A Basic Linear Algebra Compiler for Structured Matrices

Daniele G. Spampinato Markus Püschel

Department of Computer Science
ETH Zurich
{danieles, pueschel}@inf.ethz.ch

Abstract

Many problems in science and engineering are in practice modeled and solved through matrix computations. Often, the matrices involved have structure such as symmetric or triangular, which reduces the operations count needed to perform the computation. For example, dense linear systems of equations are solved by first converting to triangular form and optimization problems may yield matrices with any kind of structure. The well-known BLAS (basic linear algebra subroutine) interface provides a small set of structured matrix computations, chosen to serve a certain set of higher level functions (LAPACK). However, if a user encounters a computation or structure that is not supported, she loses the benefits of the structure and chooses a generic library. In this paper, we address this problem by providing a compiler that translates a given basic linear algebra computation on structured matrices into optimized C code, optionally vectorized with intrinsics. Our work combines prior work on the Spiral-like LGen compiler with techniques from polyhedral compilation to mathematically capture matrix structures. In the paper we consider upper/lower triangular and symmetric matrices but the approach is extensible to a much larger set including blocked structures. We run experiments on a modern Intel platform against the Intel MKL library and a baseline implementation showing competitive performance results for both BLAS and non-BLAS functionalities.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code Generation, Compilers, Optimization; G.4 [*Mathematical Software*]: Parallel and Vector Implementations, Portability

Keywords Program synthesis, Basic linear algebra, Structured matrices, DSL, Tiling, SIMD vectorization

CGO'16, March 12–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-3778-6/16/03...$15.00
http://dx.doi.org/10.1145/2854038.2854060

1. Introduction

Linear algebra computations are crucial components in many performance-critical algorithms in scientific computing, graphics, communication, control, machine learning, and other areas. For large scale dense linear algebra, high-performance software exists, usually built around the basic linear algebra subroutines (BLAS) interface [6] and LAPACK [2] or similar libraries [23]. Some shortcomings however exist. First, many computations cannot always be directly mapped to existing library functions; second, the small size computations needed in many applications are often not as optimized; third, fixed input size computations (and their potential or smaller and faster code) are usually not supported by specialized functions.

To address this problem, [21] proposed LGen, a program generator for basic linear algebra computations (BLACs). These perform fixed-size computations on matrices, vectors, and scalars using product, sum, transposition, and scalar product. LGen translates a BLAC into highly optimized C code and implements an extensible approach to generating code for vector instruction set architectures (ISAs). The generated code showed competitive performance [21]. Internally, LGen uses an approach similar to Spiral [19, 20] by using multiple stages of domain-specific languages (DSLs) to perform optimizations at the right level of abstraction.

Contributions. In this paper we extend LGen to support BLACs with structured matrices. Specifically,

- We propose a methodology for the generation of optimized code for small scale BLACs with structured matrices (sBLACs). The approach combines LGen's internal DSLs with ideas from polyhedral compilation. The methodology is extensible to include a large set of possible matrix structures; In this work, we use lower/upper triangular and symmetric as prototypical examples.
- We implemented the methodology in the LGen framework to exploit redundancy and remove unnecessary computations, ensuring compatibility with the generation of vector code. The artifact is available at [1].
- We show benchmarks of LGen-generated code with Intel MKL and as baseline naïve code compiled with the Intel compiler.

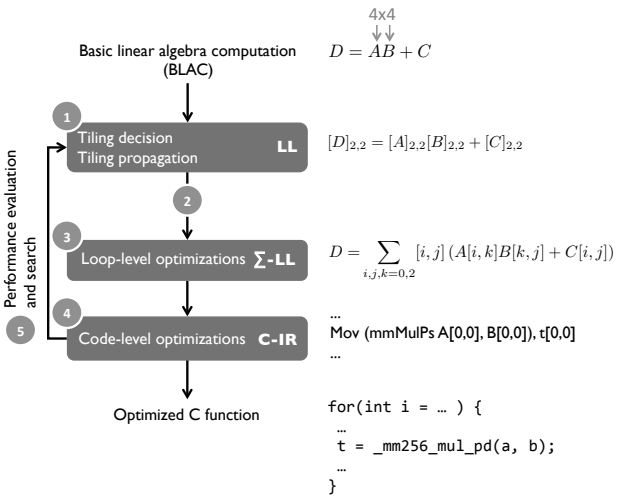


Figure 1. Architecture of LGen [21].

2. Background: LGen

We introduce notation and provide a brief overview on the original LGen. More details can be found in [21].

Basic linear algebra notation. In the following, we denote matrices with $A, B, ...$, (column) vectors with $x, y, ...$, and scalars with $\alpha, \beta, ...$ A basic linear algebra computation (BLAC) on these computes a single output using product, addition, transposition, and scalar product. Examples include $y = A^T x + \alpha z$ or $C = \alpha AB + C + D$.

For the purpose of this paper we introduce structured matrices as addition to the type and thus expand the notion of BLACs to sBLACs. Specifically, we consider lower/upper triangular, symmetric, and all-zero matrices and denote the associated types with $\mathcal{L}, \mathcal{U}, \mathcal{S}$, and $\mathcal{Z}$. For convenience we will often denote such matrices with L, U, S, and Z. An unstructured matrix is of type $\mathcal{G}$. Finally, we also expand the set of operators with the triangular solve written as $x = L\backslash y$.

Program generation with LGen. For a given BLAC, LGen assumes that the operands have fixed (and compatible) sizes and data types (float or double) and generates optimized C code, optionally vectorized using intrinsics. Its input language is called LL (linear algebra language) and does not accommodate structured operands. The main five steps of its generation flow are shown in Fig. 1 and are briefly described next. We use as example

$$D = AB + C, \qquad A, B, C, D \in \mathbb{R}^{4\times 4}. \tag{1}$$

Step 1: Tiling in LL. The first step formally tiles the BLAC recursively with fixed parameters (a degree of freedom; perfect divisibility is not required) and propagates the tiling decision to the operands. If code for a ν-way vector ISA is desired, the lowest level block size has to be ν to decompose the computation into pieces, called ν-BLACs, that can be mapped well to vector code. In our example, we consider only one level of tiling with $\nu = 2$:

$$[D = AB + C]_{\nu,\nu} \overset{\nu=2}{=} [D]_{2,2} = [A]_{2,2}[B]_{2,2} + [C]_{2,2}. \tag{2}$$

Step 2: From LL to Σ-LL. The second step takes the fully tiled BLAC in LL and rewrites it into a second DSL called Σ-LL. This representation is still mathematical and makes loops and data accesses explicit. The latter are captured as explicit gather and scatter operators (functions) on matrices to allow for reasoning and fusion through rewriting. A gather g extracts a smaller matrix from a matrix, a scatter s writes a smaller matrix into a larger all-zero matrix. Formally,

$$g = [i,j]_{k,\ell}^{m,n} : \quad \mathbb{R}^{m\times n} \to \mathbb{R}^{k\times \ell}, \\ A \mapsto Ag = A[i : i+k-1, j : j+\ell-1],$$

where the latter is Matlab notation. Note that we write the function on the right (as common for indexing) because gathers operate from the right. Indeed, if $g' = [i',j']_{u,v}^{k,\ell}$ and $gg' = [i,j]_{k,\ell}^{m,n}[i',j']_{u,v}^{k,\ell} = [i+i', j+j']_{u,v}^{m,n}$, then $(Ag)g' = A(gg')$.

The scatter is the dual of the gather:

$$s = {}_{m,n}^{k,\ell}[i,j] : \quad \mathbb{R}^{k\times \ell} \to \mathbb{R}^{m\times n},\ A \mapsto sA,$$

where $B = sA$ is defined through $B[i : i+k-1, j : j+\ell-1] = A$ and B is zero elsewhere. In this case $s'(sA) = (s's)A$ for the natural definition of $s's$. We will often omit the domain and range parameters to simplify representation.

In our example, from (2) we would obtain the following Σ-LL BLAC:

$$D = \sum_{i,j=0,2} {}_{4,4}^{2,2}[i,j]\,(\\ \left(\sum_{l,r,k=0,2} {}_{4,4}^{2,2}[l,r] \left(A[l,k]_{2,2}^{4,4} B[k,r]_{2,2}^{4,4} \right) \right) [i,j]_{2,2}^{4,4} \\ + C[i,j]_{2,2}^{4,4} \\ ). \tag{3}$$

Step 3: Loop transformations. At this step a Σ-LL BLAC can be transformed by manipulating summations, gathers, and scatters. In the final code this would correspond, e.g., to loop fusions or loop exchange. For example, starting from (3) we can fuse loops by distributing the first gather $[i,j]_{2,2}^{4,4}$ (second line) over the innermost summation to obtain

$$D = \sum_{i,j=0,2} [i,j] \left(\sum_{k=0,2} A[i,k]B[k,j] + C[i,j] \right). \tag{4}$$

Step 4: From Σ-LL to C-IR. At this point the Σ-LL representation in (4) has the following features: (a) the number of summations and their order is defined, (b) (if

```
A = Matrix(4, 4);     L = LowerTriangular(4);
S = Symmetric(L, 4); U = UpperTriangular(4);

A = L*U+S;
```

Table 1. LL implementation of BLAC (5).

vector ISA) the entire formulation is decomposed into ν-BLACs, meaning that all operations are performed on ν-tiles of the input matrices. The translation between Σ-LL and C-IR (LGen's C-like IR) is performed by mapping summations to loops, ν-BLACs to codelets, and gathers and scatters to data accesses. ν-BLACs are the 18 single-operation BLACs that operate on tiles of size $\nu \times \nu, 1 \times \nu, \nu \times 1$ with the four BLAC operators [21]. They are preimplemented once for every vector ISA. The gathers and scatters are associated to a collection of vectorized data access basic blocks called Loaders and Storers, that are used to perform low level optimizations including handling leftovers [17].

Step 5: Performance test and autotuning. Finally, LGen unparses the C-IR into vectorized C code and tests its performance. Autotuning is used to find a good result among available variants.

Introducing structures. Assume now the goal of generating code for the sBLAC

$$A = LU + S, \qquad A, L, U, S \in \mathbb{R}^{4\times 4}, \tag{5}$$

which is analogous to the generic (1). By going through the same process, LGen would produce the expression

$$A = \sum_{i,j=0,2} [i,j] \left(\sum_{k=0,2} L[i,k]U[k,j] + S[i,j] \right). \tag{6}$$

It is easy to notice that certain computations are redundant, e.g., $L[0,2]U[2,j]$ and $L[2,2]U[2,0]$. Also, the standard for symmetric matrices stores only one side of the matrix, e.g., the lower one. In this case, access to $S[0,2]$ should be replaced with $S[2,0]^T$. In the following sections we will show how to perform these analyses and transformations with LGen. We start with explaining our approach for describing structures.

3. Structured Matrices

In this section we discuss how structures are defined in LGen. The approach is designed to be extensible: adding a new structure to LGen requires the inclusion of two different interfaces, one towards the user and one towards LGen.

From a user perspective a structured matrix is just another type of matrix within an LL input program. For example, Table 1 shows a simple LL implementation of (5).

However a structured matrix also needs an internal interface to LGen to enable its decomposition in Σ-LL. We build this interface using the integer set library (isl) formalism from [25].

Polyhedral sets and maps. The two essential concepts used in our definition of structures are sets and relations (called maps in [25]) of n-tuples of integers bounded by m affine constraints. A set of such n-tuples is defined as

$$\sigma = \cup_i \{t \in \mathbb{Z}^n | \exists c \in \mathbb{Z}^e : A_i t + E_i c + z_i \geq 0\}, \tag{7}$$

where $A_i \in \mathbb{Z}^{m\times n}$, $E_i \in \mathbb{Z}^{m\times e}$, $z_i \in \mathbb{Z}^m$, and $\geq$ is componentwise. The existential quantifier allows us to identify tuples at a stride. For example, the following sets

$$\begin{aligned} \sigma_1 &= \{(i,j) \mid 0 \leq i < 4 \wedge 0 \leq j < 4\}, \\ \sigma_2 &= \{(i,j) \mid \exists a,b : 0 \leq i,j < 4 \wedge i = 2a \wedge j = 2b\} \end{aligned} \tag{8}$$

can be used to represent all integer points in a square of size 4×4 (σ_1) or those at a stride 2 (σ_2). The first set would be given by

$$A_0 = \begin{bmatrix} 1 & \\ -1 & 1 \\ & -1 \end{bmatrix}, \; E_0 = 0, \; z_0 = (0\; 0\; 3\; 3)^T.$$

The second set requires the inclusion of $i - 2a \geq 0,\; i - 2a \leq 0,\; j - 2b \geq 0$, and $j - 2b \leq 0$.

Maps in [25] are relations between sets and defined as:

$$\begin{aligned} \rho = \cup_i \{(t_0, t_1) \in \mathbb{Z}^{n_0} \times \mathbb{Z}^{n_1} \mid \\ \exists c \in \mathbb{Z}^e : A_i t_0 + B_i t_1 + E_i c + z_i \geq 0\}. \end{aligned}$$

We will use polyhedral sets to represent regions in matrices and iteration spaces of computations and polyhedral maps to represent access patterns of matrices and reorder iteration spaces.

Internal representation of structures. We associate every matrix with a pair of dictionaries called `SInfo` and `AInfo`.

`SInfo` associates regions of a matrix to structures. Its entries have the form $\mathcal{M} : \sigma$. For example, a matrix of type $\mathcal{L}$ has the following `SInfo`:

$$\texttt{L.SInfo} = \begin{Bmatrix} \mathcal{G} : \{(i,j) \mid 0 \leq i < 4 \wedge 0 \leq j \leq i\} \\ \mathcal{Z} : \{(i,j) \mid 0 \leq i < 4 \wedge i < j < 4\} \end{Bmatrix}.$$

This means that every scalar element in region $\texttt{L.SInfo}[\mathcal{G}]$ has general structure, while every element in $\texttt{L.SInfo}[\mathcal{Z}]$ has a zero structure. Note that this method allows the definition of blocked structures (e.g., the top left quadrant is symmetric), which appear in several applications.

Similarly we define the `SInfo` dictionaries of A, U, and S:

$$\begin{aligned} \texttt{U.SInfo} &= \begin{Bmatrix} \mathcal{G} : \{(i,j) \mid 0 \leq i < 4 \wedge i \leq j < 4\} \\ \mathcal{Z} : \{(i,j) \mid 0 \leq i < 4 \wedge 0 \leq j < i\} \end{Bmatrix}, \\ \texttt{A.SInfo} &= \texttt{S.SInfo} = \{\mathcal{G} : \{(i,j) \mid 0 \leq i,j < 4\}\}. \end{aligned}$$

`AInfo` associates regions of a matrix to information on how to access blocks in that region. Entries for `AInfo` have the general form

$$\sigma : (g : \mathbb{R}^{m\times n} \mapsto \mathbb{R}^{r\times c}, p : \mathbb{R}^{r\times c} \mapsto \mathbb{R}^{r\times c}),$$

$$\mathcal{M} \star \mathcal{M} \to \mathcal{M}, \quad \mathcal{M} \in \{\mathcal{G}, \mathcal{L}, \mathcal{U}\}, \quad \star \in \{+, \cdot\} \tag{9}$$

$$\alpha\mathcal{M} \to \mathcal{M}, \quad \mathcal{M} \in \{\mathcal{G}, \mathcal{L}, \mathcal{U}, \mathcal{S}\} \tag{10}$$

$$\mathcal{L}^T = \mathcal{U},\ \mathcal{U}^T = \mathcal{L},\ \mathcal{S}^T = \mathcal{S} \tag{11}$$

$$MM^T \text{ is } \mathcal{S}, \quad M \text{ is } \mathcal{M} \in \{\mathcal{G}, \mathcal{L}, \mathcal{U}, \mathcal{S}\} \tag{12}$$

$$M \text{ is } \mathcal{L}, \mathcal{U} \Rightarrow [M]_{r,r} \text{ is } \mathcal{L}, \mathcal{U} \tag{13}$$

Table 2. Examples of structure inference rules.

where g is a gather and p a permutation operator that can be applied to a gathered block. In other terms, in region σ a block should be accessed using the composed operator $p(g(\cdot))$. For example, assuming a symmetric S stores only its lower part, it has the following `AInfo`:

$$\left\{ \begin{matrix} \{(i,j) \mid 0 \le i < 4, 0 \le j \le i\} : ([i,j]_{1,1}^{4,4}, id) \\ \{(i,j) \mid 0 \le i < 4, i < j < 4\} : ([j,i]_{1,1}^{4,4}, id) \end{matrix} \right\},$$

where id is the identity permutation. Accessing element $(0,3)$ would yield $id(S[3,0]) = S[3,0]$. For matrices A, L, and U the accesses are unmodified:

$$\texttt{A.AInfo} = \left\{ \{(i,j) \mid 0 \le i,j < 4\} : ([i,j]_{1,1}^{4,4}, id) \right\},$$

$$\texttt{L.AInfo} = \left\{ \{(i,j) \mid 0 \le i < 4 \wedge 0 \le j \le i\} : ([i,j]_{1,1}^{4,4}, id) \right\},$$

$$\texttt{U.AInfo} = \left\{ \{(i,j) \mid 0 \le i < 4 \wedge i \le j < 4\} : ([i,j]_{1,1}^{4,4}, id) \right\}.$$

Type inference rules. Finally, structures need to be propagated through the tree of intermediate computations. We do this with type inference rules from well-known mathematical properties (Table 2).

4. Code Generation

In this section we present the main contribution of the paper: an approach, and its implementation within LGen, to generate optimized code for sBLACs. As running example we will use the sBLAC in (5), which contains three differently structured matrices. In the example we assume scalar (non-vectorized) code as output.

The steps in the generation closely follow the ones of the original LGen provided in Fig. 1, however with several important changes as described here.

Step 1: Tiling and structure inference. Given the LL program in Table 1 as an input, Step 1 proceeds as discussed in Section 2. In addition, structure information is propagated following the inference rules in Table 2. In our example, both LU and $LU + S$ are $\mathcal{G}$.

Step 2: From LL to Σ-LL. The rewriting system mentioned in Section 2 is substituted with an intermediate new module called Σ-CLooG based on the CLooG generator [3]. Next we briefly describe its design and how it is inserted into the generation flow.

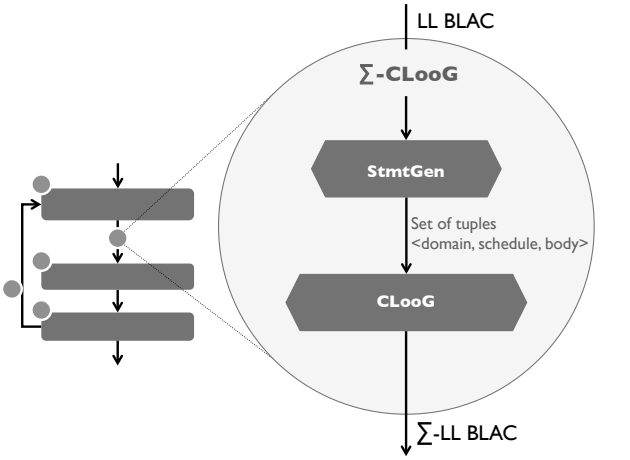


Figure 2. Architecture of the Σ-CLooG rewriting system.

Σ-CLooG: Overview. Σ-CLooG is schematically shown in Fig. 2. It consists of two main components: (1) the statement generator `StmtGen` and (2) CLooG. We extended the latter for this work with a backend to output Σ-LL.

The input to Σ-CLooG is an LL sBLAC from Step 1 and its output a translation of the input into an equivalent Σ-LL formulation. For example, given the sBLAC (5), the following Σ-LL expression is a possible output:

$$A = \sum_{i=0}^{2} \left(\sum_{j=0}^{i} [i,j](L[i,0]U[0,j] + S[i,j]) \right. \tag{14}$$

$$\left. + \sum_{j=i+1}^{3} [i,j](L[i,0]U[0,j] + S[j,i]) \right) \tag{15}$$

$$+ \sum_{j=0}^{3} [3,j](L[3,0]U[0,j] + S[3,j]) \tag{16}$$

$$+ \sum_{k=1}^{3} \sum_{i=k}^{3} \sum_{j=k}^{3} [i,j](L[i,k]U[k,j]). \tag{17}$$

Note that redundant multiplications (with zero) do not occur and that the symmetry of S is taken into account (i.e., only the part below the diagonal is accessed).

To achieve this, the input sBLAC is transformed using the information `SInfo` and `AInfo` of the matrices. This information is used to produce a set of CLooG statements. Every such statement is a triplet `<domain, schedule, body>` where: (a) `domain` is a polyhedral set σ representing the iteration space of the statement; (b) `schedule` is a polyhedral map ρ that determines the traversal or scanning order of the `domain`'s tuples; (c) `body` is a Σ-LL expression B. For example, the statement

$$\begin{aligned} s = \langle \sigma &= \{(i,k,j) \mid k = 0 \wedge 0 \le i < 4 \wedge 0 \le j \le i\}, \\ \rho &= ((i,k,j),(k,i,j)), \\ B &= [i,j](L[i,k]U[k,j] + S[i,j]) \rangle \end{aligned} \tag{18}$$

is used to generate (14) and (16) (which is $i = 3$ in (14), split off). In particular, the `domain` specifies the range of the indices appearing in the `body` and the `schedule` their order. Next we describe how `StmtGen` recursively creates statements such as (18) by bottom-up processing the sBLAC expression tree. We first explain the creation of domain and bodies. Once the entire tree is processed, the schedule is fixed.

Step 2.1: Generating domains and bodies for operations on leaves. The first operation performed by `StmtGen` is the creation of a unique index space for the input sBLAC. For our running example, three indices are needed:

$$A_{i,j} = L_{i,k}U_{k,j} + S_{i,j}.$$

Such an index space is then used to expand the `SInfo` and `AInfo` dictionaries (see end of Section 3) of the occurring matrices. In our case, the regions of the structured matrices are expanded to prisms:

$$\texttt{L.SInfo} = \begin{Bmatrix} \mathcal{G} : \{(i,k,j) | 0 \le i < 4 \wedge 0 \le k \le i\} \\ \mathcal{Z} : \{(i,k,j) | 0 \le i < 4 \wedge i < k < 4\} \end{Bmatrix} \quad (19)$$

$$\texttt{U.SInfo} = \begin{Bmatrix} \mathcal{G} : \{(i,k,j) | 0 \le k < 4 \wedge k \le j < 4\} \\ \mathcal{Z} : \{(i,k,j) | 0 \le k < 4 \wedge 0 \le j < k\} \end{Bmatrix} \quad (20)$$

$$\texttt{A.SInfo} = \texttt{S.SInfo} = \{\mathcal{G} : \{(i,k,j) \mid 0 \le i,j < 4\}\}. \quad (21)$$

Similarly, `AInfo` is computed:

$$\texttt{L.AInfo} = \Big\{\{(i,k,j) \mid 0 \le i < 4 \wedge 0 \le k \le i\} : ([i,k]_{1,1}^{4,4}, id)\Big\},$$

$$\texttt{U.AInfo} = \Big\{\{(i,k,j) \mid 0 \le k < 4 \wedge k \le j < 4\} : ([k,j]_{1,1}^{4,4}, id)\Big\},$$

$$\texttt{S.AInfo} = \begin{Bmatrix} \{(i,k,j) | 0 \le i < 4, 0 \le j \le i\} : ([i,j]_{1,1}^{4,4}, id) \\ \{(i,k,j) | 0 \le i < 4, i < j < 4\} : ([j,i]_{1,1}^{4,4}, id) \end{Bmatrix}, \quad (22)$$

$$\texttt{A.AInfo} = \Big\{\{(i,k,j) \mid 0 \le i,j < 4\} : ([i,j]_{1,1}^{4,4}, id)\Big\}.$$

Next `StmtGen` builds a set of statements for every operator in the input sBLAC bottom-up, starting from the inputs. In our case the first operation is LU. To build statements for LU we begin by determining its iteration space. In general, the iteration space for matrix multiplication is a cuboid (Fig. 3(a)). However given the presence of zero regions in (19) and (20), the redundant zero computations can be excluded (Fig. 3(b)) by computing the iteration space as

$$\mathit{iterSpace}_{LU} = \texttt{L.SInfo}[\mathcal{G}] \cap \texttt{U.SInfo}[\mathcal{G}] = \{(i,k,j) \mid 0 \le k < 4 \wedge k \le i,j < 4\}.$$

In general (e.g., for vectorization), our approach computes the iteration spaces for all combinations of nonzero operands (e.g., $\mathcal{GG}$, $\mathcal{GL}$, ...) using Algorithm 1.

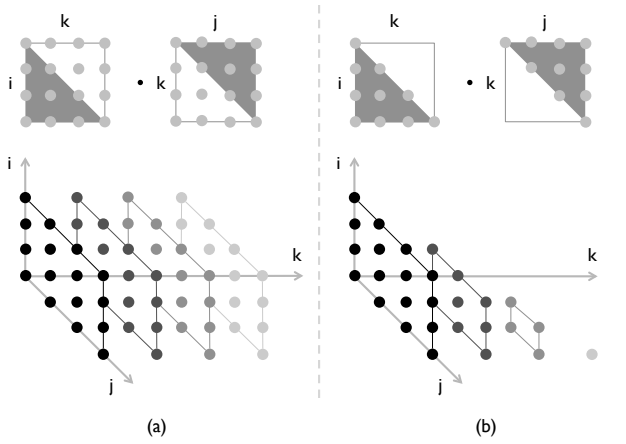


Figure 3. Iteration space of LU with redundant zero computations (a) and without (b).

Data: Inputs I_0, I_1.
Result: Iteration space (*iterSpace*) of $I_0 I_1$.
iterSpace $\leftarrow \emptyset$;
for $(\mathcal{M}_0 : \sigma_0) \in \texttt{I}_0\texttt{.SInfo} : \mathcal{M}_0 \neq \mathcal{Z}$ **do**
  for $(\mathcal{M}_1 : \sigma_1) \in \texttt{I}_1\texttt{.SInfo} : \mathcal{M}_1 \neq \mathcal{Z}$ **do**
    `// iteration space based on all pairs`
    `// of input non-zero regions.`
    iterSpace $=$ *iterSpace* $\cup (\sigma_0 \cap \sigma_1)$;
  end
end

Algorithm 1: Computing the iteration space for matrix multiplication.

The next task is to separate initial accesses to the output array from subsequent accumulations by splitting the iteration space (see Fig. 4). This information is readily available from the representation. In our example, we would split into the two iteration spaces

$$\mathit{iterSpace}_{LU}^{\mathit{init}} = \{(i,0,j) \mid 0 \le i < 4 \wedge 0 \le j < 4\},$$
$$\mathit{iterSpace}_{LU}^{\mathit{acc}} = \{(i,k,j) \mid 1 \le k < 4 \wedge k \le i,j < 4\}.$$

To derive the final domain and body of two CLooG statements for the computation of LU, these need to be intersected with the regions in the respective `AInfo` dictionaries, which explain how matrices are accessed. Since in our examples only symmetric matrices have special access, nothing changes:

$$\mathit{dom}_{LU}^{\mathit{init}} = \mathit{iterSpace}_{LU}^{\mathit{init}},$$
$$\mathit{dom}_{LU}^{\mathit{acc}} = \mathit{iterSpace}_{LU}^{\mathit{acc}},$$

and using the gathers from `AInfo` we construct the associated bodies (which in this case are the same):

$$B_{LU}^{\mathit{init}} = B_{LU}^{\mathit{acc}} = [i,j](L[i,k]U[k,j]).$$

The two statements are thus obtained:

$$s_{LU}^{\mathit{init}} = \langle \mathit{dom}_{LU}^{\mathit{init}}, \emptyset, B_{LU}^{\mathit{init}} \rangle,$$
$$s_{LU}^{\mathit{acc}} = \langle \mathit{dom}_{LU}^{\mathit{acc}}, \emptyset, B_{LU}^{\mathit{acc}} \rangle.$$

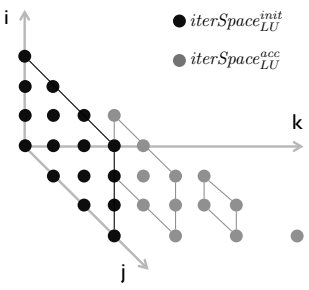


Figure 4. Iteration space of LU split into output initialization (black dots) and output accumulation (gray dots) space.

The schedules are left empty as they will be computed last. The general version of this approach for arbitrarily structured inputs is shown in Algorithm 2.

Data: *iterSpace*^{init}, *iterSpace*^{acc}, I_0, I_1, and T.
Result: CLooG statements (*stmts*) for $T = I_0 I_1$.
stmts $\leftarrow \emptyset$;
for $(\sigma_0 : (g_0, p_0)) \in$ `I0.AInfo` **do**
  for $(\sigma_1 : (g_1, p_1)) \in$ `I1.AInfo` **do**
    for $(\sigma_T : (g_T, p_T)) \in$ `T.AInfo` **do**
      for $\sigma_{space} \in \{$*iterSpace*^{init}, *iterSpace*$^{acc}\}$ **do**
        dom $\leftarrow \sigma_0 \cap \sigma_1 \cap \sigma_T \cap \sigma_{space}$;
        if *dom* $\neq \emptyset$ **then**
          `// Gather + permute inputs and multiply.`
          $m \leftarrow p_0(g_0(I_0)) \cdot p_1(g_1(I_1))$;
          `// Permute + scatter output.`
          $B \leftarrow g_T^{-1}(p_T^{-1}(m))$;
          `// Save new statement.`
          stmts $\leftarrow$ *stmts* $\cup \{\langle dom, \emptyset, B\rangle\}$;
        end
      end
    end
  end
end

Algorithm 2: Building CLooG statements for matrix multiplication. One statement is created for every combination of input and output regions that intersect the iteration space. Schedules are generated separately.

Step 2.2: Generating domains and bodies for operations recursively. As mentioned, the generation of domains and bodies is bottom up. In our example, the operation following LU is the addition $LU + S$. For the computation of its CLooG statements, `StmtGen` uses an approach similar to the one used before. However, as LU is not an input matrix, its set of (already generated) CLooG statements is used as input this time. As before we compute first the iteration space. Using (21) we get the trivial result

$$\begin{aligned} \textit{iterSpace} &= \{(i,k,j) \in \sigma \mid (\mathcal{M} : \sigma) \in \texttt{A.SInfo}, \mathcal{M} \neq \mathcal{Z}\} \\ &= \{(i,k,j) \in \texttt{A.SInfo}[\mathcal{G}]\} \\ &= \{(i,k,j) \mid 0 \le i,j < 4\}, \end{aligned}$$

where A is the output of the operation.

Next, we derive the CLooG statements, i.e, a possible splitting into domains and the associated bodies using a general algorithm for matrix addition that operates analogous to Algorithm 2. We do this by intersecting *iterSpace* with (a) the domain of s_{LU}^{init} and (b) the regions from `S.AInfo` in (22). Since there are two such regions (here denoted with $\sigma_{S,0}$ and $\sigma_{S,1}$) we obtain two domains. Both have initialization accesses only and accumulating accesses do not occur:

$$\begin{aligned} \textit{dom}_0 &= \textit{iterSpace} \cap \textit{dom}_{LU}^{init} \cap \sigma_{S,0} \\ &= \{(i,0,j) \mid 0 \le i < 4, 0 \le j \le i\}, \\ \textit{dom}_1 &= \textit{iterSpace} \cap \textit{dom}_{LU}^{init} \cap \sigma_{S,1} \\ &= \{(i,0,j) \mid 0 \le i < 4, i < j < 4\}. \end{aligned}$$

Using s_{LU}^{init} and `S.AInfo` we compute the associated two bodies:

$$\begin{aligned} B_0 &= [i,j]\Big(\overbrace{[i,j](L[i,k]U[k,j])}^{B_{LU}^{init}}[i,j] + S[i,j]\Big) \\ &= [i,j]\Big(L[i,k]U[k,j] + S[i,j]\Big), \\ B_1 &= [i,j]\Big(\overbrace{[i,j](L[i,k]U[k,j])}^{B_{LU}^{init}}[i,j] + S[j,i]\Big) \\ &= [i,j]\Big(L[i,k]U[k,j] + S[j,i]\Big). \end{aligned}$$

With the new domains and bodies we can finally construct the statements that lead to the final output in (14)–(17):

$$s_0 = \langle dom_0, \emptyset, B_0\rangle, \quad s_1 = \langle dom_1, \emptyset, B_1\rangle, \quad s_2 = s_{LU}^{acc}.$$

Before feeding the statements to CLooG, `StmtGen` needs to complete them with schedules.

We emphasize that the method sketched here on a simple example can correctly derive and exploit intermediate structures including blocks in multi-level blocking of expressions as complex as, e.g., $A = (L_0 + L_1)S_1 + xx^T$.

Step 2.3: Building the schedules. After Step 2.2 the root operator contains all necessary statements for the given sBLAC albeit without schedules. To add the schedules we first compute a global order over the index space of the sBLAC. This can be done by assuming performance models for the operators as those discussed in [10, 26]. For our example, we assume the order (k,i,j), yielding `schedule` $= \{((i,k,j),(k,i,j))\}$. Completing s_0, s_1, and s_2 with `schedule` CLooG produces the expression in (14)–(17) as the input to the next step in LGen.

Steps 3 to 5: From Σ-LL to output code. For scalar code generation the remaining three steps are similar to the original LGen (Section 2). From the Σ-LL in (14)-(17) we generate the code in Table 3.

```
for( int i = 0; i <= 2; i++ ) {
  for( int j = 0; j <= i; j++ ) {
    A[4*i+j] = L[4*i] * U[j] + S[4*i+j];
  }
  for( int j = i + 1; j <= 3; j++ ) {
    A[4*i+j] = L[4*i] * U[j] + S[i+4*j];
  }
}

for( int j = 0; j <= 3; j++ ) {
  A[j+12] = L[12] * U[j] + S[j+12];
}

for( int k = 1; k <= 3; k++ ) {
  for( int i = k; i <= 3; i++ ) {
    for( int j = k; j <= 3; j++ ) {
      A[4*i+j] += L[4*i+k] * U[4*k+j];
    }
  }
}
```

Table 3. Output C code for sBLAC (5).

5. Vectorization

Enabling vectorization introduces at least one level of tiling for ν-BLACs as discussed in Section 2, Step 1. We now discuss how this affects the internal representation of structures. We again use our example sBLAC (5) assuming we want to vectorize for a machine with a 2-way vector ISA ($\nu = 2$).

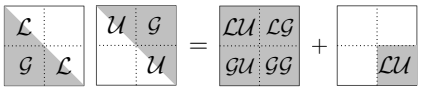
Internal representation of tiled structures. When a structured matrix is ν-tiled for vector instructions, it is viewed as a matrix of $\nu \times \nu$ blocks. Viewed like this, the matrix will still have structure. For example an $\mathcal{L}$ or $\mathcal{U}$ matrix will retain its structure. In principle, this could be derived automatically. We chose to incorporate this information into our system by providing the associated definitions of `SInfo` and `AInfo` for the blocked matrix in each case and for a generic block size. This definition can then be instantiated for specific cases. For example, for a ν-tiled symmetric matrix (instantiated for $\nu = 2$) one gets

$$\texttt{[S]}_{2,2}\texttt{.SInfo} = \left\{ \begin{matrix} \mathcal{G} : \{(0,2),(2,0)\} \\ \mathcal{S} : \{(0,0),(2,2)\} \end{matrix} \right\},$$

$$\texttt{[S]}_{2,2}\texttt{.AInfo} = \left\{ \begin{matrix} \{(0,0),(2,0),(2,2)\} : ([i,j]_{2,2}^{4,4}, id) \\ \{(0,2)\} : ([j,i]_{2,2}^{4,4}, (\cdot)^T) \end{matrix} \right\}.$$

This specifies, for example, that the tile at $(0,2)$ is accessed as $S[2,0]^T$. Next we sketch how these new definitions interact with the approach of Σ-CLooG.

Σ-CLooG and vectorization. Vectorization introduces a coarser basic block definition for the matrices. From the CLooG perspective, this only means the construction of sparser domains of the statements, where polyhedral points are accessed at a stride as in (8). The approach taken in Algorithm 2 (and similarly those taken by the other operators) would then derive more structure combinations. For example, consider the computation of $[L]_{2,2}[U]_{2,2}$. The iteration space would be constructed based on the following combination of structures:



This yields four initialization statements for the four different structure combinations (i.e., $\mathcal{LU}$, $\mathcal{LG}$, $\mathcal{GU}$, and $\mathcal{GG}$ in the first output square) and a single accumulation statement. Completing with addition, and using the schedule defined in Step 2.3, it produces the following Σ-LL output:

$$\begin{aligned} A = {} & [0,0](L[0,0]U[0,0] + S[0,0]) \\ & + [0,2](L[0,0]U[0,2] + (S[2,0])^T) \\ & + [2,0](L[2,0]U[0,0] + S[2,0]) \\ & + [2,2](L[2,0]U[0,2] + S[2,2]) \\ & + [2,2](L[2,2]U[2,2]). \end{aligned}$$

The above expression is completely decomposed into ν-BLACs and thus in principle mappable to vector code. However, it features different kinds of tiles (e.g., $L[0,0]$ is $\mathcal{L}$, $L[2,0]$ is $\mathcal{G}$, and $S[0,0]$ is $\mathcal{S}$) that enforce different kinds of computations (e.g., $L[0,0]U[0,0]$ is an $\mathcal{LU}$ multiplication while $L[2,0]U[0,0]$ is $\mathcal{GU}$). Simply ignoring the structure by using generic ν-BLACs is not possible since, by convention, data accesses above the diagonal are not allowed for $\mathcal{L}, \mathcal{U}, \mathcal{S}$.

Mapping structures to vector code. As explained in Section 2, the translation between Σ-LL and C-IR is based on three collections of codelets called Loaders, Storers, and ν-BLACs. The first two handle data accesses while the latter does the computation. When mapping structured ν-BLACs to vector code we use the generic computation but extend the Loaders and Storers to prevent illegal accesses. For example, consider the load of the lower triangular block $L[0,0]$. The expected behavior of the Loader would be the following:

$$\begin{bmatrix} \ell_{0,0} & x \\ \ell_{1,0} & \ell_{1,1} \end{bmatrix} \xrightarrow{Load} \begin{bmatrix} \ell_{0,0} & 0 \\ \ell_{1,0} & \ell_{1,1} \end{bmatrix}. \tag{23}$$

Here, a 0 is inserted by the Loader in place of x and used in the computation. Once matrices are loaded, the computations can be performed using the original 18 ν-BLACs (a slight inefficiency) introduced in Section 2.

6. Discussion

In this section we discuss some limitations and the extensibility of our approach and generator.

Limitations. The first limitation is that right now we do not take advantage of structure in ν-BLACs (e.g., as appears in the $\nu \times \nu$ blocks on the diagonal after blocking an L or U). For small ν the penalty is likely negligible, but for larger ν and relatively small matrices, some performance could be gained by implementing special ν-sBLACs.

Second, our approach always assumes matrices represented as full $m \times n$ arrays, with redundant regions not to

be accessed. For the structures discussed here, this is standard. However, for diagonal or tridiagonal matrices, a special format (storing only the nonzero elements) could be advantageous. In the generator we did implement a separation between physical and logical layout of matrices but did not use it yet for this case. Further, the structure should be describable by the affine index equations (7) provided by isl.

Extensibility. We designed our approach to be extensible to new structures. Provided a structure satisfies the constraints mentioned above, extension requires the addition of:

- *A structure definition*: `SInfo` and `AInfo` dictionaries (see Section 3).
- *A set of Loaders and Storers*: vectorized codelets for accessing ν-sized matrices with the new structure.

As an example we briefly discuss the addition of banded matrices, which have a non-zero region within two delimiting diagonals of the matrix. As in the triangular case, the scalar definition of their `SInfo` would contain two regions, a general region for the band and one or two zero regions outside the band. Producing vector code for a ν-way vector ISA would require special Loaders and Storers at the border of the band. For example, consider a banded matrix of size $4\nu \times 4\nu$ with bandwidth k. If $\nu | k$ then we would need only (unit) triangular matrices at the border:

$$\begin{bmatrix} \mathcal{G} & \mathcal{G} & \mathcal{L} & \mathcal{Z} \\ \mathcal{G} & \mathcal{G} & \mathcal{G} & \mathcal{L} \\ \mathcal{U} & \mathcal{G} & \mathcal{G} & \mathcal{G} \\ \mathcal{Z} & \mathcal{U} & \mathcal{G} & \mathcal{G} \end{bmatrix}, \tag{24}$$

otherwise, if $\nu \nmid k$ and assuming $k < \nu$, we would need

$$\begin{bmatrix} \mathcal{B} & \mathcal{K} & \mathcal{Z} & \mathcal{Z} \\ \mathcal{J} & \mathcal{B} & \mathcal{K} & \mathcal{Z} \\ \mathcal{Z} & \mathcal{J} & \mathcal{B} & \mathcal{K} \\ \mathcal{Z} & \mathcal{Z} & \mathcal{J} & \mathcal{B} \end{bmatrix}, \tag{25}$$

with $\mathcal{B}$, $\mathcal{J}$, and $\mathcal{K}$ respectively band, "almost" upper, and "almost" lower triangular structures. If a banded matrix is also symmetric then support can be added by combining the `SInfo` description sketched in (24) and (25) with an `AInfo` similar to the one described for the general case in Section 3.

Blocked structures as, for example,

$$\begin{bmatrix} \mathcal{G} & \mathcal{L} \\ \mathcal{S} & \mathcal{U} \end{bmatrix}$$

can be added by recursively fusing the `SInfo` and `AInfo` dictionaries of the occurring structures. This is possible since isl supports unions of regions as shown in (7).

7. Experiments

In this section we provide an experimental evaluation of our approach. We divide our experiments into the three categories shown in Table 4, a division according to express the compatibility of the sBLACs with the BLAS. Specifically we chose two sBLACs that match BLAS, two sBLACs that are available in BLAS but without support for structures (BLAS-like), and an sBLAC that can only be implemented using more than one BLAS or BLAS-like function (Non-BLAS). Matrices are implemented using double precision arrays 32-bytes aligned. All of them are fully stored in row-major order and in the case of triangular and symmetric matrices only half of the matrices are used.

Category	Label	sBLAC	Sizes
BLAS	*dsyrk*	$S_u = AA^T + S_u$	$A \in \mathbb{R}^{n \times 4}$
	dtrsv	$x = L \backslash x$	$L \in \mathbb{R}^{n \times n}$
BLAS-like	*dlusmm*	$A = LU + S_l$	$L, U \in \mathbb{R}^{n \times n}$
	dsylmm	$A = S_u L + A$	$S_u, L \in \mathbb{R}^{n \times n}$
Non-BLAS	*composite*	$A = (L_0 + L_1)S_l + xx^T$	$L_0, S_l \in \mathbb{R}^{n \times n}$

Table 4. Experimental categories. For S we specify whether lower (l) or upper (u) part is used.

Experimental setup. We executed our tests on an Intel Core i7-2600 CPU (Sandy Bridge microarchitecture) 3.3 GHz, AVX, 32 kB L1 D-cache, 256 kB L2 cache, under Ubuntu 14.04 with Linux kernel v3.13. We disabled Intel Turbo Boost to minimize measurement instabilities. We compare with: (a) the Intel MKL library v11.2, (b) naïve code compiled with Intel icc v15, and (c) code generated by LGen without support for structures. MKL was reported as the most competitive alternative to the previous LGen [21]. Further, starting with v11.2 it adds support for small-scale, double precision matrix multiplication (dgemm). BLAS tests are implemented using the matching BLAS functions. BLAS-like tests are implemented using `dtrmm` (*dlusmm*) and `dsymm` (*dsylmm*). The *composite* test is implemented using `MKL_Somatadd`, `dsyr`, and `dsymm`. We do not rearrange matrices when testing MKL (e.g., zeroing a half triangular matrix when used in place of a general one).

Naïve code is scalar, unoptimized, handwritten, straightforward code with hardcoded sizes of the matrices. The goal is to compare with compiler optimizations.

All tests were compiled using icc with flags -O3 -xHost -fargument-noalias -fno-alias -no-ipo -no-ip. The MKL tests use flags obtained from the Intel MKL Link Line Advisor[1].

Measuring approach. All plots show performance in flops per cycles (f/c) on the y-axis and the size parameter n in doubles on the x-axis. Assuming balanced additions and multiplications, the peak performance of the CPU is 8 f/c. The parameter n is always increased up to the L2 cache boundaries. All tests were run with warm cache. Every point on the graphs is the median of 30 repetitions and quartile informations are reported with whiskers. In most cases, however, these are too small to be visible.

[1] `https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor`

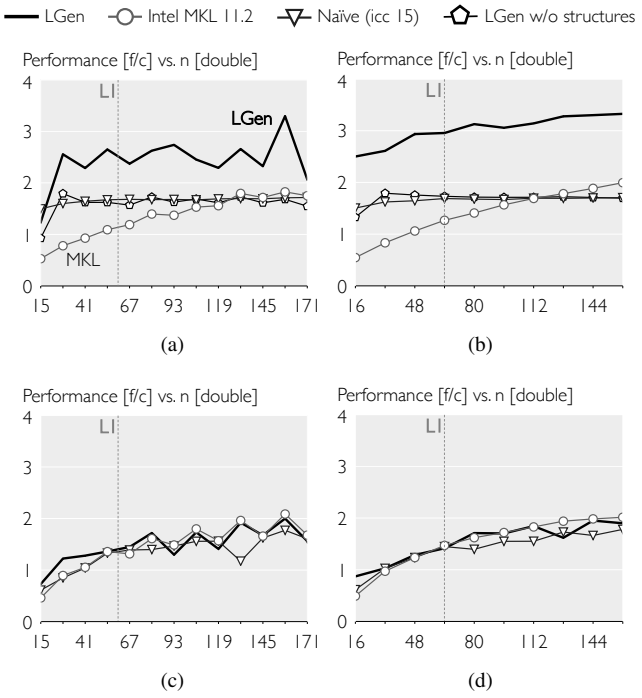


Figure 5. BLAS category: (a)–(b) *dsyrk* ($f = 4n^2 + 4n$) and (c)–(d) *dtrsv* ($f = n^2 + n$). In (b) and (d) all sizes are multiple of the vector length ($\nu = 4$). LGen w/o structures is missing in (c) and (d) as the triangular solve operator is not supported by such an approach.

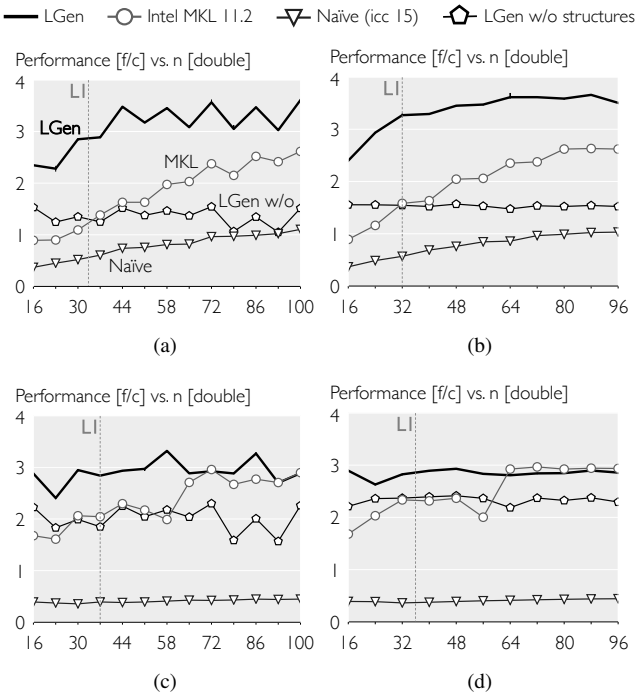


Figure 6. BLAS-like category: (a)–(b) *dlusmm* ($f = \frac{1}{3}(2n^3 + n) + n^2$) and (c)–(d) *dsylmm* ($f = n^3 + n^2$). In (b) and (d) all sizes are multiple of the vector length ($\nu = 4$).

Remarks on plots. We compute performance as the ratio of flop count taking structures into account (f underneath each plot) to measured time to solution. This way the plots can provide an estimate of the CPU utilization as well as carrying information about time speedup (as $\frac{f/c_1}{f/c_2} = \frac{c_2}{c_1}$).

Every plot shows the L1 boundary determined by working set size (size of all inputs and outputs of an sBLAC). All plots have the same legend show on top of each figure.

BLAS category. For *dsyrk* (Figs. 5(a)–(b)) LGen is up to $2.5\times$ faster than MKL when data fit in L1 and around $1.6\times$ when data fit in L2. Comparing with icc-compiled code LGen is in general $1.6\times$ faster. In all cases, icc performs unrolling and vectorization of the innermost loop of length four. However, icc does not modify the loop nest to increase reuse by blocking. In the case of *dtrsv* (Figs. 5(c)–(d)) all competitors perform equally. For larger sizes casting the computation in terms of matrix-vector multiplication becomes more crucial, an optimization that also icc applies by unrolling and vectorizing the innermost of the two loops in the handwritten code. In this case we could not generate code using the old LGen approach as it lacks the structure support required by the triangular solve.

BLAS-like category. The first test in the BLAS-like category is *dlusmm* (Figs. 6(a)–(b)). Here LGen is up to $3.5\times$ faster than icc and up to $2\times$ faster than MKL for data in L1 ($1.4\times$ for data in L2). In this case exploiting the structure of both L and U avoids about one third of redundant computations. icc on the other hand fails to perform and take advantage of proper tiling for locality. In *dsylmm* (Figs. 6(c)–(d)) LGen is up to $7\times$ faster than icc-compiled code and, for sizes up to the L1 boundary, about $1.4\times$ faster than MKL. Further investigations revealed that code generated with LGen produces high pressure on the shuffle unit of the CPU. This could be due to an excessive amount of transpositions in the innermost loops and could be handled by introducing block permutations in between (non-fused) gathers.

Non-BLAS category. Fig. 7 shows results for *composite*. Although more complicated, this sBLAC contains a multiplication term structurally similar to the one in *dsylmm*, thus the similarity between the two performance profiles.

8. Related work

In this section we describe three lines of related work: linear algebra libraries, program generators, and polyhedral optimization.

Structured matrices and linear algebra libraries. High performance linear algebra libraries such as Intel MKL [15] and libraries derived with the FLAME approach [14] provide functions for structured matrices but focus on those defined by BLAS [11]. Further, they focus on the larger sizes needed by LAPACK in scientific computing. BLIS [22] is a framework for instantiating a set of functions larger than BLAS

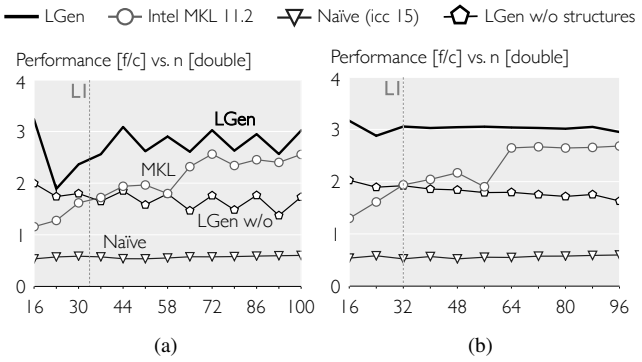


Figure 7. Non-BLAS category: (a)–(b) *composite* ($f = n^3 + \frac{5}{2}(n^2 + n)$). In (b) all sizes are multiple of the vector length ($\nu = 4$).

from a set of microkernels but does not focus on structured matrices. Our approach is fully generative, taking only the sBLAC as input, and especially targets small scale and fixed size computations.

Domain-specific languages and generators. Eigen [13] provides vectorized (up to SSE 4) code generation based on expression templates. It supports a variety of functionalities and exposes structures such as band, triangular, and symmetric matrices to the programmer. However, an approach based on C++ metaprogramming lacks autotuning capabilities and requires a non-negligible effort in extensibility. Further, the optimization favors certain computations [21].

The CLACK compiler [7] and the DxTer system [18] take linear algebra computations as input and manipulate them into a form suitable for efficient mapping to BLAS functions. They do not generate BLAS or BLAC functions, with or without structures.

VOBLA [4] is a linear algebra DSL with the goal of generating high-performance OpenCL code for BLAS and LAPACK functionalities. The DSL describes linear algebra computations using basic operators and array access patterns. Access patterns are used to separate matrix structures from storage formats. VOBLA finally relies on polyhedral compilation techniques to generate parallel GPU code.

The work in [9, 24] presents two extensions of Spiral for the generation of general, large-matrix multiplication code.

Polyhedral optimization. The work in [16] presents a multi-level tiling algorithm capable of separating full and partial tiles with non-fixed tile sizes. Optimizing compilers based on the polyhedral model [8], such as [5] and [12], reschedule computation and data accesses to enhance locality and expose parallelization and vectorization opportunities. In this work we use polyhedral tools (i.e., isl [25] and CLooG [3]) to manipulate BLACs at a higher level of abstraction.

9. Conclusion

In this paper we addressed the problem of generating efficient code for small, basic linear computations where structured matrices appear, such as triangular or symmetric. Structures enable the elimination of redundant accesses and computations and the introduction of new functions such as the solution of triangular systems. We introduced a methodology to represent and manipulate such computations based on the polyhedral representation of the structures they contain. We extended an existing compiler for small, basic linear algebra computations with our approach and showed competitive results when applied to computations with triangular and symmetric matrices. However, the approach is not limited to the structures just mentioned and we discussed how the formalism could handle more of them, such as band and composite matrices. For future work we plan to include more matrix storage formats in our system and exploit structures for the generation of higher level linear algebra functions.

Acknowledgments

The authors would like to thank Victoria Caparrós Cabezas for many helpful discussions.

References

[1] LGen: A basic linear algebra compiler. Available at `http://spiral.net/software/lgen.html`.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, third edition, 1999.

[3] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 7–16, 2004.

[4] U. Beaugnon, A. Kravets, S. van Haastregt, R. Baghdadi, D. Tweed, J. Absar, and A. Lokhmotov. VOBLA: A vehicle for optimized basic linear algebra. In *Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 115–124, 2014.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Programming Language Design and Implementation (PLDI)*, pages 101–113, 2008.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[7] D. Fabregat-Traver and P. Bientinesi. A domain-specific compiler for linear algebra operations. In *High Performance Computing for Computational Science (VECPAR 2012)*, volume 7851 of *Lecture Notes in Computer Science (LNCS)*, pages 346–361. Springer, 2013.

[8] P. Feautrier and C. Lengauer. *Encyclopedia of Parallel Computing*, chapter Polyhedron Model. Springer, 2011.

[9] F. Franchetti, F. Mesmay, D. Mcfarlin, and M. Püschel. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain-Specific Languages (DSL WC)*, volume 5658 of *Lecture Notes in Computer Science (LNCS)*, pages 385–410. Springer, 2009.

[10] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12:1–12:25, 2008.

[11] K. Goto and R. A. van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)*, 35(1):4:1–4:14, 2008.

[12] T. Grosser, A. Groesslinger, and C. Lengauer. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[13] G. Guennebaud, B. Jacob, *et al.* Eigen v3. `http://eigen.tuxfamily.org`.

[14] J. A. Gunnels, F. G. Gustavson, G. Henry, and R. A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*, 27(4): 422–455, 2001.

[15] Intel math kernel library (MKL). `http://software.intel.com/en-us/intel-mkl`.

[16] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. M. Strout. Multi-level tiling: M for the price of one. In *Supercomputing (SC)*, pages 1–12, 2007.

[17] N. Kyrtatas, D. G. Spampinato, and M. Püschel. A basic linear algebra compiler for embedded processors. In *Design, Automation and Test in Europe (DATE)*, pages 1054–1059, 2015.

[18] B. Marker, J. Poulson, D. Batory, and R. van de Geijn. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *High Performance Computing for Computational Science (VECPAR 2012)*, volume 7851 of *Lecture Notes in Computer Science (LNCS)*, pages 362–378. Springer, 2013.

[19] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93 (2):232–275, 2005.

[20] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[21] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. In *International Symposium on Code Generation and Optimization (CGO)*, pages 23–32, 2014.

[22] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating blas functionality. *ACM Transactions on Mathematical Software (TOMS)*, 41(3):14:1–14:33, 2015.

[23] F. G. Van Zee, E. Chan, R. A. van de Geijn, E. S. Quintana-Orti, and G. Quintana-Orti. The libFLAME library for dense matrix computations. *IEEE Design & Test*, 11(6):56–63, Nov. 2009.

[24] R. Veras and F. Franchetti. Capturing the expert: Generating fast matrix-multiply kernels with Spiral. In *High Performance Computing for Computational Science (VECPAR 2014)*, volume 8969 of *Lecture Notes in Computer Science (LNCS)*, pages 236–244. Springer, 2015.

[25] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software (MS)*, volume 6327 of *Lecture Notes in Computer Science (LNCS)*, pages 299–302. Springer, 2010.

[26] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, 2005.