# Layout-Oblivious Compiler Optimization for Matrix Computations

HUIMIN CUI, CAS
QING YI, University of Colarodo at Colorado Springs
JINGLING XUE, University of New South Wales
XIAOBING FENG, CAS

Most scientific computations serve to apply mathematical operations to a set of preconceived data structures, e.g., matrices, vectors, and grids. In this article, we use a number of widely used matrix computations from the LINPACK library to demonstrate that complex internal organizations of data structures can severely degrade the effectiveness of compiler optimizations. We then present a data-layout-oblivious optimization methodology, where by isolating an abstract representation of the computations from complex implementation details of their data, we enable these computations to be much more accurately analyzed and optimized through varying state-of-the-art compiler technologies. We evaluated our approach on an Intel 8-core platform using two source-to-source compiler infrastructures, Pluto and EPOD. Our results show that while the efficiency of a computational kernel differs when using different data layouts, the alternative implementations typically benefit from a common set of optimizations on the operations. Therefore separately optimizing the operations and the data layout of a computation could dramatically enhance the effectiveness of compiler optimizations compared with the conventional approaches of using a unified representation.

Categories and Subject Descriptors: D.3.4 [**Processors**]: Compilers, Optimization

General Terms: Design, Performance

Additional Key Words and Phrases: Compiler, optimization, pattern, matrix computation, layout

**ACM Reference Format:**
Cui, H., Yi, Q., Xue, J., and Feng, X. 2013. Layout-Oblivious compiler optimization for matrix computations. ACM Trans. Architec. Code Optim. 9, 4, Article 35 (January 2013), 20 pages.
DOI = 10.1145/2400682.2400694 http://doi.acm.org/10.1145/2400682.2400694

## 1. INTRODUCTION

A majority of scientific applications, in spite of their varying domains, heavily rely on a number of data-intensive kernels to compute desired solutions. These kernels typically serve to apply a sequence of domain-specific operations to a preconceived set of compound data structures, e.g., matrices, vectors, grids, and graphs. The operations may read or modify different portions of these data and then combine their values in varying ways. To effectively optimize these computations, e.g., to parallelize operations on

This work is partially supported by the NSFC grant 61202055, the Chinese National Basic Research grant 2011CB302504, the Innovation Research Group of NSFC 60921002, the National Science and Technology Major Projects of China 2011ZX01028-001-002, State 863 project 2012AA010902, the National Science Foundation of USA under grants 0833203 and 0747357, USA Department of Energy under grant DE-SC001770, and Australian Research Council (ARC) grant DP0987236 and DP110104628.

Authors' addresses: H. Cui (corresponding author), SKL Computer Architecture; Institute of Computing Technology, CAS; email: huiman.cui@gmail.com; Q. Yi, University of Colorado at Colorado Springs; J. Xue, School of Computer Science and Engineering, University of New South Wales; X. Feng, SKL Computer Architecture; Institute of Computing Technology, CAS.

DOI 10.1145/2400682.2400694 http://doi.acm.org/10.1145/2400682.2400694

| Matrix Type | Triangular Matrix | Banded Matrix | Banded Triangular Matrix |
|---|---|---|---|
| (a) Abstract Layout | $\begin{bmatrix} A_{00} & 0 & 0 & \ldots & & 0 \\ A_{10} & A_{11} & 0 & 0 & \ldots & 0 \\ A_{20} & A_{21} & A_{22} & 0 & \ldots & 0 \\ A_{30} & A_{31} & A_{32} & A_{33} & 0 & 0 \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & 0 \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} \end{bmatrix}$ | $\begin{bmatrix} A_{00} & A_{01} & 0 & \ldots & & 0 \\ A_{10} & A_{11} & A_{12} & 0 & \ldots & 0 \\ 0 & A_{21} & A_{22} & A_{23} & 0 & 0 \\ 0 & 0 & A_{32} & A_{33} & A_{34} & 0 \\ 0 & 0 & 0 & A_{43} & A_{44} & A_{45} \\ 0 & 0 & 0 & 0 & A_{54} & A_{55} \end{bmatrix}$ | $\begin{bmatrix} A_{00} & A_{01} & 0 & \ldots & & 0 \\ 0 & A_{11} & A_{12} & 0 & \ldots & 0 \\ 0 & 0 & A_{22} & A_{23} & 0 & 0 \\ 0 & 0 & 0 & A_{33} & A_{34} & 0 \\ 0 & 0 & 0 & 0 & A_{44} & A_{45} \\ 0 & 0 & 0 & 0 & 0 & A_{55} \end{bmatrix}$ |
| (b) Packed Layout | $[A_{00}\ A_{10}\ A_{11}\ A_{20}\ \ldots\ A_{55}]$ | $[0\ A_{00}\ A_{01}\ A_{10}\ A_{11}\ A_{12}\ A_{21} \ldots\ A_{55}\ 0]$ | $[A_{00}\ A_{01}\ A_{11}\ A_{12}\ \ldots\ A_{55}\ 0]$ |

Fig. 1. Data layout for triangular, banded, and banded triangular matrices.

different portions of a matrix, a compiler must be able to correctly decipher the dependence relations among statements operating on different data, e.g., by accurately distinguishing operations on different coordinates of a matrix [Allen and Kennedy 2001].

While compilers have used *dependence analysis* as the foundation of optimization for more than thirty years, the accuracy of dependence analysis could be severely degraded by complex data structures. Take the well-understood dense matrix multiplication, for instance. When the matrix is rectangular and stored in a single-dimensional array, a general-purpose compiler can attain a level of performance [Yotov et al. 2003; Chen et al. 2005] comparable to that achieved by empirically tuned ATLAS [Whaley et al. 2001]. However, nonrectangular matrices such as triangular, banded, and banded triangular ones [Mullhaupt and Riedel 2001] are typically stored in compacted forms to save space, as illustrated in Figure 1. When storing a nonrectangular matrix in a *packed* layout such as those in Figure 1(b), the array subscripts used to reference elements of the matrix could be extremely complex, which could easily overwhelm the internal dependence analysis of a compiler and thus thwart all optimizations to the matrix computation code. Section 2 shows an example of such failures in more detail.

We present a layout-oblivious optimization methodology to overcome these difficulties. In particular, our method seeks to isolate the high-level semantics of operations from the organization details of compound data structures and thereby to derive an abstract specification of operations to be applied to the data. The simplified computation can then be accurately analyzed and optimized through varying state-of-the-art compiler technologies. Finally, the optimized operations are combined with the previously isolated implementation details of their data to generate the final optimized code of the original computation.

To evaluate our methodology, we optimized 10 matrix computations using two state-of-the-art source-to-source optimizing compilers: Pluto [Bondhugula et al. 2008] and EPOD [Cui et al. 2011], which include automatic loop parallelization and memory performance optimizations for multicore architectures. The experimental results have confirmed the effectiveness of our approach in optimizing computations that operate on complex data structures and have shown that although the efficiency of the computation could differ when implementing matrices with different layouts, the alternative implementations typically benefit from a common set of optimizations on the operations. Therefore separately optimizing operations and the data layout of a computation can significantly enhance the effectiveness of compiler optimizations compared with the conventional approaches of using a unified representation.

Our contributions include the following.

—We propose a layout-oblivious methodology which promotes a separation of concerns between the abstract semantics of operations to be applied and the organizational details of compound data. We demonstrate the effectiveness of this methodology by optimizing 10 matrix computations using different data layouts. While our implementation currently targets only matrix computations, it can be extended to accommodate other general-purpose data structures in a straightforward fashion.

—We have fully implemented our optimization methodology using the POET language [Yi 2012] and developed a framework so that the implementation can be connected with arbitrary source-to-source compilers. We demonstrate the effectiveness of the framework by integrating it with two state-of-the-art source-to-source compilers, Pluto [Bondhugula et al. 2008] and EPOD [Cui et al. 2011].

—Our methodology includes a symbolic evaluation algorithm which automatically recognizes the high-level semantics of complex data structure operations based on a limited collection of user annotations and thereby to derive a normalized abstract specification of the relevant operations. In sync with the normalization algorithm, a denormalization algorithm is presented to reintegrate the optimized abstract operations with different implementations of complex data structures.

The rest of the article is organized as follows. Section 2 uses an example to demonstrate how complex data layout schemes can effectually paralyze an otherwise capable optimizing compiler. Section 3 shows our overall optimization methodology. Section 4 presents our annotation language for users to describe the intended semantics of their data structures. Section 5 introduces our symbolic evaluation algorithm and the algorithm to combine optimized abstract operations with different organizations of data. Section 6 presents experimental results. Finally, Section 7 discusses related work, and Section 8 concludes.

## 2. A MOTIVATING EXAMPLE

Figure 1 illustrates three special forms of dense matrices commonly used in linear algebra. These matrices require special treatment because they contain a large number of *zero* elements, and the locations of the zero elements are known before runtime. To save space, typically only the nonzero elements are stored for these matrices, as demonstrated by Figure 1(b), whereas the matrices are said to be stored in their *packed* forms. In the following, we examine the *abstract* and packed layout forms for each of the three special types of matrices.

—*Triangular Matrix*. This is where all the nonzero elements are below the diagonal of the matrix, which is known as a lower triangular matrix, or above the diagonal of the matrix, which is known as upper triangular. When stored in the packed layout, since only the nonzero elements of a $n \times n$ triangular matrix need to be saved, only an array of size ($\Sigma_{k=1}^{n} k$ = n*n/2) is required.

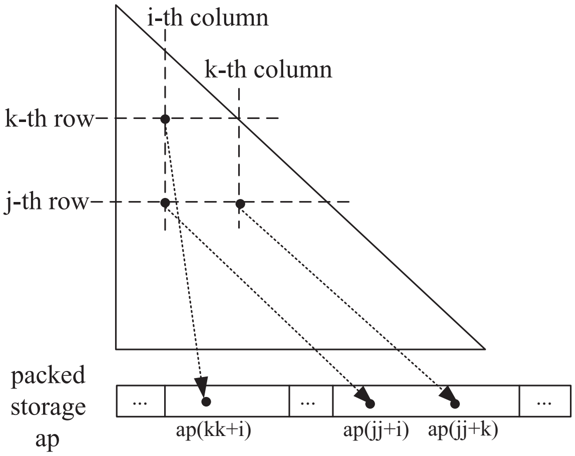
—*Banded Matrix*. This has all the nonzero elements within a narrow band diagonally bordered by two constants $k1, k2 \geqslant 0$ such that $a_{ij} \neq 0$ only if $i-k1 \leqslant j \leqslant i+k2$. When stored in a packed form, a banded matrix requires only an array of size $n\times(k1+k2+1)$, as illustrated in Figure 1(b).

—*Banded Triangular Matrix*. This is a special form of the banded matrix and has all the nonzero elements below or above the main diagonal; i.e., it is a banded matrix with $k1 = 0$ or $k2 = 0$.

The condensed storage formats for the aforesaid special matrix forms can pose overwhelming challenges for compilers to optimize their computations, e.g., to parallelize their operations or reuse their memory references in the cache or the registers. To illustrate such a situation, Figure 2(a) shows a code segment selected from the LINPACK `sppfa` routine, which operates on a packed lower triangular matrix shown in Figure 2(b). Since all the nonzero elements in the lower triangular matrix are

```
jj = 0;
for (j = 0; j < n; j++) {
   kk = 0;
   jj += j;
   for (k = 0; k < j; k++) {
      kk += k;
      for (i = 0; i < k; i++)
         ap[jj+k] = ap[jj+k] - ap[jj+i] * ap[kk+i];
   }
}
```

(a) Example for packed triangular matrix



(b) Access pattern for packed triangular matrix

```
for (j = 0; j < n; j++)
   for (k = 0; k < j; k++)
      for (i = 0; i < k; i++)
         ap(j, k) = ap(j, k) - ap(j, i) * ap(k, i);
```

(c) Mapping to the abstract form
(ap(k,i) for the element in k-th row and i-th column)

Fig. 2. Example of packed matrix access.

condensed into a single one-dimensional array ap, the k-th row of the matrix is mapped to a contiguous region in the array starting at location $\sum_{i=0}^{k} i$ and ending at $\sum_{i=0}^{k} i + k$. In Figure 2(a), the variables $jj$ and *kk* are used to keep track of the starting locations of the rows of the original matrix referenced by loop index variables $j$ and $k$, respectively. The values of $jj$ and *kk* are modified at each iteration of the $j$ and $k$ loops, respectively. Because they have a nonlinear relation with the values of the corresponding loop index variables, no state-of-the-art compiler can accurately determine the relations among array references $ap[jj + k]$, $ap[jj + i]$, and $ap[kk + i]$. As a result, the compiler has to determine that arbitrary elements of the array *ap* could be accessed by each reference and thus refrain from applying any reordering optimizations to the given code.

Although analyzing nonlinear array subscripts is beyond the ability of existing compiler technologies, the code in Figure 2(a) can be effectively optimized by a compiler if it

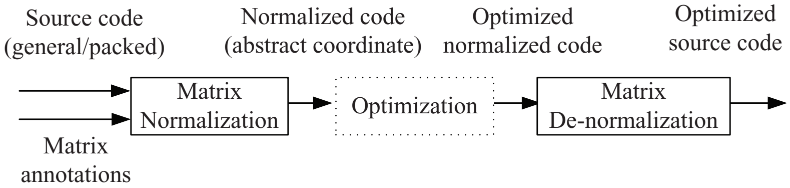


Fig. 3. Overview of optimization approach.

is translated to the form in Figure 2(c), which specifies the intended matrix operations by (a). The conversion, however, requires the domain-specific knowledge that the array *ap* in Figure 2(a) implements the packed storage form of a lower triangular matrix. With such knowledge, a symbolic evaluation algorithm can be applied to translate each access to array *ap* into the intended matrix coordinates, e.g., by mapping the single-dimensional access $ap[kk+i]$ to an abstract matrix access $ap(k, i)$, which indicates that the element at the k-th row and i-th column of the matrix is being accessed. In the following, we present a new optimization methodology which automatically supports the optimization of the code fragment in Figure 2(a) by first converting it to (c) and then converting the optimized code of (c) back into using the original condensed storage format.

## 3. THE OVERALL APPROACH

Figure 3 shows an overview of our approach, which takes two inputs, the computational kernel to be optimized and a set of annotations that specify the internal layout of matrices used in the kernel. It produces optimized code through the following steps.

(1) *Matrix normalization.* This step seeks to isolate the high-level semantics of matrix operations from implementation details of matrices within the input computation, by automatically converting all the relevant array references to a higher-level representation which uses (row,column) coordinates to access each abstract matrix, e.g., by converting the code in Figure 2(a) to Figure 2(c).
(2) *Optimization.* This step applies state-of-the-art compiler optimizations, e.g., loop parallelization, tiling, array copying, unroll-and-jam, and scalar replacement, to abstract matrix operations within the normalized code.
(3) *Matrix denormalization.* This step takes the optimized code from step (2) and modifies all the references of abstract matrix coordinates to instead access the corresponding locations in their original condensed matrix storage forms. In particular, this step can be used to convert the optimized code of Figure 2(c) back to using packed triangular matrices as shown in Figure 2(a).

Within our approach, step (2) (the optimization step) is not bound to any specific compiler and only requires the compiler to be source-to-source, so that its output code could be further modified by step (3). Note that some layout-sensitive optimizations, e.g., SSE vectorization, which requires vectorized array references to be contiguous and properly aligned with cache line boundaries, need to be turned off at step (2) and should be instead applied after the layout-oblivious optimizations and the matrix denormalization step have been completed. In the following, we first introduce our matrix annotation language and then present algorithmic details of automatically normalizing and denormalizing the matrix operations. Since our framework can be used to support arbitrary source-to-source compiler optimizations as long as these optimizations do not have special requirements on the layout of the abstract matrices, details of these optimizations are omitted in this article.

```
(1) Function definition:
    %fun_name(p1,...pn) = e1 if (c)
                            |e2 otherwise;
(2) Type definition:
    @type_name(p1,...,pn) =
                    @matrix : (row:[l..u],col:[l..u]) ⇒ e;
(3) Variable annotation:
    $x : @type_name(e1,...,en);

* p1,...,pn: parameter names;
* e, e1, e2, ..., en: integer expressions;
* c : a boolean expression;
* [l..u] : an integer type with values ≥ l and ≤ u;
* fun_name: the name of a local function;
* type_name: type name of special matrices;
* x: name of a matrix variable in the input code;
```

Fig. 4. Matrix annotation language.

```
/* %lpack_func(k) = 0                      if k==0
                  | k + lpack_func(k-1)    otherwise;
@lower_tri(n) = @matrix(row:[0..n-1],col:[0..i]) ⇒ (lpack_func(row)+col);
$ap:lower_tri(n);*/
jj = 0;
for (j = 0; j < n; j++){
   kk = 0;
   jj += j;
   for (k = 0; k < j; k++) {
      kk += k;
      for (i = 0; i < k; i++)
         ap[jj+k] = ap[jj+k] - ap[jj+i] * ap[kk+i];
   }
}
```

Fig. 5. Example annotations.

## 4. MATRIX ANNOTATIONS

Figure 4 shows our matrix annotation language designed to allow programmers to easily define the intended semantics of data structures in matrix computations, e.g., which variables have matrices as their values, whether the matrix is stored in a packed form, and given the coordinate of a matrix element, how to access its element from the packed form. The abstract syntax of the language is shown in Figure 4, which includes three types of annotations: function definitions, type definitions, and variable annotations. Figure 5 shows some example annotations for the lower triangular matrix computation in Figure 2(a). The following explains each type of annotation in detail.

*Function Definitions.* Shown at line (1) of Figure 4, function definitions are used by our annotation language to express complex nonlinear functions that may be required to annotate access functions of different matrix storage forms. A function definition can be self-recursive or recursive in terms of other functions, where the *if* and *otherwise* keywords can be used to define the base and recursive cases of the function. In Figure 5, the function $lpack\_func(k)$ is defined recursively to compute the starting address of the

$i$th row of a lower triangular matrix and is subsequently used to define the access function of the corresponding matrix type at line (2).

*Type Definitions.* Shown at line (2) of Figure 4, type definitions are used by our annotation language to define a special matrix form and specify how to locate each matrix element at an arbitrary $(row, col)$ coordinate in its packed storage layout. The syntax *@matrix* is a keyword, $(row[l1..u1], col[l2..u2])$ specifies the *coordinate* parameters of an abstract matrix, where $l1/u1$ and $l2/u2$ represent the lower/upper bounds of the row and column dimensions, and $e$ specifies the location (parameterized in terms of $row$ and $col$) of the matrix coordinate in the packed layout. The matrix type could be parameterized, e.g., by the sizes of its dimensions. In Figure 5, *lower_tri(n)* defines a $n*n$ lower triangular matrix type, and its access function is defined to map each abstract matrix coordinate (row,col) to the offset $lpack\_func(row) + col$ of the packed array.

*Variable Annotations.* Shown at line (3) of Figure 4, variable annotations in our language serve to declare additional type information for matrix variables, especially those that use special storage forms, inside a given input code. These matrix annotations can be embedded inside the input code by enclosing them in C language comments if desired, so that they can be ignored by other compilers. For example, the annotation at line (3) of Figure 5 declares the array *ap* to be a lower triangular matrix of size *n*. This annotation can be recognized by our matrix normalization algorithm, presented in Section 5, so that the complex array references of *ap* in the original input code can be converted to abstract matrix operations, shown in Figure 2(c), which are much easier to analyze and optimize by compilers.

## 5. MATRIX NORMALIZATION AND DENORMALIZATION

Based on the matrix annotations supplied by the users, our framework aims to extract the higher-level semantics of the input code through a *matrix normalization* step which converts complex array of references within the input code to simple accesses of abstract matrices using abstract matrix coordinates. The normalized code can then be fed into a source-to-source compiler for optimization. Finally, a denormalization step is applied to convert the compiler optimized code, which continues to reference all matrices in their abstract forms, back to referencing all matrices in their original storage formats. The following explains both the normalization and denormalization steps in detail.

### 5.1. Matrix Normalization

As shown in Algorithm 1, our matrix normalization algorithm takes two parameters: the input code (*input*) and the associated matrix annotations (*matrix_annot*). It aims to convert each array reference $A[r]$, where $A$ is declared as a special matrix in the annotations, to an abstract matrix access in the form of $A(row, col)$ in the input code.

*5.1.1. Algorithm Overview.* To normalize the matrix accesses, our algorithm first invokes symbolic evaluation (line 1 of Algorithm 1) to perform symbolic evaluation of the input code so that for each integer expression contained inside loops, a symbolic value is derived in terms of the index variables of its surrounding loops and other variables that are invariant across the surrounding loops. The computed symbolic values are then stored inside a hash table named *prog_table*. More details about the symbolic evaluation will be discussed in Section 5.1.2.

After the symbolic evaluation for the input program is completed, the algorithm traverses the whole program to normalize each annotated matrix variable ($A$). For each array reference $A[r]$ that needs to be converted to abstract matrix coordinates, the symbolic value of $r$ (line 5) is matched against the matrix access function provided by user annotations (line 6). In particular, line 6 invokes *match_pattern* to determine

**ALGORITHM 1:** Normalizing matrix accesses.

```
procedure Normalize_Matrix(input, matrix_annot)
 1: prog_table = symbolic_evaluation(input,empty_table);
 2: for each A:matrix in matrix_annot do
 3:    access = lookup_access_map(matrix_annot,matrix);
 4:    for each array access A[r] in input do
 5:       r_val = Lookup_symbolic_value(prog_table, r);
 6:       coord = match_pattern(r_val, access);
 7:       if coord $\neq$ empty then
 8:          add (A[r] => A(coord) to repl_A;
 9:       else
10:          repl_A = empty; break out of loop;
11:       end if
12:    end for
13:    if repl_A $\neq$ empty then
14:       apply all replacements in repl_A to input;
15:    end if
16: end for
```

whether the current array reference subscript $r$ shares the designated access pattern of the annotated matrix type of $A$ (more details discussed in Section 5.1.3). If the pattern matching is successful, an instantiated matrix coordinate $coord = (row, col)$ is returned and then used by the algorithm to replace $A[r]$ with $A(coord)$ (lines 7–8).

*5.1.2. The Symbolic Evaluation Algorithm.* Symbolic evaluation refers to the process of tracking the symbolic values of variables within a program by symbolically evaluating expressions along all reachable program paths. The analysis has been widely used in programming tools for debugging, validation, verification, and optimization [Cheatham et al. 1979]. We leverage symbolic evaluation to track the symbolic values of integer expressions while summarizing all the program paths leading to the evaluation of the expression [Cheatham et al. 1979]. The algorithm is shown in Algorithm 2 and takes two parameters, the input code to analyze, and a symbol table which stores the existing result of analysis. It modifies the input symbol table to remember analysis results for all the integer expressions contained in the input code and then returns the modified table as result.

A key idea of Algorithm 2 is to enforce that each variable and each expression in the input code has a unique symbolic value in the current hash table when evaluating an arbitrary expression of the input code. Each symbolic value can be expressed in one of the following forms:

—$c$, a constant value or the name of a global variable that is never modified within the current scope;
—$op(v1, \ldots vn, )$, the result of applying a built-in operation (e.g., $+, -, *$) to a list of symbolic values $v1, \ldots$ vn;
—$if(v1, v2, v3)$, the result of evaluating *if (v1) then v2 else v3*, where $v1$,$v2$, $v3$ are symbolic values;
—$idx_i(l, u, s)$, the value of a loop variable ranging from $l$ to $u$, with increment $s$ at each iteration;
—$Reduction(op, init, idx)$, the value of a variable initialized with *init* and then combined with a loop variable *idx* using a reduction operation *op* (e.g., +,*) at each iteration of the loop.

**ALGORITHM 2:** Symbolic Evaluation

```
procedure symbolic_evaluation(input, table)
  case $input$ is a function definition $f(params, body)$:
    $table2 = symbolic\_evaluation(body, empty\_table)$;
    $table[input] = FunctionDef(params, table2))$;
  case input is a sequence of statements:
    for each $stmt$ in $input$ do
      $table = symbolic\_evaluation(stmt, table)$;
    end for
  case input is an assignment $(lhs = rhs)$:
    $table = symbolic\_evaluation(rhs, table)$;
    $table[lhs] = table[rhs]$;
    if ($lhs$ is a reference to variable $x$) then
      $table[x] = table[lhs]$;
    end if
  case input is a loop $((i = (l, u, s)), body)$:
    $table2 = symbolic\_evaluation(body, (i \Rightarrow idx_i(l, u, s)))$;
    $table[input] = summarize\_loop(table2)$;
  case input is a conditional if (c1) then b1 else b2:
    $table$ = symbolic_evaluation(c1,table);
    $table1$ = symbolic_evaluation(b1,table);
    $table2$ = symbolic_evaluation(b2,table);
    $table[input]$ = summarize_if(c1,table1,table2);
  case input is a built-in operation $op(opd1, \ldots, opdn)$:
    $table = symbolic\_evaluation(opd1, table); \ldots.$
    $table[input] = op(table[opd1], \ldots table[opdn])$;
  case $input$ is a function call $f(args)$:
    $summarize\_function\_call(f, args, table)$;
  case $input$ is a variable reference of $x$:
    $table[input] = table[x]$;
  return $table$;
```

As shown in Algorithm 2, our algorithm recursively evaluates each expression, statement, and control flow structure, by summarizing the new values that they have computed for each memory location. Whenever a new control flow block, e.g., an if-else conditional or a loop, is encountered, a new empty table is created so that all variables not modified by the block can be treated as constants. Values computed by different control flow branches are then summarized into a unique value using the $if$ and $Reduction$ operators. We use the classical reduction recognition algorithm by Pottenger [1995] to recognize special assignment patterns such as $a = a + B(\beta(i, j, k, \ldots))$ and omit the details of their algorithm in this article.

Figure 6 uses an example to demonstrate the process, where (a) shows the input source-code, and (b) shows the symbolic evaluation results. In particular, Figure 6(b) shows that the symbolic value for $ap[jj + i]$ is $ap[Reduction(+, 0, idx_j) + idx_i]$, which matches the access function, $Reduction(+, 0, row) + col$, where $row$ and $col$ are parameters representing the coordinate of an abstract matrix, of the $lower\_packed$ matrix in the user-supplied matrix annotations shown in Figure 7.

*5.1.3. Pattern Matching.* Figure 7 uses an example to illustrate the key steps involved in mapping the array reference $ap[jj + i] = Reduction(+, 0, idx_j) + idx_i$ in Figure 6 to the corresponding abstract matrix access $ap(j, i)$. First, the symbolic value of $ap[jj + i]$ is computed as $Reduction(+, 0, idx_j) + idx_i$, which is matched against

```
/*$ap: lower_pack(n);*/
int packed_mm(float *ap, int n)
{
    jj = 0;
    for (j = 0; j < n; j++)
    {
        kk = 0;
        jj += j;
        for (k = 0; k < j; k++)
        {
            kk += k;
            for (i = 0; i < k; i++)
            ap[jj+k] = ap[jj+k] - ap[jj+i] * ap[kk+i];
        }
    }
}
```

(a) source code

<table>
<tr><td rowspan="8">table1<br>(for pack_mm)</td><td>jj</td><td colspan="4">0</td></tr>
<tr><td>j</td><td colspan="4">$idx_j(0,n,1)$*</td></tr>
<tr><td rowspan="6">table2§<br>(for loop nest j)</td><td>kk</td><td colspan="3">0</td></tr>
<tr><td>jj</td><td colspan="3">Reduction(+,0, $idx_j$) #</td></tr>
<tr><td>k</td><td colspan="3">$idx_k(0,idx_j,1)$</td></tr>
<tr><td rowspan="3">table3<br>(for loop nest k)</td><td>i</td><td colspan="2">$idx_i(0,idx_k,1)$</td></tr>
<tr><td>kk</td><td colspan="2">Reduction(+,0,$idx_k$))</td></tr>
<tr><td>table4<br>(for loop nest i)</td><td>ap[jj+k]</td><td>ap[Reduction(+,0,$idx_j$)+$idx_k$] -<br>ap[Reduction(+,0,$idx_j$)+$idx_i$] *<br>ap[Reduction(+,0,$idx_k$)+$idx_i$]</td></tr>
</table>

(b) symbolic evaluation results

* This *$idx_j$* expression represents the value of a loop variable ranging from *0* to *n*, with increment *1* at each iteration;

§ A new table is created when entering a new scope.

# The *Reduction* expression represents the value of a variable initialized with *0* and then applied with the operation of + at each iteration by a loop variable *$idx_j$*

Fig. 6. Example of symbolic evaluation.

$Reduction(+, 0, row) + col$, the access function of the lower_tri matrix, the matrix type of *ap* declared in the user annotations. Since both expressions contain the same reduction operation, their pattern matching succeeds after matching the two coordinate parameters, $row$ and $col$, with $idx_j$ and $idx_i$ respectively. Then, we can bind the coordinate parameters $(row, col)$ with $(j, i)$ and generate the normalized abstract matrix references, shown in Figure 2(c).

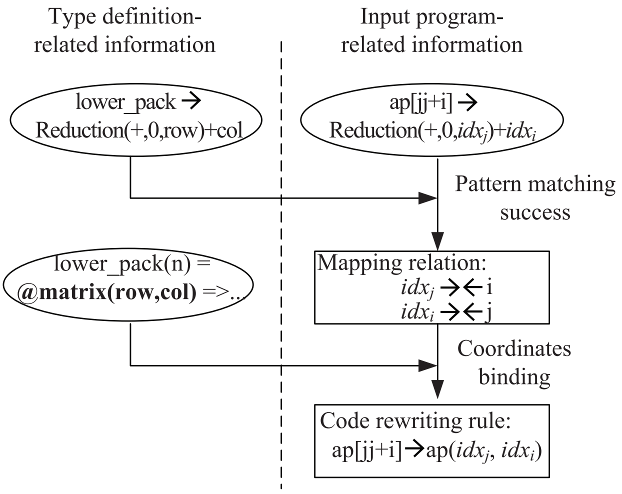

Type definition-related information
Input program-related information
lower_pack → Reduction(+,0,row)+col
ap[jj+i] → Reduction(+,0,$idx_j$)+$idx_i$
Pattern matching success
lower_pack(n) = @matrix(row,col) =>..
Mapping relation:
$idx_j$ →← i
$idx_i$ →← j
Coordinates binding
Code rewriting rule:
ap[jj+i]→ap($idx_j$, $idx_i$)

Fig. 7. Pattern matching example.

### 5.2. Matrix Denormalization

After matrix normalization, the normalized input code is converted into a valid C code by changing the type declarations of all the normalized matrices and then referencing all the converted abstract matrices using two-dimensional array accesses. The converted C code can be fed into an arbitrary C/C++ source-to-source compiler for optimization. Then, the optimized code, still in C syntax, needs to go through another denormalization process so that all the abstract matrix variables are translated back to their original storage formats.

While denormalization is the inverse of matrix normalization, it can be implemented much more easily than its counterpart. The process mostly involves translating a set of higher-level notations, specifically abstract matrix references, to their lower-level implementations based on the access functions specified for each matrix type used in the input code. Since each access function is parameterized by a pair of coordinate variables of the abstract matrix, the process merely involves substituting the coordinate variables with their actual parameters within the access function definition and then using the instantiated expression to substitute the original abstract matrix access in the input code. To translate the *Reduction* operation, which needs a loop to evaluate the desired symbolic value, we generate the required loop nest on-the-fly and then save the desired symbolic value into a temporary scalar variable. The scalar variable is then used to substitute the corresponding *Reduction* operation inside the instantiated matrix access functions.

While converting higher-level abstract matrix notations to concrete lower-level array references, our denormalization step is aware of the runtime cost of varying array subscript calculations and takes care to minimize such cost. For example, for packed triangular matrices, the subscript calculation includes a recursive function which needs to be computed using a loop. Our algorithm generates such loops by evaluating all the loop-invariant expressions outside of the generated loops to eliminate redundancy.

### 5.3. Putting It Together

We have implemented both the matrix normalization and denormalization algorithms using POET [Yi 2012], a scripting language designed to support the program transformation needs of source-to-source optimizing compilers. The normalization process generates syntactically correct C code as output, where each abstract matrix is implemented using a two-dimensional array. The normalized C code can then be optimized by an arbitrary source-to-source C compiler, which generates optimized code while maintaining all the abstract matrix references. Finally, the optimized C code is used as input to our matrix denormalizer, which converts all the abstract matrices back to using their original packed storage forms.

Our algorithms are conservative in that the optimized code is guaranteed to be correct if the user-supplied annotations can be assumed to be correct. Although our symbolic evaluation algorithm is not guaranteed to successfully recognize all forms of referencing a packed matrix, the matrix normalization algorithm will modify a matrix to its abstract form only if all the relevant references can be successfully recognized and converted; otherwise, the original packed storage form will remain, and none of the original matrix references will be modified. Since only matrices annotated by the user will be converted to abstract forms, and all the matrix conversions are strictly based on the user-supplied access functions, the normalized code is guaranteed to have identical semantics as those of the original input code. The correctness of the matrix denormalization step can be argued in a similar fashion.

Our approach is effective in optimizing packed matrix computations where the matrix layout can be easily expressed with our annotation interface. To handle more complex data structures that cannot be easily annotated using our existing interface, our infrastructure may need to be extended significantly. The generality of the approach is limited by two critical pieces within our infrastructure: symbolic evaluation and pattern matching. In our current implementation, these two components are built to specifically target matrix computations. However, symbolic evaluation has been widely used in the area of software engineering and thus proven to be extensible to more general cases. Pattern matching is known to be customizable based on varying needs. Therefore both components can be extended to handle broader application domains, although how to design a more general algorithm is still an open question.

While Section 6 will demonstrate the effectiveness of our approach in optimizing packed matrix computations, it remains a challenge to apply the layout-oblivious optimization scheme to other data structures such as sparse matrices, lists, and trees. We consider the tackling of these issues a subject of our future research.

## 6. EVALUATION

We have connected our matrix normalization and denormalization implementations with two source-to-source parallel optimizing compilers, Pluto [Bondhugula et al. 2008] and EPOD [Cui et al. 2011], both of which include automatic loop parallelization and memory locality optimizations for multicores, and have applied them to optimize 10 matrix computation kernels shown in Table I. Both Pluto and EPOD support loop parallelization by automatically generating OpenMP pragmas. For the `mm` and `mv` kernels, we studied three nonrectangular matrix types, triangular, banded, and banded triangular, each using two different matrix storage formats: abstract and packed layouts (see Figure 1). For the `sm` and `sv` kernels, banded matrices are not used as both solvers require that their input matrices must be triangular.

Our evaluation aims to verify that our approach is indeed effective in optimizing matrix computations where nonrectangular matrices are stored in complex packed storage forms. The following first introduces our experimental design and then presents our results of using both the Pluto and EPOD optimizing compilers to optimize the 10 benchmarks in Table I. Finally, Section 6.4 summarizes our experimental results.

Table I. Benchmarks

| Computation | MatrixType | Routine | Layout |
|---|---|---|---|
| matrix matrix multiply<br>$C = A \times B$ | triangular | mm.tri | abstract |
| | | | packed |
| | banded | mm.band | abstract |
| | | | packed |
| | banded<br>triangular | mm.bandtri | abstract |
| | | | packed |
| matrix vector multiply<br>$Y = A \times X$ | triangular | mv.tri | abstract |
| | | | packed |
| | banded | mv.band | abstract |
| | | | packed |
| | banded<br>triangular | mv.bandtri | abstract |
| | | | packed |
| matrix matrix solver $A \times X = B$ | triangular | sm.tri | abstract |
| | | | packed |
| | banded<br>triangular | sm.bandtri | abstract |
| | | | packed |
| matrix vector solver $A \times X = Y$ | triangular | sv.tri | abstract |
| | | | packed |
| | banded<br>triangular | sv.bandtri | abstract |
| | | | packed |

### 6.1. Experiment Design

We have used the Pluto [Bondhugula et al. 2008] and the EPOD [Cui et al. 2011] source-to-source C compilers to optimize all the 10 benchmarks in Table I, with each benchmark implemented alternatively using two different matrix layouts: abstract and packed, and each benchmark optimized with or without first invoking the matrix normalization component. The Pluto compiler uses the polyhedral model to analyze the loop iteration space of its input code and then applies three loop optimizations: parallelization, two-level loop tiling, and unroll-and-jam. EPOD is a pattern-based optimizer, where we have written a EPOD script to define the desired optimization sequence for each computation pattern. In particular, the 10 benchmarks have been grouped into four EPOD patterns: `mm`, `mv`, `sm`, `sv`. The EPOD script for `mm` includes loop parallelization, 2-level loop tiling, array copying, unroll-and-jam, loop unrolling, and SSE vectorization.

We used the `icc` compiler version 11.0 to compile all of our benchmarks to machine code with the `-openmp` and `-fast` command-line options, which instruct the `icc` compiler to apply processor-specific optimizations (e.g., SSE3 vectorization) based on `-O3`. We evaluated the `mm` and `sm` benchmarks using randomly initialized 2048*2048 matrices, and evaluated the `mv` and `sv` benchmarks using a $10240 * 10240$ matrix and a vector of size 10240. We have run all of our experiments on an 8-core Intel platform with two 2.33 GHz quad-core Xeon 5410 processors. Each processor has 32KB L1 data cache per core, 32KB L1 instruction cache per core, and 6MB L2 unified data/instruction cache shared by all cores. All the experiments are repeated 5 times, and the average GFLOPS rates attained across the 5 runs are reported as the final results. Variations among different runs are minor, as the underlying machine was otherwise idle when running our experiments.

### 6.2. Optimizing Results Using Pluto

Figure 8 shows the performance (GFLOPS) attained when using `Pluto` and `icc` combined with our layout-oblivious framework to optimize the `mm` and `sm` benchmarks, and

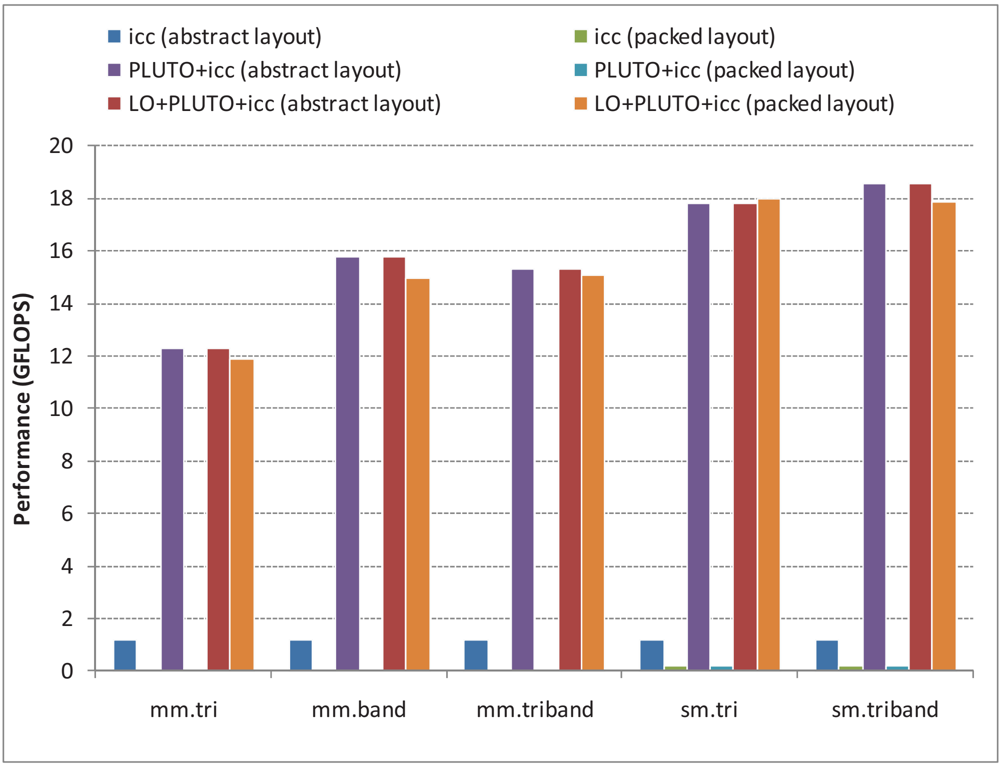


Fig. 8. Performance of the mm and sm benchmarks using our framework + PLUTO.

Figure 9 shows the performance (GFLOPS) attained when optimizing the mv and sv benchmarks. For each benchmark, icc(abstract layout)/(packed layout) stand for performance attained when using icc alone to compile the original source-code with the respective matrix layouts; PLUTO+icc(abstract layout)/(packed layout) stand for the performance attained when using both Pluto and icc to optimize the benchmarks *without* invoking our matrix normalizer; LO+PLUTO+icc (abstract)/(packed) stand for the performance attained when using Pluto and icc combined with our matrix normalizer and denormalizer for both matrix layouts.

From Figure 8, for the mm and sm matrix kernels, the performance attained by icc(abstract) icc(packed) and PLUTO+icc(packed) is extremely poor (lower than 0.1GFLOPS) compared to that achieved by PLUTO+icc (abstract). A key observation for the poor performance is that the packed layout prevents PLUTO from applying loop tiling. However, after applying matrix normalization to the packed matrices, the LO+PLUTO+icc versions have attained a similar level of performance not only for abstract matrix layouts, but also for packed matrices, due to the effective application of loop tiling and other optimizations. The icc compiler did not attain a reasonable performance for the mm and sm kernels even when using abstract matrix layout, as it fails to apply loop tiling to the nontriangular loop nest. A similar pattern can be observed for the mv and sv kernels, although the performance difference is less dramatic, as shown in Figure 9.

From Figure 8 and Figure 9, our layout-oblivious approach was able to significantly enhance the effectiveness of the Pluto compiler when optimizing computations that use packed matrix layouts. Also note that when using the packed matrix layouts,

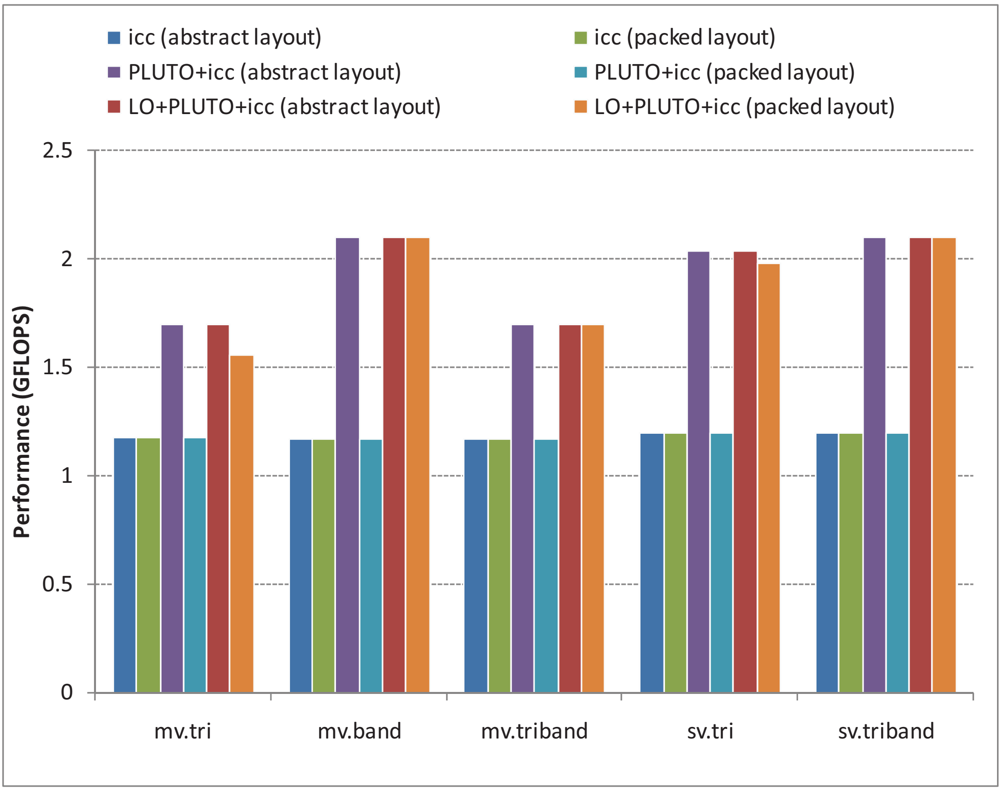


Fig. 9. Performance of the mv and sv benchmarks using our framework + PLUTO.

the optimized computations have comparable performance to that of the same kernel when using abstract layout, with under 3% slowdown in all cases. This indicates that the optimizations applied by Pluto are not sensitive to the underlying layouts of the matrices, and a common set of optimizations should be applied to the same computation irrespective of what layout forms are used to implement the matrices.

### 6.3. Optimizing Results Using EPOD

Figure 10 and Figure 11 show the performance results when using our approach together with EPOD to optimize the benchmarks. Here for each benchmark, the notations of different optimization configurations have the same meaning as Figure 8 and Figure 9, except that EPOD is used in place of PLuto.

From Figure 10 and Figure 11, almost identical patterns can be observed as in Figure 8 and Figure 9, although EPOD has attained a higher performance level for the mm and sm kernels when effective but attained slightly lower performance for the mv and sv kernels. Again the complex subscripted references within loops when using the packed matrix layout have overwhelmed the EPOD pattern recognition algorithm and thus disabled its optimizations. Our approach solves this problem and enables alternative implementations of the same matrix computation to benefit from a common set of optimizations.

The EPOD script for mm includes loop parallelization, 2-level loop tiling, array copying, unroll-and-jam, loop unrolling, and SSE vectorization. Note that this pattern-oriented

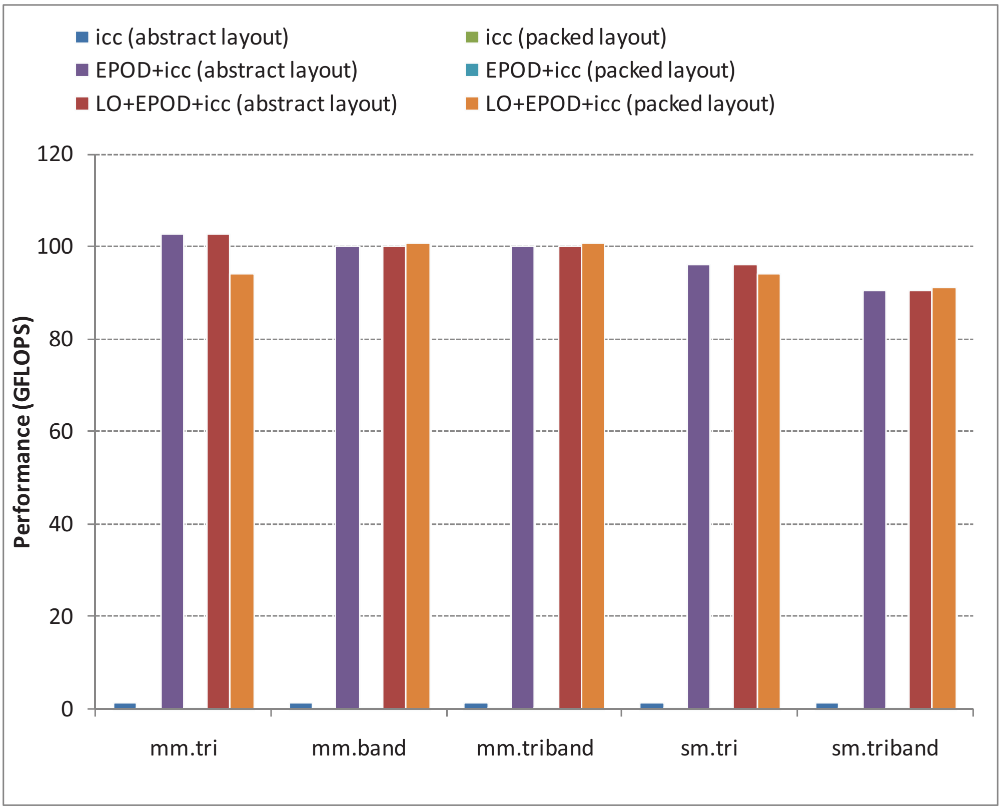


Fig. 10. Performance of the mm and sm benchmarks using our framework + EPOD.

optimization sequence includes a special step “array copying”, which copies small data blocks from the normalized abstract matrix to a preallocated space so that data within these blocks are stored contiguously in memory, as illustrated in Figure 12. This step essentially reorganizes the matrices in a block-major format [Whaley et al. 2001], thus allowing SSE vectorization to be applied to the matrix computations even for packed form matrices and turning SSE vectorization from a layout-sensitive optimization into layout-oblivious. As a result, in Figure 10 and Figure 11, we are able to achieve similar performance for both packed and abstract matrices when using the “LO+EPOD+icc” optimization configuration.

### 6.4. Summary Results

From our performance study, we make the following observations about our optimization framework.

—Computational kernels that cannot be effectively analyzed by existing optimizing compilers due to complex memory access patterns can benefit from our framework by having their data accesses normalized to a standard form and thus become optimizable.

—When a single algorithm is implemented using different data structures, the alternative implementations can often benefit from a common set of optimizations on the operations, irrespective of the different data layout schemes being used.

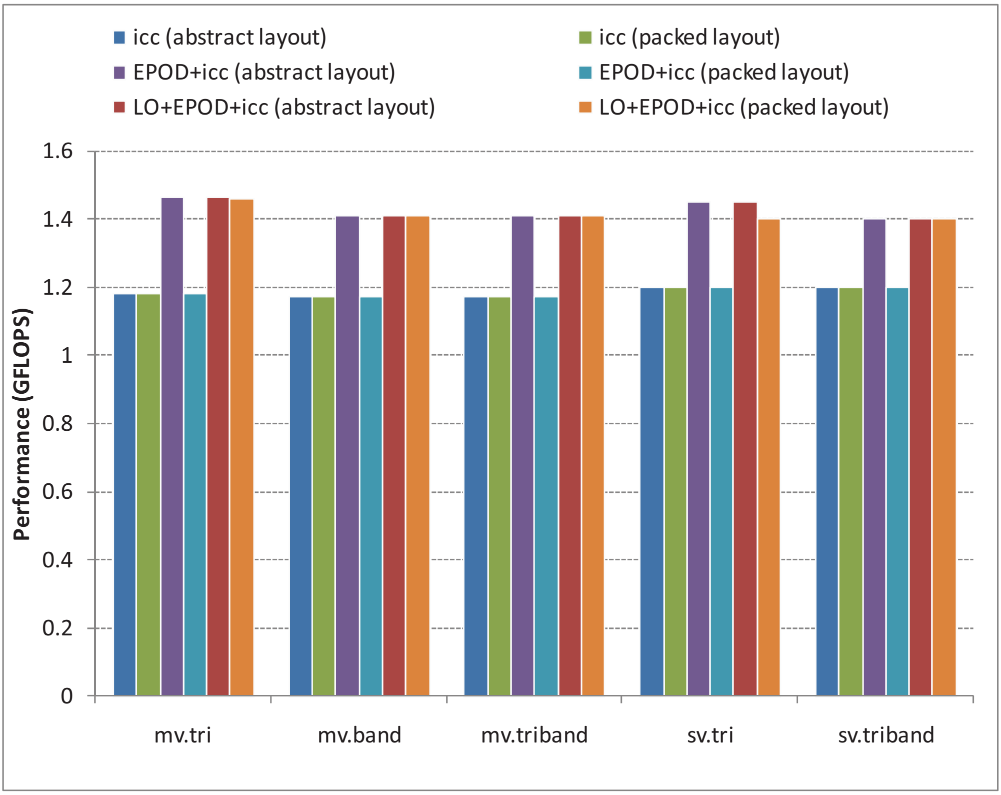


Fig. 11. Performance of the `mv` and `sv` benchmarks using our framework + EPOD.

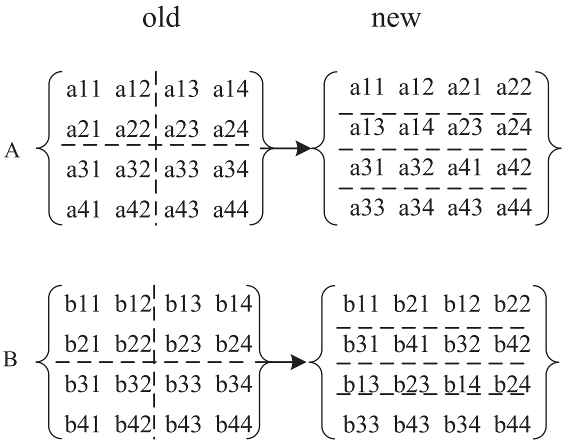


Fig. 12. Applying the `array copying` optimization to the `mm` kernel using EPOD.

—Our framework is not specific to any optimizing compiler. Although the impact of optimization may vary when using different optimizing compilers, our framework can be used to enhance the effectiveness of their optimizations by allowing them to be oblivious of the varying data structure implementations.

## 7. RELATED WORK

Dense matrix kernels belong to an important domain of scientific computing, where the BLAS and LINPACK libraries [Dongarra et al. 1990] provide standard implementations of basic vector and matrix operations. The performance of matrix computation kernels has been aggressively tuned both manually [Hu et al. 2006; Volkov and Demmel 2008] and automatically via frameworks such as PHiPAC [Bilmes et al. 1997] and ATLAS [Whaley et al. 2001].

Recursive algorithms have been used to exploit special optimization opportunities for triangular matrices stored in a packed layout [Anderson et al. 2001; Gustavson et al. 2008; Elmroth et al. 2004]. Agarwal et al. developed a parallel blocking algorithm for banded matrix stored in packed layout [Agarwal et al. 1995]. However, this body of work mostly focuses on manual algorithm optimizations for special matrix layouts, and the techniques have not been effectively integrated into optimizing compilers.

Compiler researchers have developed effective techniques to automatically optimize dense matrix computations [Chen et al. 2005; Cui et al. 2011; Ding et al. 2011; Yi 2011; Xue 2000; Xue et al. 2005], which are known to benefit from a large number of loop optimizations commonly integrated within optimizing compilers either through a pure dependence-based approach [Allen and Kennedy 2001; Wolfe 1989] or through more general integer programming models such as the Polyhedral framework [Bondhugula et al. 2008]. All of these frameworks, however, will fail when analyzing subscripted array references that are nonlinear to the index variables of the surrounding loops, as demonstrated by Figure 2. Our layout-oblivious approach aims to complement these existing compiler optimizations by rendering them insensitive to the complex internal implementations of data structures used in their input applications.

Pingali et al. [2011] introduced a *data-centric* formulation of algorithms, called the operator formulation, to express an algorithm in terms of action and data structures. With the data-centric abstraction, they introduced Tao-analysis which can unify seemingly unrelated parallelization techniques. From the perspective of programming, it is natural to support the separation of concerns in the design and development of data structures and algorithms [Sitaraman et al. 2000]. However, such separation of concerns has not been leveraged by compiler optimization techniques, which our approach aims to accomplish.

A number of compilers, e.g., Broadway [Guyer and Lin 2005] and DyC [Grant et al. 2000], used *annotation languages* to guide dynamic compilation and domain-specific optimizations of scientific codes [Wu et al. 1999; Guo et al. 2011]. Our use of developer-supplied annotations to summarize the semantics of different matrix storage formats serve to provide compilers additional domain-specific knowledge about user applications.

Symbolic evaluation, also called symbolic execution, is a case of abstract interpretation and has been widely used in programming tools for debugging, validation, verification, and optimization [Cheatham et al. 1979; Clarke 1976; Pasareanu and Visser 2009]. It analyzes programs by tracking symbolic rather than actual values and representing values of variables as symbolic expressions, thus allowing the output values to be expressed as functions of the input symbolic values. Our symbolic evaluation algorithm is an application of the existing algorithms and can similarly be used for other purposes as well.

## 8. CONCLUSION AND FUTURE WORK

This article presents a data-layout-oblivious optimization methodology to significantly enhance the effectiveness of compiler optimizations when encountering complex data access patterns that cannot be deciphered by state-of-the-art dependence analysis techniques. By isolating the implementation details of data structures and documenting their semantics through user-supplied annotations, we present a normalization algorithm which can automatically extract a higher-level representation of the computations, which can then be readily understood and optimized by compilers. Our methodology can be used to significantly enhance the effectiveness of arbitrary source-to-source optimizing compilers, as long as the optimized source-code maintains the relevant abstract data references generated by our normalization algorithm, where a denormalization process can be applied to automatically convert the abstract data structures back to using their original implementations.

We have applied our methodology to optimize 10 matrix computation kernels using two source-to-source optimizing compilers. Our results show that separately optimizing the operations and the data layout of a computation could significantly enhance the effectiveness of compiler optimizations when compared with the conventional approaches of using a unified representation.

Our future work will seek to extend our methodology to optimize matrix computations within large-scale scientific applications and to extend its applicability to other application domains beyond dense matrix computations. For example, we will seek to support sparse matrices using our annotation language and will seek to apply the layout-oblivious optimization methodology to other general-purpose data structures, e.g., lists, trees, etc.

## REFERENCES

Agarwal, R. C., Gustavson, F. G., Joshi, M. V., and Zubair, M. 1995. A scalable parallel block algorithm for band cholesky factorization. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing (PPSC'95).* 430–435.

Allen, R. and Kennedy, K. 2001. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann.

Anderson, B. S., Wasniewski, J., and Gustavson, F. G. 2001. A recursive formulation of cholesky factorization of a matrix in packed storage. *IEEE Trans. Math. Softw. 27,* 214–244.

Bilmes, J., Asanovic, K., Chin, C., and Demmel, J. 1997. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *Proceedings of the International Conference on Supercomputing (ICS'97).*

Bondhugula, U., Hartono, A., Ramanujan, J., and Sadayappan, P. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08).*

Cheatham, T. E., Holloway, J. G. H., and Townley, H. A. 1979. Symbolic evaluation and the analysis of programs. *IEEE Trans. Softw. Engin. 5*, 4.

Chen, C., Chame, J., and Hall, M. W. 2005. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the Annual IEEE/ACM Symposium on Code Generation and Optimization (CGO'05).*

Clarke, L. A. 1976. A program testing system. In *Proceedings of the Annual Conference.*

Cui, H., Wang, L., Xue, J., Yang, Y., and Feng, X. 2011a. Automatic library generation for blas3 on gpus. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'11).*

Cui, H., Xue, J., Wang, L., Yang, Y., Feng, X., and Fan, D. 2011b. Extendable pattern-oriented optimization directives. In *Proceedings of the Annual IEEE/ACM Symposium on Code Generation and Optimization (CGO'11).*

Ding, X., Wang, K., and Zhang, X. 2011. Ulcc: A user-level facility for optimizing shared cache performance on multicores. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'11).*

Dongarra, J., Croz, J. D., Duff, I., and Hammarling, S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1–17.

ELMROTH, E., GUSTAVSON, F. G., JONSSON, I., AND KAGSTROM, B. 2004. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev.* 3–45.

GRANT, B., MOCK, M., PHILIPOSE, M., CHAMBERS, C., AND EGGERS, S. J. 2000. DyC: An expressive annotation-directed dynamic compiler for c. *Theor. Comput. Sci. 248,* 1–2, 147–199.

GUO, J., STILES, M., YI, Q., AND PSARRIS, K. 2011. Enhancing the role of inlining in effective interprocedural parallelization. In *Proceedings of the International Conference on Parallel Processing (ICPP'11).*

GUSTAVSON, F. G., WASNIEWSKI, J., AND DONGARRA, J. J. 2008. Rectangular full packed format for choleskys algorithm: Factorization, solution and inversion. Tech. rep., LAPACK working note 199.

GUYER, S. Z. AND LIN, C. 2005. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proc. IEEE 93*, 2, 342–357.

HU, Z., CUVILLO, J., ZHU, W., AND GAO, G. R. 2006. Optimization of dense matrix multiplication on ibm Cyclops-64: Challenges and experiences. In *Proceedings of the International European Conference on Parallel Processing (Euro-Par'06).*

MULLHAUPT, A. AND RIEDEL, K. 2001. Banded matrix fraction representation of triangular input normal pairs. *IEEE Trans. Autom. Control 46*, 12.

PASAREANU, C. S. AND VISSER, W. 2009. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transfer 11*, 4.

PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M. A., KALEEM, R., LEE, T., LENHARTH, A., MANEVICH, R., M'ENDEZ-LOJO, M., PROUNTZOS, D., AND SUI, X. 2011. The tao of parallelism in algorithms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11).*

POTTENGER, W. M. 1995. Induction variable substitution and reduction recognition in the Polaris parallelizing compiler. Tech. rep.

SITARAMAN, M., WEIDE, B. W., LONG, T. J., AND ODGEN, W. F. 2000. A data abstraction alternative to data structure/algorithm modularization. In *International Seminar on Generic Programming.*

VOLKOV, V. AND DEMMEL, J. 2008. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the Conference on Supercomputing (SC'08).*

WHALEY, R. C., PETITET, A., AND DONGARRA, J. 2001. Automated empirical optimizations of software and the atlas project. *Parallel Comput. 27*, 3–35.

WOLFE, M. J. 1989. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA.

WU, P., MOREIRA, J. E., MIDKIFF, S. P., GUPTA, M., AND PADUA, D. A. 1999. Semantic inlining: The compiler support for java in technical computing. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing (PPSC'99).*

XUE, J. 2000. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Boston, MA.

XUE, J., HUANG, Q., AND GUO, M. 2005. Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences. In *Proceedings of the International Conference on Parallel Processing (ICPP'05).* 107–115.

YI, Q. 2011. Automated programmable control and parameterization of compiler optimizations. In *Proceedings of the Annual IEEE/ACM Symposium on Code Generation and Optimization (CGO'11).*

YI, Q. 2012. Poet: A scripting language for applying parameterized source-to-source program transformations. *Softw. Pract. Exper.* 675–706.

YOTOV, K., LI, X., REN, G., CIBULSKIS, M., DEJONG, G., GARZARAN, M., PADUA, D. A., PINGALI, K., STODGHILL, P., AND WU, P. 2003. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03).*

Received June 2012; revised September 2012; accepted October 2012