{max} - LB_{min} + 1) - 1/2(a_{h1} - 1)(a_{h2} - 1) - 1/2(a_{l1} - 1)(a_{l2} - 1)$$

Example 16:

for $i = 1$ to 20 do
    for $j = 1$ to 20 do
(f_1) $\cdots A(3i + 7j - 10) \cdots$
(f_2) $\cdots A(4i - 3j + 60) \cdots$
    endfor
endfor

Here the dependences are distance vectors and an exact dependence cannot be obtained. We give lower and an upper bound on the number of distinct accesses on the array A. We have the upper and lower bounds on both the functions f_1 and f_2.

$$LB_1 \leq f_1 \leq UB_1$$

$$LB_2 \leq f_2 \leq UB_2$$

$$LB_1 = 0, LB_2 = 4, UB_1 = 190, UB_2 = 137.$$

Therefore $LB_{min} = 0$, $UB_{max} = 190$.

Upper bound on the number of references = $190 - 0 + 1 = 191$
Lower bound on the number of references = $191 - (3-1)(7-1) - (3-1)(7-1) = 179$.

The actual number of references is 181. So a close bound on the number of distinct accesses to the array can be obtained by the above algorithm.

- Case 2: If the coefficients of the array access is much larger or comparable to the loop limits, the above algorithm will not yield a close bound since the range obtained by Banerjee's inequalities will not have all the elements within that range accessed by the access function. The large number of "holes" in the range would yield a grossly over estimated figure for the number of distinct accesses.

Consider the general loop as below:

for $i = 1$ to N_1
  for $j = 1$ to N_2
$(f_1) \cdots A[x_1 i_1 + y_1 i_2 + c_1] \cdots$
$(f_2) \cdots A[x_2 i_1 + y_2 i_2 + c_2] \cdots$
  endfor
endfor

The following assumptions are made: $x_i, y_i \gg N_1, N_2$ and there are no self dependences. $i_1, i_2 \neq 0$.

We restrict our work to two statements to show the underlying concept of determining the elements accessed for such loops. Work is under way to generalize this n references. The following algorithm gives a upper and lower bound to the number of distinct elements accessed.

We have from Banerjee's inequalities,

$$LB_1 \leq f_1 \leq UB_1 \qquad\qquad LB_2 \leq f_2 \leq UB_2$$

Note that there are four possibilities for the relative values for the lower and upper bounds of both access functions. They are:

1. $lb_2 > lb_1$ and $ub_2 > ub_1$
2. $lb_2 < lb_1$ and $ub_2 < ub_1$
3. $lb_2 < lb_1$ and $ub_2 > ub_1$
4. $lb_2 > lb_1$ and $ub_2 < ub_1$

Also note that since the access functions are linear there can be a dependence (flow, anti or output) from statement S_1 to S_2 only or from statement S_2 to S_1 only, or there could be a dependence from statement S_1 to S_2 until certain distance through the iteration and a reversal of direction (that is, from S_2 to S_1) for the remainder of the loop. This is true even though the actual dependence vector is a direction vector. Therefore we have four more cases depending on whether there is a change of direction of the dependence vector within the iteration of space. We derive the value of mid which is the outer loop index of the point where this reversal of direction of the dependence takes place. We also derive values of k_1, k_2, k_3 and k_4 which is the outer loop index beyond (or before) which the sink of dependence falls outside the iteration space for that statement. Using the above concept we obtain reuse for the 16 cases. Also note that if ($lb_2 > ub_1$ or $lb_1 > ub_2$) there is no overlap of elements within the iteration space or there is no dependence between the two statements and therefore the reuse is *zero*.

Case 1: ($x_1 > 0$ and $x_2 > 0$)

if($lb_2 > lb_1$ and $ub_2 > ub_1$)

$$\text{reuse} = \frac{(N1 - k1) \times N2 \times N2}{x_1}$$

if($lb_2 < lb_1$ and $ub_2 < ub_1$)

$$\text{reuse} = \frac{(N1 - k2) \times N2 \times N2}{x_2}$$

if($lb_2 < lb_1$ and $ub_2 > ub_1$)

$$\text{reuse} = \frac{(mid - k_2) \times N_2 \times N_2}{x_1} + \frac{(N_1 - mid) \times N_2 \times N_2}{x_2}$$

if($lb_2 > lb_1$ and $ub_2 > ub_1$)

$$\text{reuse} = \frac{(N_1 - mid) \times N_2 \times N_2}{x_1} + \frac{(mid - k_1) \times N_2 \times N_2}{x_2}$$

Case 2:($x_1 > 0$ and $x_2 < 0$)

if($lb_2 > lb_1$ and $ub_2 > ub_1$)

$$\text{reuse} = \frac{(N_1 - mid) \times N_2 \times N_2}{x_1} + \frac{(N_1 - mid) \times N_2 \times N_2}{x_2}$$

if($lb_2 < lb_1$ and $ub_2 < ub_1$)

$$\text{reuse} = \frac{(k_2 - mid) \times N_2 \times N_2}{x_1} + \frac{(k_3 - mid) \times N_2 \times N_2}{x_2}$$

if($lb_2 < lb_1$ and $ub_2 > ub_1$)

$$\text{reuse} = \frac{(k_2 - mid) \times N_2 \times N_2}{x_1} + \frac{(N_1 - mid) \times N_2 \times N_2}{x_2}$$

if($lb_2 > lb_1$ and $ub_2 < ub_1$)

$$\text{reuse} = \frac{(N_1 - mid) \times N_2 \times N_2}{x_1} + \frac{(k_3 - mid) \times N_2 \times N_2}{x_2}$$

Case 3:($x_1 < 0$ and $x_2 < 0$)

if($lb_2 > lb_1$ and $ub_2 > ub_1$)

$$\text{reuse} = \frac{(k_4 - mid) \times N_2 \times N_2}{x_1} + \frac{(k_1 - mid) \times N_2 \times N_2}{x_2}$$

if($lb_2 < lb_1$ and $ub_2 < ub_1$)

$$\text{reuse} = \frac{(N_1 - mid) \times N_2 \times N_2}{x_1} + \frac{(N_1 - mid) \times N_2 \times N_2}{x_2}$$

if($lb_2 < lb_1$ and $ub_2 > ub_1$)

$$\text{reuse} = \frac{(k_4 - mid) \times N_2 \times N_2}{x_1} + \frac{(N_1 - mid) \times N_2 \times N_2}{x_2}$$

if($lb_2 > lb_1$ and $ub_2 < ub_1$)

$$\text{reuse} = \frac{(N_1 - mid) \times N_2 \times N_2}{x_1} + \frac{(k_1 - mid) \times N_2 \times N_2}{x_2}$$

Case 4:($x_1 < 0$ and $x_2 < 0$)

if($lb_2 > lb_1$ and $ub_2 > ub_1$)

$$\text{reuse} = \frac{(N_1 - k_4) \times N_2 \times N_2}{x_1}$$

if($lb_2 < lb_1$ and $ub_2 < ub_1$)

$$\text{reuse} = \frac{(N_1 - k_3) \times N_2 \times N_2}{x_2}$$

if($lb_2 < lb_1$ and $ub_2 > ub_1$)

$$\text{reuse} = \frac{(mid - k_4) \times N_2 \times N_2}{x_1} + \frac{(N_1 - mid) \times N_2 \times N_2}{x_2}$$

if($lb_2 > lb_1$ and $ub_2 < ub_1$)

$$\text{reuse} = \frac{(N_1 - mid) \times N_2 \times N_2}{x_1} + \frac{(mid - k_3) \times N_2 \times N_2}{x_2}$$

where:

$$k_1 = \frac{lb_2 - c_1 - y_1 \times N_2}{x_1}$$

$$k_2 = \frac{lb_1 - c_2 - y_2 \times N_2}{x_2}$$

$$k_3 = \frac{ub_2 - c_1 - y_1 \times N_2}{x_1}$$

$$k_4 = \frac{ub_1 - c_2 - y_2 \times N_2}{x_2}$$

and

$$mid = \frac{(c_2 - c_1) - (y_2 - y_1) \times N_2}{x_1 - x_2}$$

Number of distinct elements:
Lower Bound = $N1 \times N2 -$ reuse
Upper Bound = $N1 \times N2 -$ reuse $+ N2 \times N2$

4.5 ALGORITHM 1

Multiple references of 1 dimension array accessed in 2 nested loop:
Consider the following loop:

for $i = l_1$ to u_1 do
  for $j = l_2$ to u_2 do
$S_1 : \cdots A[\lambda_1 i + \lambda_2 j + c_1] \cdots$
$S_2 : \cdots A[\lambda_1 i + \lambda_2 j + c_2] \cdots$
  $\ldots$
  $\ldots$
$S_r : \cdots A[\lambda_1 i + \lambda_2 j + c_r] \cdots$
  endfor
endfor

The above loop has a total of $r(r-1)+1$ dependences. They are of the form:

$$S_1 \xrightarrow{\vec{d_o}} S_1$$

$$S_2 \xrightarrow{\vec{d_{21}}} S_1$$

$$\ldots$$

$$\ldots$$

$$S_r \xrightarrow{\vec{d_{r1}}} S_1$$

— —

$$S_1 \xrightarrow{\vec{d_{12}}} S_2$$

$$S_2 \xrightarrow{\vec{d_o}} S_2$$

$$\ldots$$

$$\ldots$$

$$S_r \xrightarrow{d_{r2}} S_2$$

— —

$$\ldots$$
$$\ldots$$
$$\ldots$$
$$\ldots$$

$$- -$$

$$S_1 \xrightarrow{\vec{d_{1r}}} S_r$$
$$S_2 \xrightarrow{\vec{d_{2r}}} S_r$$
$$\ldots$$
$$\ldots$$
$$S_r \xrightarrow{\vec{d_o}} S_r$$

$$- -$$

where $\vec{d_o}$ is the output dependence

$$d_o = \begin{pmatrix} \lambda_2 \\ -\lambda_1 \end{pmatrix}$$

Also $\vec{d_o} = \vec{d_{ij}} + \vec{d_{ji}}$
and $\vec{d_{ij}}$ is the dependence from S_i to S_j
The total number of accesses to the array A is

$$(u_2 - l_2 + 1)(u_1 - l_1 + 1) \times r$$

Now,

$$\vec{d_{ij}} \equiv \begin{pmatrix} d_{ij1} \\ d_{ij2} \end{pmatrix}$$

This is obtained by solving
Now we calculate the amount of reuse on array A due to S_j. Or, find all the elements of A whose sink of the dependence is on
Reuse on S_j:

Step 1: For all i: Find $min(d_{ij1}) \equiv d_{ij1_{min1}}$ and choose that dependence vector.

$$\vec{d_{ij1_{min}}} \equiv \begin{pmatrix} d_{ij1_{min1}} \\ d_{ij1_{min2}} \end{pmatrix}$$

If $min(d_{ij1})$ is not unique choose the vector with minimum $|d_{ij1_{min2}}|$

Step 2: For all i: Find $min(|d_{ij2}|)$. Let the vector be

$$\vec{d_{ij2_{min}}} \equiv \begin{pmatrix} d_{ij2_{min1}} \\ d_{ij2_{min2}} \end{pmatrix}$$

under the following conditions.

1. $d_{ij2_{min}}$ should be distinct from $d_{ij1_{min}}$.
2. If not unique
   - choose the one where $d_{ij2_{min2}}$ is opposite sign of $d_{ij1_{min2}}$.
   - if no vector exists, choose the one with $min(|d_{ij2_{min1}}|)$

Step 3: We define a term diff(j) as follows:

$$\text{diff}(j) = \begin{cases} 0 & \text{if } d_{ij1_{min2}} = 0 \\ |d_{ij1_{min2}}| & \text{if } d_{ij2_{min2}} = 0 \\ |d_{ij2_{min2}}| & \text{if } d_{ij1_{min2}} \text{ \& } d_{ij2_{min2}} \text{ are of opposite signs} \\ |d_{ij1_{min2}}| - |d_{ij2_{min2}}| & \text{if } |d_{ij2_{min2}}| < |d_{ij1_{min2}}| \text{ \& they are of same sign} \\ 0 & \text{if } |d_{ij2_{min2}}| \geq |d_{ij1_{min2}}| \text{ \& they are of same sign} \end{cases}$$

$$\text{Reuse} = \sum_{j=1}^{r} d_{ij1_{min1}}(N_2 - d_{ij1_{min2}}) + diff(j)(N_1 - d_{ij2_{min1}})$$

The number of distinct elements = $N_1 \times N_2 \times r -$ reuse
Example 17:
for $i = 1$ to 10 do
  for $j = 1$ to 20 do
    $\cdots A[2i + 5j] \cdots$
    $\cdots A[2i + 5j + 1] \cdots$
    $\cdots A[2i + 5j + 5] \cdots$
    $\cdots A[2i + 5j + 7] \cdots$
  endfor
endfor

Dependences:
$S_2 \rightarrow S_1(\; 3 \quad -1 \;)$
$S_3 \rightarrow S_1(\; 0 \quad 1 \;)$
$S_4 \rightarrow S_1(\; 1 \quad 1 \;)$
$S_1 \rightarrow S_2(\; 2 \quad -1 \;)$
$S_3 \rightarrow S_2(\; 2 \quad 0 \;)$
$S_4 \rightarrow S_2(\; 3 \quad 0 \;)$
$S_1 \rightarrow S_3(\; 5 \quad -3 \;)$
$S_2 \rightarrow S_3(\; 3 \quad -2 \;)$

$S_4 \rightarrow S_3(\ 1 \quad 0\ )$
$S_1 \rightarrow S_4(\ 4 \quad -3\ )$
$S_2 \rightarrow S_4(\ 2 \quad -2\ )$
$S_3 \rightarrow S_4(\ 4 \quad -2\ )$
Using the above algorithm we have
Reuse due to S_1 : $10 \times 19 + 7 \times 1 = 197$.
Reuse due to S_2 : $8 \times 20 = 160$.
Reuse due to S_3 : $9 \times 20 = 180$.
Reuse due to S_4 : $8 \times 18 = 144$.
Total reuse = 681
Number of distinct accesses = $10 \times 20 \times 4 - 681 = 119$

4.6 ALGORITHM 2

Multiple references of uniformly generated 2 dimensional array in a 3 nested loop. Consider the loop:

for $i = 1$ to N_1 do
  for $j = 1$ to N_2 do
    for $k = 1$ to N_3 do
S_m : $\cdots A[h(i,j,k) + \vec{c_m}] \cdots$
    endfor
  endfor
endfor

h is a 3×2 matrix and $\vec{c_m}$ is in Z^2. The number of references to the array A is $N_1 \times N_2 \times N_3 \times r$ As in *Algorithm 1* we have $r(r-1)+1$ dependences. The dependences are arranged so that the sink of the dependence to each statement are grouped together. Let all the dependences which have its sink on S_j be:

$$\begin{pmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,r-1} & d_{1o} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,r-1} & d_{2o} \\ d_{3,1} & d_{3,2} & \cdots & d_{3,r-1} & d_{3o} \end{pmatrix}$$

The self dependence is denoted by $\vec{d_o} \equiv \begin{pmatrix} d_{1o} \\ d_{2o} \\ d_{3o} \end{pmatrix}$

We have the total reuse as
$\sum_{j=1}^{r}(N_1 - d_{1j})(N_2 - |d_{2j}|)(N_3 - |d_{3j}|)$
$+ (N_1 - d_{1o})\sum_{j=1}^{r}(N_3 - |d_{3o}|)(\text{reuse}(2,j)) + (N_2 - |d_{2o}|)(\text{reuse}(3,j))$
where

$$\text{reuse}(2,j) = \begin{cases} |d_{2j}| & \text{if } d_{2j} \text{ and } d_{2o} \text{ are of opposite signs} \\ |d_{2j}| - |d_{2o}| & \text{if } d_{2j} \text{ and } d_{2o} \text{ are of the same sign and } |d_{2j}| > |d_{2o} \\ 0 & \text{if } |d_{2j}| \leq |d_{2o}| \end{cases}$$

and

$$\text{reuse}(3,j) = \begin{cases} |d_{3j}| & \text{if } d_{3j} \text{ and } d_{3o} \text{ are of opposite signs} \\ |d_{3j}| - |d_{3o}| & \text{if } d_{3j} \text{ and } d_{3o} \text{ are of the same sign and } |d_{3j}| > |d_{3o} \\ 0 & \text{if } |d_{3j}| \leq |d_{3o}| \end{cases}$$

and the number of distinct accesses as $N_1 \times N_2 \times N_3 \times r -$ reuse

Example 18:

for $i = 1$ to 10 do
  for $j = 1$ to 20 do
    for $k = 1$ to 30 do
      $\cdots A[3i + k, j + k] \cdots$
      $\cdots A[3i + k + 5, j + k + 1] \cdots$
    endfor
  endfor
endfor

The dependence matrix is as follows: D = $\begin{pmatrix} 1 & 0 & 1 \\ -1 & 4 & 3 \\ 2 & -5 & -3 \end{pmatrix}$. Here we have, $\text{reuse}(2,1) = 1$, $\text{reuse}(3,1) = 2$, $\text{reuse}(2,2) = 1$, and $\text{reuse}(3,2) = 2$.

Reuse on $S_1 = 10 \times 16 \times 25 + 9 \times 27 \times 1 + 9 \times 17 \times 2 = 4000 + 243 + 306 = 4549$

Reuse on $S_2 = 9 \times 19 \times 28 + 9 \times 27 \times 1 + 9 \times 17 \times 2 = 4788 + 243 + 306 = 5337$

Total reuse = 4549 + 5337 = 9886

Number of distinct elements accessed = $10 \times 20 \times 30 \times 2 - 9886 = 2114$

4.7 ALGORITHM 3

Consider the following triangular loop:

for $i = 1$ to n_1 do
  for $j = i$ to n_2 do
    $\cdots A[\lambda_1 i + \lambda_2 j + c] \cdots$
  endfor
endfor

Let the dependence vector (reuse direction) be (d_1, d_2). The algorithm to estimate the distinct accesses is based on the n-dimensional cube of the iteration space which represents the reused elements defined by the dependence vector. We will consider the following cases.

Case 1: $d_1 > 0$ and $d_2 \leq 0$

- Let $|d_2| \leq n_2 - n_1$ (See Figure 4.3). The reused elements are given by

$$\sum_{d_1+1}^{n_1} (n_2 - d_2 - i + 1)$$

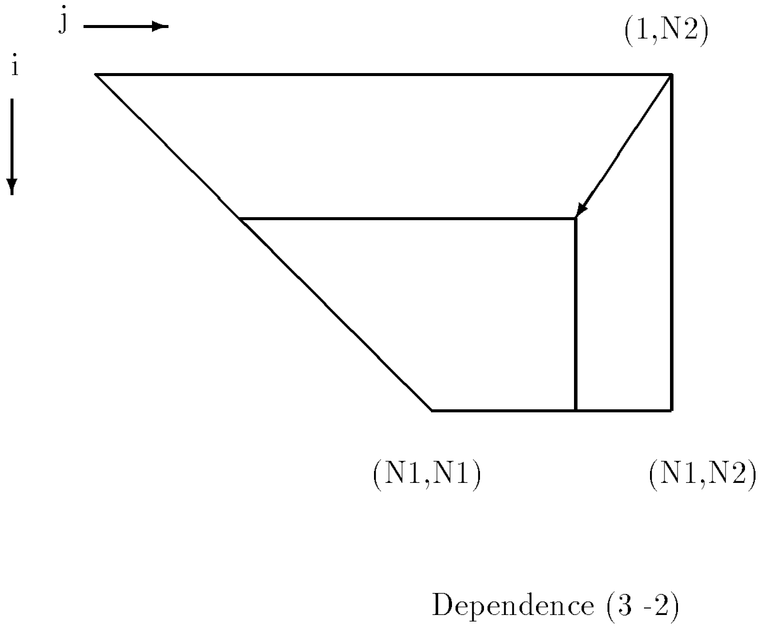


Figure 4.3: Triangular iteration space - Case 1.1

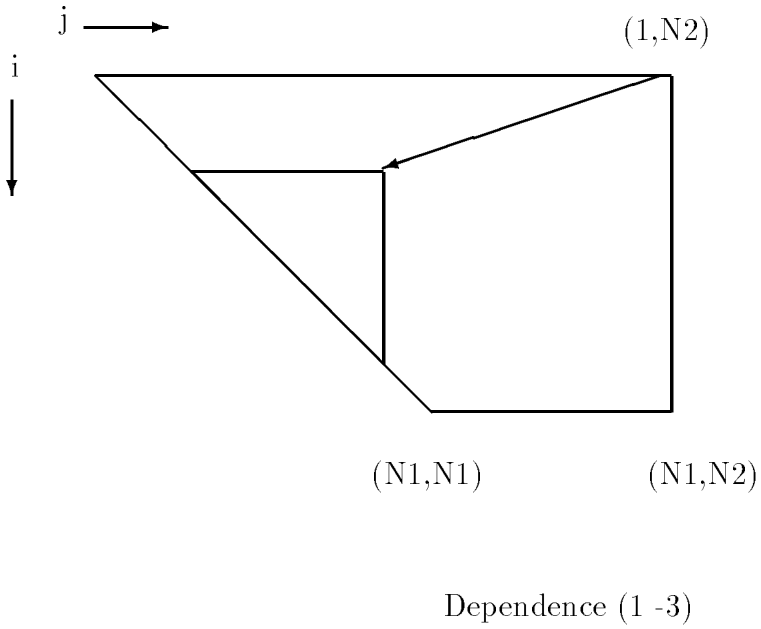


Figure 4.4: Triangular iteration space - Case 1.2

or,

$$\text{Reuse} = \sum_{d_1+1}^{n_1} (n_2 - d_2 + 1) - \sum_{d_1+1}^{n_1} i$$

i.e.,

$$\text{Reuse} = (n_2 - d_2 + 1)(n_1 - d_1) - (\sum_{1}^{n_1} i - \sum_{1}^{d_1} i)$$

Simplifying we get

$$\text{Reuse} = (n_2 - d_2 + 1)(n_1 - d_1) - \frac{1}{2}(n_1 - d_1)(n_1 + d_1 + 1)$$

- $|d_2| > n_2 - n_1$ (See Figure 4.4) Here the reused elements are given by

$$\sum_{d_1+1}^{n_2-d_2} (n_2 - d_2 - i + 1)$$

or,

$$\text{Reuse} = \sum_{d_1+1}^{n_2-d_2} (n_2 - d_2 + 1) - \sum_{d_1+1}^{n_2-d_2} i$$

i.e.,

$$\text{Reuse} = (n_2 - d_2 + 1)(n_2 - d_2 - d_1) - (\sum_{1}^{n_2-d_2} i - \sum_{1}^{d_1} i)$$

Simplifying we get

$$\text{Reuse} = (n_2 - d_2 + 1)(n_2 - d_2 - d_1) - \frac{1}{2}(n_2 - d_2 - d_1)(n_2 - d_2 + d_1 + 1)$$

Case 2: $d_1 \geq 0$ and $d_2 \geq 0$

1. $\frac{d_2}{d_1} \geq 1$
   - $|d_2| \leq n_2 - n_1$ (See Figure 4.5). The reused elements are given by

$$\sum_{d_1+1}^{n_1} (n_2 - d_2 - i + 1)$$

which has been already calculated as

$$\text{Reuse} = (n_2 - d_2 + 1)(n_1 - d_1) - \frac{1}{2}(n_1 - d_1)(n_1 + d_1 + 1)$$

- $d_2 > n_2 - n_1$ (See Figure 4.6). Here the reused elements are given by

$$\sum_{d_1+1}^{n_2-d_2} (n_2 - d_2 - i + 1)$$

and,

$$\text{Reuse} = (n_2 - d_2 + 1)(n_2 - d_2 - d_1) - \frac{1}{2}(n_2 - d_2 - d_1)(n_2 - d_2 + d_1 + 1)$$

2. $\frac{d_2}{d_1} < 1$ (See Figure 4.7). The reused elements are given by

$$\sum_{d_1+1}^{n_1} (n_2 - i + 1)$$

or

$$\text{Reuse} = \sum_{d_1+1}^{n_1} (n_2 + 1) - \sum_{d_1+1}^{n_1} i$$

i.e.,

$$\text{Reuse} = (n_2 + 1)(n_1 - d_1) - (\sum_{1}^{n_1} i - \sum_{1}^{d_1} i)$$

Simplifying we get

$$\text{Reuse} = (n_2 + 1)(n_1 - d_1) - \frac{1}{2}(n_1 - d_1)(n_1 + d_1 + 1)$$

Similarly we can get an estimate on other triangular loops.
The number of distinct accesses are calculated as $n_1 \times n_2 -$ reuse

4.8 SUMMARY

We discussed the problem of estimating the number of distinct references to an array in this chapter. We illustrated a method to compute the exact number of distinct array elements for single and multiple references for cases where the dimensions of the array is equal to or less than the loop nesting level. We showed the relationship between the range obtained by using Banerjee's inequalities and the exact figure of the number of distinct references. For uniformly generated references, we show that exact results can be obtained by using the concept of the n-dimensional cube formed by the dependence vector. Gannon *et al.* [9] present a framework where the number of elements are estimated looking at the image of the iteration space on the array space and they give lower and upper bounds. Ferrante *et al.* [8] give an estimate for single references of uniformly generated data dependences and use correction factors for Banerjee's inequalities for non-uniformly generated references.

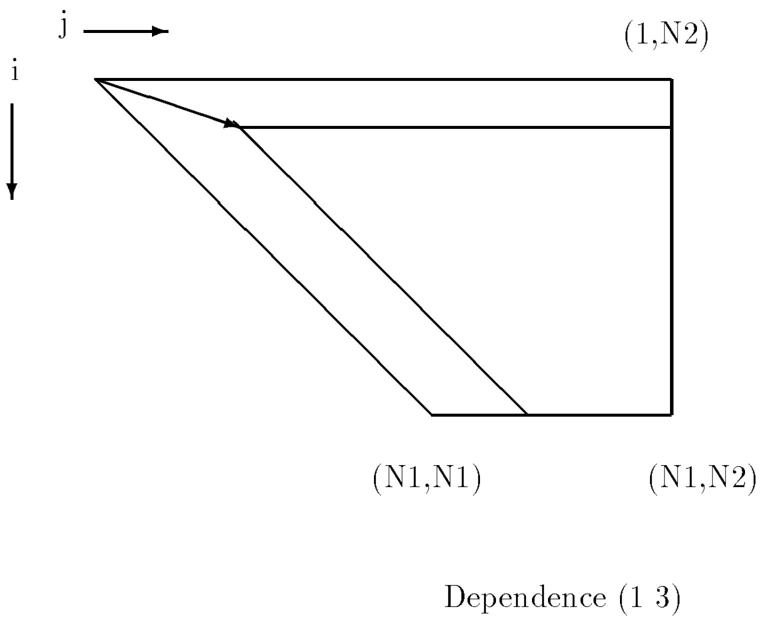


Figure 4.5: Triangular iteration space - Case 2.1.1

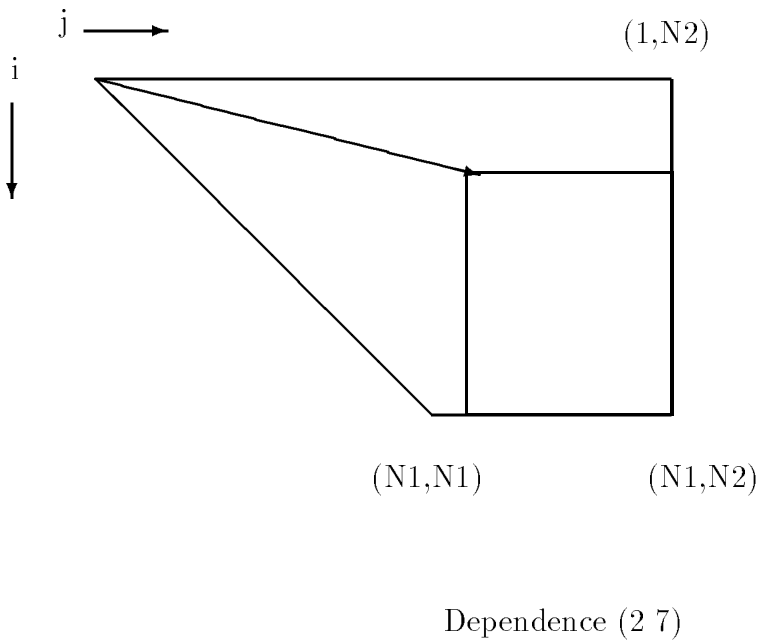
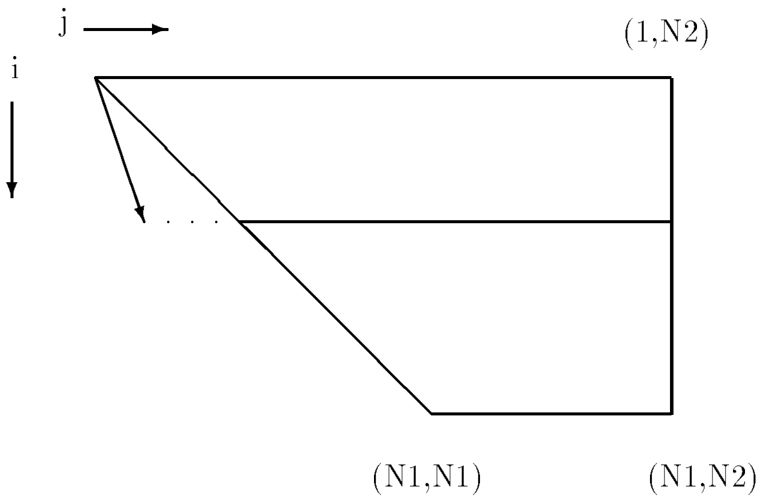


Figure 4.6: Triangular iteration space - Case 2.1.2



Dependence (3 1)

Figure 4.7: Triangular iteration space - Case 2.2.1

CHAPTER 5

SUMMARY AND CONCLUSIONS

Data locality is highly critical in achieving high performance on parallel machines. Accesses to data which are non-local results in communication which greatly affects performance. We addressed this problem by deriving compound loop transformation s which enhance data locality. In Chapter 2, we presented a technique for choosing transformations that enhance locality while enabling parallelization in the case of uniformly generated array references. We used the reference window size associated with a given loop structure as a measure of reuse; transformations were derived in order to minimize the reference window size as a means of exploiting reuse to achieve locality.

We showed a method of estimating the window size as a function of an arbitrary unimodular transformation applied to the loop nest. We showed two examples to illustrate how our technique works. Unlike the works of Eisenbeis *et al.* [6] and Li and Pingali [14] which focus only on input and output dependences, our technique exploits the entire dependence structure to derive transformations that optimize locality. The work presented here was restricted to uniformly generated references and the transformation was assumed to be unimodular. Work is in progress in extending this technique to general references and also in deriving non-unimodular transformations.

When a number of accesses are to be made across processors on a distributed memory machine or different levels of hierarchy in memory, it is more efficient to use block transfers to reduce the communication startup overhead on the machine. Also a good data distribution along with a suitable computation allocation onto the processors to optimize communication is an important goal of any optimizing compiler. We have attempted to achieve this goal by deriving compound loop transformation using the access functions of the references within the loop. Chapter 3 illustrated an algorithm which derives the terms in the transformation matrix which gives the best locality and minimum communication on distributed memory machines. We used the concept of a *data reference matrix* for individual array references. Using this as the starting point we systematically derived the best set of transformation matrices which give both good locality while enabling parallelism. Unlike [14], where a *padding matrix* is used along with an arbitrary set of rows in a *basis matrix,* we generate a transformation matrix systematically. We showed the example of matrix multiplication where we proved that no transformation is required to get the best performance; appropriate data distribution is derived using our algorithm. Also we gave two different transformation matrices for the matrix multiplication problem with equally good performance. The algorithm also gives an optimal distribution of arrays on to the processors such that block transfers are enabled to reduce inter-processor communication. The input to the algorithm is the data reference matrices of all the references to all the arrays.

In this work we have considered distribution of data along one dimension only. However complex distributions where the allocation of arrays with allocation functions having more than one index can be extended using the above algorithm. The block transfers were set up just before executing the innermost loop. This can be generalized to be set up before any appropriate iteration depending on the dependences prevailing in the program. An extension of the algorithm presented here can determine the position of block transfers of array elements within the loop. Also, we have only considered the allocation of the outermost loop iterations to each processor. This can be extended to allocate multiple iteration levels onto a single processor. In other words, the distribution function of the iterations over the processors can be more complex.

We discussed the problem of estimating the number of distinct references in an array in Chapter 4. In common scientific programs an array element accessed in a nested loop are usually repeatedly referenced during the execution of the loop. It is of interest to look at the mapping of the iteration space onto the array space. We illustrated a method to compute the exact number of distinct array elements for single and multiple references for cases where the dimensions of the array is equal to or less than the loop nesting level. We showed the relationship between the range obtained by using Banerjee's inequalities and the exact figure of the number of distinct references. For uniformly generated references, we show that exact results can be obtained by using the concept of the n-dimensional cube formed by the dependence vector. Gannon *et al.* [9] present a framework where the number of elements are estimated looking at the image of the iteration space on the array space and they give lower and upper bounds. Ferrante *et al.* [8] give an estimate for single references of uniformly generated data dependences and use correction factors for Banerjee's inequalities for non-uniformly generated references. We exploit the closeness of the expression using Banerjee's inequalities to the number of accesses for non-uniformly generated references for commonly occurring scientific code and give very close bounds.

We have addressed the problem of estimating the distinct accesses for specific cases in loops which have uniformly generated or non-uniformly generated references. For uniformly generated references, it was shown that exact estimates were possible for cases where the loop nesting n is the same as the dimension d of the array accessed as well as when $d = n - 1$. Further work can be done for arbitrary values of n and d. In such cases, though the dependence vectors are constant, the null space vectors for a single access function are in more than one direction and suitable changes needs to be done in the algorithm. For non-uniformly generated references, we gave close estimates when the coefficients of the access functions of the arrays are small compared to the loop limits which is common in most scientific programs. However, for cases where the coefficients are comparable or larger than the loop limits we have shown close bounds for two references. Further work can be done for arbitrary number of non-uniformly generated references with an extension of the concept introduced here.

BIBLIOGRAPHY

[1] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Programming Languages and Systems,* Vol. 9, No. 4, 1987, pp. 491-542.

[2] C. Ancourt and F. Irigoin, Scanning polyhedra with DO loops. In *Proc. Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP),* pages 39–50, April 1991.

[3] Jennifer M. Anderson and Monica S. Lam, Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, June 1993.

[4] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Massachusetts, 1988.

[5] U. Banerjee. Unimodular Transformation of Double Loops. *Advances in Languages and Compilers for Parallel Processing,* A. Nicolau et al., (Eds.), Pitman, London, 1991, pp. 192–219.

[6] C. Eisenbeis, W. Jalby, D. Windheiser and F. Bodin. A strategy for array management in local memory. *Advances in Languages and Compilers for Parallel Computing,* MIT Press, pp. 130–151, 1991.

[7] Z. Fang and M. Lu. A solution to cache thrashing problem in RISC based parallel processing systems. *Proc. 1991 International Conf. Parallel Processing,* August 1991.

[8] J. Ferrante, V. Sarkar and W. Thrash. On Estimating and Enhancing Cache Effectiveness. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing,* August 1991.

[9] K. Gallivan, W. Jalby and D. Gannon. On the Problem of Optimizing Data Transfers for Complex Memory Systems. *Proc. 1988 ACM International Conference on Supercomputing,* St. Malo, France, pp. 238-253.

[10] D. Gannon, W. Jalby and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformations. *Journal of Parallel and Distributed Computing,* Vol. 5, No. 5, October 1988, pp. 587-616.

[11] F. Irigoin and R. Triolet. Supernode Partitioning. *Proc. 15th Annual ACM Symp. Principles of Programming Languages,* San Diego, CA, Jan. 1988, 319-329.

[12] K. Kennedy and K. McKinley. Optimizing for parallelism and data locality. *Proc. 1992 ACM International Conference on Supercomputing,* Washington DC, July 1992.

[13] M. Lam, E. Rothberg and M. Wolf. The cache performance and optimization of blocked algorithms. *Proc. 4th Intl. Conf. Architectural Support for Programming Languages and Operating Systems,* April 1991.

[14] W. Li and K. Pingali. A Singular Loop Transformation Framework Based on Non-singular Matrices. *Proc. 5th Workshop on Languages and Compilers for Parallel Computing,* August 1992.

[15] J. Ramanujam. *Compile-time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors.* Ph.D. Thesis, The Ohio State University, September 1990.

[16] J. Ramanujam and P. Sadayappan. Multidimensional Iteration Space Tiling for Non-shared Memory Machines. in *Proc. Supercomputing 91,* pages 111–120, November 1991.

[17] J. Ramanujam, A. Narayan and G. Vaidyanathan. Loop restructuring for data locality. Technical Report 92–09–01, Dept. of Electrical and Computer Engineering, Louisiana State University, September 1992.

[18] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. Parallel and Distributed Syst.,* October 1991.

[19] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation,* pp. 30–44, June 1991.

[20] M. Wolfe and U. Banerjee. Data Dependence and Its Application to Parallel Processing. *International Journal of Parallel Programming,* Vol. 16, No. 2, pages 137–178, 1987.

[21] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, London and the MIT Press, Cambridge, Massachusetts, 1989.

[22] M. Wolfe. More Iteration Space Tiling. *Proc. Supercomputing 89,* Reno, Nevada, pages 655–664, November 1989.

[23] H. Zima and B. Chapman. *Supercompilers for parallel and vector supercomputers.* ACM Press Frontier Series, 1990.

VITA

VITA

Ashish Narayan was born in Mandya, India on October 19, 1964. He graduated from Bachelor of Engineering from Sri Jayachamarajendra College of Engineering, Mysore, India in Electronics and Communications Engineering in December 1986. He graduated from Master of Science in Electrical Engineering from Louisiana State University, Baton Rouge in May 1994. His research interests include Parallelizing Compilers, Computer Architecture and Neural Networks.

Narayan, Ashish, B.E., Sri Jayachamarajendra College of Engineering, 1986
Master of Science in Electrical Engineering, Spring Commencement, 1994
Major: Electrical Engineering
Optimizing Data Locality and Parallelism for Scalable Multiprocessors
Thesis directed by Assistant Professor Jagannathan Ramanujam
Pages in Thesis, 102. Words in Abstract, 283.

ABSTRACT

As processor speeds continue to increase at a much higher rate than memory speeds, improving memory performance becomes critical in realizing high performance in parallel machines. The aim is to have most of the memory accesses to faster memory in the memory hierarchy resulting in high levels of data locality. In many cases, it is best left to an optimizing compiler to restructure the program to extract good memory performance. The programmer will otherwise have to write code to exploit the parallelism inherent in the input code and also to improve data locality. In scientific codes, since most of the CPU time is spent in loops, we study the effect of restructuring loops on the performance. Loop transformations to exploit parallelism and to improve data locality are the most important techniques used today. Data locality and parallelism are conflicting goals and there is always a trade-off between the two. Improving data locality is aimed at reducing communication between processors or between different levels in the memory hierarchy. An important aspect in good memory performance is the estimation of the amount of local memory needed for an array accessed in a nested loop. Data transfers between levels and processors has to optimized for the best performance.

This thesis presents loop transformations to automatically restructure code at compile time which aims in improving data locality. We also present an algorithm to obtain an optimum compound loop transformation for task allocation and data distribution on distributed memory machines which yields in optimal communication and efficient use of block transfers, resulting in message vectorization. Finally, we discuss a technique to estimate the number of distinct references to an array which facilitates efficient data transfers to local memory.