Reshaping Access Patterns for Improving Data Locality

Aart J.C. Bik
Computer Science Department, Indiana University
Lindley Hall 215, Bloomington, Indiana 47405-4101, USA
ajcbik@cs.indiana.edu

Peter M.W. Knijnenburg
Computer Science Department, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, the Netherlands
peterk@cs.leidenuniv.nl

Abstract

In this paper, we present a method to construct a loop transformation that simultaneously reshapes the access patterns of several occurrences of multi-dimensional arrays along certain desired access directions. First, the method determines a direction through the original iteration space along which these desired access directions are induced. Subsequently, a unimodular transformation is constructed that changes the iteration space traversal accordingly. Finally, data dependences are accounted for. In particular, this reshaping method can be used to improve data locality in a program.

1 Introduction

It is well known that the performance of programs heavily depends on the degree of locality exhibited by the accesses in the program. If a program exhibits good temporal and spatial locality, most memory references will be to data present in the cache. Since the speed of processors and main memory tend to grow apart more and more, good locality is of ever greater importance to performance. It is therefore important to be able to restructure the program in such a way that locality is improved. The traditional way to improve locality is by *blocking* or *tiling* iteration spaces. In this approach the iteration space is cut up into pieces so that each piece only refers to a subset of the array that fits into the cache. Several approaches try to estimate the amount of locality in inner-loops, for example by means of windows [10], to drive heuristic strategies for applying blocking, permutation and fusion of loops [10, 8, 7, 11]. Recently, sophisticated multilevel blocking algorithms have been devised that take into account registers and multilevel caches [15]. Other approaches try to construct unimodular loop transformations. In the context of NUMA architectures, Li and Pingali [14] construct a transformation for access normalization in order to to localize array references; (affine) index functions are mapped onto target loop indices in such a way that inner target loops only refer to array elements local to the processor. Wolf and Lam [19] present an advanced method incorporating estimates of locality that includes unimodular transformations and tiling. Recently, algorithms that compute both computation and data decomposition have been proposed proposed [1, 2]. These algorithms, however, focus on communication minimization rather than cache behavior optimization, like the present paper. More recently, approaches that try to rearrange the layout of arrays for improving data locality have been proposed [16]. Note that this last approach is different from the previously cited ones in that the loop structures in the program remain unaffected.

In this paper, we present a novel approach to improving spatial locality, namely, we want to *reshape access patterns* of arrays. The objective is to reshape access patterns so that they become either column-wise or row-wise, depending on whether column-major (e.g., Fortran), or row-major storage of arrays is used. In this approach, the iteration space of a loop nest is transformed in such a way that the innermost loop will access arrays along columns or rows, respectively. In case of column-major storage, this means that the first index expression in an array reference should contain the loop index of the innermost loop, and the other index expressions should not. If we are able to reshape access patterns in this way, then obviously the locality of a program will be improved. We present a general method to construct a valid loop transformation that simultaneously reshapes the access pattern of several occurrences of (possibly different) multi-dimensional arrays along certain desired access directions. The method can be successively applied to each individual loop nest, thereby improving the efficiency of the program as a whole.

As far as the authors are aware, the present approach is new. In a number of cases, the proposed construction yields a loop interchange, like the strategies reported in [8]. The difference is that in our case this interchange is computed, whereas in [8] is selected by a heuristic. The construction in [14] constructs a transformation based on affine access functions present in the loop, like our approach. However, in the latter work an entire index expression is mapped onto one loop index in the target loop. This means, in general, that all index expressions in one array reference in the target loop may refer to this loop index. This should be contrasted to our approach in which only one index expression will refer to the innermost loop index, namely, the leftmost expression in case of column-major storage. This is obviously preferable for locality purposes. An interesting extension of the present work is to study how the techniques of blocking and reshaping access patterns can be combined. Reshaping access patterns could be used as a preprocessing transformation for the blocking algorithms cited above: the transformed loop will have 'compact' reference windows in inner loops.

Improving data locality is only one application of access pattern reshaping. We are able to present the construction of the transformation in a general setting: the transformation reshapes each access pattern along *an arbitrary preferred access direction.* This means that we are able to reshape access patterns so that they become column-wise, row-wise, along a diagonal, or along an arbitrary other line through the array. There are several reasons for taking this approach.

First, enforcing column-wise or row-wise access are just special instances of the same technique. Therefore, this technique can be incorporated in compilers for arbitrary languages.

Second, in [16] a notion of data transformation is discussed that changes the layout of an array and propagates this change throughout the program. Such a data transformation is given by a non-singular matrix D. In that paper, it has been observed that loop transformations and data transformation are, in a certain sense, the dual of each other: given a data transformation D and a loop transformation U, an access matrix W is transformed to DWU^{-1}. Since we are able to transform a loop so that the accesses to an array will be diagonal-wise, for instance, after this transformation the layout of the array can be skewed by means of a data transformation so that the accesses become column-wise in the transformed array. In this way, the access to an array can be made column-wise, even if a single loop or data transformation cannot be found that does the job. The precise connection between data and loop transformations as well as how they can be used together for improving locality will be the topic of future research.

Finally, the reshaping method has been used in a completely different context, namely, in the automatic conversion of a dense program into semantically equivalent sparse code [5]. Because the entries in a sparse storage scheme can usually only be generated efficiently along one particular direction, here it is important to be able to change the access direction along arbitrary preferred directions before sparse storage schemes are selected by the compiler. In [6], a preliminary version of the reshaping method of this paper (restricted to two-dimensional arrays) has been used for this purpose.

The outline the paper is as follows. Preliminaries are given first in section 2. The reshaping method is presented in section 3. In section 4 we explore how this reshaping method can be used to enhance data locality in perfectly nested loop and we provide some measurements. Finally, we state some conclusions in section 5.

2 Preliminaries

In this section, we first give some definitions. Thereafter, we discuss the effect of applying a unimodular transformation to a perfectly nested loop on the order in which the iterations of this loop are executed.

2.1 Definitions

We will use the following definitions [3, 20, 21]. The relation '$\prec_k$' on $\mathcal{Z}^d$ is defined as follows, where $\vec{\imath}, \vec{\jmath} \in \mathcal{Z}^d$ and $1 \leq k \leq d$:

$$\vec{\imath} \prec_k \vec{\jmath} \quad \Leftrightarrow \quad i_1 = j_1, \ldots, i_{k-1} = j_{k-1}, i_k < j_k$$

The **lexicographical order** '$\prec$' on $\mathcal{Z}^d$ is defined in terms of this relation:

$$\vec{\imath} \prec \vec{\jmath} \quad \Leftrightarrow \quad \exists\ 1 \leq k \leq d : \vec{\imath} \prec_k \vec{\jmath}$$

We have $\vec{\imath} \preceq \vec{\jmath}$, if either $\vec{\imath} \prec \vec{\jmath}$ or $\vec{\imath} = \vec{\jmath}$. The relations '$\succ_k$', '$\succ$' and '$\succeq$' are defined similarly.

In this paper, we will use the framework of **unimodular transformations** [3, 9, 13]. Given a perfectly nested loop with stride-1 DO-loops, index vector $\vec{\mathtt{I}} = (\mathtt{I}_1, \ldots, \mathtt{I}_d)$, and iteration space $IS \subseteq \mathcal{Z}^d$, any combination of loop interchanging, loop skewing and loop reversal (see e.g. [17, 20, 21]) that transforms this loop into another loop with index vector $\vec{\mathtt{I}}' = (\mathtt{I}'_1, \ldots, \mathtt{I}'_d)$ and iteration space $IS' \subseteq \mathcal{Z}^d$ can be modeled by a linear transformation $U : IS \rightarrow IS'$ that is defined by a $d \times d$ unimodular matrix U.[1] This means that an iteration $\vec{\imath} \in IS$ is mapped to an iteration $\vec{\imath} \in IS'$ as follows:

$$\vec{\imath}' = U\vec{\imath}$$

We assume that all data dependences occurring in the original loop are represented by a set of **distance vectors** $\mathcal{D} \subseteq \mathcal{Z}^d$, where $\vec{d} \in \mathcal{D}$ implies that a statement instance executed during iteration $\vec{\jmath} \in IS$ depends on an instance executed during iteration $\vec{\imath} \in IS$, where $\vec{\jmath} = \vec{\imath} + \vec{d}$. Note that $\vec{d} \succeq \vec{0}$ holds for all $\vec{d} \in \mathcal{D}$.

Application of a loop transformation defined by U is valid if and only if $U\vec{d} \succeq \vec{0}$ holds for all $\vec{d} \in \mathcal{D}$. Converting the *original* loop with index vector $\vec{\mathtt{I}}$ into the *target* loop with index vector $\vec{\mathtt{I}}'$ is implemented by (i) rewriting the loop-body of the original loop according to $\vec{\mathtt{I}} = U^{-1}\vec{\mathtt{I}}'$, and (ii) generating new loops that induce a lexicographical traversal of the target iteration space. If the original iteration space IS can be represented by an integer system $A\vec{\mathtt{I}} \leq \vec{b}$, the second step consists of generating loop bounds that induce a lexicographical traversal of all discrete points in $AU^{-1}\vec{\mathtt{I}}' \leq \vec{b}$. This generation is usually accomplished using Fourier-Motzkin elimination.

Fourier-Motzkin elimination [3, 20] can be used to test the consistency of a *reasonably small* system of linear inequalities $A\vec{x} \leq \vec{b}$, or to convert this system into a form in which the lower and upper bounds of each variable x_i are expressed in terms of the variables $x_1, \ldots, x_{i-1}$ only. Consistency indicates that the system has at least one *rational* solution, and can be used as a necessary (but not sufficient) condition under which an *integer* solution exists. An extension to Fourier-Motzkin elimination that can test for the existence of integer solutions is given by the Omega-test [18].

2.2 Iteration Space Traversal

Because iterations in both the original and target iteration space are traversed in lexicographical order, a unimodular transformation effectively changes the order in which iterations are executed. Since the index vectors of the original and target loop are related as $\vec{\mathtt{I}}' = U\vec{\mathtt{I}}$, we can make the following observations:

(a) Let $\vec{u} \in \mathcal{Z}^d$ denote the first row of U. Then, for fixed $\mathtt{I}'_1 = i'_1$, the more inner DO-loops of the target loop execute iterations in IS' that correspond to all iterations of IS in the hyperplane $\{\vec{\mathtt{I}} \in \mathcal{Z}^d | \vec{u} \cdot \vec{\mathtt{I}} = i'_1\}$.

[1] An integral matrix U that satisfies $\det(U) = \pm 1$ is called a unimodular matrix.

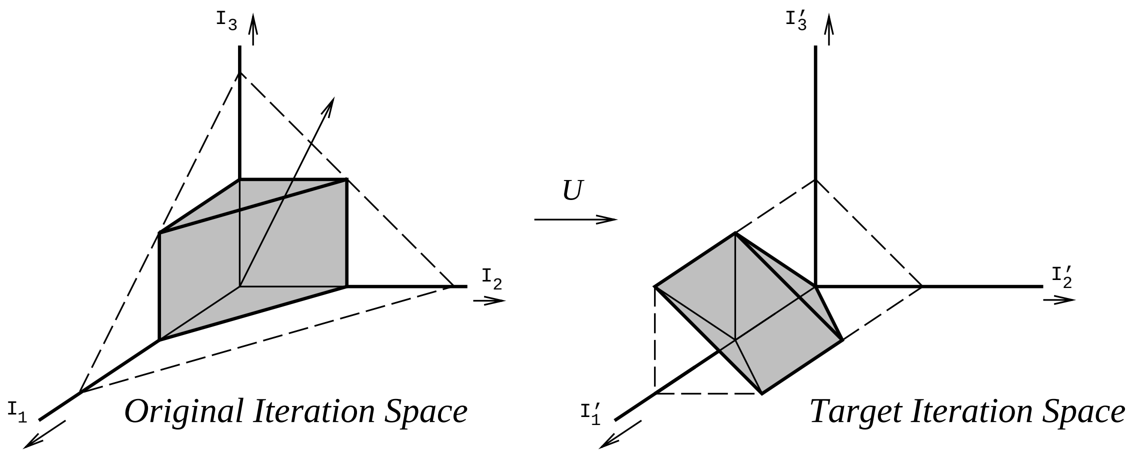

Figure 1: Application of a Unimodular Transformation

(b) Let $\vec{u}' \in \mathcal{Z}^d$ denote the last column of the inverse matrix U^{-1}. Then, for fixed $\mathtt{I}'_1 = i'_1, \ldots, \mathtt{I}'_{d-1} = i'_{d-1}$, the innermost DO-loop of the target loop successively executes iterations in IS' that correspond to iterations of IS along a single straight line with direction $\vec{u}'$.

Inner loop concurrentization methods [4, 12, 20] exploit the first observation by enforcing a successive traversal of hyperplanes in the original iteration space in successive iterations of the outermost DO-loop of the target loop such that all iterations in each individual hyperplane are completely independent. The second observation, on the other hand, forms the basis of the reshaping method presented in this paper.

EXAMPLE: Consider the following matrices:

$$U = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & \Leftrightarrow 1 & \Leftrightarrow 1 \end{pmatrix}$$

These unimodular matrices define the following loop transformation:

```
                                    DO I'1 = 0, 100
DO I1 = 0, 50                         DO I'2 = 0, MIN(50,I'1)
  DO I2 = 0, 50 - I1                    DO I'3 = MAX(0,I'1-I'2-50),
    DO I3 = 0, 50                  +                MIN(50-I'2,I'1-I'2)
      B(I1,I2,I3)            →            B(I'2,I'3,I'1-I'2-I'3)
    ENDDO                               ENDDO
  ENDDO                               ENDDO
ENDDO                               ENDDO
```

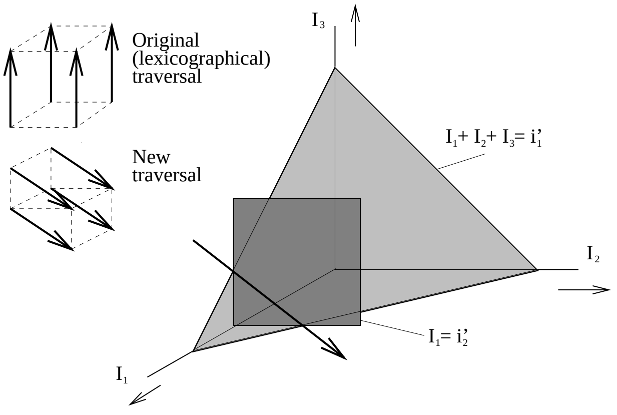

Figure 2: Traversal of the *Original* Iteration Space

The transformation of the original iteration space IS into the target iteration space IS' is illustrated in figure 1. In figure 2, we illustrate how the *original* iteration space IS is effectively traversed *after* the transformation.

Observation (a) reveals that because the first row of U is $(1, 1, 1)$, for fixed $\mathtt{I}'_1 = i'_1$ the target loop executes all iterations corresponding to iterations of IS in the plane $\mathtt{I}_1 + \mathtt{I}_2 + \mathtt{I}_3 = i'_1$.

For fixed $\mathtt{I}'_1 = i'_1$ and $\mathtt{I}'_2 = i'_2$, the target loop executes all iterations that correspond to iterations in the intersection of planes $\mathtt{I}_1+\mathtt{I}_2+\mathtt{I}_3 = i'_1$ and $\mathtt{I}_1 = i'_2$. Observation (b) reveals that this intersection forms a straight line with direction $(0,1,\Leftrightarrow 1)$, i.e., the last column of U^{-1}. For example, the iterations $(0,0,1)$ and $(0,1,0)$ in IS are mapped to the successively executed iterations $(1,0,0)$ and $(1,0,1)$ in IS'.

3 Reshaping Access Patterns

First, we state the objective of reshaping. Thereafter, we present the reshaping method in detail.

3.1 Objective of Reshaping

Suppose that the subscripts of an occurrence of a n-dimensional array in a perfectly nested loop with index vector $\vec{\mathtt{I}} = (\mathtt{I}_1, \ldots, \mathtt{I}_d)^T$ are represented by the following affine transformation $F : \mathcal{Z}^d \to \mathcal{Z}^n$:

$$F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$$

The last column of W is called the **access direction** $\vec{r} \in \mathcal{Z}^n$ of this occurrence: the array is traversed along lines with direction $\vec{r} = W(0,\ldots,0,1)^T$. Any loop transformation defined by a $d \times d$ unimodular matrix U changes this access direction by replacing the original subscripts $F(\vec{\mathtt{I}}) = \vec{v} + W\vec{\mathtt{I}}$ with $F'(\mathtt{I}') = \vec{v} + WU^{-1}\vec{\mathtt{I}}'$.

We say that *a loop transformation reshapes the access patterns of this occurrence along a preferred access direction $\vec{s} \in \mathcal{Z}^n$, if this latter vector and the resulting true access direction $\vec{r}\,' \in \mathcal{Z}^n$ are linearly dependent.* This implies that the following equation holds for some λ.

$$\vec{r}\,' = WU^{-1}(\underbrace{0,\ldots,0}_{d-1},1)^T = \lambda \cdot \vec{s}$$

We can use the following proposition, which says that the collection of all vectors that are linearly dependent on $\vec{s}$ are given by the kernel of $\mathcal{S}(\vec{s})$.

Proposition 3.1 *Let $\vec{r}, \vec{s} \in \mathcal{Z}^n$, where $\vec{s} \neq \vec{0}$. Let k be the least index such that $s_k \neq 0$. Then, $\exists\lambda.\vec{r} = \lambda\cdot\vec{s}$ iff $\mathcal{S}(\vec{s})\vec{r} = \vec{0}$, where $\mathcal{S}(\vec{s})$ is the following $(n \Leftrightarrow 1) \times n$ matrix:*

$$\mathcal{S}(\vec{s}) = \begin{pmatrix} s_k & & & & & & \\ & \ddots & & \vdots & & & \\ & & s_k & -s_{k-1} & & & \\ & & & -s_{k+1} & s_k & & \\ & & & \vdots & & \ddots & \\ & & & -s_n & & & s_k \end{pmatrix}$$

$$\text{with } -s_1 \text{ heading the fourth column}$$

PROOF '$\Rightarrow$' Suppose $\vec{r} = \lambda\cdot\vec{s}$, or $r_1 = \lambda\cdot s_1, \ldots, r_n = \lambda\cdot s_n$. Then necessarily $\lambda = r_k/s_k$, since by assumption $s_k \neq 0$. Substituting r_k/s_k for λ and rewriting yields the second equation.

'$\Leftarrow$' Conversely, suppose $\mathcal{S}(\vec{s})\vec{r} = \vec{0}$. This can be rewritten to $s_k \cdot r_i \Leftrightarrow s_i \cdot r_k = 0$ for all $1 \leq i \leq n$. Hence, for all i, $r_i = (r_k/s_k)\cdot s_i$. Hence, by setting $\lambda = r_k/s_k$, we infer that there exists a λ such that for all i, $r_i = \lambda\cdot s_i$. []

Note that we allow reshaping that results in 'scalar-wise' access patterns (which will improve *temporal* locality), since the equation is trivially satisfied for the vector $\vec{r} = \vec{0}$.

3.2 Method of Reshaping

In this section, we present a general method to construct a valid unimodular transformation that simultaneously reshapes the access patterns of m occurrences of, possibly different, multi-dimensional arrays in a single perfectly nested loop with index vector $\vec{\mathtt{I}} = (\mathtt{I}_1,\ldots,\mathtt{I}_d)^T$ along certain preferred access directions. We assume that the ith occurrence is a n_i-dimensional array with subscripts $F_i(\vec{\mathtt{I}}) = \vec{v}_i + W_i\vec{\mathtt{I}}$ (W_i is a $n_i \times d$ matrix) and that the preferred access direction for this occurrence is given by $\vec{s}_i \in \mathcal{Z}^{n_i}$.

We also assume the data dependence structure of this loop is represented by a set $\mathcal{D} \subseteq \mathcal{Z}^d$ of dependence distance vectors.

3.2.1 Preferred Iteration Directions

According to proposition 3.1, a loop transformation defined by a $d \times d$ unimodular matrix U reshapes the access patterns of the ith occurrence in the loop along the preferred access direction $\vec{s}_i \in \mathcal{Z}^{n_i}$, if the last column $\vec{\alpha} \in \mathcal{Z}^d$ of the inverse matrix U^{-1} satisfies the following objective:

$$\mathcal{S}(\vec{s}_i)\cdot W_i\,\vec{\alpha} = \vec{0}$$

After application of such a transformation, the original iteration space is effectively traversed along straight lines with the direction $\vec{\alpha} \in \mathcal{Z}^d$ (cf. observation (b) in section 2). Therefore, this vector is referred to as the **preferred iteration direction**.

All access patterns are reshaped simultaneously if $\vec{\alpha} \in Z^d$ satisfies $S\vec{\alpha} = \vec{0}$ for the following **objective matrix**:

$$S = \begin{pmatrix} \mathcal{S}(\vec{s}_1) & & \\ & \ddots & \\ & & \mathcal{S}(\vec{s}_m) \end{pmatrix} \begin{pmatrix} W_1 \\ \vdots \\ W_m \end{pmatrix}$$

Since the elements in each row or column of a unimodular matrix must be relatively prime, any integer solution of $S\vec{\alpha} = \vec{0}$ with $\gcd(\alpha_1, \ldots, \alpha_d) = 1$ may be used as preferred iteration direction, which gives rise to the following set $\Delta \subseteq \mathcal{Z}^d$:

$$\Delta = \{\vec{\alpha} \in \mathcal{Z}^d \mid S\vec{\alpha} = 0 \text{ and } \gcd(\alpha_1, \ldots, \alpha_d) = 1\}$$

Using the integer echelon reduction algorithm of [3, p32-39] to compute the unimodular matrix R such that RS^T is in echelon form (yielding $r = \text{rank}(S)$ as side-effect), all integer solutions of the homogeneous integer system $S\vec{\alpha} = \vec{0}$ are given by the following formula for arbitrary $\lambda_i \in \mathcal{Z}$ [3, p59-66]:

$$\vec{\alpha} = [\ (\underbrace{0, \ldots, 0}_{r}, \lambda_{r+1}, \ldots, \lambda_d)R\ ]^T \tag{1}$$

This observation provides a simple condition for the existence of a (possibly invalid) unimodular transformation that performs the preferred reshaping:

Proposition 3.2 *There exists a $d \times d$ unimodular matrix U for which the last column $\vec{\alpha} \in \mathcal{Z}^d$ of U^{-1} satisfies $S\vec{\alpha} = \vec{0}$ for some $n \times d$ matrix S if and only if rank$(S) < d$.*

If $r = d$, then the reshaping method fails. If, on the other hand, $r < d$ holds, then the last $d \Leftrightarrow r$ rows of R form a basis of the linear subspace consisting of all solutions of the homogeneous system.

We now have to choose a vector $\vec{\alpha} \in \Delta$ for constructing a unimodular matrix U^{-1} having $\vec{\alpha}$ as its last column.

It is not immediately obvious which $\vec{\alpha}$ to choose: there may be infinitely many solutions with components that are relatively prime, and we want to find one such that eventually we can construct a corresponding valid transformation. We have found that for practical cases, it is sufficient to restrict our attention to the set of basis vectors:

$$\Delta' = \{\vec{\alpha} = [(\underbrace{0, \ldots, 0}_{k-1}, 1, \underbrace{0, \ldots, 0}_{d-k})R\ ]^T \mid r < k \leq d\}$$

If none of these can be used to construct a valid transformation (see the next section), the reshaping method fails.

However, there remain cases in which this heuristic fails because access patterns can only be reshaped using an $\alpha \in \Delta \Leftrightarrow \Delta'$.

3.2.2 Construction of Valid Transformation

For any $\vec{\alpha} \in \Delta'$, we can use a completion method [3][5, p15-19] to construct a $d \times d$ unimodular matrix U for which the last column of U^{-1} consists of this preferred iteration direction.

Given these unimodular matrices, we can exploit the fact that for any $(d \Leftrightarrow 1) \times (d \Leftrightarrow 1)$ unimodular matrix Y and integer $z \in \{\Leftrightarrow 1, +1\}$, the following V is still a unimodular matrix for which the last column of the inverse is $\pm\vec{\alpha}$:

$$\begin{array}{rcl} V & = & \begin{pmatrix} & & & 0 \\ & Y & & \vdots \\ & & & 0 \\ 0 & \ldots & 0 & z \end{pmatrix} U \\ V^{-1} & = & U^{-1} \begin{pmatrix} & & & 0 \\ & Y^{-1} & & \vdots \\ & & & 0 \\ 0 & \ldots & 0 & z \end{pmatrix} \end{array} \tag{2}$$

Consequently, if for a given $\vec{\alpha} \in \Delta'$ and corresponding U we can construct a unimodular matrix Y and integer $z \in \{\Leftrightarrow 1, +1\}$ that define a matrix V with $V\vec{d} \succeq \vec{0}$ for all $\vec{d} \in \mathcal{D}$, then a valid transformation performing the preferred reshaping has been found.

Otherwise, another $\vec{\alpha} \in \Delta'$ is tried until either this construction is successful, or the set Δ' has been exhausted. In the latter case, the reshaping method fails.

For $\mathcal{D} = \emptyset$, the construction is trivial, since we can use the loop transformation defined by $V = U$ (viz. $Y = I$ and $z = 1$). Otherwise, we define the following set $\tilde{\mathcal{D}} \subseteq \mathcal{D}$:

$$\tilde{\mathcal{D}} = \{\vec{d} \in \mathcal{D} \mid U\vec{d} = (\underbrace{0,\ldots,0}_{d-1},\lambda)^T, \lambda \in \mathcal{Z}\}$$

The set of dependence distance vectors $\mathcal{D}$ can be partitioned into $\tilde{\mathcal{D}}$ and $\mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$. We will see that dependence distance vectors in the former set determine the selection of the integer $z \in \{\Leftrightarrow 1, +1\}$, whereas dependence distance vectors in the latter set must be dealt with during the construction of the matrix Y.

Lemma 3.1 *Given a $d \times d$ unimodular matrix U and a set $\tilde{\mathcal{D}} \subseteq \mathcal{Z}^d$ where each $\vec{d} \in \tilde{\mathcal{D}}$ satisfies $\vec{d} \succeq \vec{0}$ and $U\vec{d} = (0,\ldots,0,\lambda)^T$ for some $\lambda \in \mathcal{Z}$, then either:*

1. *for all $\vec{d} \in \tilde{\mathcal{D}}$ we have $U\vec{d} \succeq \vec{0}$, or*
2. *for all $\vec{d} \in \tilde{\mathcal{D}}$ we have $U\vec{d} \preceq \vec{0}$.*

Proof Assume that there are $\vec{d}_1, \vec{d}_2 \in \tilde{\mathcal{D}}$ such that

- $U\vec{d}_1 = (0,\ldots,0,\lambda_1)^T$ for $\lambda_1 > 0$, and
- $U\vec{d}_2 = (0,\ldots,0,\lambda_2)^T$ for $\lambda_2 < 0$.

Obviously, $\vec{d}_i = U^{-1}(0,\ldots,0,\lambda_i)^T$ implies that both $\vec{d}_1 = \lambda_1 \cdot \vec{v}$ and $\vec{d}_2 = \lambda_2 \cdot \vec{v}$ hold for a fixed $\vec{v} \neq \vec{0}$, namely, for $\vec{v}$ equal to the last column of U^{-1}. This is in contradiction with the assumption that both $\vec{d}_1$ and $\vec{d}_2$ are lexicographically positive. Consequently, either:

1. for all $\vec{d} \in \tilde{\mathcal{D}}$, $U\vec{d} = (0,\ldots,0,\lambda)^T$ for some $\lambda \geq 0$, i.e., $U\vec{d} \succeq \vec{0}$, or
2. for all $\vec{d} \in \tilde{\mathcal{D}}$, $U\vec{d} = (0,\ldots,0,\lambda)^T$ for some $\lambda \leq 0$, i.e., $U\vec{d} \preceq \vec{0}$.

[]

This lemma provides a convenient method to select a suitable integer $z \in \{\Leftrightarrow 1, +1\}$.

Proposition 3.3 *Given a $d \times d$ unimodular matrix U and a set $\tilde{\mathcal{D}} \subseteq \mathcal{Z}^d$ where each $\vec{d} \in \tilde{\mathcal{D}}$ satisfies $\vec{d} \succeq \vec{0}$ and $U\vec{d} = (0,\ldots,0,\lambda)^T$ for some $\lambda \in \mathcal{Z}$, then for any $(d \Leftrightarrow 1) \times (d \Leftrightarrow 1)$ matrix Y there exists an integer $z \in \{\Leftrightarrow 1, +1\}$ such that (2) defines a matrix V with $V\vec{d} \succeq \vec{0}$ for all $\vec{d} \in \tilde{\mathcal{D}}$,*

Proof Since the first $d \Leftrightarrow 1$ components of $V\vec{d}$ are zero for any $\vec{d} \in \tilde{\mathcal{D}}$ and any matrix Y, the proposition follows directly from Lemma 3.1: We select $z = \Leftrightarrow 1$ if $U\vec{d} \prec \vec{0}$ for any $\vec{d} \in \tilde{\mathcal{D}}$, and $z = 1$ otherwise. []

Using Proposition 3.3, we have found a method for finding the required $z \in \{\Leftrightarrow 1, +1\}$. In order to construct the unimodular matrix Y, only the dependence distance vectors in $\mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$ need to be used. Let $\tilde{U}$ consist of the first $d \Leftrightarrow 1$ rows of the unimodular matrix U:

$$\tilde{U} = \begin{pmatrix} 1 & & & 0 \\ & \ddots & & \vdots \\ & & 1 & 0 \end{pmatrix} U$$

Because for any unimodular Y and $\vec{d} \in \mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$ the inequality $Y\tilde{U}\vec{d} \neq \vec{0}$ holds, we have to find a unimodular matrix Y that satisfies the following constraint for all $\vec{d} \in \mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$:

$$Y\tilde{U}\vec{d} \succ \vec{0} \tag{3}$$

Such a matrix Y does not always exist. For instance, for $\tilde{U} = (1, \Leftrightarrow 1)$ and a set of remaining dependence distance vectors $\mathcal{D} \Leftrightarrow \tilde{\mathcal{D}} = \{(1,0)^T, (0,1)^T\}$, neither $Y = (+1)$ nor $Y = (\Leftrightarrow 1)$ satisfies the objective.

We use the following property [3]:

Proposition 3.4 *Given a set $\mathcal{D}_U \subseteq \mathcal{Z}^{d-1}$ with $\vec{d} \succ \vec{0}$ for all $\vec{d} \in \mathcal{D}_U$, then there exists a $(d \Leftrightarrow 1) \times (d \Leftrightarrow 1)$ unimodular matrix F such that $F\vec{d} \succ_1 \vec{0}$ for all $\vec{d} \in \mathcal{D}_U$.*

Consequently, if there exists a unimodular matrix Y that satisfies equation (3) for all $\vec{d} \in \mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$, then this proposition implies that another unimodular matrix $\tilde{Y}$ exists such that the constraint $\tilde{Y}\tilde{U}\vec{d} \succ_1 \vec{0}$ holds for all $\vec{d} \in \mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$. Hence, we can safely focus on finding this latter matrix $\tilde{Y}$ directly.

First, we determine whether there is a vector $\vec{y} \in \mathcal{Z}^{d-1}$ with $\gcd(y_1, \ldots, y_{d-1}) = 1$ such that the inequality $\vec{y} \cdot \tilde{U}\vec{d} > 0$ holds for all $\vec{d} \in \mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$. This problem is equivalent to finding a suitable solution of the following system of inequalities, where the rows of the integer matrix M are formed of the vectors $\tilde{U}\vec{d}$ for all $\vec{d} \in \mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$:

$$\Leftrightarrow M \begin{pmatrix} y_1 \\ \vdots \\ y_{d-1} \end{pmatrix} \leq \begin{pmatrix} \Leftrightarrow 1 \\ \vdots \\ \Leftrightarrow 1 \end{pmatrix}$$

We use Fourier-Motzkin elimination to test the consistency of this system. The construction of Y fails if the system is inconsistent (e.g. we have $1 \leq y_1$ and $y_1 \leq \Leftrightarrow 1$ for the example above). If the system is consistent, however, any *rational* solution $\vec{y}^{\,r}$ (which can be obtained as side-effect of the elimination) can be scaled to an integer solution of which the components are relatively prime (viz. $\vec{y} = \lambda \cdot \vec{y}^{\,r}$ for λ equal to the least common multiple of the denominators).

Eventually, a matrix completion method as described in [3][5, p15-19] is used to construct a unimodular matrix $\tilde{Y}$ with $\vec{y} \in \mathcal{Z}^{d-1}$ as its first row. Obviously, $\tilde{Y}\tilde{U}\vec{d} \succ_1 \vec{0}$ holds for all $\vec{d} \in \mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$.

3.3 Summary

Summarizing, the following steps are applied. First, we construct the objective matrix S using the subscripts and preferred access directions. If $\text{rank}(S) = d$, then the reshaping method fails (proposition 3.2). Otherwise, for each preferred iteration direction $\vec{\alpha} \in \Delta'$, a corresponding $d \times d$ unimodular matrix U of which the last column of U^{-1} is equal to this direction is constructed and the following steps are applied until either the method succeeds or this set has been exhausted, in which case the reshaping method fails:

- Select $z = \Leftrightarrow 1$ if $U\vec{d} \prec \vec{0}$ for any $\vec{d} \in \tilde{\mathcal{D}}$ or select $z = 1$ otherwise (Proposition 3.3).
- Find a $\vec{y} \in \mathcal{Z}^{d-1}$ with $\gcd(y_1, \ldots, y_{d-1}) = 1$ such that $\vec{y} \cdot \tilde{U}\vec{d} > 0$ for all $\vec{d} \in \mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$.

If this vector $\vec{y} \in \mathcal{Z}^{d-1}$ exists, then a completion method [3][5, p15-19] is used to construct a unimodular matrix Y with this vector as first row.

The resulting matrix Y together with Y^{-1} and integer $z \in \{\Leftrightarrow 1, +1\}$ define matrices V and V^{-1} according to (2) such that $V\vec{d} \succeq \vec{0}$ holds for all $\vec{d} \in \mathcal{D}$. Hence, application of the loop transformation defined by this matrix is valid and reshapes the access patterns of each ith occurrence along the preferred direction $\vec{s}_i \in \mathcal{Z}^2$.

EXAMPLE: Consider the following triple loop:

```
DO I1 = 10, 15
  DO I2 = 1, 3
    DO I3 = 10, 15
      A(  I1 + 3*I2 + I3,   I1 + I3) = ...
      B(2*I1 +          I3, 2*I2     ) = ...
      C(  I1 - 3*I2      ,         I3) = ...
    ENDDO
  ENDDO
ENDDO
```

We assume that the data dependences of this loop are represented by the following set:

$$\mathcal{D} = \{(1,0,0)^T, (0,1,0)^T, (0,0,1)^T, (3,1,\Leftrightarrow 6)^T\}$$

Now, suppose that row-wise access patterns are desired for these three occurrences, i.e. $\vec{s}_i = (0,1)^T$ for $1 \leq i \leq 3$. Reshaping the access patterns accordingly seems to be a non-trivial task at first sight. However, the reshaping method proceeds as follows.

First, the objective matrix S is constructed:

$$S = \begin{pmatrix} 1 & 0 & & & & \\ & & 1 & 0 & & \\ & & & & 1 & 0 \end{pmatrix} \left(\begin{array}{ccc} 1 & 3 & 1 \\ 1 & 0 & 1 \\ \hline 2 & 0 & 1 \\ 0 & 2 & 0 \\ \hline 1 & \Leftrightarrow 3 & 0 \\ 0 & 0 & 1 \end{array} \right)$$

Echelon reduction of S^T yields the following form for $E = RS^T$:

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & \Leftrightarrow 1 \\ 3 & 1 & \Leftrightarrow 6 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 3 & 0 & \Leftrightarrow 3 \\ 1 & 1 & 0 \end{pmatrix}$$

Because $\text{rank}(S) = 2$, all integer solutions of the homogeneous system $S\vec{\alpha} = \vec{0}$ are given by $\vec{\alpha} = [(0,0,\lambda_3)R]^T$ for arbitrary $\lambda_3 \in \mathcal{Z}$. Hence, we have $\Delta' = \{(3,1,\Leftrightarrow 6)^T\}$. The following matrices are constructed with a completion method as described in [3][5, p15-19]:

$$U = \begin{pmatrix} \Leftrightarrow 1 & 3 & 0 \\ 0 & 6 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad U^{-1} = \begin{pmatrix} \Leftrightarrow 1 & 0 & 3 \\ 0 & 0 & 1 \\ 0 & 1 & \Leftrightarrow 6 \end{pmatrix}$$

Let $\tilde{U}$ denote the matrix consisting of the first 2 rows of U. Then $\tilde{\mathcal{D}} = \{\vec{d} \in \mathcal{D} | \tilde{U}\vec{d} = \vec{0}\}$ is equal to $\{(3, 1, \Leftrightarrow 6)^T\}$. Since $U(3, 1, \Leftrightarrow 6)^T = (0, 0, 1)^T$, we select $z = 1$.

Finding an integer vector $\vec{y} \in \mathcal{Z}^2$ such that $\vec{y} \cdot \tilde{U}\vec{d} > 0$ for all $\vec{d} \in \mathcal{D} \Leftrightarrow \tilde{\mathcal{D}}$ is equivalent to solving the following system of linear inequalities:

$$\begin{pmatrix} \Leftrightarrow 1 & 0 \\ 3 & \Leftrightarrow 6 \\ 0 & \Leftrightarrow 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \leq \begin{pmatrix} \Leftrightarrow 1 \\ \Leftrightarrow 1 \\ \Leftrightarrow 1 \end{pmatrix}$$

Obviously, this system is consistent, and a solution $(1, 1)^T$ can be used directly for $\vec{y}$. The following matrices results:

$$Y = \begin{pmatrix} 1 & 1 \\ \Leftrightarrow 1 & 0 \end{pmatrix} \qquad Y^{-1} = \begin{pmatrix} 0 & \Leftrightarrow 1 \\ 1 & 1 \end{pmatrix}$$

These matrices and $z = 1$ give rise to the following V and V^{-1}:

$$V = \begin{pmatrix} 1 & 3 & 1 \\ \Leftrightarrow 1 & 3 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad V^{-1} = \begin{pmatrix} 0 & \Leftrightarrow 1 & 3 \\ 0 & 0 & 1 \\ 1 & 1 & \Leftrightarrow 6 \end{pmatrix}$$

Application of the loop transformation defined by V yields the following target loop in which row-wise access patterns result for *all* occurrences:

```
 DO I1' = 23, 39

   DO I2' = MAX(-12,I1'-15,16-I1'), MIN(-1,I1'-30,33-I1')

     DO I3' = MAX(⌈(I1'+I2'-15)/6⌉,⌈(I2'+10)/3⌉),
+             MIN(⌊(I1'+I2'-10)/6⌋,⌊(I2'+15)/3⌋)

       A(I1'      , I1'      - 3*I3') = ...
       B(I1' - I2',            2*I3') = ...
       C(    - I2', I1' + I2' - 6*I3') = ...
     ENDDO
   ENDDO
 ENDDO
```

4 Improving Data Locality

Because Fortran uses column-major storage of arrays, accessing arrays along columns enhances the spatial locality of a program. Hence, a straightforward approach to improve the performance of a program is to apply the reshaping method to each perfectly nested loop in a Fortran program, where we set the preferred access direction of each multi-dimensional array to $(1, 0, \ldots, 0)^T$. If the reshaping method fails for a particular loop nest, either because the reshaping is impossible or because data dependences prohibit the reshaping, then we can simply continue with the next perfectly nested loop in the program, or we can reapply the reshaping method to the same loop with a smaller set of occurrences, for example, by discarding an occurrence with unique subscripts before occurrences having identical subscripts. If the reshaping method still fails, another occurrence is discarded until reshaping is successful or all occurrences have been discarded.

Below, we present some experiments that have been conducted on an SGI Indy 4600. All fragments are compiled by the native FORTRAN compiler using the flag `-O2`. Although in the experiments the value of `N` varies, the actual shape of each array is kept constant (and this shape envelops the parts that may be accessed for all values).

EXAMPLE: Consider the following implementation of the operation $C \leftarrow C + AB$.

```
DO I1 = 1, N
  DO I2 = 1, N
    DO I3 = 1, N
      C(I1,I2) = C(I1,I2) + A(I1,I3) * B(I3,I2)
    ENDDO
  ENDDO
ENDDO
```

First, we construct the objective matrix S using the preferred access direction $\vec{s} = (1, 0)^T$ for all occurrences:

$$\begin{pmatrix} 0 & 1 & & & & & & \\ & & 0 & 1 & & & & \\ & & & & 0 & 1 & & \\ & & & & & & 0 & 1 \end{pmatrix} \begin{pmatrix} W_{\mathtt{C}} \\ W_{\mathtt{C}} \\ W_{\mathtt{A}} \\ W_{\mathtt{B}} \end{pmatrix}$$

Hence, the objective matrix has the following form:

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

Echelon reduction of S^T yields the following form for $E = RS^T$:

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

Because $\text{rank}(S) = 2$, we obtain the set $\Delta' = \{(1,0,0)^T\}$. Under the assumption that the data dependences due to the accumulation of elements in array `C` may be ignored, the following matrices can be used directly to apply the appropriate reshaping:

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

This transformation gives rise to a simple loop permutation where $\mathtt{I}_1$ becomes the index of the outermost DO-loop. In figure 3, we show the execution times of the original and reshaped version for varying values of `N`.

EXAMPLE: Consider the following triple loop in which some three-dimensional arrays and some two-dimensional arrays are referenced:

```
DO I1 = 1, N
  DO I2 = 1, N
    DO I3 = 1, N
      C(I1,I2,I3) = B(I1,I3,I2) - D(I3-I1,I3) * A(I1,I1)
    ENDDO
  ENDDO
ENDDO
```

The objective matrix S is constructed as illustrated below:

$$\begin{pmatrix} 0 & 1 & 0 & & & & & & & \\ 0 & 0 & 1 & & & & & & & \\ & & & 0 & 1 & 0 & & & & \\ & & & 0 & 0 & 1 & & & & \\ & & & & & & 0 & 1 & & \\ & & & & & & & & 0 & 1 \end{pmatrix} \begin{pmatrix} W_{\mathtt{C}} \\ W_{\mathtt{B}} \\ W_{\mathtt{D}} \\ W_{\mathtt{A}} \end{pmatrix}$$

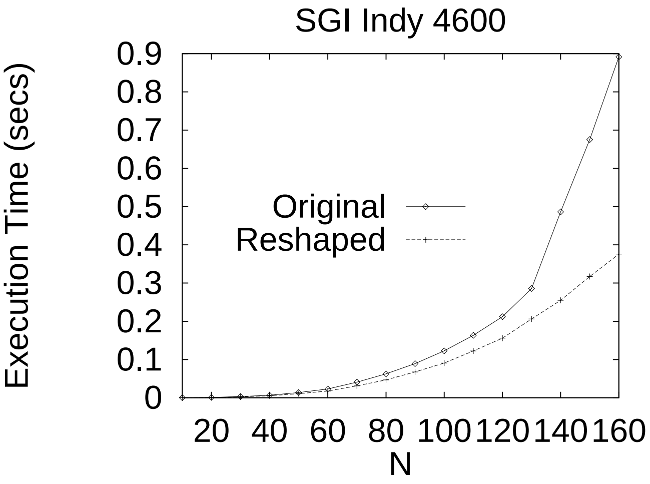

Figure 3: Computation of $C \leftarrow C + AB$

Hence, the resulting objective matrix has the following form:

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

Echelon reduction of S^T, however, reveals that $\text{rank}(S) = 3$, indicating that making all access patterns column-wise is impossible. However, after we discard the occurrence of `A` from consideration (which is simply implemented by discarding the last row of S), then the $\text{rank}(S)$ becomes 2 and the reshaping method can be applied successfully. In this case, the same loop permutation as in the previous example results, making all except the access patterns of `A` column-wise. In figure 4, we show the performance improvement obtained by this transformation.

Finally, we briefly sketch an extension to our approach that is given by *iterating* the construction. First observe that, in case of column-wise reshaping, all reshaped access matrices become of the form, where $w_{1d} \neq 0$:

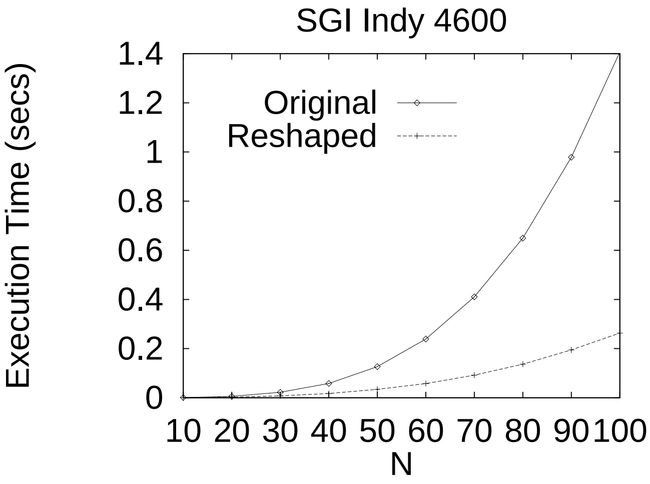

Figure 4: Performance of first Triple Loop

$$W = \begin{pmatrix} w_{11} & \dots & w_{1,d-1} & w_{1d} \\ w_{21} & & w_{2,d-1} & 0 \\ \vdots & \ddots & & \vdots \\ w_{c1} & & w_{c,d-1} & 0 \end{pmatrix}$$

or:

$$W = \begin{pmatrix} w_{11} & \dots & w_{1,d-1} & 0 \\ w_{21} & & w_{2,d-1} & 0 \\ \vdots & \ddots & & \vdots \\ w_{c1} & & w_{c,d-1} & 0 \end{pmatrix}$$

Ideally, in the later case we would like to make $w_{2,d-1} = 0, \dots, w_{c,d-1} = 0$ as well. In order to obtain this form we re-apply the reshaping method to all occurrences for which W has a zero last column. First, we construct a $(d \Leftrightarrow 1) \times (d \Leftrightarrow 1)$ unimodular transformation V using the method from section 3 with as input the *reduced* access matrices of these occurrences, formed by dropping the last column from the original access matrices. Subsequently, this loop transformation is applied to the $d \Leftrightarrow 1$ outermost loops. Likewise, we construct a $(d \Leftrightarrow 2) \times (d \Leftrightarrow 2)$ unimodular transformation V' for handling the next iterator, and so on. It is easy to see that in this way we can obtain an 'ideal' access matrix, as described above.

EXAMPLE: Consider the following triple loop:

```
DO I1 = 1, N
  DO I2 = 1, N
    DO I3 = 1, N
       A(I1+I2,I2,I1-I3) = B(I3,I2) * C(I1-I3,I2)
    ENDDO
  ENDDO
ENDDO
```

Making the access patterns of all these occurrences column-wise gives rise to the following objective matrix:

$$S = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & \Leftrightarrow 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Echelon reduction of S^T yields the following form for $E = RS^T$:

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

Because $\text{rank}(S) = 2$, we obtain the set $\Delta' = \{(1,0,1)^T\}$. Since there are no data dependences in the loop, the following matrices can be used directly to apply the reshaping:

$$U = \begin{pmatrix} 1 & 0 & \Leftrightarrow 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad U^{-1} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ \Leftrightarrow 1 & 0 & 1 \end{pmatrix}$$

Applying the loop transformation defined by these matrices results in the fragment shown below:

```
DO I1' = 1-N, N-1
  DO I2' = 1, N
    DO I3' = MAX(I1'+1,1), MIN(I1'+N,N)
       A(I2'+I3',I2',I1') = B(I3'-I1',I2') * C(I1',I2')
    ENDDO
  ENDDO
ENDDO
```

The resulting access matrices reveal that column-wise access patterns have been obtained for both the occurrences of `A` and `B`:

$$W_{\mathtt{A}} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad W_{\mathtt{B}} = \begin{pmatrix} \Leftrightarrow 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

The access patterns of C, on the other hand, have become scalar-wise. In fact, this occurrence is accessed along rows in the outermost two loops:

$$W_{\mathtt{C}} = \left(\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array} \right)$$

Applying the reshaping method to the reduced access matrix gives rise to the following objective matrix:

$$S = \left(\begin{array}{cc} 0 & 1 \end{array} \right) \left(\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right) = \left(\begin{array}{cc} 0 & 1 \end{array} \right)$$

Hence, we obtain $\Delta' = \{(1,0)^T\}$, which gives rise to a simple loop interchanging of the outermost two DO-loops:

$$U = \left(\begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array} \right) \quad U^{-1} = \left(\begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array} \right)$$

In figure 5, we show the execution time of the original, first reshaped, and final reshaped fragment for varying values of N. This experiment clearly shows how an extra iteration of reshaping can be used to further improve the performance of a loop. Moreover, the experiment shows that the overhead associated with evaluating MIN and MAX functions is outweighted by the improvements obtained with respect to the memory hierarchy.

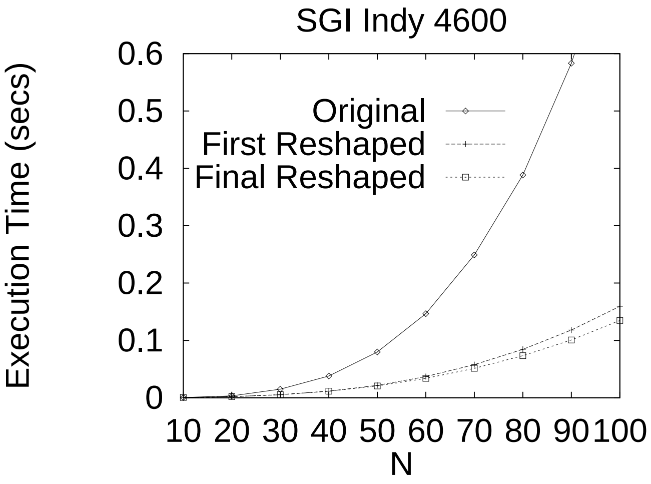

Figure 5: Performance of second Triple Loop

5 Conclusions

In this paper, we have presented the construction of a valid transformation of a perfectly nested loop that simultaneously reshapes the access patterns of several occurrences of multi-dimensional arrays along certain desired access directions. In particular, we have shown how this reshaping method can be used to improve data locality in programs. If for a particular loop nest the reshaping method fails, the compiler re-tries the reshaping method with a smaller collection of occurrences. Moreover, we have shown that in some cases, iterating the reshaping method can further improve performance.

Note that in this paper we have focused on improving *spatial* locality by making loops refer to neighboring elements in successive iterations. Obviously, this method can be combined with methods that are aimed at enhancing temporal locality, like loop blocking. Moreover, future research will be aimed at combining the reshaping method of this paper with the framework of *data structure* transformations presented in [16].

References

[1] J.M. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pages 112–125, 1993.

[2] B. Appelbe, S. Doddapaneni, and C. Hardnett. A new algorithm for global optimization for parallelism and locality. In *Proc. 7th Ann. Workshop on Languages and Compilers for Parallel Computing*, 1994.

[3] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Boston, 1993.

[4] U. Banerjee. *Loop Parallelization.* Kluwer Academic Publishers, Boston, 1994.

[5] A.J.C. Bik. *Compiler Support for Sparse Matrix Computations.* PhD thesis, Leiden University, 1996.

[6] A.J.C. Bik, P.M.W. Knijnenburg, and H.A.G. Wijshoff. Reshaping access patterns for generating sparse codes. In *Proc. 7th Ann. Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, pages 406–422. Springer Verlag, Berlin, 1995.

[7] F. Bodin, C. Eisenbeis, W. Jalby, T. Montaut, P. Rabain, and D. Windheiser. Algorithms for data locality optimizations. APPARC Deliverable CoD2, 1994.

[8] S. Carr, K.S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proc. 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1994.

[9] M.L. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16:157–171, 1990.

[10] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *J. Parallel and Distributed Computing*, 5:587–616, 1988.

[11] K. Kennedy and K.S. McKinley. Optimizing for parallelism and data locality. In *Proc. ICS*, 1992.

[12] L. Lamport. The parallel execution of DO loops. *Comm. of the ACM*, 17(2):83–93, 1974.

[13] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. In *Proc. 5th In'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 285–295, 1992.

[14] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Int'l J. of Parallel Programming*, 22(2):183–205, 1994.

[15] J.J. Navarro, A. Juan, and T. Lang. MOB forms: A class of multilevel block algorithms for dense linear algebra operations. In *Proc. ICS*, 1994.

[16] M.F.P O'Boyle and P.M.W. Knijnenburg. Nonsingular data transformations: Definition, validity, applications. In *Proc. 6th Workshop on Compilers for Parallel Computers*, 1996.

[17] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Comm. of the ACM*, 29(12):1184–1201, 1986.

[18] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Comm. of the ACM*, 8:102–114, 1992.

[19] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. on Programming Languages Design and Implementation*, pages 30–44, 1991.

[20] M.J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[21] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.