A Basic Linear Algebra Compiler

Daniele G. Spampinato
Dept. of Computer Science, ETH Zurich
danieles@inf.ethz.ch

Markus Püschel
Dept. of Computer Science, ETH Zurich
pueschel@inf.ethz.ch

ABSTRACT

Many applications in media processing, control, graphics, and other domains require efficient small-scale linear algebra computations. However, most existing high performance libraries for linear algebra, such as ATLAS or Intel MKL are more geared towards large-scale problems (matrix sizes in the hundreds and larger) and towards specific interfaces (e.g., BLAS). In this paper we present LGen: a compiler for small-scale, basic linear algebra computations. The input to LGen is a fixed-size linear algebra expression; the output is a corresponding C function optionally including intrinsics to efficiently use SIMD vector extensions. LGen generates code using two levels of mathematical domain-specific languages (DSLs). The DSLs are used to perform tiling, loop fusion, and vectorization at a high level of abstraction, before the final code is generated. In addition, search is used to select among alternative generated implementations. We show benchmarks of code generated by LGen against Intel MKL and IPP as well as against alternative generators, such as the C++ template-based Eigen and the BTO compiler. The achieved speed-up is typically about a factor of two to three.

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, Small matrices, DSL, Tiling, SIMD vectorization

1. INTRODUCTION

After decades of experience in high performance computing it is well understood how to develop fast dense linear algebra code. For basic computations, commercial and non-commercial high performance libraries exist that comply with the BLAS (basic linear algebra subroutines) interface; higher level linear algebra routines such as LU or Cholesky factorization are built on top of BLAS. These libraries provide excellent performance for the large problem sizes needed in scientific computing and many other domains.

However, many performance-critical applications require linear algebra computations at a smaller scale, often for specific input sizes, and sometimes with no direct match to BLAS functions. Examples include optimization algorithms, Kalman filters, geometric transformations and other routines that occur in media processing, computer vision, control, and graphics. To provide better library support for these, we propose a compiler that translates a basic linear algebra computation (BLAS or not BLAS) into efficient code. To better explain our contribution, we first give a brief review of the related work.

High performance basic linear algebra libraries. Several highly optimized BLAS libraries [7, 6] exist including ATLAS [34], GotoBLAS [15], and Intel MKL [23]. However, for smaller sizes, their performance can be suboptimal compared to what is achievable (e.g., [29] and the results in this paper) and the interface may not match a desired computation. For this reason, Intel IPP [22] includes a section called Intel IPP MX devoted to small scale linear algebra operations with a non-BLAS interface. IPP and MKL will be among our benchmarks.

Generators for linear algebra. The libraries mentioned above are implemented and optimized by hand. Various approaches have worked on automation. Among the earliest efforts are PHiPAC [5] and the ATLAS generator [34], which iteratively tune implementation parameters, such as block size and loops order, using the runtime as feedback (autotuning). Both are focused on BLAS and large sizes.

The Build to Order BLAS (BTO) [30, 3] is a domain-specific compiler for matrix computations. BTO is not bound to the BLAS interface and optimizes for loop fusion, data partitioning, and parallelism using autotuning. However, BTO relies on compilers for vectorization. A different generative approach is adopted by Eigen [18], uBLAS [33], and the Matrix Template Library (MTL) [17]. They use C++ expression templates to optimize the code at compile time. Optimizations include loop fusion, unrolling, and SIMD vectorization [16]. However, they lack runtime feedback and hence do not support autotuning. We will use BTO and Eigen as benchmarks.

FLAME [19] provides a methodology for automatically deriving algorithms for higher level linear algebra functions [4] given as mathematical equations. The supported functions

CGO '14 February 15 - 19 2014, Orlando, FL, USA
Copyright 2014 ACM 978-1-4503-2670-4/14/02 ...$15.00.
http://dx.doi.org/10.1145/2544137.2544155.

are mostly those covered by the LAPACK library [1] and the generated algorithms rely on the availability of an efficient BLAS library. At an even higher level of abstraction operates the linear algebra compiler presented in [8]. It decomposes a linear algebra target equation into a sequence of computations provided by BLAS or LAPACK and generates associated Matlab code. The approach exploits domain knowledge and properties of the operands by rewriting and inference rules. Both works are similar in spirit to ours (since they start with a mathematical description) but target more complex functions compared to the basic linear algebra computations considered here.

Generators in other domains. Other program generators were developed for signal processing. For example, genfft [13] generates the small size FFTs (codelets) needed in FFTW [14], and Spiral [28, 27] uses domain-specific languages (DSLs) for optimizations such as loop merging [11] and vectorization [10]. Our work aims to build a functionality similar to genfft in [13] for basic linear algebra using an approach similar to Spiral.

Optimizing compilers. A third approach to fast linear algebra code is the use of optimizing compilers. Polyhedral compilers can perform loop optimizations, tiling, and vectorization for imperfectly nested loops [21, 24]. Other vectorization techniques for loops include [26, 25, 2]. All these apply code transformations to expose code portions amenable to optimizations such as vectorization. Since the scope is more general, a specific linear algebra compiler should yield better results. We benchmark against straightforward loop code compiled with a state-of-the-art vendor compiler.

Contributions. We make the following main contributions:

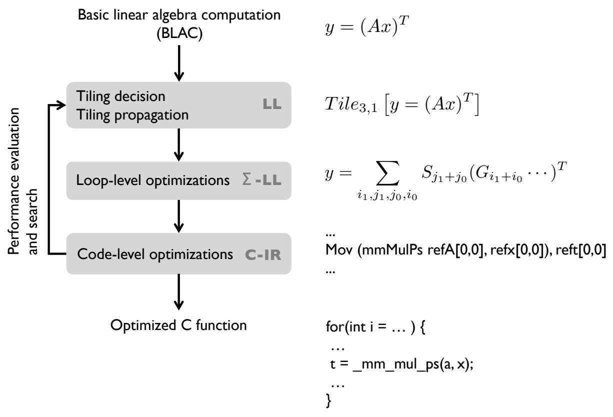
- We present a novel approach to generating efficient code for basic linear algebra computations. The approach consists of two levels of mathematical DSLs that are used to perform loop optimizations and vectorization. We explain the design and discuss limitations.
- We present a vectorization methodology that is easily portable to new vector architectures and that handles left-over code efficiently.
- We show performance benchmarks with libraries (Intel MKL and IPP), generators (BTO and Eigen), and compiler-optimized code. The results show a significant speed-up in many cases.

2. OVERVIEW

We present LGen, a compiler for performance-optimized basic linear algebra computations (BLACs) of fixed size. By "basic linear algebra" we mean computations on matrices, vectors (which are viewed as special matrices), and scalars that are composed from matrix multiplication, matrix addition, transposition, and scalar multiplication. We denote scalars with $\alpha, \beta, \dots$, matrices with $A, B, \dots$, and (column) vectors with $x, y, \dots$. BLACs are specified by equations such as $\alpha = x^T y$, $y = Ax$, $A = BC + \alpha D + A$, or

$$\beta = (Ax + \alpha y)^T z + \beta. \tag{1}$$

All terms on the right-hand side are inputs and the left-hand side is the output (which can also be an input). A valid input to LGen is for example (1) plus the data type (currently float or double) plus the sizes of the objects, e.g.,


Basic linear algebra computation (BLAC)
Tiling decision
Tiling propagation
LL
Loop-level optimizations
Σ-LL
Code-level optimizations
C-IR
Optimized C function
Performance evaluation and search
$y = (Ax)^T$
$Tile_{3,1}\left[y = (Ax)^T\right]$
$y = \sum_{i_1,j_1,j_0,i_0} S_{j_1+j_0}(G_{i_1+i_0}\cdots)^T$

```
...
Mov (mmMulPs refA[0,0], refx[0,0]), reft[0,0]
...
```

```
for(int i = … ) {
 …
 t = _mm_mul_ps(a, x);
 …
}
```

Figure 1: Architecture of LGen.

A is 5×9, x has length 9, y and z have length 5. The output is a C function (optionally vectorized using intrinsics) implementing the given BLAC.

Next we give a high level overview of the code generation approach. The details are explained in Sections 3 and 4.

Code generation overview. The basic structure of LGen is shown in Fig. 1. The input is a BLAC as specified above in what we call linear algebra language (LL). In the first step a tiling strategy is chosen. This is done by annotating the BLAC with a tile size (e.g., 3×2), which is then propagated to derive the tile sizes of all in- and outputs. Hierarchical tiling is possible. Next, the resulting LL expression is converted to another language called Σ-LL that is still based on linear algebra but makes access patterns and loops explicit. At this level, loop merging and possible loop exchanges are performed. Next, a C-intermediate representation (C-IR) is generated to perform loop unrolling, scalar replacement, and conversion into SSA form. Finally, the C function is generated; its performance is used in a feedback-driven search (autotuning).

If vectorized code (with intrinsics) is desired, the vector length ν of the ISA is an input together with the BLAC and impacts the tiling decision. The generated code uses a small set of pre-implemented building blocks, called ν-BLACs. The process is explained in Section 4.

Discussion. A few aspects are worth clarifying about our approach.

- *Optimizations and problem size:* In principle, our approach can generate code for any (but fixed) input size. However, the optimizations performed by LGen (register-level blocking, code style, vectorization) are right now geared towards small, cache resident sizes.
- *Relationship between BLAS and BLAC:* BLAS contains only a specific set of linear algebra computations: those needed by LAPACK. For these, the input sizes are parameters, and strided data layouts are supported. In contrast, our BLACs allow for a larger set of computations but are specialized to the input sizes, and, at the moment, support only contiguous data.
- *Relationship to Spiral:* LGen is designed closely after Spiral and also uses ideas from FLAME and HTA [20]. The input is a mathematical description and Σ-LL is a generalization of Σ-SPL [11], which is also used for loop optimizations. In contrast to OL [9], a previous

```
beta  = Scalar()
A     = Matrix(5, 9)
x     = Matrix(9, 1)
alpha = Scalar()
y     = Matrix(5, 1)
z     = Matrix(5, 1)
Generate(beta = (A*x+alpha*y)^T*z + beta, opts)
```

Table 1: BLAC expression (1) as input to LGen.

Spiral-like attempt at linear algebra, LL is not point-free, which has the benefit of yielding a more natural representation.

3. SCALAR CODE GENERATION

We now describe step by step the program generation process (Fig. 1) for scalar (non-vectorized) C code.

3.1 Input in LL

The input is a BLAC expressed in LL. Since the syntax is straightforward we only show (1) as an example in Table 1. The parameter `opts` (last line) specifies if scalar or vectorized code is desired, the precision (float or double), and the search strategy. For the following explanations, we use a simpler BLAC as running example, namely matrix-vector multiplication of the form

$$y = Ax + y. \tag{2}$$

We will consider different sizes depending on the details to be illustrated. The input is parsed into an expression graph. We implemented a type system that determines the matrix associated with every node of the graph and makes sure that the expression is well-formed.

3.2 Step 1: Tiling in LL

Tiling is a crucial locality optimization in linear algebra [5, 34] for all levels of the memory hierarchy. Thus, the first step in LGen (Fig. 1) is to fix a tiling strategy (the search will then be able to explore different ones), which has to be done in a way that works for all BLACs.

For a given BLAC, such as (2), tiling is defined as an annotation in LL with two fixed tile size parameters r and c. For example, tiling (2) with $r = 2$, $c = 1$ is expressed as

$$\text{Tile}_{2,1}(y = Ax + y) \equiv [y = Ax + y]_{2,1}.$$

We use the square brackets for brevity.

In the next step, the top tiling decision is propagated in the expression graph to derive the associated tiling decision for the in- and output matrices. This is done using rewriting with the rules shown in Table 2. Note that matrix multiplication introduces a degree of freedom (k) that will be included in the search space. For a given BLAC, any tile size r, c is allowed. This is important, as even expressions with poor divisibility can then be tiled; accordingly, LGen has to handle the left-over code efficiently, a challenge that is most interesting for vectorized code (see Section 4).

As a tiling example we consider (2) where A is 4×4, tiled with $r = 2, c = 1$:

$$\begin{aligned}[y = Ax + y]_{2,1} &\rightarrow [y]_{2,1} = [Ax + y]_{2,1} \\ &\rightarrow [y]_{2,1} = [Ax]_{2,1} + [y]_{2,1} \\ &\rightarrow [y]_{2,1} = [A]_{2,k}[x]_{k,1} + [y]_{2,1},\end{aligned} \tag{3}$$

$[\langle e_L\rangle = \langle e_R\rangle]_{r,c}$	$\rightarrow$	$[\langle e_L\rangle]_{r,c} = [\langle e_R\rangle]_{r,c}$
$[\langle e_L\rangle + \langle e_R\rangle]_{r,c}$	$\rightarrow$	$[\langle e_L\rangle]_{r,c} + [\langle e_R\rangle]_{r,c}$
$[\langle \text{scalar}\rangle \cdot \langle e\rangle]_{r,c}$	$\rightarrow$	$[\langle \text{scalar}\rangle]_{1,1} \cdot [\langle e\rangle]_{r,c}$
$[\langle e_L\rangle \cdot \langle e_R\rangle]_{r,c}$	$\rightarrow$	$[\langle e_L\rangle]_{r,k} \cdot [\langle e_R\rangle]_{k,c}\,,\ 1 \le k \le \#\text{cols}(\langle e_L\rangle)$
$\left[\langle e\rangle^T\right]_{r,c}$	$\rightarrow$	$[\langle e\rangle]^T_{c,r}$

Table 2: Rewrite rules to propagate tiling decisions; $\langle e_L\rangle, \langle e_R\rangle$, and $\langle e\rangle$ are BLAC expressions.

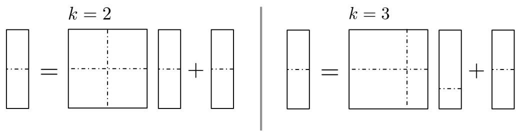


Figure 2: Visualization of (3) for $k = 2$ (homogeneous tiling) and $k = 3$ (heterogeneous tiling).

with $1 \le k \le 4$. The initial decision produces four tiling variants for the inputs A and x, all of which are considered in the search. Since we consider small problem sizes, we tile scalar code for locality in the registers (and for vectorization as explained later). In this case, following ideas from [35], we require $rc \le N_R$ (the number of logical registers) and bound k depending on the size of the L1 I-cache.

If we apply an $r \times c$ tiling to an $m \times n$ matrix A, then $[A]_{r,c}$ has dimensions $\lceil \frac{m}{r} \rceil \times \lceil \frac{n}{c} \rceil$. The resulting matrix is homogeneous if $r|m$ and $c|n$ or heterogeneous if tiles at the borders have a different size. For example, Fig. 2 depicts the structures resulting from two possible choices of k in (3). The tiled matrix $[A]_{2,k}$ is 2×2 in both cases, but is homogeneous for $k = 2$ and heterogeneous for $k = 3$.

Multilevel tiling is done by applying the tiling rules in Table 2 to previously tiled equations. The only constraint is that the created new tiles are not composed of subtiles of different sizes. For example, further tiling of $[A]_{2,3}$ in Fig. 2 with $(r, c) \in \{(1, 2), (2, 2)\}$ would not be allowed. Multilevel tiling is used for vectorization in Section 4.

3.3 Step 2: Loop optimizations in Σ-LL

After the tiling has been fixed we translate the resulting LL expression into a language called Σ-LL, which makes access patterns and loops explicit as matrices and matrix sums, respectively. Σ-LL is still purely mathematical and hence optimizations like loop merging and loop exchange can be done without analysis. A more puristic motivation for Σ-LL is that it is a natural intermediate step when translating a BLAC into loop code. Σ-LL extends Σ-SPL that is heavily used in Spiral [28, 32].

Σ-LL. Σ-LL includes *gather* and *scatter* matrices. They are used to extract or insert submatrices from or to large matrices. To explain the basic idea we will use Matlab-like notation: $A(k : \ell, m : n)$ is the submatrix of A obtained from rows k to ℓ and columns m to n; $A(k : s : \ell, m : t : n)$ extracts rows and columns at strides s and t, respectively. Assuming A is 3×3, then the the top left 2×2 submatrix can be extracted using gather matrices as

$$A(0:1, 0:1) = G_L A G_R, \quad G_L = \left[\begin{smallmatrix}1&0&0\\0&1&0\end{smallmatrix}\right], G_R = \begin{bmatrix}1&0\\0&1\\0&0\end{bmatrix}.$$

Similarly, defining the scatter matrices $S_L = G_R$ and $S_R =$

Matlab-like	Σ-LL
$B = A(b:s:e, b':s':e')$	$B = G_L(h_{b,s}^{d\to r}) A G_R(h_{b',s'}^{d'\to r'})$
$\mathrm{A} = \mathrm{zeros}(r, r')$ $A(b:s:e, b':s':e') = B$	$A = S_L(h_{b,s}^{d\to r}) B S_R(h_{b',s'}^{d'\to r'})$

Table 3: Gathers and scatters associated with extracting and inserting matrices. A is $r \times r'$ and B is $d \times d'$. In the first column $e = b + sd$ and $e' = b' + s'd'$.

G_L, we can insert a 2×2 matrix B into a 3×3 matrix A as

$$A = S_L B S_R.$$

In general, we parametrize gathers and scatters with a symbolic index function h. Specifically, for $r \geq d$,

$$h_{b,s}^{d\to r}: \mathbb{R}^d \to \mathbb{R}^r,\ i \mapsto b + is.$$

The associated gathers are now defined such that the second row in Table 3 holds. The scatters are analogously defined as $S_L(h) = G_R(h), S_R(h) = G_L(h)$, such that the third row in Table 3 holds.

To simplify notation in the following, we will omit L and R, and write for example $G_L(h_{b,s}^{d\to r}) = G_{b,s}^{d,r}$ or simply $G_{b,s}$ if the dimensions are evident or simply G_b if $s = 1$.

Using gathers and scatters, tiled computations can be expressed as summations. Consider, for example, the tiled matrix-vector multiplication $[y]_{2,1} = [A]_{2,2} \cdot [x]_{2,1}$ with $A \in \mathbb{R}^{4\times 4}$ and $x \in \mathbb{R}^4$. The computation on the tiles is visualized in Fig. 3. In Σ-LL, this is expressed as

$$\begin{aligned} y &= S_0^{2,4}\left(G_0^{2,4} A G_0^{2,4}\right) S_0^{2,4} S_0^{2,4}\left(G_0^{2,4} x\right) \\ &\quad + S_0^{2,4}\left(G_0^{2,4} A G_2^{2,4}\right) S_2^{2,4} S_2^{2,4}\left(G_2^{2,4} x\right) \\ &\quad + S_2^{2,4}\left(G_2^{2,4} A G_0^{2,4}\right) S_0^{2,4} S_0^{2,4}\left(G_0^{2,4} x\right) \\ &\quad + S_2^{2,4}\left(G_2^{2,4} A G_2^{2,4}\right) S_2^{2,4} S_2^{2,4}\left(G_2^{2,4} x\right) \\ &= S_0\left(G_0 A G_0\right)\left(G_0 x\right) + \cdots + S_2\left(G_2 A G_2\right)\left(G_2 x\right) \\ &= \sum_{i=0,2}^{3} \left[S_i\left(G_i A G_0\right)\left(G_0 x\right) + \cdots + S_i\left(G_i A G_2\right)\left(G_2 x\right)\right] \\ &= \sum_{i=0,2}^{3} \sum_{j=0,2}^{3} S_i\left(G_i A G_j\right)\left(G_j x\right). \end{aligned}$$

We use $\sum_{i=0,k}$ for summations with increment $k > 1$. In the equations we applied simplification properties shown in Table 4. If we want to express the computation down to the scalar level, we need to apply the same reasoning to the

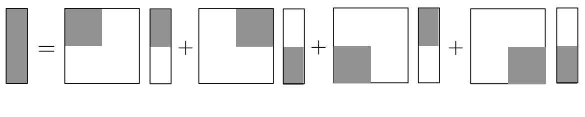


Figure 3: Tiled matrix-vector multiplication $[y]_{2,1} = [A]_{2,2}\,[x]_{2,1}$ with $A \in \mathbb{R}^{4\times 4}$ and $x \in \mathbb{R}^4$. White regions, created using scatters, contain zeros.

$$h_{b,s}^{n',N} \circ h_{b',s'}^{n,n'} = h_{b+sb',ss'}^{n,N} \quad (4)$$
$$G_L(h) = S_L(h)^T \quad (5)$$
$$G_{0,1}^{n,n} = S_{0,1}^{n,n} = I_n \quad (6)$$
$$S_L(h) \cdot S_L(h') = S_L(h \circ h') \quad (7)$$
$$G_L(h) \cdot G_L(h') = G_L(h' \circ h) \quad (8)$$
$$G_L(h_{i,1}^{s\to d}) \sum_{i'=b,s}^{d-1} S_L(h_{i',1}^{s\to d}) = G_i^{s,d} S_i^{s,d} = I_s \quad (9)$$

Table 4: Simplification properties for gathers and scatters; $\circ$ is function composition and I_n is an $n \times n$ identity matrix. (9) **holds if $i \in (b:s:d)$.**

$$[\langle e_L\rangle]_{r,c} + [\langle e_R\rangle]_{r,c} \to \sum_{i,j} S_i\left(G_i\langle e_L\rangle G_j + G_i\langle e_R\rangle G_j\right) S_j \quad (10)$$
$$[\langle e_L\rangle]_{r,k} \cdot [\langle e_R\rangle]_{k,c} \to \sum_{i,j,k} S_i\left(G_i\langle e_L\rangle G_k \cdot G_k\langle e_L\rangle G_j\right) S_j \quad (11)$$
$$[\langle \mathrm{scalar}\rangle]_{1,1} \cdot [\langle e\rangle]_{r,c} \to \sum_{i,j} S_i\left(\langle \mathrm{scalar}\rangle \cdot G_i\langle e\rangle G_j\right) S_j \quad (12)$$
$$[\langle e\rangle]_{r,c}^T \to \sum_{i,j} S_j\left(G_i\langle e\rangle G_j\right)^T S_i \quad (13)$$

Table 5: Rules to recursively translate LL into Σ-LL.

computation between submatrices:

$$\begin{aligned} y &= \sum_{i=0,2}^{3} \sum_{j=0,2}^{3} S_i \\ &\quad \times \sum_{i'=0}^{1} \sum_{j'=0}^{1} S_{i'} \Bigg(\underbrace{G_{i'} \overbrace{G_i A G_j}^{2\times 2} G_{j'}}_{\text{scalar}}\Bigg) \Bigg(\underbrace{G_{j'} \overbrace{G_j x}^{2\times 1}}_{\text{scalar}}\Bigg) \\ &= \sum_{i,j,i',j'} S_i S_{i'}\left(G_{i'} G_i A G_j G_{j'}\right)\left(G_{j'} G_j x\right) \\ &= \sum_{i,j,i',j'} S_{i+i'}\left(G_{i+i'} A G_{j+j'}\right)\left(G_{j+j'} x\right) \end{aligned}$$

The latter corresponds to a tiled implementation with four loops.

In summary, Σ-LL makes the index functions explicit as symbolic objects that can be manipulated through rewriting. Possible loops are made explicit as mathematical summations. Since the language is mathematical, rewrite rules are simply mathematical identities.

LL to Σ-LL. The translation from LL to Σ-LL is done by rewriting the expression graph with tiling information. The rewrite rules are shown in Table 5. As before $\langle e_L\rangle$ and $\langle e_R\rangle$ refers to the left- and right-hand side of an expression.

Simplifications and loop fusion. The Σ-LL expression obtained from a straightforward translation will not represent an efficient program since every read and write yields a gather or scatter object. Thus we simplify by rewriting with a set of mathematical identities (see Table 4) that reduce the number of these objects. In particular, gathers and scatters can cancel each other if subsequent reads and writes are not to overlapping locations or they can be fused by composing the index functions (a strided access of a strided access yields a strided access).

Loop exchange. The order of loops in the generated loop nests is in principle a further degree of freedom of the generation process. Instead of enlarging the search space we determine nest-wise local orderings, using what we call a priority matrix (Π). Every time a summation is created using rules from Table 5, its indices are associated with new rows of Π. Columns of Π are related to factors that can influence performance. In our study we consider three factors: instruction-level parallelism (*ilp*), temporal locality (*tl*), and spatial locality (*sl*).

Every entry $\Pi(i, f)$ estimates the (positive) impact on f of increasing index i before other indices of the summation. For example, $\Pi(i, tl) > \Pi(j, tl)$ means that increasing index i before j has better temporal locality. In general, this value can depend on the operation and on the tiling level. For example, following the discussion in [35] about loop ordering for matrix multiplication we use the following criteria for rule (11) in Table 5 assuming we are at an outer level of tiling:

- *Temporal locality:* varying k has larger impact as we keep operating on same output elements, $\Pi(k, tl) = 1$, $\Pi(p, tl) = 0$, $p \in i, j$.
- *Spatial locality:* Row-major indices have larger impact, $\Pi(i, sl) = 0$, $\Pi(k, sl) = 1$, and $\Pi(j, sl) = 2$.
- *Ilp:* moving along the dimensions of the output matrix has larger impact, $\Pi(p, ilp) = 1$, $p \in i, j$, $\Pi(k, ilp) = 0$.

Using the above criteria we obtain the following matrix Π:

Π	*tl*	*sl*	*ilp*
i	0	0	1
j	0	2	1
k	1	1	0

We determine the order of the indices in two steps: (a) we sort the columns by priority given to the performance factors; (b) we sort the rows of Π in ascending lexicographical order. In our example, assuming priority (*tl*, *sl*, *ilp*), with *tl* being highest, we would determine the order (i, j, k); giving higher priority to spatial locality, i.e., assuming priority (*sl*, *tl*, *ilp*), yields (i, k, j).

3.4 Step 3: C-IR optimizations

Σ-LL expressions are converted into a C-intermediate representation (C-IR) that we use for applying a set of code-level optimizations, such as loop unrolling of the lowest levels of tiling, scalar replacement, and conversion into SSA form. Translation into C-IR first requires binding internal matrices of a Σ-LL expression graph to arrays in memory. Input and output matrices are bound to input and output arrays, internal operators to temporary arrays, and gathers and scatters to arrays associated with the expressions they multiply.

After matrix binding, memory references are used in combination with code templates associated with the Σ-LL operators to produce C-IR code. The access patterns are deduced from the index mapping functions of the gathers and scatters and incorporated in the reference objects. For example, the code template for scalar addition can be described through the pseudocode in Table 6. The codelet makes the following assumptions: (a) `left` and `right` are scalar expressions; (b) the reference objects (e.g., `inL`) contain all the information necessary to locate the position of the scalars within eventual larger matrices.

```
genAdd(B, expr, left, right):
  // code for  expr = left + right
  inL = getReference(left)
  inR = getReference(right)
  out = getReference(expr)

  B <- Mov (Add inL[0,0], inR[0,0]), out[0,0]
```

Table 6: Code generation template for scalar addition; the object `B` refers to a basic block of the code.

Finally, the C-IR code is unparsed into C.

3.5 Step 4: Performance evaluation and search

After the C function is generated, it is executed and its performance is measured and used for autotuning. The number of functions that can be generated depends on the degrees of freedom introduced by tiling in Step 1. If more than one function can be generated, LGen explores them (at present) either by exhaustive or by random search.

4. VECTOR CODE GENERATION

In the previous section we described the generation of scalar code for fixed-size BLACs using LGen. However, to obtain high performance, vectorization for SIMD ISAs is crucial. In this section, we explain how LGen generates C code including intrinsics to explicitly use vector instructions. The vector length (e.g., 4 for SSE float) is denoted with ν.

An important feature of our approach is extensibility. This means that porting to a new vector architecture is a straightforward, non-creative effort. Our solution does this conceptually as it was done in Spiral [10]. Specifically, we identify a few basic vectorized building blocks, called ν-BLACs, that need to be available to our system: porting to a new instruction set simply requires their implementation.

The generation process extends the one for scalar code introduced in Section 3 in the following way: (a) LGen receives the input BLAC together with the vector length ν of the ISA; (b) before tiling for registers, we apply a first level of tiling (ν-tiling) to match to the ν-BLACs; (c) ν-BLACs are associated with a set of pre-implemented codelets that are generated at C-IR level; (d) left-over code (for parts smaller than ν) is also vectorized by embedding into ν-BLACs using a *pack-compute-unpack* approach.

We now describe the ν-BLACs, tiling, code generation, and the handling of left-over code.

ν-BLACs. A ν-BLAC is a BLAC with the following characteristics: (a) only one operator is used; (b) it can be efficiently implemented on a vector ISA: for this we require that every matrix involved has size $1 \times \nu$, $\nu \times 1$, or $\nu \times \nu$.

The four operators in LL (multiplication, addition, scalar multiplication, transposition) yield the 18 ν-BLACs in Table 7. Porting LGen to a new vector ISA requires only the implementation of these (and associated packing routines explained later) using intrinsics. An example ν-BLAC is shown in Table 8. Note that we use unaligned instructions. Codelets always assume that vectors and matrix rows are contiguous in memory. This means that a single codelet can be associated with more than one ν-BLAC (e.g., in the vector addition $x + y$ in Table 8, x and y can be both row or column vectors).

Tiling for ν-BLACs. Once ν is provided to LGen, it

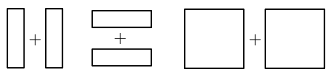
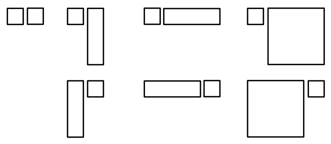
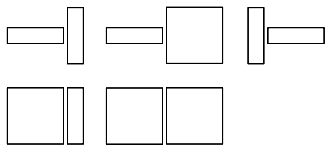
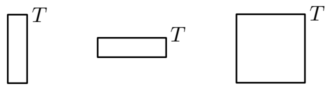
Operator	Required ν-BLACs
Addition (3 ν-BLACs)	
Scalar Multiplication (7 ν-BLACs)	
Matrix Multiplication (5 ν-BLACs)	
Transposition (3 ν-BLACs)	

Table 7: 18 required ν-BLACs to vectorize LL.

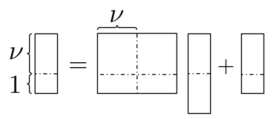
```
blac_nu2_xpy(B, refx, refy, out):
  B <- Mov (mmLoaduPd refx[0,0]), vx
  B <- Mov (mmLoaduPd refy[0,0]), vy
  B <- mmStoreuPd (mmAddPd vx, vy), out[0,0]
```

Table 8: ν-BLAC codelet for $x + y$ and $\nu = 2$; x and y are either $\nu \times 1$ or $1 \times \nu$.

performs a first level of tiling with $(r, c) \in \{(1, \nu), (\nu, 1), (\nu, \nu)\}$. For example, consider (2) where A is 3×4, $\nu = 2$, and we tile with $(r, c) = (\nu, \nu)$ to get

$$[y]_{\nu,1} = [A]_{\nu,\nu}[x]_{\nu,1} + [y]_{\nu,1},$$

or visually



For simplicity, we separately consider the Σ-LL expressions $[z]_{\nu,1} = [A]_{\nu,\nu}[x]_{\nu,1}$ and $[y]_{\nu,1} = [z]_{\nu,1} + [y]_{\nu,1}$. Following the procedure described in Section 3, we obtain

$$z = \sum_{j=0,\nu} S_0^{\nu,3}\left[\left(G_0^{\nu,3} A G_j^{\nu,4}\right)\left(G_j^{\nu,4} x\right)\right] \tag{14}$$

$$+ \sum_{k=0,\nu} S_2^{1,3}\left[\left(G_2^{1,3} A G_k^{\nu,4}\right)\left(G_k^{\nu,4} x\right)\right] \tag{15}$$

$$\text{and} \quad y = S_0^{\nu,3}\left(G_0^{\nu,3} z + G_0^{\nu,3} y\right) \tag{16}$$

$$+ S_2^{1,3}\left(G_2^{1,3} z + G_2^{1,3} y\right) \tag{17}$$

Eqs. (14)–(15) describe the same computation but performed using tiles of different size. The same holds for (16)–(17). In particular, (14)–(16) map directly to ν-BLAC codelets, while (17) needs additional work to be mapped. We describe both situations next.

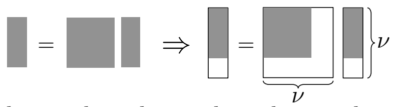
ν-BLAC code generation. The codelets for the ν-BLACs are pre-implemented and are retrieved using the parameter ν and the required precision (to date we support

```
genAdd(B, expr, left, right, opts):
  // code for  expr = left + right
  inL = getReference(left)
  inR = getReference(right)
  out = getReference(expr)

  nu, prec = opts[nu], opts[precision]
  vecSize = sizeof(left)
  nublac = getNuBLAC(Add, vecSize, nu, prec)

  nublac(B, inL, inR, out)
```

Table 9: Code generation template for addition; the object B refers to a basic block of the code.

SSE). Hence the code generation template for scalar code in Table 6 is extended as shown in Table 9.

Left-over code handling. Expressions such as $G_2^{1,3}z + G_0^{1,3}y$ (adding two vectors of length one) in (17) do not conform to any ν-BLAC with $\nu = 2$. The basic idea is to embed these BLACs into a (larger) ν-BLAC of appropriate type. For example a small matrix-vector multiplication is embedded as shown here:



In LGen, this is done by pack and unpack routines that perform the embedding of the operands. These are also pre-implemented using intrinsics and selected upon code generation. Since the BLAC code is unrolled, the compiler will be able to perform some dead code elimination (e.g., right above the last row of the matrix and associated operations can be removed).

5. EXPERIMENTS

In this section we show performance benchmarks of BLAC code generated by LGen.

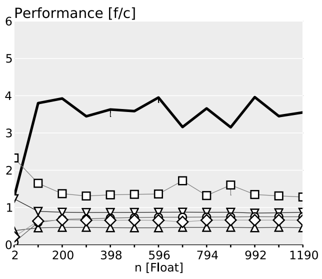
Experimental setup. We divide our experiments into four categories depending on the type of functionality generated:

- *Simple BLACs:* $y = Ax$ (smv) and $C = AB$ (smm).
- *BLACs that closely match BLAS:* $y = \alpha x + y$ (saxpy), $y = \alpha Ax + \beta y$ (sgemv), and $C = \alpha AB + \beta C$ (sgemm).
- *BLACs that need more than one BLAS call:* $y = \alpha Ax + \beta Bx$ (sgesummv), $\alpha = x^T Ay$ (sblinf), and $C = \alpha(A_0 + A_1)^T B + \beta C$ (sgemam).
- *Micro BLACs:* three BLACs from previous cases (smv, smm, and sblinf) using very small matrices and vectors.

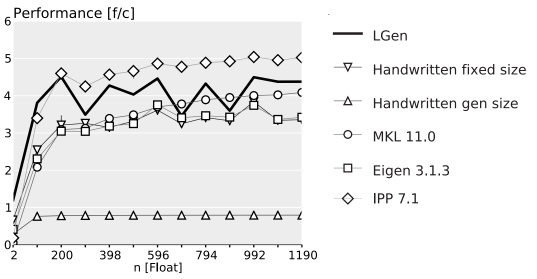
In the first three cases we use matrices with narrow rectangular shapes (panels) or small squares (blocks). This choice is due to their importance [15, 31]. The sizes are either $n \times 4$ or $4 \times n$, chosen to fit into L1 D-cache, or 4×4. For Micro BLACs, the matrices are $n \times n$ with $2 \leq n \leq 10$.

We run our tests on an Intel Xeon X5680 (Westmere EP microarchitecture), 3.3 GHz, SSE 4.2, 32 kB L1 D-cache, under RHEL Server 6 with kernel v.2.6.32. Intel's SpeedStep and Turbo Boost technologies were disabled during the tests.

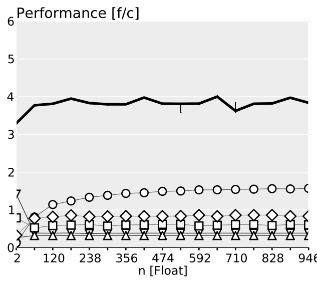
We considered only single precision code and compared against (a) Intel MKL v.11, (b) Intel IPP v.7.1, (c) Eigen v.3.1.3, (d) BTO v1.3, and (e) handwritten code. The latter is scalar, non-unrolled code and comes in two versions:



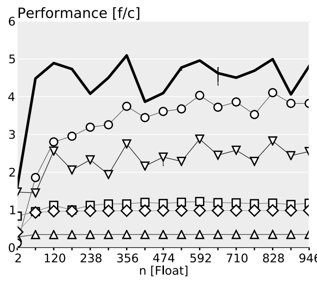
(a) A is $n \times 4$.



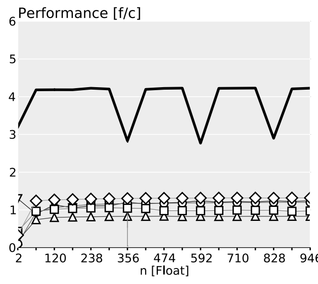
(b) A is $4 \times n$.



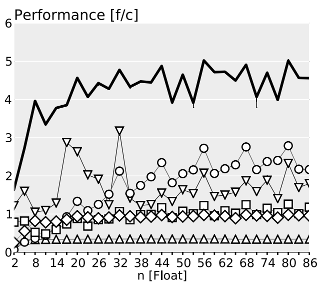
(c) A is $n \times 4$, B is 4×4.



(d) A is 4×4, B is $4 \times n$.



(e) A is $4 \times n$, B is $n \times 4$.



(f) A is $n \times 4$, B is $4 \times n$.

Figure 4: Simple BLACs. (a)–(b): $y = Ax$**; (c)–(f):** $C = AB$**.**

with hardcoded problem sizes (fixed size) and with problem sizes passed as parameters (general size). MKL and IPP are provided as binary code. Code obtained from LGen, Eigen, BTO, as well as all the handwritten kernels were compiled using icc v.13.1 with flags -O3 -xHost -fargument-noalias -fno-alias -ip -ipo.

LGen uses a random search with a sample size of 10. BTO's kernels were generated disabling multithreading, and enabling loop tiling. In Eigen we used fixed-size Map interfaces to existing arrays, no-alias assignments, and enabled SSE code generation. In MKL, we implemented sgesummv with two calls to `cblas_sgemv`, sblinf as a combination of `cblas_sgemv` and `cblas_sdot`, and sgemam as a call to `MKL_Somatadd`[1] followed by `cblas_sgemm`.

All plots show performance in flops per cycle (f/c) on the y-axis, and on the x-axis the size of the largest dimension of the matrices involved. The theoretical peak performance of the platform is 8 f/c in all cases (assuming balanced adds and mults). However, our plots are scaled to 6 f/c for better readability.

Time is measured under warm-cache conditions. Because of the short execution times of the kernels, we adopt a two-loops measuring strategy. The outer loop repeats the measurement to return median and quartile information. The inner loop reduces the error by executing the target code for at least 10^8 cycles. The data points reported in the plots are always medians of 20 repetitions, and for each point, we also report using whiskers the most extreme data points that fall in the range $[1.5q_1, 1.5q_3]$ following [12] (q_1 and q_3 are respectively the lower and upper quartiles). In most cases the variation is negligible.

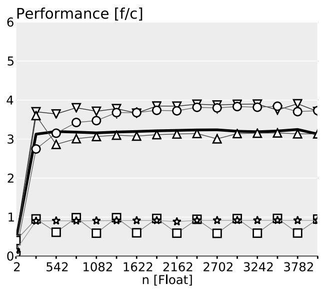
Case 1: Simple BLACs. Fig. 4 shows performance results for the BLACs smv and smm. For smv with vertical A (Fig. 4(a)), LGen performs between 1.8× and 3× better than Eigen. With horizontal A (Fig. 4(b)) and for larger n LGen performs within 10% of IPP and Eigen. For smm we consider four scenarios. In the panel-block case (Fig. 4(c)) LGen performs about 2.5× faster than MKL. In the block-panel computation (Fig. 4(d)) the improvements reduce to 10% for larger sizes. For the panel-panel products the speed-up is again a factor of about 3× over the competition in Fig. 4(e)) and about 2× for the rank-4 update in Fig. 4(f). Unfortunately, we could not compare against BTO due to exceptions raised by the generated code. Downwards spikes, such as in Fig. 4(e), are related to suboptimal tiling decisions resulting either from the random selection during search or from current multilevel tiling limitations (see Section 3.2).

Case 2: BLACs that closely match BLAS. The results are shown in Fig. 5. For saxpy (Fig. 5(a)) MKL and the icc-compiled fixed size code attain the same performance which is about 15% higher than LGen. sgemv (Figs 5(b)–(c)) and sgemm (Figs 5(d)–(g)) have a performance behavior very close to the one previously observed for smv and smm.

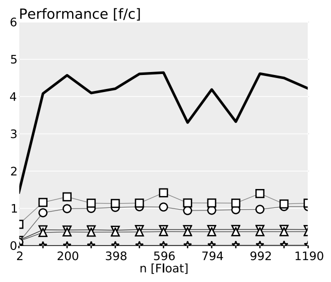
Case 3: BLACs that need more than one BLAS call. The results are shown in Fig. 6. Eigen's ability to generate fused loops results in comparable performance between BLACs in Case 2 and 3 (e.g., smv-based expressions in Figs. 6(d), 6(b), and 5(c)). On the other hand, we notice that slight changes in computational patterns (e.g., from smv in Fig. 4(b) to sgesummv in Fig. 6(b)) can diminish the capability of icc to apply loop-level optimizations. The combination of BTO's autotuning capabilities and icc's autovectorization achieves similar performance to LGen (about 4 f/c) for the case of sgesummv with horizontal panels (Fig. 6(b)). For sgemam (Figs 6(e)–(h)) all competing curves except MKL perform below 1 f/c.

Case 4: Micro BLACs. Finally, we report on small size code in Fig. 7. In this case LGen produces fully unrolled code with vectorized left-over computations. In case of smv

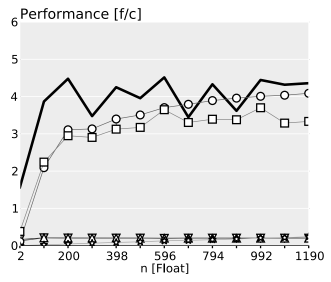
[1]`MKL_Somatadd` is a non-BLAS function provided by Intel MKL.



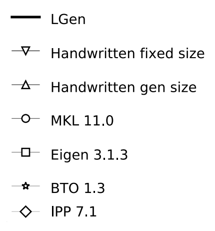
(a) x of length n.

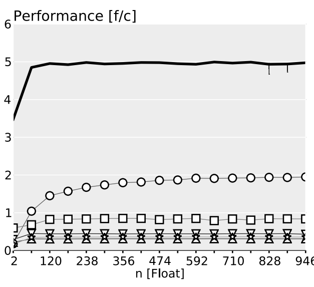


(b) A is $n \times 4$.

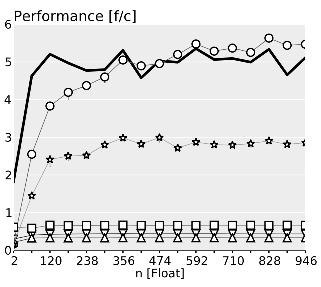


(c) A is $4 \times n$.

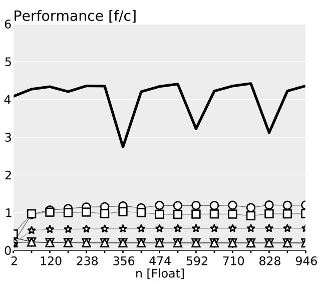




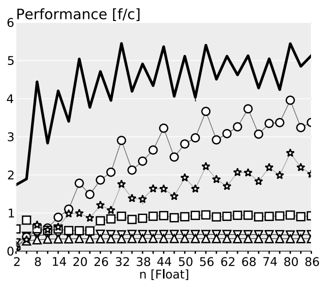
(d) A is $n \times 4$, B is 4×4.



(e) A is 4×4, B is $4 \times n$.

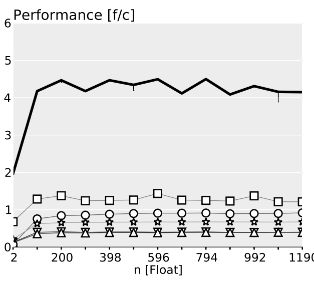


(f) A is $4 \times n$, B is $n \times 4$.

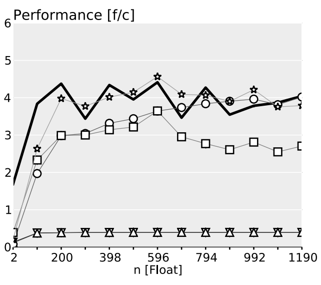


(g) A is $n \times 4$, B is $4 \times n$.

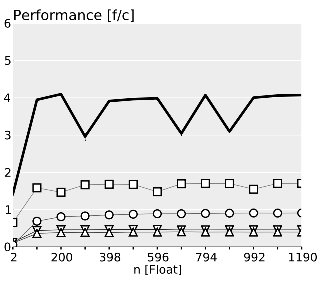
Figure 5: BLACs that closely match BLAS. (a): $y = \alpha x + y$; (b)–(c): $y = \alpha Ax + \beta y$; (d)–(g): $C = \alpha AB + \beta C$.



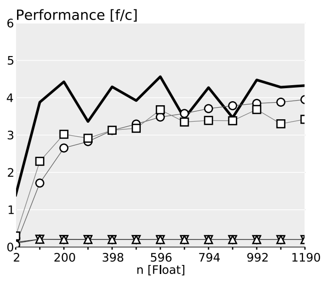
(a) A and B are $n \times 4$.



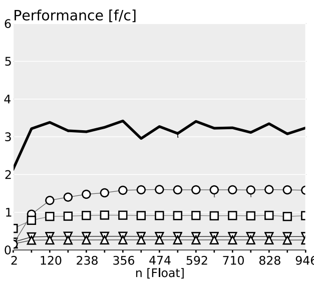
(b) A and B are $4 \times n$.



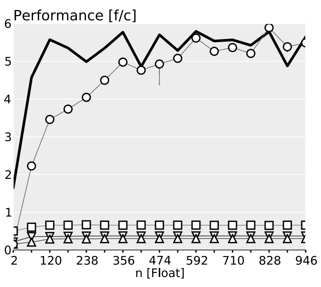
(c) A is $n \times 4$.



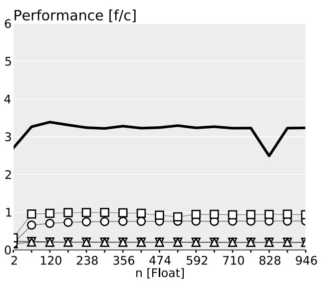
(d) A is $4 \times n$.



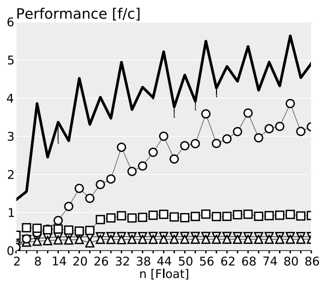
(e) A_0, A_1 are $4 \times n$, B is 4×4.



(f) A_0, A_1 are 4×4, B is $4 \times n$.



(g) A_0, A_1 are $n \times 4$, B is $n \times 4$.

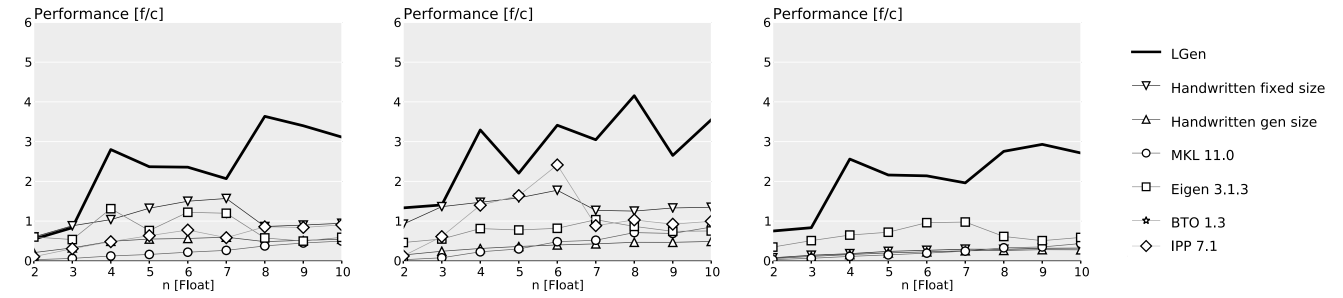


(h) A_0, A_1 are $4 \times n$, B is $4 \times n$.

Figure 6: BLACs that need more than one BLAS call. (a)–(b): $y = \alpha Ax + \beta Bx$; (c)–(d): $\alpha = x^T Ay$; (e)–(h): $C = \alpha(A_0 + A_1)^T B + \beta C$.

and smm, LGen exhibits improvements between 1.25× and 3.5× compared to icc fixed size, which is the best competing code. For sblinf we achieve a speedup as high as up to 6× with respect to Eigen.

Remarks. From the plots we can observe that certain shapes are favored by the existing libraries and generators we compared with. For instance, looking at Fig. 6 we can quickly identify that the most competitive plots (b), (d), (f), and (h) involve horizontal panels. In contrast, LGen produces across most functions and sizes a performance in the 3–6 f/c range. Also worth noting is that the compiler fails to optimize straightforward loop code, even when specialized to the problem size.



(a) $y = Ax$ (b) $C = AB$ (c) $\alpha = x^T Ay$

Figure 7: Micro BLACs. All matrices are squared of size $n \times n$.

6. LIMITATIONS AND FUTURE WORK

We discuss a list of current limitations of LGen and explain how they may be overcome in future work:

Fixed size code. LGen generates code that is specialized to the input sizes of the operands. It is important to stress that many relevant applications outside the scientific computing domain fulfill this constraint. However, for many other applications a library for general input sizes is still desirable. One possible solution to achieving this could be the recursion step closure technique developed in [32].

Data type. At present, LGen only supports real floating point data; extension to complex numbers will mainly impact the vectorization strategy.

Contiguous data. LGen only handles computations on contiguous data structures. Allowing for parametric gather and scatter operators in Σ-LL could resolve this issue.

Matrix structure. We currently assume general matrices, but special knowledge about the structure of matrices (e.g., symmetric), if it exists, can be used to define more efficient algorithms for certain operations. Structured matrices could be introduced by extending the type system of LGen and the backend.

Search methods. Our search strategies are very limited at the moment: exhaustive and random search. Thus better code may be within LGen's scope but cannot be found. We plan to extend LGen with well-known search algorithms used in various autotuning approaches.

Higher level algorithms. The current LGen only supports BLACs, i.e., no LAPACK functionality such as LU or Cholesky factorization. We believe this extension is possible using a FLAME-like approach that decomposes these functions into BLACs.

7. CONCLUSION

Currently, there is no consolidated solution that provides high performance code for arbitrary small linear algebra computations. With this paper we aimed to make a first step focusing on basic computations of fixed size. The other main goal was to do so with mathematical DSLs that enable optimizations at a high level of abstraction, in a way that closely resembles Spiral for linear transforms. In this paper, we already used the DSLs for vectorization and for simple loop optimizations. In future work we plan to extend the approach towards generalized interfaces and higher level linear algebra functions.

The performance of the code generated by LGen is competitive and mostly better than prior work on the small fixed sizes considered. However, we believe that more can be achieved with better vectorization, better search, and the possible use of models to determine optimization parameters.

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. *LAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to SIMD loop synthesis. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 123–134, 2013.

[3] G. Belter, E. R. Jessup, T. Nelson, B. Norris, and J. G. Siek. Reliable generation of high-performance matrix algebra. *Computing Research Repository (CoRR)*, abs/1205.1098, 2012.

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

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

[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] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):1–17, 1988.

[8] 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.

[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] F. Franchetti and M. Püschel. Generating SIMD vectorized permutations. In *International Conference on Compiler Construction (CC)*, volume 4959 of *Lecture Notes in Computer Science (LNCS)*, pages 116–131. Springer, 2008.

[11] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. In *Programming Language Design and Implementation (PLDI)*, pages 315–326, 2005.

[12] M. Frigge, D. C. Hoaglin, and B. Iglewicz. Some implementations of the boxplot. *The American Statistician*, 43(1):50–54, 1989.

[13] M. Frigo. A fast Fourier transform compiler. In *Programming Language Design and Implementation (PLDI)*, pages 169–180, 1999.

[14] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

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

[16] P. Gottschling and C. Steinhardt. Meta-tuning in MTL4. In *International Conference on Numerical Analysis and Applied Mathematics (ICNAAM)*, volume 1281, pages 778–782, 2010.

[17] P. Gottschling, D. S. Wise, and A. Joshi. Generic support of algorithmic and structural recursion for scientific computing. *International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)*, 24(6):479–503, 2009.

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

[19] 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.

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

[21] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *International Conference on Supercomputing (ICS)*, pages 147–157, 2009.

[22] Intel. Intel integrated performance primitives (IPP). `http://software.intel.com/en-us/intel-ipp`.

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

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

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

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

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

[28] 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.

[29] J. Shin, M. Hall, J. Chame, C. Chen, and P. Hovland. Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. In K. Naono, K. Teranishi, J. Cavazos, and R. Suda, editors, *Software Automatic Tuning*, pages 353–370. Springer New York, 2010.

[30] J. Siek, I. Karlin, and E. Jessup. Build to order linear algebra kernels. In *International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–8, 2008.

[31] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS)*. To appear.

[32] Y. Voronenko, F. de Mesmay, and M. Püschel. Computer generation of general size linear transform libraries. In *International Symposium on Code Generation and Optimization (CGO)*, pages 102–113, 2009.

[33] J. Walter, M. Koch, et al. uBLAS. `www.boost.org/libs/numeric`.

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

[35] 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.