Program Generation for Small-Scale Linear Algebra Applications

Daniele G. Spampinato
Department of Computer Science
ETH Zurich
Switzerland
danieles@inf.ethz.ch

Diego Fabregat-Traver
Aachen Institute for Advanced Study in Computational Engineering Science
RWTH Aachen University
Germany
fabregat@aices.rwth-aachen.de

Paolo Bientinesi
Aachen Institute for Advanced Study in Computational Engineering Science
RWTH Aachen University
Germany
pauldj@aices.rwth-aachen.de

Markus Püschel
Department of Computer Science
ETH Zurich
Switzerland
pueschel@inf.ethz.ch

Abstract

We present SLinGen, a program generation system for linear algebra. The input to SLinGen is an application expressed mathematically in a linear-algebra-inspired language (LA) that we define. LA provides basic scalar/vector/matrix additions/multiplications and higher level operations including linear systems solvers, Cholesky and LU factorizations. The output of SLinGen is performance-optimized single-source C code, optionally vectorized with intrinsics. The target of SLinGen are small-scale computations on fixed-size operands, for which a straightforward implementation using optimized libraries (e.g., BLAS or LAPACK) is known to yield suboptimal performance (besides increasing code size and introducing dependencies), but which are crucial in control, signal processing, computer vision, and other domains. Internally, SLinGen uses synthesis and DSL-based techniques to optimize at a high level of abstraction. We benchmark our program generator on three prototypical applications: the Kalman filter, Gaussian process regression, and an L1-analysis convex solver, as well as basic routines including Cholesky factorization and solvers for the continuous-time Lyapunov and Sylvester equations. The results show significant speed-ups compared to straightforward C with Intel icc and clang with a polyhedral optimizer, as well as library-based and template-based implementations.

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168812

CCS Concepts • **Mathematics of computing** → **Mathematical software**; **Computations on matrices**; • **Software and its engineering** → **Compilers**; **Source code generation**; *Domain specific languages*; • **General and reference** → *Performance*;

Keywords Performance, SIMD vectorization, Rewriting

ACM Reference Format:
Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. 2018. Program Generation for Small-Scale Linear Algebra Applications. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3168812

1 Introduction

A significant part of processor time worldwide is spent on mathematical algorithms that are used in simulations, machine learning, control, communication, signal processing, computer vision, and other domains. The problem sizes and computers used range from the very large (e.g., simulations on a supercomputer or learning in the cloud) to the very small (e.g., a Kalman filter or Viterbi on an embedded processor). Both scenarios have in common the need for fast code, for example to save energy, to enable real-time, or to maximize processing resolution. The mathematics used in these domains may differ widely, but the actual computations in the end often fall into the domain of linear algebra, meaning sequences of computations on matrices and vectors.

For large-scale linear algebra applications, the bottleneck is usually in cubic-cost components such as matrix multiplication, matrix decompositions, and solving linear systems. High performance is thus attained by using existing highly optimized libraries (typically built around the interfaces of BLAS [8] and LAPACK [1]). For small-scale applications, the same libraries are not as optimized, may incur overhead due to fixed interfaces and large code size, and introduce dependencies [41]. The small scale is the focus of this paper.

Table 1. Kalman filter (one iteration at time step k). Matrices are upper case, vectors lower case. During prediction (steps (1) and (2)) a new estimate of the system state x is computed (1) along with an associated covariance matrix P. During update (steps (3) and (4)), the predictions are combined with the current observation z.

$$x_{k|k-1} = Fx_{k-1|k-1} + Bu \tag{1}$$

$$P_{k|k-1} = FP_{k-1|k-1}F^T + Q \tag{2}$$

$$x_{k|k} = x_{k|k-1} + P_{k|k-1}H^T \times (HP_{k|k-1}H^T + R)^{-1}(z_k - Hx_{k|k-1}) \tag{3}$$

$$P_{k|k} = P_{k|k-1} - P_{k|k-1}H^T \times (HP_{k|k-1}H^T + R)^{-1}HP_{k|k-1} \tag{4}$$

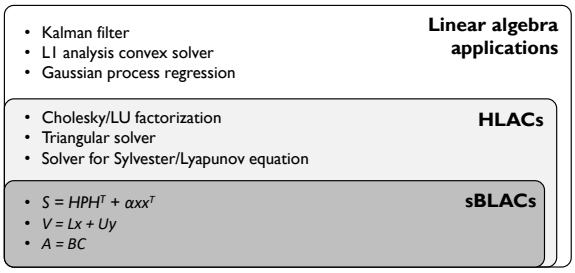


Figure 1. Classes of linear algebra computations used in this paper with examples.

Program generation for small-scale linear algebra. In an ideal world, a programmer would express a linear algebra computation as presented in a book from the application domain, and the computer would produce code that is both specialized to this computation and highly optimized for the target processor. With this paper, we make progress towards this goal and introduce SLinGen, a generator for small-scale linear algebra applications on fixed-size operands. The input to SLinGen is the application expressed in a linear algebra language (LA) that we introduce. LA provides basic scalar/-vector/matrix operations, higher level operations including triangular solve, Cholesky and LU factorization, and loops. The output is a single-threaded, single-source C code, optionally vectorized with intrinsics.

As illustrating example, we consider the Kalman filter, which is ubiquitously used to control dynamic systems such as vehicles and robots [39], and a possible input to SLinGen. Table 1 shows a basic Kalman filter, which performs computations on matrices (upper case) and vectors (lower case) to iteratively update a (symmetric) covariance matrix P. The filter performs basic multiplications and additions on matrices and vectors, but also requires a Cholesky factorization and a triangular solve to perform the matrix inversions. Note that operand sizes are typically fixed (number of states and measurements in the system) and in the 10s or even smaller. We show benchmarks with SLinGen-generated code for the Kalman filter later.

Classification of linear algebra computations. Useful for this paper, and for specifying our contribution, is an organization of the types of computations that our generator needs to process to produce single-source code. We consider the following three categories (Fig. 1):

- *Basic linear algebra computations, possibly with structured matrices (sBLACs, following [41]):* Computations on matrices, vectors, and scalars using basic operators: multiplication, addition, and transposition. Mathematically, sBLACs include (as a subset) most of the computations supported by the BLAS interface.
- *Higher-level linear algebra computations (HLACs):* Cholesky and LU factorization, triangular solve, and other direct solvers. Mathematically, HLAC algorithms (in particular when blocked for performance) are expressed as loops over sBLACs.
- *Linear algebra applications:* Finally, to express and support entire applications like the Kalman filter, this class includes loops and sequences of sBLACs, HLACs, and auxiliary scalar computations.

SLinGen supports the latter class (and thus also HLACs), and thus can generate code for a significant class of real-world linear algebra applications. In this paper we consider three case studies: the Kalman filter, Gaussian process regression, and L1-analysis convex optimization.

Contributions. In this paper, we make the following main contributions.

- The design and implementation of a domain-specific system that generates performant, single-source C code directly from a high-level linear algebra description of an application. This includes the definition of the input language LA, the use of synthesis and DSL (domain-specific language) techniques to optimize at a high, mathematical level of abstraction, and the ability to support vector ISAs. The generated code is single-threaded since the focus is on small-scale computations.
- As a crucial component, we present the first generator for a class of HLACs that produces single-source C (with intrinsics), i.e., does not rely on a BLAS implementation.
- Benchmarks of our generated code for both single HLACs and application-level linear algebra programs comparing against hand-written code: straightforward C optimized with Intel icc and clang with the polyhedral Polly, the template-based Eigen library, and code using libraries (Intel's MKL, ReLAPACK, RECSY).

As we will explain, part of our work builds on, but considerably expands, two prior tools: the sBLAC compiler LGen and Cl1ck. LGen [41, 42] generates vectorized C code for single sBLACs with fixed operand sizes; in this paper we move considerably beyond in functionality to generate code for an entire linear algebra language that we define and that

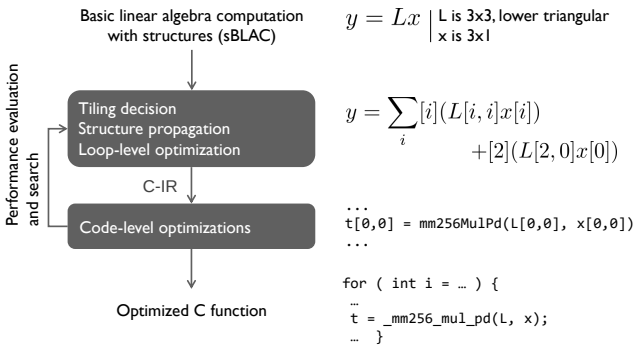


Figure 2. The architecture of LGen including a sketched small example.

includes HLAC algorithms (which themselves involve loops over sequences of sBLACs, some with changing operand size) and entire applications. Cl1ck [9, 10] synthesizes blocked algorithms (but not C code) for HLACs, represented in a DSL assuming an available BLAS library as API. Our goal is to generate single-source, vectorized C code for HLACs and entire linear algebra applications containing them.

2 Background

We provide background on the two prior tools that we use in our work. LGen [25, 41, 42] compiles single sBLACs (see Fig. 1) on fixed-size operands into (optionally vectorized) C code. Cl1ck [9, 10] generates blocked algorithms for (a class of) HLACs, expressed at a high level using the BLAS API.

In the following, we use uppercase, lowercase, and Greek letters to denote matrices, vectors, and scalars, respectively.

2.1 LGen

An sBLAC is a computation on scalars, vectors, and (possibly structured) matrices that involves addition, multiplication, scalar multiplication, and transposition. Examples include $A = BC$, $y = Lx + Uy$, $S = HPH^T + \alpha xx^T$, where S is symmetric and L/U are lower/upper triangular. The output is on the left-hand side and may also appear as input.

Overview. LGen translates an input sBLAC on fixed-size operands into C code using two intermediate compilation phases as shown in Fig. 2. During the first phase, the input sBLAC is optimized at the mathematical level, using a DSL that takes possible matrix structures into account. These optimizations include multi-level tiling, loop merging and exchange, and matrix structure propagation. During the second phase, the mathematical expression obtained from the first phase is translated into a C-intermediate representation. At this level, LGen performs additional optimizations such as loop unrolling and scalar replacement. Finally, since different tiling decisions lead to different code versions of the same computation, LGen uses autotuning to select the fastest version for the target computer.

Explicit vectorization. LGen allows for multiple levels of tiling and arbitrary tile sizes. If vectorization is enabled, the innermost level of tiling decomposes an expression into so-called ν-BLACs, where ν is the vector length (e.g., $\nu = 4$ for double precision AVX). There are 18 ν-BLACs (all single operations on $\nu \times \nu$ matrices and vectors of length ν), which are pre-implemented once for a given vector ISA together with vectorized data access building blocks, called Loaders and Storers, that handle leftovers and structured matrices.

2.2 Cl1ck

Cl1ck [9, 10] is an algorithm generator that implements the FLAME methodology [4]. The input to the generator is an HLAC expressed in terms of the standard matrix operators—addition, product, transposition and inversion—and matrix properties such as orthogonal, symmetric positive definite, and triangular. The output is a family of loop-based algorithms which make use of existing BLAS kernels; while all the algorithms in a family are mathematically equivalent, they provide a range of alternatives in terms of numerical accuracy and performance.

Overview. As illustrated in Fig. 3, the generation takes place in three stages: PME Generation, Loop Invariant Identification, and Algorithm Construction. In the first stage, the input equation (e.g., $U^T * U = S$, where U is the output), is blocked symbolically to obtain one or more recursive formulations, called "Partitioned Matrix Expression(s)" (PMEs); this stage makes use of a linear algebra knowledge-base, as well as pattern matching and term rewriting. In our example, blocking yields[1]

$$\begin{pmatrix} U_{TL}^T & 0 \\ U_{TR}^T & U_{BR}^T \end{pmatrix} \begin{pmatrix} U_{TL} & U_{TR} \\ 0 & U_{BR} \end{pmatrix} = \begin{pmatrix} S_{TL} & S_{TR} \\ S_{BL} & S_{BR} \end{pmatrix},$$

from which the three dependent equations shown are generated. In the second stage, Cl1ck identifies possible loop invariants for the yet-to-be-constructed loop-based algorithms. One example here is the predicate $U_{TL}^T U_{TL} = S_{TL}$ that has to be satisfied at the beginning and the end of each loop iteration. Finally, each loop invariant is translated into an algorithm that expresses the computation in terms of recursive calls, and BLAS- and LAPACK-compatible operations. As the process completes, the input equation becomes "known": it is assigned a name, is added to the knowledge base, and can be used as a building block for more complex operations.

Formal correctness. The main idea behind Cl1ck and the underlying FLAME methodology is the simultaneous construction of a loop-based blocked algorithm and its proof of correctness. This is accomplished by identifying the loop invariant that the algorithm will have to satisfy, *before* the algorithm exists. From the invariant, a template of a proof of correctness is derived, and the algorithm is then constructed to satisfy the proof.

[1] T, B, L, and R stand for Top, Bottom, Left, and Right, respectively.

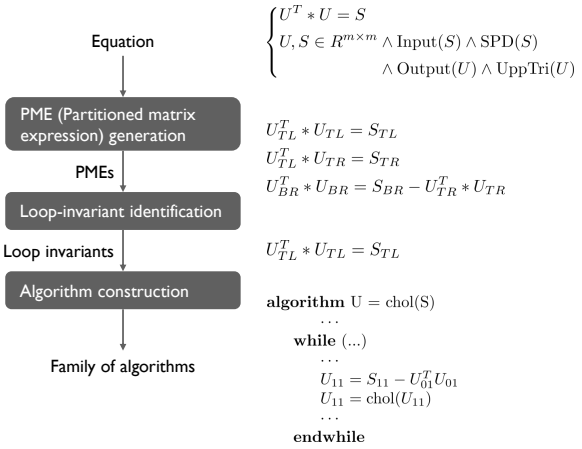


Figure 3. CL1CK: For a target input computation, a family of algorithms is generated in three stages.

⟨*la-program*⟩ ::= {⟨*declaration*⟩} {⟨*statement*⟩}

⟨*declaration*⟩ ::= 'Mat' ⟨*id*⟩ '(' ⟨*size*⟩ ',' ⟨*size*⟩ ')' '<' ⟨*iotype*⟩ { ',' ⟨*property*⟩ } [',' ⟨*ow*⟩] '>;'
| 'Vec' ⟨*id*⟩ ... | 'Sca' ⟨*id*⟩ ...

⟨*iotype*⟩ ::= 'In' | 'Out' | 'InOut'

⟨*property*⟩ ::= 'LoTri' | 'UpTri' | 'UpSym' | 'LoSym'
| 'PD' | 'NS' | 'UnitDiag'

⟨*ow*⟩ ::= 'ow(' ⟨*id*⟩ ')'

⟨*statement*⟩ ::= ⟨*for-loop*⟩ | ⟨*sBLAC*⟩ | ⟨*HLAC*⟩ ';'

⟨*for-loop*⟩ ::= 'for (i = ...) {' {⟨*statement*⟩$_i$ } '}'

⟨*sBLAC*⟩ ::= ⟨*id*⟩ '=' ⟨*expression*⟩

⟨*HLAC*⟩ ::= ⟨*expression*⟩ '=' ⟨*expression*⟩
| ⟨*id*⟩ '=' '(' ⟨*id*⟩ ')$^{-1}$'

Figure 4. Grammar for the LA language. An LA program consists of the declaration of a number of operands, and a sequence of computational statements. Operands may be scalars, vectors, or matrices, which are declared as either input ('In') or output ('Out') and of a certain size. The non-terminals ⟨*id*⟩ and ⟨*size*⟩ are any variable name and fixed size integer respectively. Matrices can have one or more properties. Structural properties beginning with 'Up' and 'Lo' specify respectively upper and lower storage format for both triangular and symmetric matrices. 'PD' and 'NS' stand for positive definiteness and non-singularity respectively. ⟨*expression*⟩ represents any well-defined combination of input and output scalars, vectors and matrices with operators '+', '-', '*', '$(\cdot)^T$' (transposition). If an expression is defined exclusively over scalars then it can include also the division ('/') and square root ('$\sqrt{}$') operators. When an ⟨*id*⟩ appears alone on the left-hand side, it must be an output element and it can also appear on the right-hand side ('InOut'). The notation ⟨*statement*⟩$_i$ indicates that data accesses in a statement might depend on the induction variable of the surrounding loop.

```
1 Mat H(k, n) <In>;
2 Mat P(k, k), R(k, k) <In, UpSym, PD>;
3 Mat S(k, k) <Out, UpSym, PD>;
4 Mat U(k, k) <Out, UpTri, NS, ow(S)>;
5 Mat B(k, k) <Out>;
6
7 S = H * H^T + R;
8 U^T * U = S;
9 U^T * B = P;
```

Figure 5. Example LA program for given constants n and k. Note the explicit specification of input and output matrices.

2.3 Challenges in connecting CL1CK and LGEN

In this paper we address two main research challenges. The first is how to connect CL1CK and LGEN to generate code for HLACs; the second is then how to generate code for entire applications consisting of BLACs and HLACs. The first challenge already requires 1) an extension from single sBLACs (the domain of LGEN) to the entire DSL used by CL1CK to express its loop-based outputs with multiple statements (see Fig. 3), and 2) a way to automatically synthesize HLACs on small blocks (Sec. 3.1). If the generation and inlining of C code is performed independently for each statement in CL1CK's output, certain optimizations cannot be applied, thus missing important opportunities for better vectorization (Sec. 3.2) and locality (Sec. 3.3). In Sec. 4, we demonstrate the efficacy of our solution compared to previous work on selected benchmarks. For a set of HLAC benchmarks (as previously mentioned, CL1CK only takes HLACs as an input), we include the implementation of CL1CK's generated algorithms using optimized BLAS and LAPACK functions among our competitors.

3 SLINGEN

We present SLINGEN, a program generator for small-scale linear algebra applications. The input to SLINGEN is a linear algebra program written in LA, a MATLAB-like domain-specific language described by the grammar in Fig. 4. The output is a single-source optimized C function that implements the input, optionally vectorized with intrinsics.

LA programs are composed of sBLACs and HLACs over scalars, vectors, and matrices of a fixed size. As an example, the program in Fig. 5 shows a slightly modified fragment of the Kalman filter in Table 1. In this program, statement 7 corresponds to an sBLAC that computes a matrix multiplication and an addition of symmetric matrices. Statements 8 and 9 are two HLACs: the Cholesky decomposition of S and a triangular linear system with unknown B. Note that, without an explicit input and output specification, it would be impossible to tell the difference between 8 and 9.

Overview. The general workflow of SLINGEN is depicted in Fig. 6. Given an input LA program, the idea is to perform a number of lowering steps, until the program is expressed in

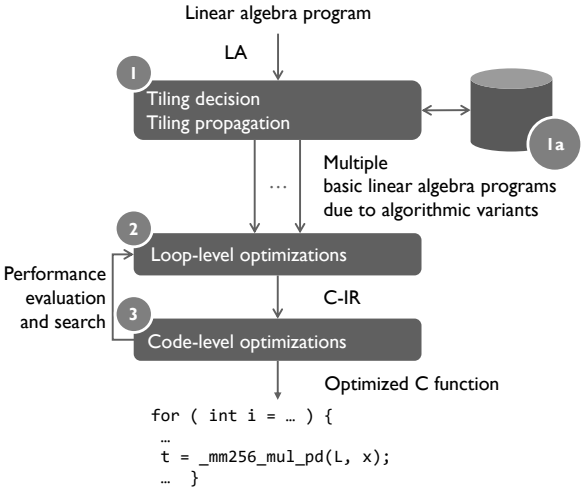


Figure 6. Architecture of SLinGen.

terms of pre-implemented vectorized codelets and auxiliary scalar operations (divisions, square roots, ...). In the first stage, the input program is transformed into one or more alternative *basic LA programs*—implementations that rely exclusively on sBLACs and scalar operations. To achieve this, loop-based algorithms are synthesized for each occurring HLAC. Each of these expands the HLAC into a computation on sBLACs and scalars. Since more than one algorithm is available for each HLAC, autotuning can be used to explore different alternatives.

Next, each basic LA program is processed in two additional stages, as shown in Fig. 6. In Stage 2, each implementation is translated into a C-like intermediate representation (C-IR); if vectorization is enabled, rewriting rules are used to increase vectorization opportunities. For example, a group of scalar statements can be rewritten as a vectorizable sBLAC. C-IR includes (1) special pointers for accessing portions of matrices and vectors, (2) mathematical operations on the latter, and (3) `For` and `If` constructs with affine conditions on induction variables. In Stage 3 of Fig. 6, each C-IR program is unparsed into C code and its performance is measured. During the translation, code-level optimizations are applied. As an example, if vectorization is desired, a domain-specific load/store analysis is applied to replace explicit memory loads and stores with more efficient data rearrangements between vector variables. Finally, the code with best performance is selected as the final output.

The three stages are now explained in greater detail.

3.1 Stage 1: Basic Linear Algebra Programs Synthesis

In Stage 1, SLinGen identifies all HLACs in the input program and synthesizes for each of them algorithms that consist of *basic building blocks*, i.e., sBLACs and scalar operations.

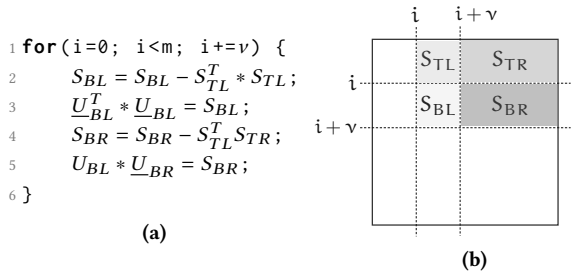


Figure 7. (a) One possible synthesized LA fragment for the HLAC in (5). (b) shows partitions of S involved in the computation; U is partitioned analogously. For better readability we underline output matrices in the HLACs (lines 3 and 5).

Identifying HLACs. SLinGen traverses all statements in the input LA program collecting all HLACs. As described by the LA grammar in Fig. 4, HLACs are characterized by the presence of an expression on their left-hand side, or by the use of a non-basic operator on the right-hand side (in our case we only consider the inverse). As an example, given the LA input in Fig. 5, SLinGen would collect the two HLACs on lines 8 and 9.

Translation into basic form. Next, SLinGen derives for each HLAC a number of loop-based algorithms. This process is implemented using an iterative extension of the Cl1ck methodology (see Sec. 2.2), which we describe using as running example line 8 of Fig. 5:

$$U^T U = S. \tag{5}$$

Both matrices are of general but fixed size $m \times m$. As we explain next, the translation first decomposes the HLAC into operations on sBLACs and small HLACs of vector-size ν, and then constructs codelets that consist of small sBLACs and scalar operations.

1) Refinement of HLACs. The refinement partitions the HLAC along each of its dimensions until only sBLACs and vector-size HLACs remain. Therefore, the first decision is how to partition the dimensions. For the example in (5), SLinGen decides to partition along rows and columns of both matrices to carry on and exploit the symmetry of S and the triangular structure of U. While any number of levels of partitioning are possible, for the sake of brevity, we assume that only two levels of partitioning are desired, with blockings of size ν (the architecture vector width) and 1. With the chosen partitioning, three algorithms are obtained for (5) (associated with three possible loop invariants); we continue with the one shown in Fig. 7. This algorithm relies on two matrix multiplications (sBLACs on lines 2 and 4), one HLAC of size $\nu \times m$ (i.e., the linear system on line 5), and one HLAC of vector-size (i.e., the recursive instance of (5) on line 3).

```
1 for(j=i+ν; j<m; j+=ν) {
2     U_BL^T * U_BR(:, j:j+ν) = S_BR(:, j:j+ν);
3 }
```

Figure 8. LA code synthesized for the HLAC in line 5 of Fig. 7.

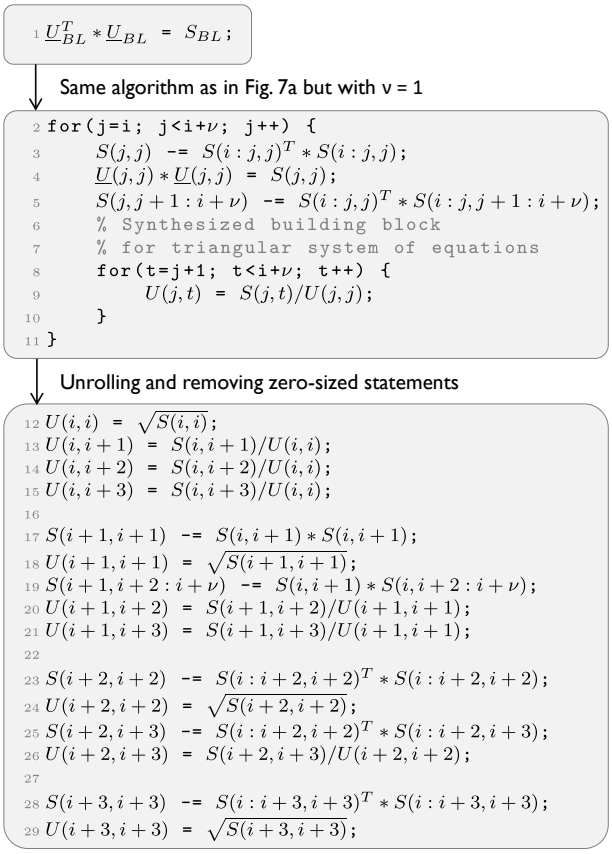


Figure 9. Code synthesis for the vector-sized HLAC in line 3 of Fig. 7. We denote with $i:j$ the interval $[i, j)$.

.

The HLAC corresponding to the linear system needs to be further lowered. Since one dimension is already of size ν, the HLAC is partitioned along the other dimension (the columns of submatrices U_{BR} and S_{BR}). SLinGen synthesizes the algorithm in Fig. 8, which is inlined in the algorithm in Fig. 7 replacing line 5. Now, the computation of the HLAC in (5) is reduced to computations on sBLACs and HLACs of size $\nu \times \nu$; SLinGen proceeds by generating the corresponding vector-size codelet for the latter.

2) Automatic synthesis of HLAC codelets. The remaining task is to generate code for the vector-sized HLACs; we use line 3 of Fig. 7 as example. Using the same process as before but with block size 1, SLinGen obtains the algorithm in Fig. 9, lines 2–11. This algorithm is then unrolled, as shown in lines 12–29, and then inlined.

No further HLACs remain and SLinGen is ready to proceed with optimizations and code generation.

Algorithm reuse. We have discussed how the iterative process of building basic linear algebra programs out of an initial LA program can require multiple algorithmic synthesis steps per HLAC in input. If two HLACs only differ in the sizes of their inputs and outputs but share the same functionality (e.g., both solve a system $L^T X = B$ for X, where L is lower triangular) their LA formulations would be based on the same algorithms. Such reuse can occur even within the same HLAC, as we have shown for the case of (5) where the building block in Fig. 9 was created based on the same algorithm initially derived for the whole computation shown in Fig. 7. For this reason SLinGen stores information about the algorithms required for building basic linear algebra forms of HLACs in a database that is queried before starting a new algorithmic synthesis step (Stage 1a in Fig. 6).

3.2 Stage 2: sBLAC tiling and vectorization

The aim is to lower the basic linear algebra programs produced in the previous stage to C-IR form. To do this, SLinGen decomposes sBLACs into vectorizable codelets (ν-BLACs), and performs optimizations to increase vectorization efficiency when possible.

In a basic linear algebra program, every statement is either an auxiliary scalar computation or an sBLAC. SLinGen proceeds first by tiling all sBLACs and decomposing them into vector-size sBLACs that can be directly mapped onto ν-BLACs, using the LGen approach described in Sec. 2.1. Next it improves the vectorization efficiency of the resulting implementation. For instance, the codelet in Fig. 9 is composed of sBLACs that can be mapped to vectorized ν-BLACs, but also of several scalar computations that could result in a low vectorization efficiency. Thus, SLinGen searches for opportunities to combine multiple scalar operations of the same type into one single sBLAC with a technique similar to the one used to identify superword-level parallelism [26].

For instance consider the pair of rules R_0 and R_1 in Table 2. R_0 combines two scalar divisions into an element-wise division of a vector by a scalar, while R_1 transforms such an element-wise division into a scalar division followed by the scaling of a vector. The application of rules R_0 and R_1 to lines 13–15 and 20–21 in Fig. 9 yields two additional sBLACs for the multiplication of a scalar times a vector as shown in Fig. 10. Similar rules for other basic operators create new sBLACs to improve code vectorization.

The basic linear algebra programs are now mapped onto ν-BLACs and translated into C-IR code.

3.3 Stage 3: Code-level optimization and autotuning

In the final stage, SLinGen performs optimizations on the C-IR code generated by the previous stage. These are similar to, or extended version of those done in LGen [42]. We focus on one extension: an improved scalarization of vector accesses enabled by a domain-specific load/store analysis. The goal of the analysis is to replace explicit memory loads and stores by shuffles in registers in the final vectorized C code. The

Table 2. Example of rewriting rules to expose more ν-BLACs. $x, b \in \mathbb{R}^k$; $\beta_i, \chi_i, \lambda, \tau \in \mathbb{R}$. Statement S_0 appears in the computation before S_1 and no operation writes to χ_1, β_1, or λ in between.

$$R_0 : \frac{S_0 : \chi_0 = \beta_0/\lambda, \quad S_1 : \chi_1 = \beta_1/\lambda}{\mathtt{x} = [\chi_0 \mid \chi_1], \mathtt{b} = [\beta_0 \mid \beta_1], \mathtt{x} = \mathtt{b}/\lambda} \tag{6}$$

$$R_1 : \frac{\mathtt{op(x)} = \mathtt{op(b)}/\lambda, \quad \mathtt{op}(\cdot) = (\cdot) \text{ or } \cdot^{\mathsf{T}}}{\tau = 1/\lambda, \quad \mathtt{op(x)} = \tau * \mathtt{op(b)}} \tag{7}$$

$\tau_0 = 1/U(i, i)$;
$U(i, i+1 : i+\nu) = \tau_0 S(i, i+1 : i+\nu)^T$;

(a)

$\tau_1 = 1/U(i+1, i+1)$;
$U(i+1, i+2 : i+\nu) = \tau_1 S(i+1, i+2 : i+\nu)^T$;

(b)

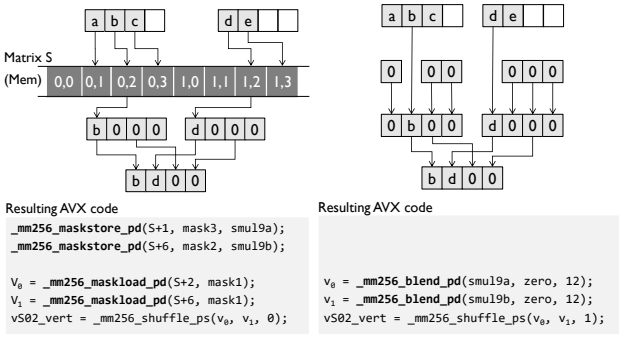
Figure 10. Application of rules in Table 2 to (a) lines 13–15 and (b) lines 20–21 in the code in Fig. 9, which yields additional ν-BLACs (second line of both (a) and (b)).

```
// Store sca. mul. in Fig. 9a. U overwrites S.
Vecstore(S+1, smul9a, [0, 1, 2], hor);
...
// Store sca. mul. in Fig. 9b. U overwrites S.
Vecstore(S+6, smul9b, [0, 1], hor);
...
// Load from Fig. 12, l.23
__m256d vS02_vert = Vecload(S+2, [0, 1], vert);
```

Figure 11. C-IR code snippet for the access $S(i : i+2, i+2)$ with $i = 0$ on line 23 of Fig. 9. `Vecstore(addr, var, [`$p_0, p_1, \ldots$`], hor/vert)` (and analogous Vecload) is a C-IR vector instruction with the following meaning: Store vector variable `var` placing the element at position p_i at the ith memory location starting from address addr in horizontal (vertical) direction.

technique is domain-specific as memory pointers are associated with the mathematical layout. We explain the approach with an example. Consider the access $S(i : i+2, i+2)$ in Fig. 9, line 23. The elements gathered from S were computed in lines 13–15 and 20-21, which were rewritten to the code in Fig. 10 as explained before. Note that in this computation U overwrites S (see specification in Fig. 5, line 4). The associated C-IR store/load sequence for $i = 0$ is shown in Fig. 11, and would yield the AVX code in Fig. 12a. However, by analyzing their overlap, we can deduce that element 1 of the first vector (`smul19a`) goes into element 0 of the result one, while element 0 of the second vector (`smul19b`) into element 1. This means the stores/loads can be replaced by a blend instruction in registers as shown in Fig. 12b.

Autotuning. Finally, SLinGen unparses the optimized C-IR code into C code and its performance is measured. If several algorithms are available for the occurring HLACs, autotuning is used to select the fastest.



(a) **(b)**

Figure 12. Resulting AVX code for the C-IR snippet in Fig. 11 without (a) and with (b) load/store analysis. In (a) `vS02_vert` is obtained by explicitly storing to and loading from memory while in (b) by shuffling vector variables.

Table 3. Selected HLAC benchmarks. All matrices $\in \mathbb{R}^{n \times n}$. A is symmetric positive definite, S is symmetric, and L and U are lower and upper triangular, respectively. X is the output in all cases.

Name	Label	Computation
Cholesky dec.	*potrf*	$X^T X = A$, X upper triangular
Sylvester eq.	*trsyl*	$LX + XU = C$, X general matrix
Lyapunov eq.	*trlya*	$LX + XL^T = S$, X symmetric
Triangular inv.	*trtri*	$X = L^{-1}$, X lower triangular

4 Experimental Results

We evaluate SLinGen for two classes of computations: HLACs and linear algebra applications (see Fig. 1).

HLACs. We selected four HLACs common in applications: the Cholesky decomposition (*potrf*), the solution of triangular, continuous-time Sylvester and Lyapunov equations (*trsyl* and *trlya*, respectively), and the inverse of a triangular matrix (*trtri*). We provide their definitions in Table 3.

Applications. We selected three applications from different domains: (a) The Kalman filter (*kf*) for control systems which was introduced in Sec. 1, (b) a version of the Gaussian process regression [38] (*gpr*) used in machine learning to compute the predictive mean and variance for noise free test data, and (c) an L1-analysis convex solver [2] (*l1a*) used, for example, in image denoising and text analysis. We list the associated LA programs in Fig. 13.

4.1 Experimental setup

All tests are single-threaded and executed on an Intel Core i7-2600 CPU (Sandy Bridge) running at 3.3 GHz, with 32 kB L1 D-cache, 256 kB L2 cache, and support for AVX, under

Input: F, B, Q, H, R, P, u, x, z
Output: P, x

$y = F * x + B * u$;
$Y = F * P * F^T + Q$;
$v_0 = z - H * y$;
$M_1 = H * Y$;
$M_2 = Y * H^T$;
$M_3 = M1 * H^T + R$;
$\underline{U}^T * \underline{U} = M_3$;
$U^T * \underline{v_1} = v_0$;
$U * \underline{v_2} = v_1$;
$U^T * \underline{M_4} = M_1$;
$U * \underline{M_5} = M_4$;
$x = y + M_2 * v_2$;
$P = Y - M_2 * M_5$;

(a) Program *kf*

Input: K, X, x, y
Output: ϕ, ψ, λ

$\underline{L} * \underline{L}^T = K$;
$L * \underline{t_0} = y$;
$L^T * \underline{t_1} = t_0$;
$k = X * x$;
$\phi = k^T * t_1$;
$L * \underline{v} = k$;
$\psi = x^T * x - v^T * v$;
$\lambda = y^T * t_1$;

(b) Program *gpr*

Input: W, A, x_0, y, v_1, z_1, v_2, z_2, α, β, τ
Output: v_1, z_1, v_2, z_2

$y_1 = \alpha * v_1 + \tau * z_1$;
$y_2 = \alpha * v_2 + \tau * z_2$;
$x_1 = W^T * y_1 - A^T * y_2$;
$x = x_0 + \beta * x_1$;
$z_1 = y_1 - W * x$;
$z_2 = y_2 - (y - A * x)$;
$v_1 = \alpha * v_1 + \tau * z_1$;
$v_2 = \alpha * v_2 + \tau * z_2$;

(c) Program *l1a*

Figure 13. Selected application benchmarks. The declaration of input and output elements is omitted but we underline output matrices and vectors in all HLACs. All matrices and vectors are of size $n \times n$ and n, respectively. Both *kf* and *l1a* are iterative algorithms and we limit our LA implementation to a single iteration. The original algorithms of both *gpr* and *l1a* contain additionally a small number of min, max, and log operations, which have very minor impact on the overall cost.

Ubuntu 14.04 with Linux kernel v3.13. Turbo Boost is disabled. In the case of *potrf*, *trsyl*, *trlya*, and *trtri* we compare with: (a) the Intel MKL library v11.3.2, (b) ReLAPACK [33], (c) Eigen v3.3.4 [18], straightforward code (d) compiled with Intel icc v16, and (e) clang v4 with the polyhedral Polly optimizer [16], and (f) the implementation of algorithms generated by Cl1ck implemented with MKL. For *trsyl* we also compare with RECSY [23], a library specifically designed for these solvers. In the case of *kf*, *gpr*, and *l1a* we compare against library-based implementations using MKL and Eigen. Note that starting with v11.2, Intel MKL added specific support for small-scale, double precision matrix multiplication (*dgemm*).

The straightforward code is scalar, handwritten, loop-based code with hardcoded sizes of the matrices. It is included to show optimizations performed by the compiler. For icc we use the flags -O3 -xHost -fargument-noalias -fno-alias -no-ipo -no-ip. Tests with clang/Polly were compiled with flags -O3 -mllvm -polly -mllvm -polly-vectorizer=stripmine. Finally, tests with MKL are linked to the sequential version of the library using flags from the Intel MKL Link Line Advisor.[2] In Eigen we used fixed-size Map interfaces to existing arrays, no-alias assignments, in-place computations of solvers, and enabled AVX code generation. All code is in double precision and the working set fits in the first two levels of cache.

Plot navigation. The plots present performance in flops per cycles (f/c) on the y-axis and the problem size on the x-axis. All matrices and vectors are of size $n \times n$ and n, respectively. The peak performance of the CPU is 8 f/c. A bold black line represents the fastest SLinGen-generated code. For HLACs, generated alternative based on different Cl1ck-generated algorithms are shown using colored, dashed lines without markers. The selection of the fastest is thus a form of algorithmic autotuning.

Every measurement was repeated 30 times on different random inputs. The median is shown and quartile information is reported with whiskers (often too small to be visible). All tests were run with warm cache.

4.2 Results and analysis

HLACs. Figure 14 shows the performance for the HLAC benchmarks. In the left column of Fig. 14, we compare performance results for all competitors except Cl1ck, for which we provide a more detailed comparison reported on the plots on the right column. Cl1ck generates blocked algorithms with a variable block size n_b and uses a library for the occuring BLAS functions (here: MKL BLAS). For each HLAC benchmark we measure performance for $n_b \in \{\nu, n/2, n\}$, where $\nu = 4$ is the ISA vector length.

For *potrf* (Fig. 14a) SLinGen generates code that is on average 2×, 1.8×, and 3.8× faster than MKL, ReLAPACK, and Eigen, respectively. Compared to icc and clang/Polly we obtained a larger speedup of 4.2× and 5.6× showing the limitations of compilers. SLinGen is also about 40% faster than Cl1ck code. As expected, the performance of Cl1ck's implementation is very close to that of the MKL library. For *trsyl* (Fig. 14b) our system reaches up to 2 f/c and is typically 2.8×, 2.6×, 12×, 4×, 1.5×, and 2× faster than MKL, ReLAPACK, RECSY, Eigen, icc, and clang/Polly, respectively.

[2] software.intel.com/en-us/articles/intel-mkl-link-line-advisor

When compared with Cl1ck's implementation, SLinGen results 5.3× faster. The computation of *trlya* (Fig. 14c) attains around 1.7 f/c for the larger sizes, thus being 5× faster than the libraries and 2× than the compilers. Note that missing a specialized interface for this function, MKL performs more than 2× slower than icc. Fig. 14d shows *trtri* where SLinGen achieves up to 3.5 f/c, with an average speedup of about 2.5× with respect to MKL and Cl1ck's most performant implementation. When comparing with the other competitors, SLinGen is up to 2.3×, 21×, 4.2×, and 4.6× faster than MKL, ReLAPACK, Eigen, icc and clang/Polly, respectively.

Applications. Figure 15 shows the performance for the chosen applications. For *kf*, we show two plots. Fig. 15a varies the state size (i.e., dimension of x and P) and sets the observation size (i.e., dimension of z, H, and R) equal to it. SLinGen generates code which is on average 1.4×, 3×, and 4× faster than MKL, Eigen, and icc. Note that typical sizes for *kf* tend to lie on the left half, where we observe even larger speedups. Fig. 15b fixes the state size to 28 and varies the size of the observation size between 4 and 28.

On *gpr*, our generated code performs similarly to MKL, while being 1.7× faster than icc and Eigen. Finally, for the *l1a* test, SLinGen generated code that is on average 1.6×, 1.3×, and 1.5× faster than MKL, Eigen, and icc, respectively.

To judge the quality of the generated code, we study more in depth the code synthesized for the HLAC benchmarks.

Bottleneck analysis. We examined SLinGen's generated code with ERM [7], a tool that performs a generalized roofline analysis to determine hardware bottlenecks. ERM creates the computation DAG using the LLVM Interpreter [17, 27] and a number of microarchitectural parameters capturing throughput and latency information of instructions and the memory hierarchy. We ran ERM with the parameters describing our Sandy Bridge target platform to analyze our four HLAC benchmarks.

Table 4 summarizes for each routine and three sizes the hardware bottleneck found with ERM. In general we notice that the generated code is limited by two main factors. For small sizes it is the cost of divisions/square roots, which on Sandy Bridge can only be issued every 44 cycles. The fraction of these is asymptotically small (approximately $1/n^2$ for *potrf*, and $1/n$ for *trsyl*, *trlya*, and *trtri*) but matters for small sizes as they are also sequentially dependent. We believe that this effect plagues our *gpr*, which suffers from divisions by Cholesky decomposition and triangular solve.

For larger sizes, the number of loads and stores between registers and L1 grows mostly due to spills. SLinGen tends to fuse several innermost loops of several neighbouring sBLACs, which can result in increased register pressure due to several intermediate computations. This effect could be improved by better scheduling strategies based on analytical models [28].

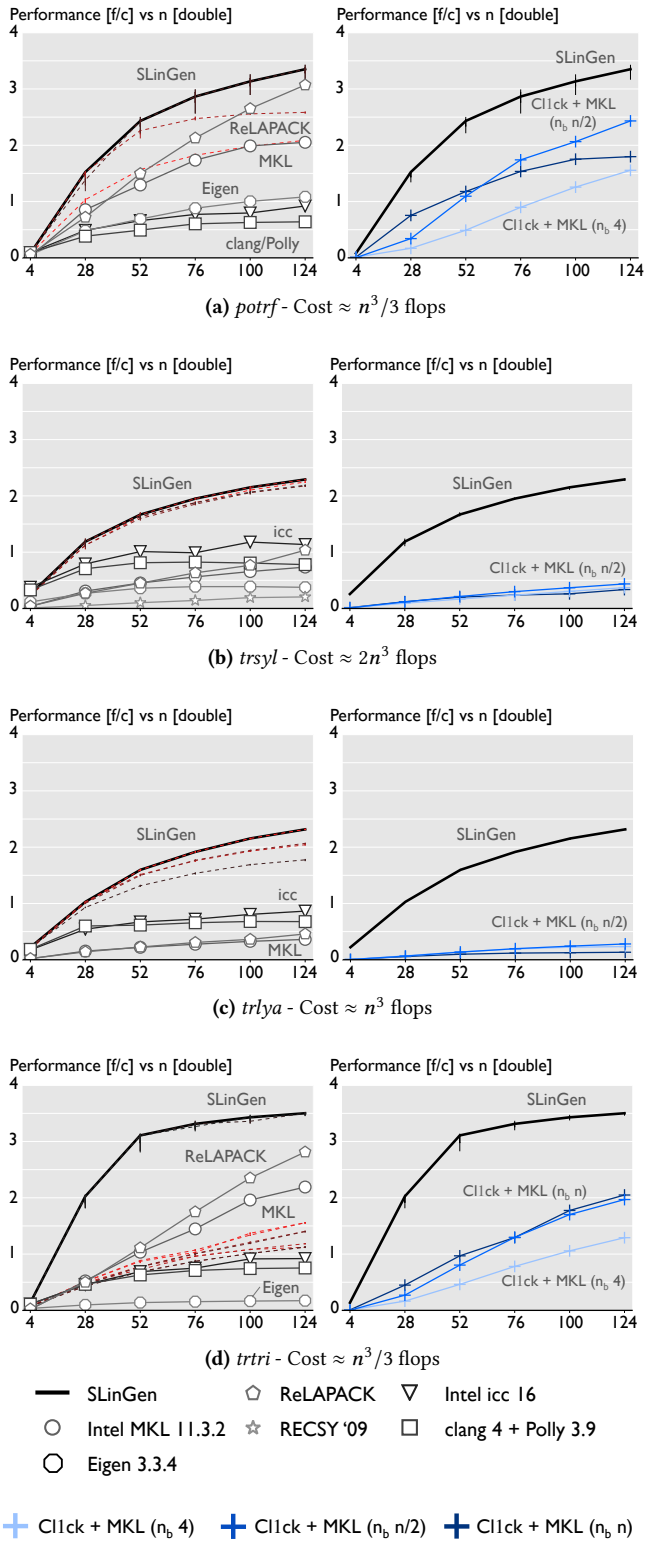
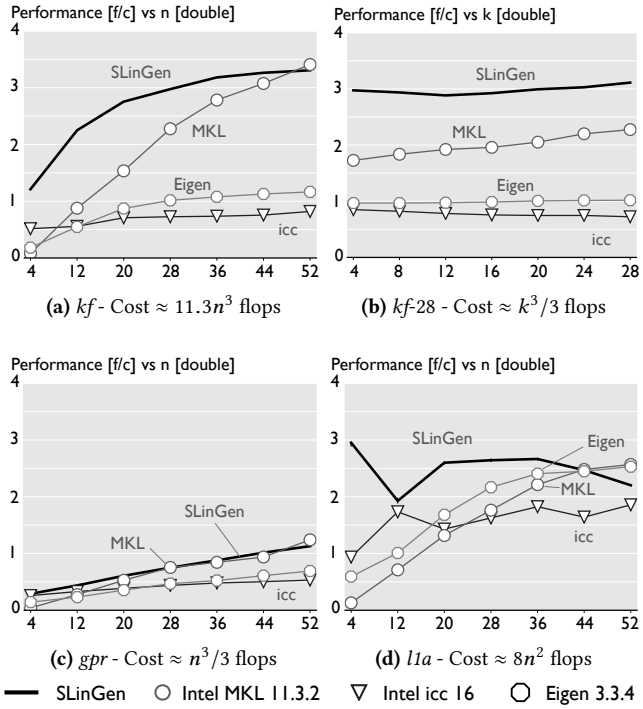


Figure 14. Performance plots for the HLAC benchmarks: (a) *potrf*, (b) *trsyl*, (c) *trlya*, and (d) *trtri*. On the left column: colored, dashed lines without markers indicate different SLinGen-generated algorithms.



(a) *kf* - Cost $\approx 11.3n^3$ flops **(b)** *kf*-28 - Cost $\approx k^3/3$ flops

(c) *gpr* - Cost $\approx n^3/3$ flops **(d)** *l1a* - Cost $\approx 8n^2$ flops

Figure 15. Performance plots for the application-level programs in fig 13. *kf*-28 is based on the same *kf* program but fixes the number of state parameters to 28. Inputs z, H, and R have now size k, $k \times 28$, and $k \times k$, respectively.

Finally, the third column in Table 4 shows the ratio of issued shuffles and blends to the total number of issued instructions, excluding loads and stores. This number can be considered as an estimate of how much data rearrangement overhead is introduced by SLinGen's vectorization strategy. As estimated by ERM's bottleneck analysis, the peak performance would almost never be affected by the introduced shuffles and blends (forth and fifth column).

5 Limitations and Future Work

SLinGen is a research prototype and naturally has limitations that future work can address.

First, SLinGen is currently restricted to fixed input and output sizes and expects structured matrices to use a full storage scheme. Many relevant applications, e.g., in signal processing, control, and optimization fulfil this constraint. However, for others a library for general input sizes is still desirable.

Second, when a linear algebra computation is composed of many small subcomputations, SLinGen could take advantage of modern multicore systems. The idea is to identify task parallelism among independent subcomputations and allocate them to different cores of a processor.

Third, we only consider computations on a single input. However, SLinGen could be extended to handle batched computations, i.e., a group of independent computations that can fit into cache together and could be processed by one single function to better take advantage of data and task parallelism.

Finally, the generated code is certainly not performance-optimal. More could be achieved by additional lower level optimizations that address the issues identified in our bottleneck analysis.

Table 4. Summary of bottleneck analysis with ERM. The shuffle/-blend issue rate is the ratio of shuffle/blend issues to the total issued instructions (excluding loads and stores). The achievable peak performance when taking shuffles/blends into account is shown in the last columns.

Computation	Sizes	Bottleneck	Issue rate Shuffles & blends	Perf limit (f/c) Shuffles	Blends
potrf	4	divs/sqrt	50%	6.5	8
	76	L1 stores	15%	8	8
	124	L1 stores	10%	8	8
trsyl	4	divs	52%	5.2	8
	76	divs	35%	8	8
	124	L1 loads	35%	8	8
trlya	4	divs	55%	4.5	8
	76	divs	40%	6.7	8
	124	L1 loads	37%	6.8	8
trtri	4	divs	62%	3.2	8
	76	L1 loads	32%	8	8
	124	L1 loads	32%	8	8

6 Related Work

Here we review the existing body of work on the optimization of linear algebra computations, performed either by hand or automatically.

Hand-optimized linear algebra libraries. Multiple libraries offer high-performance implementations of BLAS and LAPACK interfaces. Prominent examples include the commercial Intel MKL [22] and the open-source OpenBLAS [47]. BLIS [44] is a framework for instantiating a superset of the BLAS operations from a set of microkernels; however, it does not does cover higher-level operations or entire applications. ReLAPACK [33] and RECSY [23] provide recursive high-performance implementations of LAPACK and Sylvester-type computations, respectively.

Algorithm Synthesis. The Formal Linear Algebra Methods Environment (FLAME) [19] provides a methodology for automatically deriving algorithms for higher level linear algebra functions [4] given as mathematical equations. The supported functions are mostly those covered by the LAPACK library and the generated algorithms rely on the availability of a BLAS library. The methodology is completely automated by the Cl1ck compiler [9, 10] which we used and extended in this work.

Library focus on small computations. Recently, the problem of efficiently processing small to medium size computations has attracted the attention of hardware and library developers. The LIBXSMM library [21] provides an assembly code generator for small dense and sparse matrix multiplication, specifically for Intel platforms. Intel MKL has introduced fast general matrix multiplication (GEMM) kernels on small matrices, as well as a specific functionality for batches of matrices with same parameters, such as size and leading dimensions (GEMM_BATCH). The very recent [15] provides carefully optimized assembly for small-scale BLAS-like operations. The techniques used in writing these kernels could be incorporated into SLinGen.

DSL-based approaches. PHiPAC [5] and ATLAS [46] are two of the earliest generators, both aiming at high-performance GEMM by parameter tuning. Higher-level generators for linear algebra include the CLAK compiler [11, 12], the DxTer system [29], and BTO [40]. CLAK finds efficient mappings of matrix equations onto building blocks from high-performance libraries such as BLAS and LAPACK. SLinGen could benefit from a CLAK-like high-level front-end to perform mathematical reasoning on more involved matrix computations that require manipulation. DxTer transforms blocked algorithms such as those generated by Cl1ck, and applies transformations and refinements to output high-performance distributed-memory implementations. BTO focuses on memory bound computations (BLAS 1-2 operations) and relies on a compiler for vectorization. LINVIEW [30] is a framework for incremental maintenance of analytical queries expressed in terms of linear algebra programs. The goal of the system is to propagate within a (large) computation only the changes caused by (small) variations in the input matrices.

The MATLAB Coder [43] supports the generation of C and C++ functions from most of the MATLAB language but produces only scalar code without explicit vectorization. Julia [3] is is a high-level dynamic language that targets scientific computing. Linear algebra operations in Julia are mapped to BLAS and LAPACK calls.

Also related are expression template-based DSLs like the Eigen [18] and the uBLAS [45] libraries. In particular, Eigen provides vectorized code generation, supporting a variety of functionalities including Cholesky and LU factorizations. However, libraries based on C++ metaprogramming cannot take advantage of algorithmic or implementation variants. Another approach based on metaprogramming is taken by the Hierarchically Tiled Arrays (HTAs) [20], which offer data types with the ability to dynamically partition matrices and vectors, automatically handling situations of overlapping areas. HTAs priority, however, is to improve programmability reducing the amount of code required to handle tiling and data distribution in parallel programs, leaving any optimization to the programmer (or program generator).

Finally, our approach is in concept related to Spiral, a generator for the different domain of linear transforms [35, 36] and also uses ideas from the code generation approach of the LGen compiler [25, 41, 42]. Spiral is a generator for linear transforms (like FFT, a very different domain) and not for the computations considered in this paper. There was an effort to extend it to linear algebra [14] but was shown only for matrix-matrix multiplication. The connection between SLinGen and Spiral is in concept: The translation from math to code and the use of DSLs to apply optimizations at a high level of abstraction.

Optimizing compilers. In Sec. 4 we compare against Polly [16], an optimizer for the clang compiler based on the polyhedral model [13]. This and other techniques reschedule computation and data accesses to enhance locality and expose parallelization and vectorization opportunities [6, 24]. Multi-platform vectorization techniques such as those in [31, 32] use abstract SIMD representations making optimizations such as alignment detection portable across different architectures. The work in [37] leverages whole-function vectorization at the C level, differently from other frameworks that deviate from the core C language, such as Intel ispc [34]. The scope of these and other optimizing compilers is more general than that of our generator. On the other hand, we take advantage of the specific domain to synthesize vectorized C code of higher performance.

7 Conclusions

This paper pursues the vision that performance-optimized code for well-defined mathematical computations should be generated directly from a mathematical representation. This way a) complete automation is achieved; b) domain knowledge is available to perform optimizations that are difficult or impossible for compilers at a high level of abstraction; c) porting to new processor architectures may be simplified. In this paper we proposed a prototype of such a system, called SLinGen, for small-scale linear algebra as needed in control, signal processing, and other domains. While limited in scope, it goes considerably beyond prior work on automation for linear algebra, which mainly focused on library functions for BLAS or LAPACK; SLinGen compiles entire linear algebra programs and still obtains competitive or superior performance to handwritten code. The methods we used are a combination of DSLs, program synthesis and generation, symbolic computation, and compiler techniques. There is active research on these topics, which should also provide ever better tools and language support to facilitate the development of more, and more powerful generators like SLinGen.

Acknowledgments

Financial support from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through grants GSC 111 and BI 1533/2–1 is gratefully acknowledged.

References

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

[2] S. R. Becker, E. J. Candès, and M. C. Grant. 2011. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical Programming Computation* 3, 3 (2011), 165.

[3] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98.

[4] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. 2005. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software (TOMS)* 31, 1 (2005), 1–26.

[5] J. Bilmes, K. Asanovic, C. W. Chin, and J. Demmel. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing (ICS)*. 340–347.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Programming Language Design and Implementation (PLDI)*. 101–113.

[7] V. Caparrós Cabezas and M. Püschel. 2014. Extending the Roofline Model: Bottleneck Analysis with Microarchitectural Constraints. In *IEEE International Symposium on Workload Characterization (IISWC)*. 222–231.

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

[9] D. Fabregat-Traver and P. Bientinesi. 2011. Automatic Generation of Loop-Invariants for Matrix Operations. In *International Conference on Computational Science and its Applications (ICCSA)*. 82–92.

[10] D. Fabregat-Traver and P. Bientinesi. 2011. Knowledge-Based Automatic Generation of Partitioned Matrix Expressions. In *Computer Algebra in Scientific Computing*. 144–157.

[11] D. Fabregat-Traver and P. Bientinesi. 2013. Application-tailored Linear Algebra Algorithms: A Search-Based Approach. *International Journal of High Performance Computing Applications (IJHPCA)* 27, 4 (2013), 425 – 438.

[12] D. Fabregat-Traver and P. Bientinesi. 2013. A Domain-Specific Compiler for Linear Algebra Operations. In *High Performance Computing for Computational Science (VECPAR 2012) (Lecture Notes in Computer Science (LNCS))*, Vol. 7851. 346–361.

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

[14] F. Franchetti, F. Mesmay, D. Mcfarlin, and M. Püschel. 2009. Operator Language: A Program Generation Framework for Fast Kernels. In *IFIP Working Conference on Domain-Specific Languages (DSL WC) (Lecture Notes in Computer Science (LNCS))*, Vol. 5658. Springer, 385–410.

[15] G. Frison, D. Kouzoupis, A. Zanelli, and M. Diehl. 2017. BLASFEO: Basic linear algebra subroutines for embedded optimization. *CoRR* abs/1704.02457 (2017). http://arxiv.org/abs/1704.02457

[16] T. Grosser, A. Groesslinger, and C. Lengauer. 2012. Polly — Performing Polyhedral Optimizations On A Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.

[17] LLVM Developer Group. 2017. The LLVM Compiler Infrastracture. http://llvm.org/.

[18] G. Guennebaud, B. Jacob, et al. 2017. Eigen. http://eigen.tuxfamily.org.

[19] J. A. Gunnels, F. G. Gustavson, G. Henry, and R. A. van de Geijn. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software (TOMS)* 27, 4 (2001), 422–455.

[20] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua. 2008. Programming with tiles. In *Principles and Practice of Parallel Programming (PPoPP)*. 111–122.

[21] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[22] Intel. 2017. Intel Math Kernel Library (MKL). software.intel.com/en-us/intel-mkl.

[23] I. Jonsson and B. Kågström. 2002. Recursive Blocked Algorithms for Solving Triangular Systems — Part I: One-sided and Coupled Sylvester-type Matrix Equations. *ACM Transactions on Mathematical Software (TOMS)* 28, 4 (2002), 392–415.

[24] M. Kong, R. Veras, K. Stock, F. Franchetti, L. N. Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Programming Language Design and Implementation (PLDI)*. 127–138.

[25] N. Kyrtatas, D. G. Spampinato, and M. Püschel. 2015. A Basic Linear Algebra Compiler for Embedded Processors. In *Design, Automation and Test in Europe (DATE)*. 1054–1059.

[26] S. Larsen and S. Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Programming Language Design and Implementation (PLDI)*. 145–156.

[27] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Code Generation and Optimization (CGO)*. 75–86.

[28] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Transactions on Mathematical Software (TOMS)* 43, 2 (2016), 12:1–12:18.

[29] B. Marker, J. Poulson, D. Batory, and R. van de Geijn. 2013. Designing Linear Algebra Algorithms by Transformation: Mechanizing the Expert Developer. In *High Performance Computing for Computational Science (VECPAR 2012)*. Lecture Notes in Computer Science (LNCS), Vol. 7851. Springer, 362–378.

[30] M. Nikolic, M. ElSeidy, and C. Koch. 2014. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *Management of Data (SIGMOD)*. 253–264.

[31] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. 2011. Vapor SIMD: Auto-vectorize once, run everywhere. In *International Symposium on Code Generation and Optimization (CGO)*. 151–160.

[32] D. Nuzman, I. Rosen, and A. Zaks. 2006. Auto-vectorization of interleaved data for SIMD. In *Programming Language Design and Implementation (PLDI)*. 132–143.

[33] E. Peise and P. Bientinesi. 2016. Recursive Algorithms for Dense Linear Algebra: The ReLAPACK Collection. (2016). Available at: http://arxiv.org/abs/1207.2169.

[34] M. Pharr and W. R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar)*. 1–13.

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

[36] 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. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[37] G. Rapaport, A. Zaks, and Y. Ben-Asher. 2015. Streamlining Whole Function Vectorization in C Using Higher Order Vector Semantics. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. 718–727.

[38] C. E. Rasmussen and C. K. I. Williams. 2005. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

[39] L. L. Scharf. 1991. *Probability, Statistical Signal Processing, Detection, Estimation, and Time Series Analysis*. Addison-Wesley.

[40] J. G. Siek, I. Karlin, and E. R. Jessup. 2008. Build to order linear algebra kernels. In *International Parallel & Distributed Processing Symposium*

(IPDPS). 1–8.

[41] D. G. Spampinato and M. Püschel. 2014. A Basic Linear Algebra Compiler. In *International Symposium on Code Generation and Optimization (CGO)*. 23–32.

[42] D. G. Spampinato and M. Püschel. 2016. A Basic Linear Algebra Compiler for Structured Matrices. In *International Symposium on Code Generation and Optimization (CGO)*. 117–127.

[43] The MathWorks, Inc. 2017. MATLAB Coder. https://www.mathworks.com/products/matlab-coder.html.

[44] F. G. Van Zee and R. A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software (TOMS)* 41, 3, Article 14 (2015), 33 pages.

[45] J. Walter, M. Koch, et al. 2012. uBLAS. www.boost.org/libs/numeric/ublas.

[46] R. C. Whaley and J. J. Dongarra. 1998. Automatically tuned linear algebra software. In *Supercomputing (SC)*. 1–27.

[47] X. Zhang. 2017. The OpenBLAS library. http://www.openblas.net/.