Transforming Complex Loop Nests For Locality

Qing Yi  Ken Kennedy

Computer Science Department
Rice University

Abstract

Because of the increasing gap between the speeds of processors and standard memory chips, many compiler techniques have been developed to enhance locality of applications. This paper focuses on automatically optimizing complicated loop structures, for which existing techniques are either ineffective or require too much computation time to be practical for a commercial compiler. Building on traditional unimodular transformations on perfectly nested loops, we have developed a novel transformation called dependence hoisting. This transformation facilitates fusion of a set of arbitrarily nested loops at the outermost position of a code segment containing these loops. This transformation is especially useful when the loops to be fused are nested inside one another and when some loops cannot be legally distributed before fusion. We have also developed a transformation framework called computation slicing, which applies dependence hoisting to block arbitrary loop nests for better locality. In terms of both asymptotic complexity, which is comparable to that of standard unimodular loop transformations, and actual running time, as measured in our experimental results, computation slicing should be efficient enough for inclusion in commercial production compilers. We have implemented the framework as a Fortran source-to-source translator. Our implementation has successfully blocked four numerical benchmark kernels: Cholesky, QR, LU factorization without pivoting, and LU factorization with partial pivoting. The automatically-blocked benchmarks achieved performance improvements similar to those attained by manually blocked programs in LAPACK [3]. The automatic blocking of QR and LU with partial pivoting is a notable achievement because these benchmarks include loop nests that are considered difficult — to our knowledge, no previous compiler implementation has completely automated the blocking of QR and LU with pivoting. This fact indicates that our technique can in practice match or exceed the effectiveness of many general loop transformation frameworks.

1 Introduction

Over the past twenty years, increases in processor speed have dramatically outstripped performance increases for standard memory chips, in both latency and bandwidth. To bridge this gap, architects have inserted one or more levels of cache between the processor and memory, resulting in a memory hierarchy. The data access latency to a higher level of the memory hierarchy is often orders of magnitude less than the latency to a lower level. To achieve high performance on such machines, compilers must optimize the locality of applications so that data fetched into caches are reused before being displaced.

Blocking (or tiling) is a highly effective compiler strategy that enhances locality of applications by partitioning computations into smaller blocks. A set of unimodular loop transformations, such as loop strip-mining, reversal, skewing [23], fusion, distribution, interchange, and index-set splitting [6, 17], can be combined to automatically block computations. These transformations are inexpensive and can effectively optimize simple loop structures such as perfectly nested loops. However, on more complicated loop structures, these transformations often fail, even when an effective blocking is possible.

This paper extends previous unimodular loop transformations to block complicated loop nests. We introduce a new transformation called dependence hoisting, which facilitates fusion of a set of arbitrarily nested loops
at the outermost position of a code segment containing these loops. The complexity of applying dependence hoisting to a loop nest is $O(N^2 + NLD)$, where $N$ is the number of statements in the nest, $L$ is the maximum nesting depth, and $D$ is the size of the dependence graph. We also develop a transformation framework called computation slicing which blocks the loop nest by combining loop strip-mining with dependence hoisting. This blocking strategy requires time proportional to $N^2L + NL^2D$, which we believe is fast enough for incorporation in most commercial compilers.

We have implemented the framework in a Fortran source-to-source translator. Although it has broad applicability to general, non-perfectly nested loops, we have focused our experiments on four numerical benchmark kernels—Cholesky, QR, LU factorization without pivoting, and LU factorization with partial pivoting—because they are generally considered difficult to block automatically. To our knowledge, no previous compiler implementation has completely automated the blocking of QR and LU with pivoting. Our translator successfully blocked all four benchmarks, achieving significant performance improvements over the unblocked versions. The automatically blocked versions also achieve a performance level comparable to that of the versions in LAPACK which were hand blocked by professionals using algorithm change in some cases. This fact indicates that our technique is quite powerful and can match or exceed the real-world effectiveness (although not the theoretical effectiveness) of many general transformation frameworks [14, 19, 1, 16].

To illustrate how computation slicing extends previous transformation techniques on perfectly nested loops, we use three equivalent versions of LU factorization without pivoting, as shown in Figure 1. Dongarra, Gustavson and Karp [9] described various implementations of non-pivoting LU with different loop orderings, from which we selected the KIJ, KJI and KJI loop orderings. The KJI and KJI forms in Figure 1(a) and (b) are commonly used in scientific applications. The JKI form in Figure 1(c) is a less commonly used deferred-update version which defers all the updates to each column of the matrix until immediately before the scaling of that column. Specifically, at each iteration of the outermost $j$ loop, statement $s_2$ first applies all the deferred updates to column $j$ by subtracting multiples of columns 1 through $j - 1$; statement $s_1$ then scales column $j$ immediately after these deferred updates. This JKI form is particularly useful for blocking non-pivoting LU.

The three code fragments in Figure 1 are not only equivalent, they also contain the same dependence constraints. Thus, an optimizing compiler should in theory be able to recognize this equivalence and freely translate between any pair of these versions. For example, to translate Figure 1(a) to Figure 1(b), a compiler can distribute the $i$ loop surrounding both $s_1$ and $s_2$ in (a) and then interchange the distributed $i(s_2)$ loop ($i$ loop surrounding $s_2$) with the $j(s_2)$ loop. A compiler can also easily reverse this procedure and translate (b) back to (a).

However, traditional unimodular loop transformations cannot translate between Figure 1(b) or (a) and Figure 1(c). For example, to translate from (b) to (c), a compiler needs to distribute the outermost $k$ loop in (b), interchange the distributed $k(s_2)$ loop ($k$ loop surrounding $s_2$) with the $j(s_2)$ loop, and then fuse the distributed $k(s_1)$ loop with the interchanged $j(s_2)$ loop. To automatically perform this sequence of transformations, a compiler must be able to recognize the $k(s_1)$ loop and the $j(s_2)$ loop can be legally fused at the outermost loop level. The compiler must also find a way to distribute the outermost $k$ loop in (b) in order to fuse the distributed $k(s_1)$ loop with the $j(s_2)$ loop. However, since the outermost $k$ loop in (b) carries dependence cycles connecting $s_1$ and $s_2$, the traditional unimodular loop transformation techniques cannot distribute this loop and thus cannot establish the translation from (b) to (c).

To accomplish the translation between Figure 1(b) or (a) and Figure 1(c), we extend traditional unimodular
loop transformations with transitive dependence information between arbitrary loops independent of their original nesting structure. We can then automatically recognize the legality of fusing the \( k(s_1) \) and \( j(s_2) \) loops in (b) from the transitive dependences between these loops. To distribute the \( k(s_1, s_2) \) loop in (b), we use a novel transformation called dependence hoisting. This transformation first puts a new outermost dummy loop (with induction variable \( x \)) surrounding the original \( k \) loop in (b) and synchronizes the iterations of loops \( j(s_2) \) and \( k(s_1) \) with this new dummy loop. In particular, it inserts conditionals to execute the iteration \( j \) of loop \( j(s_2) \) when \( x = j \) and iteration \( k \) of loop \( k(s_1) \) when \( x = k \). These conditionals shift the loop-carrying levels of the dependence cycles connecting \( s_1 \) and \( s_2 \) to this new outermost \( x \) loop. Because the original \( k \) loop is now inside the \( x \) loop and no longer carries the dependence cycles connecting \( s_1 \) and \( s_2 \), we can distribute this \( k \) loop and translate (b) to (c). Section 2 describes this translation in detail.

The computation slicing framework in this paper is similar to a technique called iteration space slicing proposed by Pugh and Rosser [22]. They also used transitive dependence information to fuse related iterations of statements. Their approach manipulates the iteration spaces or data spaces of statements to summarize related computations; our approach focuses on transforming loops directly and does not manipulate any symbolic spaces of statements. Although our approach is less general than the approach by Pugh and Rosser, it is powerful enough for a large class of real-world applications and is much more efficient.

Besides iteration space slicing, several other general loop transformation frameworks [14, 19, 1, 16] can also aggressively transform complicated loop structures. These frameworks typically adopt a mathematical formulation of program dependence and loop transformations. They first map the iteration spaces of statements into some unified space, such as the time space or data space of a program. The unified space is then considered for transformation. Finally, a new program is constructed by mapping the selected transformations of the unified space onto the iteration spaces of statements. These frameworks are powerful because they incorporate the whole solution space of program transformations. However, they are also complicated and expensive to apply. Because of their high cost, these frameworks are rarely used in commercial compilers.

Our strategy is a compromise that trades a small amount of generality for substantial gains of efficiency. Thus, we do not manipulate any mathematically formulated space other than the iteration space of loops and the dependences within that space. Our technique can be integrated into traditional unimodular transformation systems and, because it is inexpensive, it should be suitable for inclusion in commercial production compilers.

The remainder of this paper is organized as follows. Section 2 first illustrates the dependence hoisting transformation by translating the KJI form of non-pivoting LU in Figure 1(b) to the JKI form in (c). Section 3 then presents some notations and definitions for further elaboration of the transformation framework. Section 4 provides an overview of the computation slicing framework and again uses the KJI form of non-pivoting LU as example to further illustrate the framework. Section 5 presents the transformation algorithms for the computation slicing framework. Section 6 presents experimental results. Section 7 introduces related work, and conclusions are drawn in Section 8.

## 2 Example: Translating Non-pivoting LU

This section illustrates how to translate the KJI form of non-pivoting LU in Figure 1(b) into the JKI form in (c). As discussed in Section 1, this translation requires fusing the \( k(s_1) \) and \( j(s_2) \) loops in (b) at the outermost loop level. To facilitate this fusion, the \( k(s_1, s_2) \) loop needs to be distributed. However, due to the dependence cycle connecting \( s_1 \) and \( s_2 \), the distribution is not legal in the original code.

We introduce a novel transformation called dependence hoisting to resolve this conflict. A dependence hoisting transformation permits fusion of arbitrarily nested loops at the outermost position of some code segment containing these loops. The transformation is particularly useful when the loops to be fused are nested inside one another and when some loops cannot be legally distributed before fusion, as is the case of fusing the \( k(s_1) \) and \( j(s_2) \) loops in Figure 1(b).

Dependence hoisting translates Figure 1(b) to (c) in three steps. First, it creates a new dummy loop (with induction variable \( x \)) surrounding the original code in (b) and then inserts conditionals in (b) so that statement \( s_1 \) is executed only when \( x = j \) and \( s_2 \) is executed only when \( x = k \). Figure 2(b) shows the result of this transformation step. Here because the conditionals synchronize the \( k(s_1) \) and \( j(s_2) \) loops with the new \( x(s_1, s_2) \) loop in a lock-step fashion, the loop-carrying level of the dependence cycle connecting \( s_1 \) and \( s_2 \) is shifted to the new outermost \( x \) loop. The \( k(s_1, s_2) \) loop can now be distributed because it no longer carries any dependence
cycle. The transformed code after distribution is shown in Figure 3(a). Finally, dependence hoisting removes all the redundant conditionals and loops. The transformed code after this cleanup step is shown in Figure 3(b).

We now discuss each of the translation steps in more detail. Figure 2(a) shows the original code and its dependence graph. Here, because loop fusion and distribution are affected only by dependence edges between $s_1$ and $s_2$, only these edges are marked with dependence conditions. Note that these conditions include relations between not only common loops surrounding $s_1$ and $s_2$, but also non-common loops such as the $j(s_2)$ loop. The extra information is necessary to precisely model dependences independent of the original loop structure. Section 3.2 describes in detail our new dependence representation.

Figure 2(b) shows the transformed code from (a) after the first step of translation. Here the new outermost loop "$x = 1, n$" unifies the iteration ranges of both loops $j(s_2)$ and $k(s_1)$. The two conditionals "$x = k$" and "$x = j$" synchronize loops $k(s_1)$ and $j(s_2)$ with loop $x(s_1, s_2)$ so that $s_1$ is executed only when $x = k$ and $s_2$ is executed only when $x = j$. Because of these forced execution conditions, loop $x(s_1)$ always has the same dependence edges as those of loop $k(s_1)$. Similarly, loop $x(s_2)$ always has the same dependence conditions as those of loop $j(s_2)$. We can then modify the dependence graph in Figure 2(a) accordingly. Figure 2(b) shows the modified dependence graph, where dependence conditions for the new outermost $x$ loop are added to the dependence edges between $s_1$ and $s_2$. Because it originally has the dependence condition $k(s_1) < j(s_2)$, the dependence edge $d(s_1, s_2)$ (from $s_1$ to $s_2$) now includes the condition $x(s_1) < x(s_2)$. Similarly, the dependence edge $d(s_2, s_1)$ now includes condition $x(s_2) = x(s_1)$. All the dependence edges still maintain the same directions, indicating that the transformation is legal. However, the dependence edge $d(s_1, s_2)$ (from $s_1$ to $s_2$) is now carried by the outermost $x$ loop. Thus we have shifted the loop level of the original dependence cycle connecting $s_1$ and $s_2$ to this new $x$ loop.

From the dependence graph in Figure 2(b), we can now distribute the $k(s_1, s_2)$ loop. Because the dependence edge $d(s_1, s_2)$ is now carried by the outermost $x$ loop, we can ignore this dependence when distributing loop $k(s_1, s_2)$. After loop distribution, statement $s_2$ is put before $s_1$. Figure 3(a) shows the transformed code.

In Figure 3(a), both loops $k(s_1)$ and $j(s_2)$ now surround only a single statement and a conditional that synchronizes them with the outermost $x$ loop. These loops have thus become redundant and can be removed together with the conditionals inside them. Before removing the $k(s_1)$ loop and the conditional "$x = k$", the loop index variable $k$ must be replaced with $x$ in statement $s_1$. Before removing the $j(s_2)$ loop and the conditional "$x = j$", the loop index variable $j$ must be replaced with $x$ in statement $s_2$. In addition, the upper bound for the $k(s_2)$ loop must be adjusted to $x - 1$, in effect because the $j(s_2)$ loop is exchanged outward before the substitution. Figure 3(b) shows the final transformed code and modified dependence graph, where the dependence edges in (a) are modified so that all the dependence conditions for the removed loops are also removed.

The final transformed code in Figure 3(b) is the same as the $JKI$ form of non-pivoting LU in Figure 1(c) except that the name of the outermost loop index variable is $x$ instead of $j$. In reality, the index variables of the new loops can often reuse those of the removed loops so that a compiler does not have to create a new loop index variable at each dependence hoisting transformation. This new loop nest can then be blocked using a process.
do $x = 1, n$
do $k = 1, n - 1$
do $i = k + 1, n$
if ($j = x$) then
s2: $a(i, j) = a(i, j) - a(i, k) * a(k, j)$
endif
endo
do $k = 1, n - 1$
do $i = k + 1, n$
if ($k = x$) then
s1: $a(i, k) = a(i, k) / a(k, k)$
endif
endo
do $x = 1, n$
do $k = 1, x - 1$
do $i = k + 1, n$
s2: $a(i, x) = a(i, x) - a(i, k) * a(k, x)$
endo
s1: $a(i, x) = a(i, x) / a(x, x)$
endo
endo
endo

(a) 

Figure 3: Translation steps (2) and (3): distribute loops and cleanup

discussed in Section 4.

3 Notations and Definitions

In this section, we present notations and definitions for further elaboration of our transformation algorithms. In particular, we present our loop transformation model and dependence representations.

3.1 Loop Transformation Model

To transform loops without being limited by their original nesting structure, we adopt a loop model similar to that by Ahmed, Mateev and Pingali [1]. We use the notation $\ell(s)$ to denote a loop $\ell$ surrounding some statement $s$. This notation enables us to treat loop $\ell(s)$ as different from loop $\ell(s')$ ($s \neq s'$) and thus to treat each loop $\ell$ surrounding multiple statements as potentially distributable. Dependence analysis will later inform us whether or not each loop can be distributed.

Each statement $s$ inside a loop is executed multiple times; each execution is called an iteration instance of $s$. Suppose that statement $s$ is surrounded by $m$ loops $\ell_1, \ell_2, \ldots, \ell_m$. For each loop $\ell_i(s)$ ($i = 1, \ldots, m$), we denote its iteration index variable as $Ivar(\ell_i(s))$ and its iteration range as $Range(\ell_i(s))$. Each value $I$ of $Ivar(\ell_i(s))$ defines a set of iteration instances of $s$, a set that can be expressed as $Range(\ell_1) \times \ldots \times Range(\ell_{i-1}) \times I \times \ldots \times Range(\ell_m)$. We denote this iteration set as iteration $I:\ell_i(s)$ or the iteration $I$ of loop $\ell_i(s)$.

In order to focus on transforming loops, we treat all the other control structures in a program as primitive statements that is, we do not transform loops inside other control structures such as conditional branches. We adopt this strategy to simplify the technical presentation of this paper. To optimize real-world applications, preparative transformations such as “if conversion” [2] must be incorporated to remove the non-loop control structures in between loops. These preparative transformations are out of the scope of this paper and will not be discussed further.

In this paper, we present loop transformations only at the outermost loop level of any code segment. To apply transformations at deeper loop levels, we hierarchically consider the code segment at each loop level and apply the same transformation algorithms at the outermost loop level of that code segment, guaranteeing that no generality is sacrificed.

3.2 Dependences and Transitive Dependences

We use the notation $d(s_x, s_y)$ to denote the set of dependence edges from some statement $s_x$ to $s_y$ in the dependence graph. Suppose that $s_x$ and $s_y$ are surrounded by $m_x$ loops $(\ell_{x_1}, \ell_{x_2}, \ldots, \ell_{x_m})$ and $m_y$ loops $(\ell_{y_1}, \ell_{y_2}, \ldots, \ell_{y_m})$ with $s_x$ at index $i$ in the former and $s_y$ at index $j$ in the latter. The transitive dependence relation is the union of the iteration instances $I_{x_i} \times I_{y_j}$ and $I_{y_j} \times I_{x_i}$, which we denote as $T_{x_i} \times T_{y_j}$.
\((\ell_1, \ell_2, \ldots, \ell_m)\) respectively. Each dependence edge from \(s_x\) to \(s_y\) is represented using an \(m \times m\) matrix \(D_{m \times m}\), which we call an Extended Direction Matrix (EDM). Each entry \(D[i, j] (1 \leq i \leq m_x, 1 \leq j \leq m_y)\) in the matrix specifies a dependence condition between loops \(\ell(s_x)\) and \(\ell(s_y)\). The dependence represented by \(D\) satisfies the conjunction of all these conditions. This EDM representation extends traditional dependence vectors \([2, 24]\) by computing a relation between two arbitrary loops \(\ell_x(s_x)\) and \(\ell_y(s_y)\) even if \(\ell_x \neq \ell_y\). The extra information is important for resolving legality of transformations independent of the original loop structure.

Given a dependence EDM \(D\) from statement \(s_x\) to \(s_y\), we use the notation \(D(\ell_x, \ell_y)\) to denote the dependence condition in \(D\) between loops \(\ell_x(s_x)\) and \(\ell_y(s_y)\). Each condition \(D(\ell_x, \ell_y)\) can have the following values: \(\leq n\), \(\geq n\), \(= n\) and \(\leq n\). The first three values \(\leq n\), \(\geq n\) and \(\leq n\) specify that the dependence conditions are \(Ivar(\ell_x) = Ivar(\ell_y) + n\), \(Ivar(\ell_x) \leq Ivar(\ell_y) + n\) and \(Ivar(\ell_x) \geq Ivar(\ell_y) + n\) respectively; the last value \(\leq n\) specifies that the dependence condition is always true. We use the notation \(\text{Div}(D(\ell_x, \ell_y))\) to denote the dependence direction (\(\leq n\), \(\leq n\), \(\geq n\) or \(\leq n\)) of the condition and use \(\text{Align}(D(\ell_x, \ell_y))\) to denote the alignment factor of the condition. Both the dependence directions and alignment factors are computed using traditional dependence analysis techniques \([2, 24, 4]\).

We use the notation \(\text{td}(s_x, s_y)\) to denote the set of transitive dependence edges from \(s_x\) to \(s_y\). This set includes all the dependence paths from \(s_x\) to \(s_y\) in the dependence graph. Each transitive dependence edge in \(\text{td}(s_x, s_y)\) has the same EDM representation as those of the dependence edges in \(d(s_x, s_y)\). Because they summarize the complete dependence information between statements, transitive dependence edges can be used directly to determine the legality of program transformations, as shown in Section 4.1. To compute these transitive dependence edges, we perform transitive analysis on the dependence graph using the same algorithm by Yi, Adve and Kennedy \([25]\).

## 4 Computation Slicing Framework

In this section, we introduce a framework, computation slicing, that applies the dependence hoisting transformation illustrated in Section 2 to optimize an arbitrary loop nest. The framework first uses transitive dependence information to find all the opportunities of applying dependence hoisting. Then it combines loop strip-mining with dependence hoisting to automatically block the original loop nest.

Although we focus on blocking complicated loop nests for locality in this paper, computation slicing is a general framework that can integrate many other transformations besides loop strip-mining. In fact, our real-world implementation of this framework also integrated traditional loop fusion, distribution and interchange (see Section 6). Although we have not implemented many other transformations such as loop skewing and index-set splitting, we believe that these transformations can also be combined with computation slicing in a straightforward fashion.

Before presenting the computation slicing framework, Section 4.1 first describes how to use transitive dependence information to determine the safety of loop fusion and interchange transformations. Section 4.2 then presents an overview of the framework. Sections 4.3 and 4.4 again use the non-pivoting LU code in Figure 1(b) to illustrate the framework.

### 4.1 Transformation Analysis

Using transitive dependence information, this section resolves the legality of transforming arbitrarily nested loops at the outermost loop level of a given code segment. In particular, we focus on two loop transformations that are useful for the computation slicing framework: shifting an arbitrary loop \(\ell(s)\) to the outermost loop level and fusing two arbitrary loops \(\ell(s_x)\) and \(\ell(s_y)\) at the outermost loop level (the fused loop will be placed at the outermost loop level). Section 4.3 provides examples of these analyses.

To decide whether or not an arbitrary loop \(\ell(s)\) can be legally shifted to the outermost loop level, we examine the self transitive dependence \(\text{td}(s, s)\). We conclude that the shifting is legal if the following equation holds:

\[
\forall D \in \text{td}(s, s), \quad D(\ell, \ell) = \text{"\leq n" or "\leq n"}, \quad \text{where } n \leq 0. \tag{1}
\]

\footnote{In \([25]\), we used an integer programming tool, the Omega Library \([11]\), to facilitate recursion transformation. However, the transitive analysis algorithm in \([25]\) is independent of specific dependence representations and does not use the Omega Library.}
The above equation indicates that each iteration $I$ of loop $\ell(s)$ depends only on itself or previous iterations of loop $\ell(s)$. Consequently, placing loop $\ell(s)$ at the outermost loop level of statement $s$ is legal because no dependence cycle connecting $s$ is reversed by this transformation.

To decide whether two loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be legally fused at the outermost loop level, we examine the transitive dependencies $td(s_x, s_y)$ and $td(s_y, s_x)$. We conclude that the fusion is legal if the following equation holds:

$$\forall D \in td(s_y, s_x), \quad \text{Dir}(D(\ell_y, \ell_x)) = "\leq" \quad \text{or} \quad "\geq" \quad \text{and}$$

$$\forall D \in td(s_x, s_y), \quad \text{Dir}(D(\ell_x, \ell_y)) = "\leq" \quad \text{or} \quad "\geq"$$  \hspace{1cm} (2)

If Equation (2) holds, we can fuse loops $\ell_x(s_x)$ and $\ell_y(s_y)$ into a single loop $\ell_f(s_x, s_y)$ s.t.

$$Ivar(\ell_f) = Ivar(\ell_x) = Ivar(\ell_y) + \text{align},$$  \hspace{1cm} (3)

where $\text{align}$ is a small integer and is called the alignment factor for loop $\ell_y$. The value of $\text{align}$ must satisfy the following equation:

$$a_y \leq \text{align} \leq -a_x, \text{where}$$

$$a_x = \text{Max}\{\text{Align}(D(\ell_y, \ell_x)), \forall D \in td(s_y, s_x)\},$$

$$a_y = \text{Max}\{\text{Align}(D(\ell_x, \ell_y)), \forall D \in td(s_x, s_y)\}.  \hspace{1cm} (4)$$

From Equation (3), each iteration $I$ of the fused loop $\ell_f(s_x, s_y)$ executes both the iteration $I: \ell_x(s_x)$ (iteration $I$ of loop $\ell_x(s_x)$) and the iteration $I - \text{align}: \ell_y(s_y)$. From Equation (2) and (4), iteration $I: \ell_x(s_x)$ depends on the iterations $\leq I + a_x: \ell_y(s_y)$, which are executed by the iterations $\leq I + a_x + \text{align}$ of loop $\ell_f(s_y)$. Similarly, iteration $I - \text{align}: \ell_y(s_y)$ depends on the iterations $\leq I - \text{align} + a_y: \ell_x(s_x)$, which are executed by the same iterations of loop $\ell_f(s_x)$. Since $a_y \leq \text{align} \leq -a_x$ from Equation (4), we have $I + \text{align} + a_x \leq I$ and $I - \text{align} + a_y \leq I$. Each iteration $I$ of the fused loop $\ell_f(s_x, s_y)$ thus depends only on itself or previous iterations. Consequently, no dependence direction is reversed by this fusion transformation.

### 4.2 Overview

Building on the loop interchange and fusion analysis in Section 4.1, this section presents an overview of our computation slicing framework. This framework first identifies all the loops that can be shifted to the outermost position of a loop nest. It then applies a sequence of dependence hoisting transformations to re-arrange the original loop structure. Each dependence hoisting is seen as partitioning the original code into a set of slices — each slice executes a single fused iteration of a set of arbitrarily nested loops. We thus name this framework computation slicing (or slicing).

We use the name computation slice (or slice) to denote the information necessary to perform a dependence hoisting transformation. Each computation slice selects a set of loops called slicing loops to shift to the outermost position of some loop nest $C$. For each statement $s$ inside $C$, a loop $\ell$ surrounding $s$ is chosen as the slicing loop for $s$, and a small integer is chosen as the alignment factor for the slicing loop. Each statement $s$ is called the slicing statement of its slicing loop. A computation slice thus has the following attributes:

- **stmt-set**: the set of statements in the slice;
- **slice-loop(s) $\forall s \in$ stmt-set**: for each statement $s$ in stmt-set, the slicing loop for $s$;
- **slice-align(s) $\forall s \in$ stmt-set**: for each statement $s$ in stmt-set, the alignment factor for slice-loop(s).

A legal computation slice must satisfy three conditions: first, it includes all the statements in $C$; second, all of its slicing loops can be legally shifted to the outermost loop level; third, each pair of slicing loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be legally fused s.t. $Ivar(\ell_x) + \text{slice-align}(s_x) = Ivar(\ell_y) + \text{slice-align}(s_y)$.

The process of constructing legal computation slices is called computation slicing analysis (or slicing analysis). The analysis starts from some arbitrary statement $s_0$ and first finds all the valid slicing loops for $s_0$. For each found slicing loop $\ell_0(s_0)$, the analysis creates a slice including $\ell_0(s_0)$ and then extends the slice to include compatible slicing loops for other statements. Section 4.3 illustrates this process in more detail.

Each computation slice can drive a single dependence hoisting transformation, which fuses all the slicing loops in the slice. As illustrated in Section 2, dependence hoisting first puts a new outermost loop $\ell_f$ surrounding $C$
and then synchronizes each slicing loop $\ell(s)$ with $\ell_f$ by inserting a conditional surrounding the slicing statement $s$ so that $s$ is executed only if "$\text{Ivar}(\ell_f) = \text{Ivar}(\ell(s)) + \text{slice-align}(s)$". Section 5.4 proves the legality of this transformation. Because the inserted conditionals shift the loop-carrying levels of a set of dependences to the new outermost loop $\ell_f$, each slicing loop can then be legally distributed. The transformation then distributes each slicing loop $\ell$ so that it encloses only its slicing statement and the conditional synchronizing $\ell$ with loop $\ell_f$. Both the slicing loops and the conditionals can then be removed because they have become redundant. Section 5.3 presents the algorithm for dependence hoisting in more detail.

The computation slicing framework achieves automatic blocking by combining dependence hoisting transformations with loop strip-mining. The blocking process first collects all the legal computation slices for the input loop nest $C$. It then uses these slices to drive a sequence of dependence hoisting transformations. Whenever it decides to block the fused loop $\ell_f$ of a slice, the algorithm strip-mines $\ell_f$ into a strip-counting loop $\ell_c$ and a strip-enumerating loop $\ell_t$ and then uses $\ell_t$ as the input loop nest for further dependence hoisting transformations, which in turn shift a new set of loops outside loop $\ell_t$ but inside loop $\ell_c$, thus blocking loop $\ell_f$. Section 4.4 illustrates this process in more detail.

4.3 Slicing Analysis for Non-pivoting LU

This section illustrates slicing analysis using the $KJI$ form of non-pivoting LU in Figure 1(b). Figure 4 presents the dependence and transitive dependence information for this code using the EDM representation described in Section 3.2. Figure 5 shows the constructed computation slices.

To construct the slices in Figure 5, we first select an arbitrary statement and find all the candidate slicing loops for this statement. Suppose that statement $s_1$ is selected. We identify candidate slicing loops for $s_1$ as each loop $\ell(s_1)$ that can be legally shifted to the outermost loop level. The legality is resolved by examining the self transitive dependence $td(s_1,s_1)$ in Figure 4(b). From $td(s_1,s_1)$, the self dependence conditions are "$= -1$" for loop $k(s_1)$ and "$= 0$ or $= -1$" for loop $i(s_1)$; both conditions satisfy Equation (1) in Section 4.1. Both loops $k(s_1)$ and $i(s_1)$ thus can be legally shifted to the outermost loop level.

We then create two computation slices, $\text{slice}_k$ and $\text{slice}_i$, where $\text{slice}_k$ selects the slicing loop $k(s_1)$ for statement $s_1$, and $\text{slice}_i$ selects the slicing loop $i(s_1)$ for $s_1$. The alignment factors for both slicing loops are 0. We then extend each slice with a compatible slicing loop for statement $s_2$. From the self transitive dependence $td(s_2,s_2)$ in Figure 4(b), all the loops surrounding $s_2$ (loops $k(s_2)$, $j(s_2)$ and $i(s_2)$) can be legally shifted to the outermost loop level and thus are candidate slicing loops for $s_2$.

To further extend the computation slice $\text{slice}_k$, we find a slicing loop for statement $s_2$ as a loop $\ell(s_2)$ that can be legally fused with loop $k(s_1)$. The legality of fusion is resolved by examining the transitive dependences...
\[
\begin{align*}
do x_c = 1, n, b \\
do x_j = x_c \min(x_c - b - 1, n) \\
do k = 1, x_j - 1 \\
do i = k + 1, n \\
s_2: a(i, x_j) = a(i, x_j) - a(i, k) \ast a(k, x_j) \\
end do \\
\Rightarrow \\
\begin{align*}
do x_c = 1, n, b \\
do x_b = x_c, n - 1 \\
if x_b \geq x_c 	ext{ and } x_b \leq x_c + b - 1 \\
\text{then} \\
do i = x_b + 1, n \\
s_1: a(i, x_b) = a(i, x_b) / a(x_b, x_b) \\
end do \\
end if \\
do x_j = \max(x_j, x_c), \min(x_c + b - 1, n) \\
do i = x_b + 1, n \\
s_2: a(i, x_j) = a(i, x_j) \ast a(x_j, x_j) \\
en do \\
end do \\
\end{align*}
\]

(a) strip-mine slice \(_j\) 
(b) shift slice \(_k\) outside 
(c) split loop \(_x_k\)

Figure 6: Blocking non-pivoting LU

td(s\(_1\), s\(_2\)) and td(s\(_2\), s\(_1\)) in Figure 4(b). From td(s\(_1\), s\(_2\)), the dependence conditions for loop \(k(s_1)\) are “\(k(s_1) \leq k(s_2)\)”, “\(k(s_1) \leq j(s_2) - 1\)” and “\(k(s_1) \leq i(s_2) - 1\)” from td(s\(_2\), s\(_1\)), the conditions are “\(k(s_2) \leq k(s_1) - 1\)”, “\(j(s_2) \leq k(s_1)\)” and “\(i(s_2) \geq k(s_1) + 1\) or “\(i(s_2) \leq k(s_1)\)”. The conditions between loops \(k(s_1)\) and \(k(s_2)\) satisfy Equation (2) in Section 4.1; thus these two loops can be legally fused. From Equation (4), the alignment factor for loop \(k(s_2)\) is 0 or 1. Similarly, loop \(j(s_2)\) can be fused with loop \(k(s_1)\) with fusion alignment 0 or -1. Loop \(i(s_2)\) cannot be fused with loop \(k(s_1)\) because of the dependence condition “\(i(s_2) \geq k(s_1) + 1\)” in td(s\(_2\), s\(_1\)).

We then duplicate slice \(_k\) with another slice slice \(_k\), extend slice \(_k\) by selecting the slicing loop \(k(s_2)\) for \(s_2\), and then extend slice \(_j\) by selecting the slicing loop \(j(s_2)\) for \(s_2\). The completed slices are shown in Figure 5, Figure 5 also shows the slice slice \(_j\) which initially selects the slicing loop \(i(s_1)\) for statement \(s_1\). Here loop \(i(s_2)\) can be legally fused with loop \(i(s_1)\) and thus is selected as the slicing loop for statement \(s_2\).

4.4 Blocking Non-pivoting LU

In this section, we illustrate how to block the KJII form of non-pivoting LU in Figure 1(b) using slice \(_j\) and slice \(_k\) in Figure 5. Figure 6 shows both the intermediate and final results of the blocking transformation.

The slice slice \(_j\) is first used to drive a dependence hoisting transformation, and the transformed code is shown in Figure 3(b). The new outermost loop “\(x = 1, n\)” in Figure 3(b) is then strip-mined, as shown in Figure 6(a).

We then use the slice slice \(_k\) to shift loops outside the strip-countering loop \(x_j\) in Figure 6(a). Because the original \(k(s_1)\) loop becomes the \(x_j(s_1)\) loop in Figure 6(a), loop \(x_j(s_1)\) is now the slicing loop for statement \(s_1\) in slice \(_j\). Following similar steps as described in Section 2, we apply a dependence hoisting transformation to fuse the loops \(x_j(s_1)\) and \(k(s_2)\) and then place the fused loop outside loop \(x_j(s_2)\). Figure 6(b) shows the blocked code.

The code in Figure 6(b) can be written into a more intuitive version shown in Figure 6(c). Here the \(x_k\) loop is split so that the conditional surrounding \(s_1\) in Figure 6(b) can be removed. From Figure 6(c), we see that the factorization is now performed one column block at a time and is thus blocked in the column direction of the matrix. To achieve blocking in the row direction, we need to use slice \(_j\) in Figure 5 to further transform this code. The process follows similar transformation steps and will not be discussed further.

5 Transformation Algorithms

Figure 7 summarizes the algorithms for our computation slicing framework. This section first elaborates the three functions in Figure 7 and then discusses the legality proof of the algorithms.

9
5.1 Blocking a Loop Nest

In Figure 7, function Block-Loop-Nest is the top level procedure to block an arbitrary loop nest and has two steps. Step (1) performs slicing analysis on the input loop nest \( C \) to collect all the legal computation slices into a variable set \( \text{slices-set} \). Step (2) then blocks \( C \) through a sequence of dependence hoisting transformations using the collected slices.

The blocking algorithm in Figure 7 uses a variable \( C_1 \) to keep track of the current code segment to transform. Following each dependence hoisting transformation on \( C_1 \), if it decides to blocking the new loop \( \ell_f \) placed outside \( C_1 \), the algorithm strip-mines \( \ell_f \) into a strip-counting loop \( \ell_c \) and a strip-enumerating loop \( \ell_t \), and then uses \( \ell_t \) as the new input loop nest for further dependence hoisting transformations by setting \( C_1 = \ell_t \). Further transformations on \( C_1 \) then shift a new set of loops outside loop \( \ell_t \) but inside loop \( \ell_c \), thus blocking loop \( \ell_f \).

Note that the algorithm automatically rearranges the original loop nesting order when extracting computation slices from \( \text{slices-set} \). To optimize locality of programs, at every iteration, the algorithm extracts from \( \text{slices-set} \) the computation slice whose slicing loops carry the least data reuses. This strategy ensures that the loops carrying fewer data reuses are nested outside. The algorithm also strip-mines a loop \( \ell_t \) only if the loop carries some data reuse. This strategy reduces the number of non-profitable loop blockings that actually degrade the performance.

5.2 Slicing Analysis

The function Slicing-Analysis in Figure 7 constructs all the legal computation slices for the input loop nest \( C \) and then puts these slices into the output variable set \( \text{slices-set} \). This function builds on the loop interchange and fusion analysis in section 4.1 and has the following three steps.

Step (1) chooses an arbitrary statement \( s_0 \) in the input loop nest as the starting point and identifies candidate slicing loops for \( s_0 \) as each loop \( \ell(s_0) \) that can be legally shifted to the outermost loop level. This step then creates a computation slice for each candidate slicing loop of statement \( s_0 \) and puts these slices into a variable set \( \text{slices-under-construction} \) for further examination.

Step (2) checks the variable \( \text{slices-under-construction} \) and exits if this variable set is empty; otherwise, this step pulls a computation slice out of \( \text{slices-under-construction} \) and extends the slice by going to step (3).

Step (3) completes the partial slice \( \text{s} \) selected by step (2) and then goes back to step (2) for other partial slices. For each statement \( s \) not yet in the partial slice \( \text{s} \), this step identifies valid slicing loops for \( s \) as each loop \( \ell(s) \) that can be shifted to the outermost loop level and can be fused with the slicing loops already in \( \text{s} \). For each found slicing loop \( \ell(s) \), an alignment factor \( \text{align} \) is computed from the fusion alignments between loop \( \ell(s) \) and other slicing loops in \( \text{s} \). If multiple slicing loops are found, step (3) duplicates the original slice to remember all the extra slicing loops and then adds the duplicated slices into the variable set \( \text{slices-under-construction} \) for further examination. At the end of step (3), if all the statements have been included in \( \text{s} \), this original slice is completed successfully and thus can be collected into the output variable \( \text{slices-set} \); otherwise, this slice is thrown away.

5.3 Dependence Hoisting Transformation

The function Dependence-Hoisting in Figure 7 performs a dependence hoisting transformation on the given loop nest \( C \) using a computation slice \( \text{s} \). The transformation process is separated into the following three steps.

Step (1) puts a new outermost loop \( \ell_f \) surrounding \( C \) and then inserts a conditional \( \text{cond}(s) \) surrounding each statement \( s \). Loop \( \ell_f \) unions the iteration ranges of all the slicing loops after proper alignments. Each conditional \( \text{cond}(s) \) forces the statement \( s \) inside it to execute only if \( \text{Ivar}(\ell_f) = \text{Ivar}(\text{slices-loop}(s)) + \text{slices-align}(s) \). Step (1) then modifies the dependence graph of \( C \) to include dependence conditions for the new loop \( \ell_f \). For each statement \( s \), \( \text{Ivar}(\ell_f(s)) \) has the same dependence conditions as those of \( \text{Ivar}(\text{slices-loop}(s)) + \text{slices-align}(s) \). Figure 2(b) illustrates this modification using the KJI form of non-pivoting LU.

Step (2) then distributes each slicing loop \( \ell(s) \) so that \( \ell(s) \) encloses only its slicing statement \( s \). Step (2.1) first applies loop index-set splitting to remove all the dependence cycles that are incident to statement \( s \) and are carried by loop \( \ell \). For each statement \( s_1 \) \((\ell(s_1) = \text{slices-loop}(s_1)) \) strongly connected with statement \( s \) in \( \text{Dep}(\ell) \) (the dependence graph of \( \ell \)), step (2.1) splits \( s_1 \) into three statements under the execution conditions
Block-Loop-Nest(C)
(1) slice-set = Ø, Slicing-Analysis(C, slice-set)
(2) C₁ = C

While slice-set ≠ Ø do
    slice = remove a slice from slice-set
    Dependence-Hoisting(C₁, slice)
    If (slice should be blocked) then
        Strip-nine the outermost loop ⋁₀ surrounding C₁
        C₁ = strip enumerating loop of ⋁₀

Slicing-Analysis(C, slice-set)
(1) slice-under-construction = Ø
    s₀ = an arbitrary statement in C
    For each loop l₁(s₀) that can be moved to outermost level
        Add a new slice slice into slice-under-construction
    Add s₀ into slice: slice-loop(s₀) = l₁(s₀), slice-align(s₀) = 0
(2) If slice-under-construction = Ø, stop.
    slice = Remove a slice from slice-under-construction
(3) For each statement s in C but not in slice
    For each loop l₁(s) that can be fused with slicing loops in slice
        and can be moved to the outermost loop level of C
        align = alignment factor for l₁(s)
        If (l₁(s) is the first loop found) slice₁ = slice
        Else slice₁ = Duplicate(slice)
            Add slice₁ into slice-under-construction
            Add s into slice: slice-loop(s) = l₁(s), slice-align(s) = align
            If (slice includes all statements in C), add slice into slice-set
    Goto step (2)

Dependence-Hoisting(C₁, slice)
(1) slice-range = \bigcup_{l ∈ slice} (Range(slice-loop(l)) + slice-align(l))
    l_f = create a new loop with Range(l_f) = slice-range
    Put l_f surrounding C
    For each statement s ∈ slice with l = slice-loop(s)
        cond₁(s) := "if (Ivar(l_f) = Ivar(l) + slice-align(l))"
        Insert cond₁(s) surrounding s
    Modify the dependence graph of C
(2) For each statement s ∈ slice with l(s) = slice-loop(s)
    Distribute l(s) to enclose only s
    (2.1) For each statement s₁ strongly connected with a in Dep(l)
        m = slice-align(s₁) + slice-align(s)
        l₁(s₁) = slice-loop(s₁)
        Split s₁ with conditions: Ivar(l₁) < Ivar(l₁) + m
            Ivar(l₁) = Ivar(l₁) + m
        and Ivar(l₁) > Ivar(l₁) + m
        Modify the dependence graph of C
    (2.2) Distribute loop l(s) so that Dep(l(s)) is strongly connected
        While (l(s) contains other statements than s) do
            Interchange loop l(s) with the loop immediately inside it
        Modify dependence graph of C
        Distribute loop l(s) so that Dep(l(s)) is strongly connected
    (3) For each statement s ∈ slice with l(s) = slice-loop(s)
        Replace Ivar(l) with Ivar(l_f) - slice-align(s) inside l(s)
        Adjust iteration ranges of loops between l_f and l(s)
        Remove cond₁(s) and cond₂(s)

Figure 7: Algorithms to block a loop nest.

Ivar(l) < Ivar(l₁) + m, Ivar(l) = Ivar(l₁) + m and Ivar(l) > Ivar(l₁) + m respectively, where m = slice-align(s₁) - slice-align(s). Step (2.2) then repeatedly applies loop distribution and interchange to remove all the dependence cycles that are incident to s and are carried by loops inside l(s). At each iteration, step (2.2) first distributes loop l(s) so that Dep(l) is strongly connected. If l(s) still encloses statements other than s, this step interchanges l(s) with the loop immediately inside l(s) and then tries again. Since steps (2.1) and (2.2) successfully remove all the dependence cycles incident to s in Dep(l), eventually the distributed loop l(s) encloses only statement s. For proof of these two steps, see section 5.4.

Step (3) then removes all the distributed slicing loops and the conditionals that synchronize them with the outermost loop ⋁₀. Before removing each slicing loop l(s), this step makes two adjustments to the original loop nest: first, it replaces the loop index variable Ivar(l(s)) with Ivar(l_f) - slice-align(s) inside loop l(s); second, it adjusts the iteration ranges of the loops between l_f and l(s) to ensure the correct iteration set of statement s. Step (3) then removes both the slicing loop l(s) and the conditional cond₁ because they have become redundant.

5.4 Legality and Complexity

The slicing analysis algorithm in Figure 7 is correct in that only legal computation slices are collected. Each collected slice satisfies three conditions: first, it includes all the statements in the input loop nest; second, all the slicing loops can be legally shifted to the outermost loop level; third, each pair of slicing loops l_x(s_x) and l_y(s_y) can be legally fused after being aligned with slice-align(s_x) and slice-align(s_y) respectively.

We prove the legality of the dependence hoisting transformation steps in Figure 7 by demonstrating that the transformed code at each step satisfies two conditions: first, each statement still executes the same set of iteration instances; second, no dependence is reversed in the dependence graph.

Step (1) of the dependence hoisting transformation puts a new loop l_f at the outermost position and then inserts a conditional surrounding each statement. Each conditional forces the statement s inside it to execute the same set of iteration instances by synchronizing the slicing loop slice-loop(s) with the outermost loop l_f in lock-step fashion with alignment slice-align(s). These conditionals shift the loop-carrying levels of a set of dependences to the outermost loop level. Since all the slicing loops can be legally shifted to the outermost loop level and can be legally fused with each other, no dependence direction is reversed.

Step (2) of the dependence hoisting transformation distributes each slicing loop l(s) so that l(s) encloses only its slicing statement s. After step (1), all the dependence paths between two statements s_x and s_y are now
carried by the outermost loop \( \ell_f \) unless the following condition holds:

\[
Ivar(\ell_x) + \text{slice-align}(s_x) = Ivar(\ell_y) + \text{slice-align}(s_y)
\]

(5)

where \( \ell_x = \text{slice-loop}(s_x) \) and \( \ell_y = \text{slice-loop}(s_y) \). Thus only the dependence paths satisfying Equation (5) need to be considered when distributing the slicing loops.

For each slicing loop \( \ell(s) \), step (2.1) applies loop index-set splitting to remove all the dependence cycles that are incident to statement \( s \) and are carried by loop \( \ell \) in \( \text{Dep}(\ell) \) (the dependence graph of \( \ell \)). For each statement \( s_1 \neq s \) inside loop \( \ell \), suppose \( \text{slice-loop}(s_1) = \ell_1 \). From Equation (5), each dependence EDM \( D \) between statements \( s_1 \) and \( s \) satisfies the condition \( D(\ell(s), \ell_1(s_1)) = "m" \), where \( m = \text{slice-align}(s_1) - \text{slice-align}(s) \). Once we split \( s_1 \) into three statements \( s'_1 \), \( s''_1 \) and \( s'''_1 \) under execution conditions \( Ivar(\ell) < Ivar(\ell_1) + m \), \( Ivar(\ell) = Ivar(\ell_1) + m \) and \( Ivar(\ell) > Ivar(\ell_1) + m \) respectively, each dependence EDM between statements \( s'_1 \) and \( s \) satisfies the condition \( Ivar(\ell(s'_1)) \leq Ivar(\ell(s)) - 1 \). Since each iteration of loop \( \ell \) depends only on itself or previous iterations in \( \text{Dep}(\ell) \), there is no dependence path from statement \( s \) to \( s'_1 \) in \( \text{Dep}(\ell) \). Similarly, there is no dependence path from statement \( s''_1 \) to \( s \) in \( \text{Dep}(\ell) \). Thus all the dependence cycles between statements \( s \) and \( s_1 \) are now between \( s \) and \( s''_1 \). Since each dependence EDM between \( s'_1 \) and \( s \) satisfies the condition \( Ivar(\ell(s)) = Ivar(\ell(s''_1)) \), no dependence cycle connecting \( s \) is carried by loop \( \ell \).

For each slicing loop \( \ell(s) \), step (2.2) of the dependence hoisting transformation further removes from \( \text{Dep}(\ell) \) all the dependence cycles that are incident to statement \( s \) and are carried by loops inside \( \ell(s) \). This step first distributes loop \( \ell(s) \) so that \( \ell(s) \) does not carry any dependence. If \( \ell(s) \) still encloses statements other than \( s \), there must be dependence cycles carried by a loop \( \ell' \) perfectly nested inside \( \ell(s) \), step (2.2) can then shift loop \( \ell \) inside \( \ell' \) to remove these dependence cycles from \( \text{Dep}(\ell) \). The distribution and interchange repeat until loop \( \ell(s) \) encloses only statement \( s \).

Since each slicing loop \( \ell(s) \) now encloses only statement \( s \) and the conditional synchronizing \( \ell(s) \) with the outermost loop \( \ell_f \), both the slicing loop and the conditional can be safely removed. Step (3) of the dependence hoisting transformation removes both the slicing loops and conditionals while making sure each statement still executes the same set of iteration instances. Therefore all the transformation steps are legal.

The complexity of function \( \text{Slicing-Analysis} \) in Figure 7 is \( O(N^2L) \), where \( N \) is the number of statements, and \( L \) is the maximum depth of the input loop nest. The complexity of the function \( \text{Dependence-Hoisting} \) in Figure 7 is \( O(N^2 + NLD) \), where \( D \) is the size of the dependence graph. Since at most \( L \) computation slices can be constructed for any loop nest, the complexity of function \( \text{Block-Loop-Nest} \) in Figure 7 is \( O(N^2L + NL^2D) \).

## 6 Experimental Results

We have implemented the algorithms in Figure 7 as a Fortran source-to-source translator constructed on top of the D System, an experimental compiler infrastructure developed at Rice. We have applied this system to four numerical linear algebra kernels—Cholesky, QR, LU factorization without pivoting, and LU factorization with partial pivoting—and two benchmark applications—\( \text{tomato} \) (a vectorized mesh generation program with 191 lines of Fortran code) and \( \text{Erlebacher} \) (a three-dimensional tridiagonal solver with 554 lines of Fortran code).

In this section, we will focus primarily on the performance results of the four linear algebra kernels for two reasons. First, these benchmarks are important linear algebra subroutines that are widely used in scientific computing applications. Second, the loop nests within them are considered difficult to block automatically. Previous (and quite powerful) compiler techniques (including both unimodular transformation strategies and general loop transformation frameworks) have been able to automatically block only Cholesky and LU without pivoting. To our knowledge, no previous translator has completely automated the blocking of either QR or LU with partial pivoting. Carr, Kennedy and Lehoucq [5, 6] hypothesized that no compiler could automatically produce the blocking of pivoting LU that is used in LAPACK [3] without specific commutativity information. Our work has not disproved this hypothesis; rather, we have succeeded in generating a substantially different blocking for pivoting LU without requiring such commutativity information (see Figure 8). Surprisingly, our blocked version exhibits only minor performance differences (see section 6.2) when compared with the LAPACK version.

Note that we are not attempting to compete with the existing hand-tuned implementations of linear algebra kernels. Instead we are using the naive versions of these kernels to demonstrate the power of the transformation framework we have developed. If our strategy can succeed in optimizing and blocking these to a level of
performance comparable to LAPACK, it can also be successful on a wide variety of similarly challenging loop
ests.

Being able to block both QR and LU with pivoting indicates that with significantly less compile-time overhead, our translator can, in certain difficult cases, succeed where previous general transformation frameworks have failed. However, this observation does not generalize in a theoretical sense. Many general frameworks, which can be shown to be sufficiently powerful to succeed on a particular complex loop nest, may nevertheless fail to identify a valid transformation. One reason this might happen is that these systems require so much computational power that some solutions cannot be found due to technical constraints. Our approach is thus valuable in that it can handle difficult loop nests such as those in QR and pivoting LU and that it can do so with complexity similar to that of reasonably inexpensive unimodular loop transformation techniques.

To evaluate the performance of these linear algebra kernels, we compare the performance of the original code with that of the code output from our blocking translator for each benchmark kernel. We also compare the performance of our automatically blocked code to that of out-of-the-box LAPACK on each linear algebra kernel except LU without pivoting (there is no such LAPACK entry). For non-pivoting LU, we compare our performance to that of a version blocked by hand following the LAPACK blocking strategy for LU with pivoting.

Our objective in the comparison with LAPACK is to show how close our automatic strategy can get to the best hand-coded versions of the same kernels. LAPACK is chosen because it has been developed by professional algorithm designers over a period of years. In some cases, these developers even applied algorithmic changes that are not available to our system because they violate the restrictions of dependence. Being able to achieve this objective indicates that our translator is very effective in optimizing complex kernels of the kinds found in these benchmarks.

We also briefly summarize the performance results for tomcatv and Erlebacher. These two benchmarks employ only simple loop structures that can also be optimized using traditional unimodular techniques alone. However, being able to optimize large benchmark programs like tomcatv and Erlebacher demonstrates that computation slicing is a general framework that can be combined with many other standard loop transformations such as fusion and interchange.

To present performance results of the benchmarks, Section 6.1 first briefly describes the original versions of the benchmarks and our strategy for blocking LU factorization with partial pivoting. Section 6.2 then presents performance measurements of the benchmarks on an SGI workstation.

6.1 Benchmarks

The original linear algebra kernels used in our study are transcribed from the simple versions found in Golub and Van Loan [10]. The original versions of LU with and without pivoting are shown in Figure 8(a) and Figure 1(b) respectively. The original versions of Cholesky and QR are written similarly. The code for QR is based on Householder transformations [10]. The original versions of tomcatv and Erlebacher are obtained from the SPEC95 and ICASE benchmark suites respectively.

Our translator automatically blocked each linear algebra kernel in less than two seconds on a SUN Sparc workstation with 330MHz processors. The two larger benchmarks, tomcatv and Erlebacher, each take the translator less than 8 seconds to transform. The above compile time is measured with our translator compiled with "-g" option and with no tuning. These facts provide evidence that the slicing framework is fast enough for use in commercial compilers.

For the four linear algebra kernels, blocking is the principal transformation applied by the translator. For Cholesky and non-pivoting LU, both the row and column dimensions of the matrix are blocked; for QR and pivoting LU, only the column dimension is blocked (the row dimension cannot be blocked for the program to be correct). The blocking for Cholesky is quite similar to the one for non-pivoting LU, discussed in section 4.4. Here we briefly describe our blocking for pivoting LU, which is also a model for blocking QR.

Figure 8(a) shows the original version of pivoting LU. This version is different from the one used in the LAPACK BLAS [5] in that the j loop surrounding statement s3 has iteration range “j = k, n” instead of “j = 1, n”. (To block the BLAS version, a preliminary step to split this loop into two loops “j = 1, k – 1” and “j = k, n” is needed. Although we believe this step can be automated, we have not yet implemented it in our translator.)

We block the input code using two computation slices: the first, slicek, selects the outermost k loop as slicing loops for all the statements; the second, slicej, selects the k loop for statements s1, s2 and e1, but selects the
do $k = 1,n-1$

$s_1$: $p(k) = k$; $mu = abs(a(k,k))$

do $i = k+1,n$

$s_2$: if ($mu < abs(a(i,k))$) then

$s_3$: endif

do $j = k,n$

$s_4$: $t = a(k,j)$

$s_5$: $a(k,j) = a(p(k),j)$

$s_6$: $a(p(k),j) = t$

do $i = k+1,n$

$s_7$: $a(i,k) = a(i,k)/a(k,k)$

do $i = k+1,n$

$s_8$: $a(i,j) = a(i,j) - a(i,k) * a(k,j)$

endo

(a) original code

do $x = 1,n$

do $k = 1,x-1$

$s_9$: $t = a(k,x)$

$s_{10}$: $a(k,x) = a(p(k),x)$

$s_{11}$: $a(p(k),x) = t$

do $i = k+1,n$

$s_{12}$: $a(i,x) = a(i,x) - a(i,k) * a(k,x)$

do $i = k+1,n$

endo

(b) after dependence hoisting

do $x_c = 1,n$

do $x_j = x_c,min(x_c - 1,n - 1)$

$s_{13}$: $t = a(x_c,x_j); a(x_c,x_j) = a(p(x_c),x_j); a(p(x_c),x_j) = t$

endo

do $i = x_j +1,n$

$s_{14}$: $a(x_j,x_j) = a(x_j,x_j) * a(x_j,x_j)$

do $i = x_j +1,n$

endo

(c) after blocking

Figure 8: Blocking LU factorization with partial pivoting

<table>
<thead>
<tr>
<th>Cycles in billions</th>
<th>L1 misses in millions</th>
<th>L2 misses in millions</th>
<th>TLB misses in thousands</th>
</tr>
</thead>
<tbody>
<tr>
<td>chol</td>
<td>lu</td>
<td>lup</td>
<td>qr</td>
</tr>
<tr>
<td>0.2</td>
<td>0.3</td>
<td>0.9</td>
<td>0.9</td>
</tr>
</tbody>
</table>

(oiginal version; sliced version; L-LAPACK version)

Figure 9: Results from measurements on a uniprocessor SGI workstation

$j$ loop for $s_3$ and $s_5$. The transformed code using slice $x_j$ is shown in Figure 8(b). Here the original $j(s_3)$ loop is split into "j = k" and "j = k+1,n" by step (2.1) of the dependence hoisting transformation in Figure 7. This code is similar to the deferred-update version of non-pivoting LU in Figure 1(c). We then strip-mine the outermost $x$ loop in Figure 8(b) and use the slice $x_c$ to further shift loops outside the strip-enumarating $x_j$ loop. Figure 8(c) shows the blocked code using both slices. This code operates on a single block of columns at a time and is similar to the blocked code of non-pivoting LU in Figure 6(c). The blocking is based on the observation that, although the code dealing with selecting pivots (statements $s_1$ and $s_2$) imposes bi-directional dependence constraints among rows of the input matrix, the dependence constraints among columns of the matrix have only one direction—from columns on the left to columns on the right.

For tomcatu and Erlebaher, loop fusion is the principal transformation applied by the translator (our translator also integrate loop fusion in the computation slicing framework). Blocking is applied to only a few loops in Erlebaher and is applied only to tomcatu with an enhanced matrix size (see section 6.2).

6.2 Performance Results

We measured the performance on an SGI workstation with a 195 MHz R10000 processor, 256MB main memory, separate 32KB first-level instruction and data caches (L1), and a unified 1MB second-level cache (L2). Both caches are two-way set-associative. The cache line size is 32 bytes for L1 and 128 bytes for L2. We used SGI's
perfex tool (which is based on two hardware counters) to count the total number of cycles, and L1, L2 and TLB misses.

All benchmarks (including their original versions, automatically transformed versions and LAPACK versions) were compiled using the SGI compiler with "-O2" option. This strategy ensures that all the versions are optimized at the same level by the SGI compiler. For each auto-blocked version, different block sizes were tested and the result of the best block size is presented. Thus, the results for automatically blocked versions of linear algebra kernels include the improvements due to an auto-tuning step similar to (but less powerful than) that used by the ATLAS system to tune the BLAS for a new architecture [21]. This is fair because the numbers reported for LAPACK in Figure 9 are based on BLAS versions with tuning parameters selected by ATLAS for the SGI workstation.

For each linear algebra kernel, we present results on a 500x500 matrix. We choose this matrix size because it is the most favorable for LAPACK. On larger matrices, the performance comparisons are consistent on LU with and without pivoting. However, larger matrix sizes are more favorable to our blocked versions on Cholesky and QR. We repeated each measurement 5 or more times and present the average across these runs. The variations across runs were very small (within 1%).

Figure 9 presents the performance results of the linear algebra kernels, where lu, lup, chol and qr denote the four kernels, and “sliced version” denotes the versions blocked by our translator. In each case, results are normalized to the performance of the original version.

All the sliced versions performed much better than the original versions because of better locality. The performance improvements are shown uniformly in the cycle count and L2 cache miss graphs. The sliced versions of chol and lu also manifest improvements in L1 cache performance of more than a factor of 10. Here because the compiler has blocked both the row and column array dimensions in these versions, their working sets are small enough to fit in L1 cache. However, these two versions also show poor TLB performance due to the penalty of accessing data with large strides.

Compared with the LAPACK versions, the sliced version of chol achieved almost identical overall performance, the sliced version of QR achieved better performance, and the sliced versions of lu and lup achieved comparable though slightly worse performance. For lup, both the sliced and LAPACK versions have only the column dimension of the matrix blocked (the row dimension cannot be blocked in order for the program to be correct), and the SGI workstation favors the loop ordering in the LAPACK version. For lu, the penalty of higher TLB misses for the sliced version negated its better L1 cache performance. Reorganizing the data layout [20] may improve this situation.

Table 1 presents the performance results of tomcatv and Erlebacher. For tomcatv, we used two matrix sizes: a moderate size 513x513 and an enhanced size 1025x1025. For Erlebacher, we used a 129x129x129 array. For these two benchmarks we used multiple transformation options in our translator (our translator is tunable by turning off certain optimizations): one option has blocking turned off to focus on fusion and the other option does not. We used multiple transformation setups because there is a trade-off between blocking and fusion—two loops will not be fused in the translator if the fusion inhibits blocking. Thus, to obtain the best performance for both tomcatv and Erlebacher, the translator needs to apply profitability analysis to determine when to favor blocking and when to favor fusion. Although this analysis can be combined with computation slicing in a straightforward fashion, our translator does not yet incorporate it, so we presents results from both setups.

In Table 1, all the performance improvements are relative to the original versions of the benchmarks. From these measurements, we see that most of the improvements for tomcatv and Erlebacher come from applying loop fusion combined with computation slicing. Blocking becomes effective for tomcatv only when we use larger data sizes that do not fit in the primary cache.
7 Related Work

Many unimodular loop transformations, such as loop blocking, fusion, distribution, interchange, skewing and index-set splitting [23, 15, 17, 8], have been proposed to improve locality of applications. These techniques are highly efficient when transforming simple loop structures such as perfectly nested loops. However, they are often ineffective when transforming complicated loop nests. Wolf and Lam [23] proposed an uniform algorithm to select compound sequences of unimodular loop transformations. This algorithm is still limited by the original loop structures of programs.

This paper extends traditional unimodular loop transformations to effectively transform complicated loop nests independent of their original nesting structure. Although it does not incorporate the entire solution space of loop transformations, our technique has demonstrated high effectiveness by blocking some of the most challenging benchmarks.

Several general loop transformation frameworks are theoretically more powerful than the technique proposed in this paper. These frameworks vary in their efficiency and effectiveness. Pugh [19] proposed a framework that finds a schedule for each statement describing the moment each statement instance will be executed. This technique requires an expensive step to find a feasible mapping from iteration spaces of statements into instance execution time. Kodukula, Ahmed and Pingali [14] proposed an approach called \textit{data shuffling}, in which a tiling transformation on a loop nest is described in terms of a tiling of key arrays in the loop nest. Here a mapping from the iteration spaces of statements into the data spaces must be computed. Lim, Cheong and Lam [16] proposed a technique called \textit{affine partition}, which maps instances of instructions into the time or processor space via an affine expression in terms of the loop index values of their surrounding loops. Finally, Ahmed, Mateev and Pingali [1] proposed an approach that embeds the iteration spaces of statements into a \textit{product space}. The product space is then transformed to enhance locality. In this approach, a mapping from iteration spaces of statements into the \textit{product space} must be computed.

All the above transformation frameworks compute a mapping from the iteration spaces of statements into some other space. The computation of these mappings is expensive and generally requires special integer programming tools such as the Omega library [11]. In contrast, our technique does not require such mappings and, hence, is much more efficient. In our earlier work [25], we used Omega library to perform recursion analysis and transformation. In our experience, the integer set and mapping operations significantly slowed down our translator. This experience motivated us to seek simpler and more efficient transformation techniques.

Rosser and Pugh [22] have proposed a technique similar to our framework. They first proposed using transitive dependences to achieve loop transformations independent of the original loop structure. They also proposed a technique called \textit{iteration space slicing}, which represents transitive dependences as integer set mappings to compute related iteration instances of statements. We use transitive dependences to relate iterations of arbitrary loops directly without manipulating any symbolic spaces of statements.

The loop model in this paper is similar to that adopted by Ahmed, Mateev and Pingali [1]. They represent the same loop surrounding different statements as different loops and use a product space to incorporate all the extra loop dimensions. Our approach uses far fewer extra loop dimensions—it temporarily adds one extra loop at each dependence hoisting transformation and then removes the extra dimension immediately.

Finally, our technique can be effectively combined with other strategies for optimizing memory performance of simple loop nests, such as automatic selection of blocking factors [8, 15, 18], loop fusion [13, 12], multi-level memory hierarchy management [7], and data layout rearrangement [20].

8 Conclusion

This paper extends previous unimodular loop transformations to fuse and block complex loop nests for better locality. Experimental results indicate that our technique is inexpensive in running time, yet highly effective—for real codes, it approaches the power of many general loop transformation frameworks.

References


