A Basic Linear Algebra Compiler for Structured Matrices

Daniele G. Spampinato, Markus Püschel

Reported by Edward Hutter
Program generation for small-scale linear algebra has resulted in:

- LGen\(^1\)
- LGen for embedded processors\(^2\)
- LGen for structured matrices\(^3\)
- SLinGen\(^4\)

\(^1\) Spampinato, Püschel; 2014; "A Basic Linear Algebra Compiler"
\(^2\) Kyrtatas, Spampinato, Püschel; 2015; "A Basic Linear Algebra Compiler for Embedded Processors"
\(^3\) Spampinato, Püschel; 2016; "A Basic Linear Algebra Compiler for Structured Matrices"
\(^4\) Spampinato, Fabregat-Traver, Püschel; 2018; "Program Generation for Small-Scale Linear Algebra Applications"
Motivation

The problem:
- Performance-critical applications in media processing, control, graphics, etc. need efficient small-scale linear algebra computations
- Most libraries are tuned for large-scale problems
- Interfaces can be rigid (BLAS)

A solution:
- An infrastructure geared towards small-scale linear algebra computations
- Able to exploit fixed-sizes known at compile-time
- Able to understand non-standard computations (non-BLAS)
Competing strategies

In 2014, did such a solution exist?

- Highly optimized BLAS libraries (Intel MKL, ATLAS, GotoBLAS)
  - tuned for large problem sizes with BLAS interface
  - Intel IPP was developed to address this

- Generators
  - PHiPAC and ATLAS generator tune parameters at runtime, focus on large problems
  - BuildToOrder BLAS (BTO) - a DSC for matrix computations, not bound to BLAS interface, relies on compiler for vectorization
  - Eigen, MTL use C++ expression templates to optimize code at compile time, lack runtime feedback

- Optimizing compilers
  - polyhedral compilers can vectorize and optimize imperfectly nested loops
  - no such compiler dedicated to linear algebra exists
Consensus:

- Vectorization, loop merging, and tiling not novel in 2014, many strategies at compile/runtime
- Each existing approach exposed a flaw

Contributions of LGen

- novel approach to generating efficient code for basic linear algebra computations
- two levels of DSLs to perform loop optimizations and vectorization, respectively
- portable vectorization strategy, left-over code handled efficiently
- speedups over competitors mentioned above
**LGen methodology**

![Diagram of LGen methodology]

- Basic linear algebra computation (BLAC)
- Tiling decision
- Tiling propagation
- Loop-level optimizations (Σ-LL)
- Code-level optimizations (C-IR)
- Optimized C function

\[
y = (Ax)^T
\]

\[
Tile_{3,1} \left[ y = (Ax)^T \right]
\]

\[
y = \sum_{i_1, j_1, j_0, i_0} S_{j_1+j_0} (G_{i_1+i_0} \cdots)^T
\]

...\[
\text{Mov (mmMulPs refA[0,0], refx[0,0]), reft[0,0]}
\]

...\[
\text{for(int i = ... ) {}
\]

...\[
\text{t = _mm_mul_ps(a, x);}
\]

...\[
}
LGen overview

- BLAC is specified in Linear algebra Language (LL)
- Tiling decision is propagated through BLAC, still in LL
- LL converted to Σ-LL, where access patterns and loops are explicit
- C-IR generated from Σ-LL to perform loop unrolling, scalar replacement, static assignment
- C function is generated (optionally vectorized), feedback loop enabled
Scalar code generation: BLACs

What is a BLAC?

- a basic linear algebra computation of fixed size, written in Linear algebra Language (LL)
- consists of matrices, vectors, and scalars
- composed of matrix multiplication, matrix addition, transposition, and scalar multiplication
- example: $\beta = (A \times a + \alpha y) z + \beta$
- more information needed: data type, fixed sizes of individual objects, use vectorization?
- parsed into an expression graph
Scalar code generation: Tiling in LL

- Scalar code tiled for locality within registers
- Acts as an annotation to the BLAC in LL
- (Any) two parameters \( r, c \) are fixed, search can explore others
  - constraints: \( rc \leq N_r, k \) produces blocks that fit in L1 cache
  - homogenous vs. heterogenous, multilevel
- Tiling decision propagates down expression tree

Simple BLAC: \( y = Ax + y \)

\[
\text{Tile}_{2,1}(y = Ax + y) \equiv [y = Ax + y]_{2,1} \\
[y = Ax + y]_{2,1} \rightarrow [y]_{2,1} = [Ax + y]_{2,1} \\
\rightarrow [y]_{2,1} = [Ax + y]_{2,1} \\
\rightarrow [y]_{2,1} = [Ax]_{2,1} + [y]_{2,1} \\
\rightarrow [y]_{2,1} = [A]_{2,k} [x]_{k,1} + [y]_{2,1}, 1 \leq k \leq 4
\]
Scalar code generation: Loop optimizations in $\Sigma$-LL

Tiled LL translated into $\Sigma$-LL

$\Sigma$-LL
- mathematical DSL, allowing loop merging/exchange by manipulation
- access patterns $\rightarrow$ matrices, loops $\rightarrow$ matrix sums
- use gather and scatter matrices to extract and insert matrices

\[
\begin{align*}
[(e_L)]_{r,c} + [(e_R)]_{r,c} & \rightarrow \sum_{i,j} S_i (G_i (e_L) G_j + G_i (e_R) G_j) S_j \\
[(e_L)]_{r,k} \cdot [(e_R)]_{k,c} & \rightarrow \sum_{i,j,k} S_i (G_i (e_L) G_k \cdot G_k (e_L) G_j) S_j \\
[(\text{scalar})]_{1,1} \cdot [(c)]_{r,c} & \rightarrow \sum_{i,j} S_i (\langle \text{scalar} \rangle \cdot G_i (c) G_j) S_j \\
[(c)]_{r,c}^T & \rightarrow \sum_{i,j} S_j (G_i (c) G_j)^T S_i
\end{align*}
\]

Table 5: Rules to recursively translate LL into $\Sigma$-LL.
Scalar code generation: Gather/Scatter

Let's start with two examples:

1. Extracting \( A(0 : 1, 0 : 1) \) from \( A_{3 \times 3} \)

\[
\begin{bmatrix}
a_{1,1} & a_{1,2} \\
a_{2,1} & a_{2,2}
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
a_{1,1} & a_{1,2} & a_{1,3} \\
a_{2,1} & a_{2,2} & a_{2,3} \\
a_{3,1} & a_{3,2} & a_{3,3}
\end{bmatrix}
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
0 & 0
\end{bmatrix}
\]

\[ A(0 : 1, 0 : 1) = G_L A G_R \]

2. Injecting \( A(0 : 1, 0 : 1) \) into \( A_{3 \times 3} \)

\[
\begin{bmatrix}
a_{1,1} & a_{1,2} & 0 \\
a_{2,1} & a_{2,2} & 0 \\
0 & 0 & 0
\end{bmatrix} =
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
a_{1,1} & a_{1,2} \\
a_{2,1} & a_{2,2}
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\]

\[ A = S_L A S_R \]

\[ G_R = S_L = G_L^T, \quad G_L = S_R = G_R^T \]
Scalar code generation: Gather/Scatter

Key concept: tiled computations \(\equiv\) matrix summations

- Gathers/Scatters parameterized by symbolic index function \(h\)
  - \(h_{b,s}^{d\rightarrow r}: \mathbb{R}^d \rightarrow \mathbb{R}^r, i \rightarrow b + is, d \leq r\)
  - \(G(h_{b,s}^{d\rightarrow r}) = G_{b,s}^{d,r} = G_{b,s} = G_b\)

Example: \(y = Ax\)
Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

- Gathers/Scatters parameterized by symbolic index function $h$
  - $h_{b,s}^{d\rightarrow r} : \mathbb{R}^d \rightarrow \mathbb{R}^r$, $i \rightarrow b + is$, $d \leq r$
  - $G(h_{b,s}^{d\rightarrow r}) = G_{b,s}^{d,r} = G_{b,s} = G_b$

Example: $y = Ax$

\[ A \]

\[ y = A x \]
Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

- Gathers/Scatters parameterized by symbolic index function $h$
  - $h_{b,s}^{d \rightarrow r} : \mathbb{R}^d \rightarrow \mathbb{R}^r$, $i \rightarrow b + is$, $d \leq r$
  - $G(h_{b,s}^{d \rightarrow r}) = G_{b,s}^d = G_b = G_{b,s}^r$

Example: $y = Ax$

\[ AG_0^{2,4} \]
Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

- Gathers/Scatters parameterized by symbolic index function $h$
  - $h_{b,s}^{d\rightarrow r} : \mathbb{R}^d \rightarrow \mathbb{R}^r$, $i \rightarrow b + is$, $d \leq r$
  - $G(h_{b,s}^{d\rightarrow r}) = G_{b,s}^{d,r} = G_{b,s} = G_b$

Example: $y = Ax$

$$G_0^{2,4} A G_0^{2,4}$$
Scalar code generation: Gather/Scatter

Key concept: tiled computations \( \equiv \) matrix summations

- Gathers/Scatters parameterized by symbolic index function \( h \)
  - \( h_{b,s}^{d\rightarrow r} : \mathbb{R}^d \rightarrow \mathbb{R}^r, \) \( i \rightarrow b + is, \) \( d \leq r \)
  - \( G(h_{b,s}^{d\rightarrow r}) = G_{b,s}^{d,r} = G_{b,s} = G_b \)

Example: \( y = Ax \)

\[
(G_0^{2,4} A G_0^{2,4}) S_0^{2,4}
\]
Scalar code generation: Gather/Scatter

Key concept: tiled computations \(\equiv\) matrix summations

- Gathers/Scatters parameterized by symbolic index function \(h\)
  - \(h^d_{b,s} : \mathbb{R}^d \rightarrow \mathbb{R}^r, i \rightarrow b + is, d \leq r\)
  - \(G(h^d_{b,s}) = G^d_{b,s} = G_{b,s} = G_b\)

Example: \(y = Ax\)

\[
S_0^{2,4}(G_0^{2,4} AG_0^{2,4}) S_0^{2,4}
\]
Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

- Gathers/Scatters parameterized by symbolic index function $h$
  - $h_{b,s}^{d\rightarrow r} : \mathbb{R}^d \rightarrow \mathbb{R}^r$, $i \rightarrow b + is$, $d \leq r$
  - $G(h_{b,s}^{d\rightarrow r}) = G_{b,s}^{d,r} = G_{b,s} = G_b$

Example: $y = Ax$

\[ S_0^{2,4} (G_0^{2,4} A G_0^{2,4}) S_0^{2,4} S_0^{2,4} (G_0^{2,4} x) \]
Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

- Gathers/Scatters parameterized by symbolic index function $h$
  - $h_{b,s}^{d\rightarrow r} : \mathbb{R}^d \rightarrow \mathbb{R}^r$, $i \rightarrow b + is$, $d \leq r$
  - $G(h_{b,s}^{d\rightarrow r}) = G_{b,s}^{d,r} = G_{b,s} = G_b$

Example: $y = Ax$

$$y = S_0^{2,4}(G_0^{2,4} AG_0^{2,4})S_0^{2,4}S_0^{2,4}(G_0^{2,4} x) + S_0^{2,4}(G_0^{2,4} AG_2^{2,4})S_2^{2,4}S_2^{2,4}(G_2^{2,4} x) + S_2^{2,4}(G_2^{2,4} AG_0^{2,4})S_0^{2,4}S_0^{2,4}(G_0^{2,4} x) + S_2^{2,4}(G_2^{2,4} AG_2^{2,4})S_2^{2,4}S_2^{2,4}(G_2^{2,4} x)$$
Scalar code generation: Gather/Scatter

Key concept: tiled computations ≡ matrix summations

- Gathers/Scatters parameterized by symbolic index function $h$
  - $h_{b,s}^{d\rightarrow r} : \mathbb{R}^d \rightarrow \mathbb{R}^r$, $i \rightarrow b + is$, $d \leq r$
  - $G(h_{b,s}^{d\rightarrow r}) = G_{b,s}^{d,r} = G_{b,s} = G_b$

Example: $y = Ax$

$$y = S_0^{2,4}(G_0^{2,4} AG_0^{2,4})(G_0^{2,4}x) + S_0^{2,4}(G_0^{2,4} AG_2^{2,4})(G_2^{2,4}x) +$$
$$S_2^{2,4}(G_2^{2,4} AG_0^{2,4})(G_0^{2,4}x) + S_2^{2,4}(G_2^{2,4} AG_2^{2,4})(G_2^{2,4}x)$$
Scalar code generation: Gather/Scatter

Key concept: tiled computations \( \equiv \) matrix summations

- Gathers/Scatters parameterized by symbolic index function \( h \)
  - \( h^d\rightarrow r : \mathbb{R}^d \rightarrow \mathbb{R}^r, i \rightarrow b + is, d \leq r \)
  - \( G(h^d\rightarrow r) = G^{d,r}_{b,s} = G_{b,s} = G_{b} \)

Example: \( y = Ax \)

\[
y = \sum_{i=0,2}^{3} S_i^{2,4}(G_i^{2,4} AG_0^{2,4})(G_0^{2,4} x) + S_i^{2,4}(G_i^{2,4} AG_2^{2,4})(G_2^{2,4} x)
\]
Scalar code generation: Gather/Scatter

Key concept: tiled computations ≡ matrix summations

- Gathers/Scatters parameterized by symbolic index function $h$
  - $h_{b,s}^{d\rightarrow r}: \mathbb{R}^d \rightarrow \mathbb{R}^r, i \rightarrow b + is, d \leq r$
  - $G(h_{b,s}^{d\rightarrow r}) = G_{b,s}^{d,r} = G_{b,s} = G_b$

Example: $y = Ax$

$$y = \sum_{i=0,2}^{3} \sum_{j=0,2}^{3} S_{i}^{2,4}(G_{i}^{2,4} A G_{j}^{2,4})(G_{j}^{2,4} x)$$
Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

- Gathers/Scatters parameterized by symbolic index function $h$
  - $h^{d \rightarrow r}_{b,s} : \mathbb{R}^d \rightarrow \mathbb{R}^r$, $i \rightarrow b + is$, $d \leq r$
  - $G(h^{d \rightarrow r}_{b,s}) = G^d_{b,s} = G_{b,s} = G_b$

Example: $y = Ax$

$$y = \sum_{i=0}^{3} \sum_{j=0}^{3} S^{2,4}_{i} \cdot \sum_{i'=0}^{1} \sum_{j'=0}^{1} S^{2,4}_{i'} (G^{2,4}_{i'} G^{2,4}_{i} A G^{2,4}_{j'} G^{2,4}_{j'}) (G^{2,4}_{j'} G^{2,4}_{j} x)$$
Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

- Gathers/Scatters parameterized by symbolic index function $h$
  - $h_{b,s}^{d \rightarrow r} : \mathbb{R}^d \rightarrow \mathbb{R}^r, i \rightarrow b + is, d \leq r$
  - $G(h_{b,s}^{d \rightarrow r}) = G_{b,s}^{d,r} = G_{b,s} = G_b$

Example: $y = Ax$

\[
y = \sum_{i,j,i',j'} S_i^{2,4} S_{i'}^{2,4} (G_{i'}^{2,4} G_{i}^{2,4} A G_{j'}^{2,4} G_{j}^{2,4}) (G_{j'} G_{j}^{2,4} x)
\]
Scalar code generation: Gather/Scatter

Key concept: tiled computations \( \equiv \) matrix summations

- Gathers/Scatters parameterized by symbolic index function \( h \)
  \[ h_{b,s}^{d \rightarrow r} : \mathbb{R}^d \rightarrow \mathbb{R}^r, i \rightarrow b + is, d \leq r \]
  \[ G(h_{b,s}^{d \rightarrow r}) = G_{b,s}^{d,r} = G_{b,s} = G_b \]

Example: \( y = Ax \)

\[
y = \sum_{i,j,i',j'} S_{i+i'}^{2,4} (G_{i+i'}^{2,4} A G_{j+j'}^{2,4})(G_{j+j'} x)
\]
Scalar code generation: Gather/Scatter summary

Key concepts:

- tiled computations \( \equiv \) matrix summations
- access patterns symbolized as matrices
- loop nests encoded as matrix summations
- \( \Sigma \)-LL DSL uses mathematical identities to optimize code before instantiation

\[
\begin{align*}
[(e_L)_{r,c} + (e_R)_{r,c}] & \rightarrow \sum_{i,j} S_i \left( G_i(e_L)G_j + G_i(e_R)G_j \right) S_j \quad (10) \\
[(e_L)_{r,k} \cdot (e_R)_{k,c}] & \rightarrow \sum_{i,j,k} S_i \left( G_i(e_L)G_k \cdot G_k(e_L)G_j \right) S_j \quad (11) \\
[\text{(scalar)}]_{1,1} \cdot [(e)_{r,c}] & \rightarrow \sum_{i,j} S_i \left( \text{(scalar)} \cdot G_i(e)G_j \right) S_j \quad (12) \\
[(e)]^T_{r,c} & \rightarrow \sum_{i,j} S_j \left( G_i(e)G_j \right)^T S_i \quad (13)
\end{align*}
\]

Table 5: Rules to recursively translate LL into \( \Sigma \)-LL.

Matrix summations in what order?
Scalar code generation: Loop fusion

Blind application of recursive rules $\rightarrow$ inefficient generated code

Simplification properties help:

\[
\begin{align*}
h_{b,s}^{n',N} \circ h_{b',s'}^{n',n'} &= h_{b+sb',ss'}^{n,N} \\
G_L(h) &= S_L(h)^T \\
G_{0,1}^{n,n} &= S_{0,1}^{n,n} = I_n \\
S_L(h) \cdot S_L(h') &= S_L(h \circ h') \\
G_L(h) \cdot G_L(h') &= G_L(h' \circ h) \\
G_L(h_{i,1}^{s \rightarrow d}) \sum_{i'=b,s}^{d-1} S_L(h_{i',1}^{s \rightarrow d}) &= G_{i}^{s,d} S_{i}^{s,d} = I_s
\end{align*}
\]

- gathers/scatters cancel if nonoverlapping reads/writes
- gathers/scatters fuse if strided access of a strided access
Scalar code generation: Loop exchange

Priority matrix used instead of enlarged search space

Three metrics:
- instruction-level parallelism (ilp)
- temporal locality (tl)
- spatial locality (sl)

Priority table for matrix multiplication:

\[
\begin{array}{c|c|c|c}
\Pi & tl & sl & ilp \\
\hline
i & 0 & 0 & 1 \\
\hline
j & 0 & 2 & 1 \\
\hline
k & 1 & 1 & 0 \\
\end{array}
\] (1)

Fastest loop index \(\rightarrow\) largest value of most important metric
Scalar code generation: C-IR optimizations

Final optimizations only after fixed tiling and loop ordering
- loop unrolling of lowest tiling level
- scalar replacement
- conversion to static single assignment form

Translation from Σ-LL to C-IR
- bind internal matrices of Σ-LL expression graph into arrays in memory
- Σ-LL operators translated into code templates using memory references
- access patterns deduced from Σ-LL expression, influences reference objects

C-IR code is parsed into C

```
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]
```
A number of C functions can be generated at compile-time
  - decided by number of valid tiling strategies
  - influenced by extra degrees of freedom introduced by certain BLACs
  - Loop orderings are fixed
  - exhaustive search or random search

Generated functions are executed and autotuned at runtime
Vector code generation: Overview

LGen requires 18 $\nu$-BLACs to vectorize LL

Porting to new architectures with different vector lengths requires only implementing these 18 operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Required $\nu$-BLACs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition (3 $\nu$-BLACs)</td>
<td>![Addition Diagram]</td>
</tr>
<tr>
<td>Scalar Multiplication (7 $\nu$-BLACs)</td>
<td>![Scalar Multiplication Diagram]</td>
</tr>
<tr>
<td>Matrix Multiplication (5 $\nu$-BLACs)</td>
<td>![Matrix Multiplication Diagram]</td>
</tr>
<tr>
<td>Transposition (3 $\nu$-BLACs)</td>
<td>![Transposition Diagram]</td>
</tr>
</tbody>
</table>
Vector code generation: Extensions from scalar

Vectorized code generation requires extra steps:

- LGen must receive vector length $\nu$ with BLAC
- $\nu$-tiling occurs before register-tiling to match $\nu$-BLACs
  - 3 tile shapes: $(1, \nu), (\nu, 1), (\nu, \nu)$
- Each $\nu$-BLAC has dedicated codelet generated at C-IR level
- Left-over code vectorized by embedding into a larger $\nu$-BLACs
  - similar to scatter operation, packed and unpacked using intrinsics
  - compiler can remove dead code after unrolling

$$y = A_{3 \times 4} x + y \text{ tiled with } (r, c) = (\nu, \nu):$$

$$\nu \begin{bmatrix} 1 \\ & & \nu \end{bmatrix} = \begin{bmatrix} \nu \\ & & \nu \\ & & 1 \end{bmatrix} + \begin{bmatrix} \nu \\ & & \nu \end{bmatrix}$$
LGen performance analysis setup

- benchmarks performed on an Intel Xeon X5680, 3.3 GHz, SSE 4.2, 32 kB L1 D-cache
- single precision
- compared against:
  - Intel MKL v.11 and Intel IPP v.7.1 binaries
  - Eigen v.3.1.3, BTO v1.3, handwritten code compiled
    - compilation flags: -O3 -xHost -fargument-noalias -fno-alias -ip -ipo
- performance measured in flops/cycle (8 is peak)
- each data point is median of 20 iterations, more advanced statistics also used
LGen performance: Simple BLACs

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

Daniele G. Spampinato, Markus Püschel

Reported by Edward Hutter
LGen performance: BLAS-like BLACs

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

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

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

Daniele G. Spampinato, Markus Püschel

Reported by Edward Hutter
Figure 7: Micro BLACs. All matrices are squared of size $n \times n$. 
Concerns with performance evaluation

Performance evaluation was incomplete:

- performance analysis mostly just reading off plots
- what are we supposed to take away from plots alone?
- search space of possible tiling and loop optimization strategies too large for reader to guess
- what LGen strategies were difference makers above the rest?
- what were the exact reasons why LGen performed poorly on certain BLACs?
- "poor tiling strategy" is not sufficient
LGen limitations

- fixed size code only
- single and double precision floats only
- contiguous data only
- simple matrix structures cannot be exploited\(^5\)
- optimization search strategies: exhaustive and random search
- no support for BLACs within higher-level algorithms\(^6\)

\(^5\)limitation addressed in 2016 work
\(^6\)limitation addressed in 2018 work
LGen for structured matrices (sLGen)

Matrix structures captured by LGen and polyhedral compilation

Optimized code generated for structured matrices (sBLACs)

Three structures supported as of 2016:
  - upper/lower triangular
  - symmetric
sLGen: Matrix structures

Let's first modify the gather and scatter operators:

- **Gather**
  - \( g = [i, j]_{k,l}^{m,n} : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{k \times l} \)
  - \( A \rightarrow Ag = A[i : i + k - 1, j : j + l - 1] \)

- **Scatter**
  - \( s = k,l_{m,n}[i, j] : \mathbb{R}^{k \times l} \rightarrow \mathbb{R}^{m \times n} \)
  - \( A \rightarrow sA = A[i : i + k - 1, j : j + l - 1] \)

- **Fusion:**
  - \( A[i, j]_{k,l}^{m,n} [i', j']_{u,v}^{k',l'} \rightarrow A[i + i', j + j']_{u,v}^{m,n} \equiv (Ag)g' \rightarrow A(gg') \)

Four matrix structures are introduced: \( L, U, S, Z, G \)

One additional operator introduced: \( G \setminus \nu \)
sLGen: Matrix structures

Let's do an example: \( A = LU + S \) \( A, L, U, S \in \mathbb{R}^{4 \times 4} \)

\[
\begin{align*}
[A = LU + S]_{2,2} & \quad \text{apply fixed tiling} \\
[A]_{2,2} = [LU + S]_{2,2} & \quad \text{propagate tiling down expression tree} \\
[A]_{2,2} = [LU]_{2,2} + [S]_{2,2} & \\
[A]_{2,2} = [L]_{2,2}[U]_{2,2} + [S]_{2,2} & \text{degree of freedom block size } k = 2
\end{align*}
\]

Convert LL to \( \Sigma\)-LL

\[
A = \sum_{i,j=0,2}^{2,2} 4,4[i,j] \left( \sum_{l,r,k=0,2}^{2,2} 4,4[l,r] (L[l,k]_{2,2}^4,4 U[k,r]_{2,2}^4,4) + S \right) [i,j]_{2,2}^4,4
\]

\[
A = \sum_{i,j=0,2}^{2,2} 4,4[i,j] \left( \sum_{k=0,2}^{2,2} \left( L[i,k]_{2,2}^4,4 U[k,j]_{2,2}^4,4 \right) + S[i,j]_{2,2}^4,4 \right) \text{ fuse loops}
\]
Translation from Σ-LL to C-IR doesn’t change:

- matrix summations mapped to loops
- $\nu$-BLACs mapped to codelets
- gathers/scatters mapped to data accesses

Structure not exploited
Polyhedral sets and maps decompose a structured matrix in $\Sigma$-LL

Sets:
- $n$-tuples bounded by constraints
- represent regions in matrices, iteration spaces in computations
- Ex. $\sigma = \{(i, j) | 0 \leq i < 4 \land 0 \leq j < 4\}$

Maps:
- relations between polyhedral sets
- represent access patterns of matrices, reorder iteration spaces
- Ex. $\rho = \bigcup_i \{(t_0, t_1) \in \mathbb{Z}^{n_0} \times \mathbb{Z}^{n_1} | \exists c \in \mathbb{Z}^e : A_i t_0 + B_i t_1 + E_i c + z_i \geq 0\}$
sLGen: Internal representation of structures

Every matrix has pair of dictionaries: SInfo, AInfo

SInfo - polyhedral set, giving each structure an associated region
- \( M : \sigma \)

AInfo - polyhedral map, mapping each region to access pattern
- \( \sigma : (g : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{r \times c}, p : \mathbb{R}^{r \times c} \rightarrow \mathbb{R}^{r \times c}) \)

Example:
- \( L.\text{SInfo} - \)
  \[
  \begin{cases}
    G : \{(i, j)|0 \leq i < 4 \land 0 \leq j \leq i\} \\
    Z : \{(i, j)|0 \leq i < 4 \land i < j < 4\}
  \end{cases}
  \]
- \( L.\text{AInfo} - \)
  \[
  \left\{ \{(i, j)|0 \leq i < 4 \land 0 \leq j \leq i\} : ([i, j]_{1,1}^{4,4}, id) \right\}
  \]

Structures propagatged through expression tree using inference rules
sLGen: Code generation

Only difference from LGen: Σ-CLooG
sLGen: Code generation

Motivation for CLooG:
1. remove redundant multiplications
2. exploit symmetry

LGen:

\[
A = \sum_{i,j=0,2}^{2,2} 4,4[i,j] \left( \sum_{k=0,2}^{L[i,k]4,4 U[k,j]4,4} + S[i,j]4,4 \right)
\]

sLGen:

\[
A = \sum_{i=0}^{2} \left( \sum_{j=0}^{i} 1,1[i,j] \left( L[i,0]4,4 U[0,j]4,4 + S[i,j]4,4 \right) + \sum_{j=i+1}^{3} 1,1[i,j] \left( L[i,0]4,4 U[0,j]4,4 + S[j,i]4,4 \right) \right)
\]

\[
+ \sum_{j=0}^{3} 1,1[i,j] \left( L[3,0]4,4 U[0,j]4,4 + S[3,j]4,4 \right) + \sum_{i=1}^{3} \sum_{i=k}^{3} 3,3[i,j] \left( L[i,k]4,4 U[k,j]4,4 \right)
\]
What are CLooG statements?

- `<domain, schedule, body>`
  - domain: polyhedral set \( \sigma \) representing the iteration space of statement (range of indices in the body)
  - schedule: polyhedral map \( \rho \) determining the traversal order of domain’s tuples (order of indices in the body)
  - body: \( \Sigma\text{-LL} \) expression \( B \)

- generated by each structure’s SInfo and AInfo

Example:

\[
\begin{align*}
\sigma &= \{(i, k, j) | k = 0 \land 0 \leq i < 4 \land 0 \leq j \leq i\} \\
\rho &= ((i, k, j), (k, i, j)) \\
B &= \frac{1,4}{4,4}[i, j] \left( L[i, k]^{4,4} U[k, j]^{4,4} + S[i, j]^{4,4} \right)
\end{align*}
\]
sLGen: Code generation

Role of StmtGen: create CLooG statements using sBLAC expression tree

1. create unique index space for input sBLAC
   - \( A_{i,j} = L_{i,k} U_{k,j} + S_{i,j} \to 3 \) indices found

2. expand SInfo/AInfo dictionaries using new index space
   - \( L.SInfo = \begin{cases} 
   G : \{ (i, k, j) | 0 \leq i < 4 \land 0 \leq k \leq i \} \\
   Z : \{ (i, k, j) | 0 \leq i < 4 \land i < k < 4 \} 
   \end{cases} \)
   - \( L.AInfo = \begin{cases} 
   \{ (i, k, j) | 0 \leq 4 \land 0 \leq k \leq i \} : ([i, k]_{1,1}^{4,4}, id) 
   \end{cases} \)

   - \( U, A, S \) change as well

3. build CLooG statement for each operator in input sBLAC

---

Daniele G. Spampinato, Markus Püschel

Reported by Edward Hutter
sLGen: Code generation

Iteration space of first leaf:

\[
\text{iterSpace}_{LU} = L\text{.SInfo}[G] \cap U\text{.SInfo}[G]
\]

\[
= \{(i, k, j)|0 \leq k < 4 \land k \leq i, j < 4\}
\]

Split iteration space: initial accesses and accumulations

\[
\text{iterSpace}_{LU}^{\text{init}} = \{(i, 0, j)|0 \leq i < 4 \land 0 \leq j < 4\}
\]

\[
\text{iterSpace}_{LU}^{\text{acc}} = \{(i, k, j)|1 \leq k < 4 \land k \leq i, j < 4\}
\]

Final domains need to be intersected with AInfo dictionaries

\[
\text{domain}_{LU}^{\text{init}} = \text{iterSpace}_{LU}^{\text{init}}
\]

\[
\text{domain}_{LU}^{\text{acc}} = \text{iterSpace}_{LU}^{\text{acc}}
\]
sLGen: Code generation

Final bodies need to be intersected with AInfo dictionaries

\[ B_{LU}^{\text{init}} = B_{LU}^{\text{acc}} = \frac{1}{4,4} \sum_{i,j} L[i,k]^{4,4} U[k,j]^{4,4} \]

Finally, we get partial CLooG statements for \( LU \):

\[ s_{LU}^{\text{init}} = < \text{dom}_{LU}^{\text{init}}, \emptyset, B_{LU}^{\text{init}} > \]
\[ s_{LU}^{\text{acc}} = < \text{dom}_{LU}^{\text{acc}}, \emptyset, B_{LU}^{\text{acc}} > \]

Remaining steps:

1. Recurse up tree
2. Initialize CLooG statements for internal nodes and root
3. Choose loop index order and recurse back down

\[ A = \sum_{i=0}^{2} \left( \sum_{j=0}^{i} \frac{1}{4,4} \sum_{i,j} (L[i,0]^{4,4} U[0,j]^{4,4} + S[i,j]^{4,4}) + \sum_{j=i+1}^{3} \frac{1}{4,4} \sum_{i,j} (L[i,0]^{4,4} U[0,j]^{4,4} + S[i,j]^{4,4}) \right) \]
\[ + \sum_{j=0}^{3} \frac{1}{4,4} \sum_{i,j} (L[3,0]^{4,4} U[0,j]^{4,4} + S[3,j]^{4,4}) + \sum_{i=1}^{3} \sum_{i=k}^{3} \sum_{j=k}^{3} \frac{1}{4,4} \sum_{i,j} (L[i,k]^{4,4} U[k,j]^{4,4} + S[i,j]^{4,4}) \]
sLGen performance analysis setup

- benchmarks performed on an Intel Sandy Bridge, 3.3 GHz, AVX, 32 kB L1 D-cache, 256 kB L2 cache
- double precision arrays, 32-byte aligned, row-wise stored
- BLAS, BLAS-like (no structure support), Non-BLAS (multiple BLAS calls required)
- compared against:
  - Intel MKL v11.2
  - naive code compiled with Intel icc v15 (scalar, unoptimized, fixed-size)
  - LGen code without support for structures
    - compilation flags: -O3 -xHost -fargument-noalias -fno-alias -ip -ipo
- performance measured in flops/cycle (8 is peak)
- each data point is median of 30 iterations, more advanced statistics also used
sLGen performance: BLAS

<table>
<thead>
<tr>
<th>Category</th>
<th>Label</th>
<th>sBLAC</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>dyrk</td>
<td>$S_u = A A^T + S_u$</td>
<td>$A \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td></td>
<td>dsyrv</td>
<td>$z = L \backslash z$</td>
<td>$L \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td>BLAS-like</td>
<td>dlauum</td>
<td>$A = L U + S_l$</td>
<td>$L, U \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td></td>
<td>dlauim</td>
<td>$A = S_u L + A$</td>
<td>$S_u, L \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td>Non-BLAS</td>
<td>composite</td>
<td>$A = (L_0 + L_1) S_l + x x^T$</td>
<td>$L_0, S_l \in \mathbb{R}^{n \times n}$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Label</th>
<th>sBLAC</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>dryrk</td>
<td>$S_u = A A^T + S_u$</td>
<td>$A \in \mathbb{R}^{n \times 4}$</td>
</tr>
<tr>
<td></td>
<td>dtrsv</td>
<td>$z = L \backslash z$</td>
<td>$L \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td>BLAS-like</td>
<td>dlmms</td>
<td>$A = L U + S_l$</td>
<td>$L, U \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td></td>
<td>dsymlm</td>
<td>$A = S_u L + A$</td>
<td>$S_u, L \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td>Non-BLAS</td>
<td>composite</td>
<td>$A = (L_0 + L_1) S_l + x z^T$</td>
<td>$L_0, S_l \in \mathbb{R}^{n \times n}$</td>
</tr>
</tbody>
</table>

![Graphs showing performance comparison](image)

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Label</th>
<th>sBLAC</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>dryrk</td>
<td>$S_a = A A^T + S_a$</td>
<td>$A \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td></td>
<td>dtrsv</td>
<td>$x = L \backslash x$</td>
<td>$L \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td>BLAS-like</td>
<td>dlammm</td>
<td>$A = L U + S_i$</td>
<td>$L, U \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td></td>
<td>dtrmm</td>
<td>$A = S_u L + A$</td>
<td>$S_u, L \in \mathbb{R}^{n \times n}$</td>
</tr>
<tr>
<td>Non-BLAS</td>
<td>composite</td>
<td>$A = (I_O + L_1) S_1 + x x^T$</td>
<td>$I_O, S_1 \in \mathbb{R}^{n \times n}$</td>
</tr>
</tbody>
</table>

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

Comments:
- detailed performance analysis, more than previous work (2014)
- autotuning procedure barely explained.
- how may iterations to decide if "good"? How expensive relative to actual computation?

Limitations
- $\nu$-BLACs not able to exploit structure
- with increasing number of vector lanes, will be more opportunity

Extensions
- very extensible to new structures
- adding a structure requires adding SInfo/AInfo and a set of loaders/storers (vectorized codelets)