A Basic Linear Algebra Compiler for Structured Matrices

Daniele G. Spampinato, Markus Püschel

Reported by Edward Hutter

Line of research

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 infastructure 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

LGen compiler

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

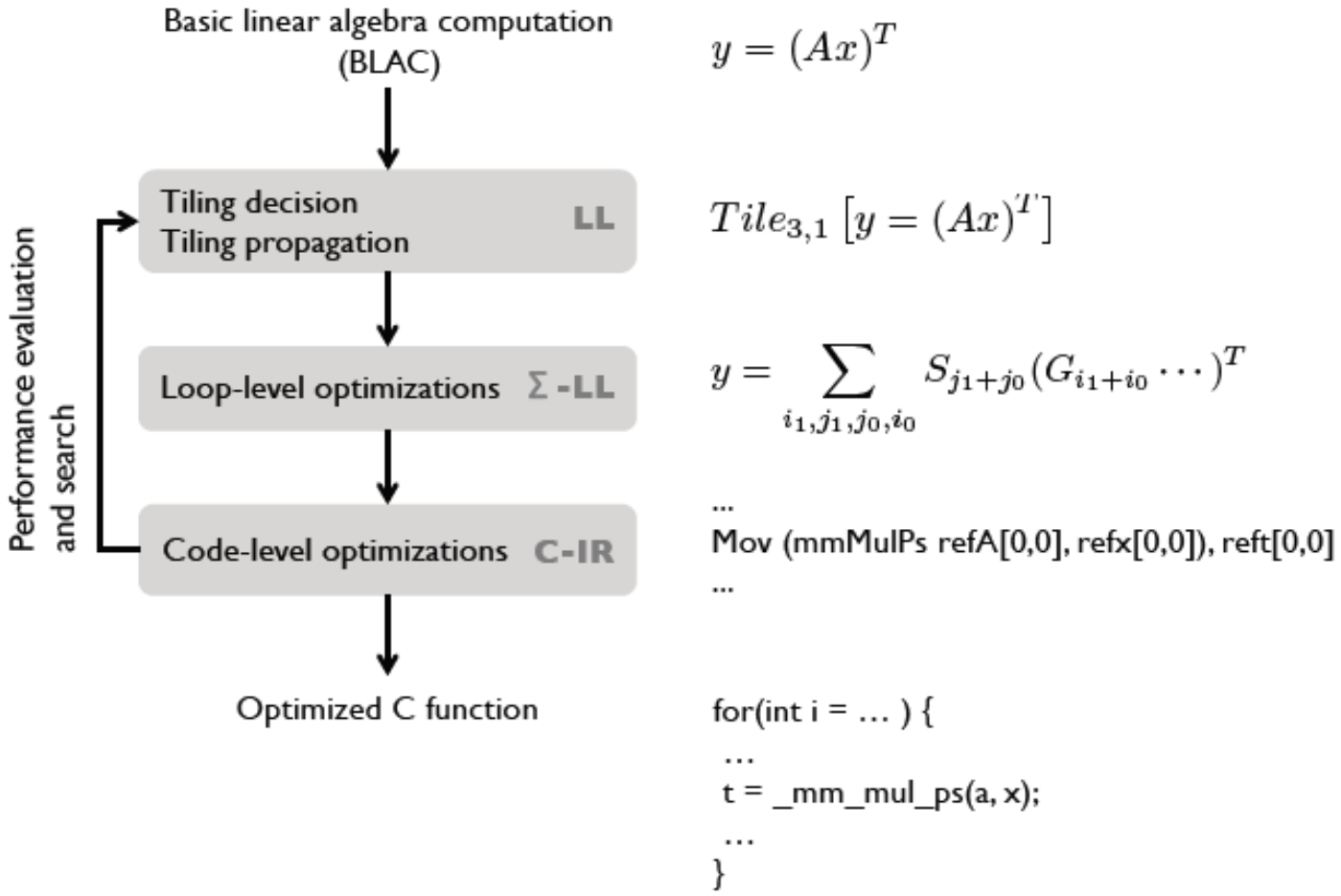


Figure 1: Architecture of LGen.

LGen overview

- BLAC is specified in Linear algebra Language (LL)
- Tiling decision is propogated 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?

```
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.

- 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: $\beta = (Ax + \alpha y)z + \beta$
- 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 propogates down expression tree

Simple BLAC: $y = Ax + y$

$$\begin{aligned}
\text{Tile}_{2,1}(y = Ax + y) &\equiv [y = Ax + y]_{2,1} \\
[y = Ax + y]_{2,1} &\rightarrow [y]_{2,1} = [Ax + y]_{2,1} \\
&\rightarrow [y]_{2,1} = [Ax + y]_{2,1} \\
&\rightarrow [y]_{2,1} = [Ax]_{2,1} + [y]_{2,1} \\
&\rightarrow [y]_{2,1} = [A]_{2,k}[x]_{k,1} + [y]_{2,1}, 1 \leq k \leq 4
\end{aligned}$$

Scalar code generation: Loop optimizations in Σ-LL

Tiled LL translated into Σ-LL

Σ-LL

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

$$[\langle e_L \rangle]_{r,c} + [\langle e_R \rangle]_{r,c} \rightarrow \sum_{i,j} S_i \left(G_i \langle e_L \rangle G_j + G_i \langle e_R \rangle G_j \right) S_j \quad (10)$$

$$[\langle e_L \rangle]_{r,k} \cdot [\langle e_R \rangle]_{k,c} \rightarrow \sum_{i,j,k} S_i \left(G_i \langle e_L \rangle G_k \cdot G_k \langle e_L \rangle G_j \right) S_j \quad (11)$$

$$[\langle \text{scalar} \rangle]_{1,1} \cdot [\langle e \rangle]_{r,c} \rightarrow \sum_{i,j} S_i \left(\langle \text{scalar} \rangle \cdot G_i \langle e \rangle G_j \right) S_j \quad (12)$$

$$[\langle e \rangle]^T_{r,c} \rightarrow \sum_{i,j} S_j \left(G_i \langle e \rangle G_j \right)^T S_i \quad (13)$$

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

Scalar code generation: Gather/Scatter

Lets 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 $A_{3\times 3}$

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

$$A = S_L A S_R$$

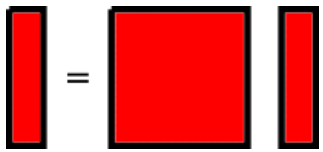
$$G_R = S_L = G_L^T,\ G_L = S_R = G_R^T$$

Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$

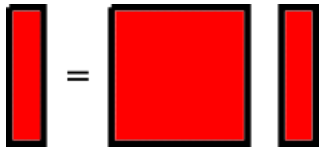


Scalar code generation: Gather/Scatter

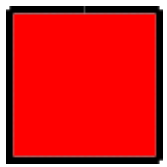
Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



A

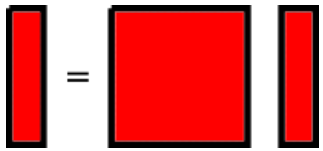


Scalar code generation: Gather/Scatter

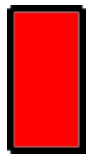
Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



$AG_0^{2,4}$

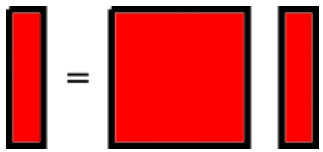


Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



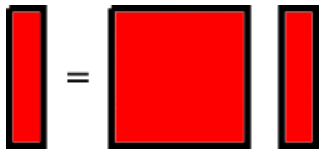
$G_0^{2,4} A G_0^{2,4}$

Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



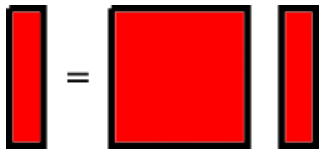
$(G^{2,4}_0 A G^{2,4}_0) S^{2,4}_0$

Scalar code generation: Gather/Scatter

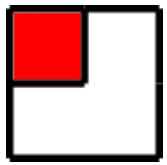
Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



$S^{2,4}_0(G^{2,4}_0 A G^{2,4}_0)S^{2,4}_0$

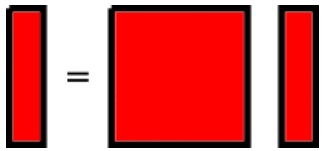


Scalar code generation: Gather/Scatter

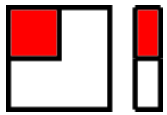
Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



$$S_0^{2,4}(G_0^{2,4} A G_0^{2,4}) S_0^{2,4} S_0^{2,4}(G_0^{2,4} x)$$

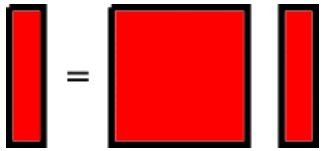


Scalar code generation: Gather/Scatter

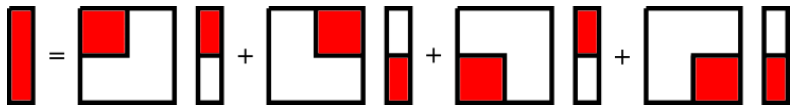
Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



$$y = S_0^{2,4}(G_0^{2,4} A G_0^{2,4}) S_0^{2,4} S_0^{2,4}(G_0^{2,4} x) + S_0^{2,4}(G_0^{2,4} A G_2^{2,4}) S_2^{2,4} S_2^{2,4}(G_2^{2,4} x) +$$
$$S_2^{2,4}(G_2^{2,4} A G_0^{2,4}) S_0^{2,4} S_0^{2,4}(G_0^{2,4} x) + S_2^{2,4}(G_2^{2,4} A G_2^{2,4}) S_2^{2,4} S_2^{2,4}(G_2^{2,4} x)$$

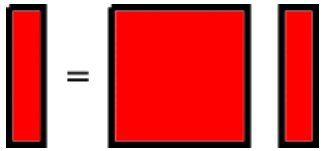


Scalar code generation: Gather/Scatter

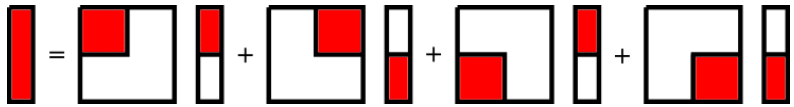
Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



$$y = S_0^{2,4}(G_0^{2,4} A G_0^{2,4})(G_0^{2,4} x) + S_0^{2,4}(G_0^{2,4} A G_2^{2,4})(G_2^{2,4} x) + S_2^{2,4}(G_2^{2,4} A G_0^{2,4})(G_0^{2,4} x) + S_2^{2,4}(G_2^{2,4} A G_2^{2,4})(G_2^{2,4} x)$$

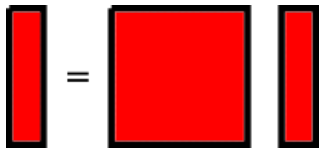


Scalar code generation: Gather/Scatter

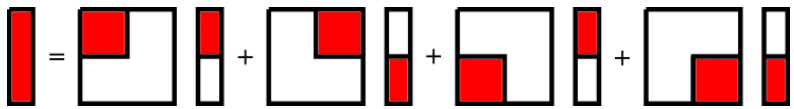
Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



$$y = \sum_{i=0,2}^{3} S_i^{2,4}(G_i^{2,4} A G_0^{2,4})(G_0^{2,4} x) + S_i^{2,4}(G_i^{2,4} A G_2^{2,4})(G_2^{2,4} x)$$

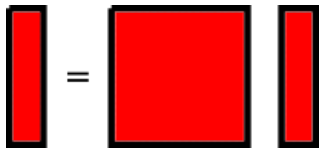


Scalar code generation: Gather/Scatter

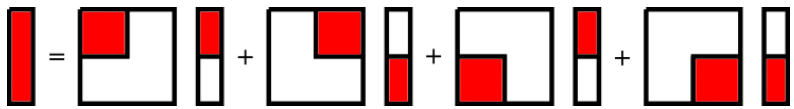
Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



$$y = \sum_{i=0,2}^{3} \sum_{j=0,2}^{3} S_i^{2,4} (G_i^{2,4} A G_j^{2,4})(G_j^{2,4} x)$$

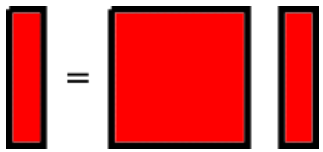


Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



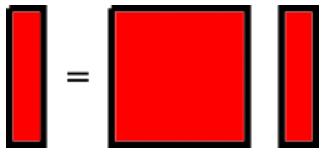
$$y = \sum_{i=0,2}^{3} \sum_{j=0,2}^{3} S_i^{2,4} \cdot \sum_{i'=0}^{1} \sum_{j'=0}^{1} S_{i'}^{2,4} (G_{i'}^{2,4} G_i^{2,4} A G_j^{2,4} G_{j'}^{2,4}) (G_{j'}^{2,4} G_j^{2,4} x)$$

Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



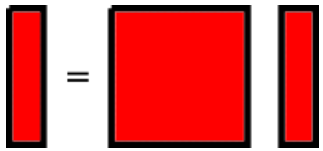
$$y = \sum_{i,j,i',j'} S^{2,4}_i S^{2,4}_{i'} (G^{2,4}_{i'} G^{2,4}_i A G^{2,4}_j G^{2,4}_{j'})(G_{j'} G^{2,4}_j x)$$

Scalar code generation: Gather/Scatter

Key concept: tiled computations $\equiv$ matrix summations

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

Example: $y = Ax$



$$y = \sum_{i,j,i',j'} S^{2,4}_{i+i'}(G^{2,4}_{i+i'} A G^{2,4}_{j+j'})(G_{j+j'} x)$$

Scalar code generation: Gather/Scatter summary

Key concepts:

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

$$[\langle e_L \rangle]_{r,c} + [\langle e_R \rangle]_{r,c} \rightarrow \sum_{i,j} S_i \left(G_i \langle e_L \rangle G_j + G_i \langle e_R \rangle G_j \right) S_j \quad (10)$$

$$[\langle e_L \rangle]_{r,k} \cdot [\langle e_R \rangle]_{k,c} \rightarrow \sum_{i,j,k} S_i \left(G_i \langle e_L \rangle G_k \cdot G_k \langle e_L \rangle G_j \right) S_j \quad (11)$$

$$[\langle \text{scalar} \rangle]_{1,1} \cdot [\langle e \rangle]_{r,c} \rightarrow \sum_{i,j} S_i \left(\langle \text{scalar} \rangle \cdot G_i \langle e \rangle G_j \right) S_j \quad (12)$$

$$[\langle e \rangle]^T_{r,c} \rightarrow \sum_{i,j} S_j \left(G_i \langle e \rangle G_j \right)^T S_i \quad (13)$$

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

Matrix summations in what order?

Scalar code generation: Loop fusion

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

Simplification properties help:

$$h^{n',N}_{b,s} \circ h^{n,n'}_{b',s'} = h^{n,N}_{b+sb',ss'}$$
$$G_L(h) = S_L(h)^T$$
$$G^{n,n}_{0,1} = S^{n,n}_{0,1} = I_n$$
$$S_L(h) \cdot S_L(h') = S_L(h \circ h')$$
$$G_L(h) \cdot G_L(h') = G_L(h' \circ h)$$
$$G_L(h^{s \rightarrow d}_{i,1}) \sum_{i'=b,s}^{d-1} S_L(h^{s \rightarrow d}_{i',1}) = G^{s,d}_i S^{s,d}_i = I_s$$

- 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:

Π	tl	sl	ilp
i	0	0	1
j	0	2	1
k	1	1	0

(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]
```

Scalar code generation: Performance evaluation and search

A number of C functions can be generated at compile-time

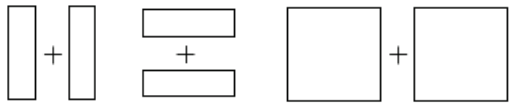
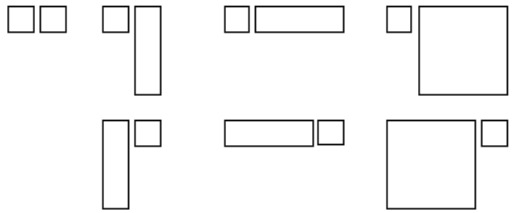
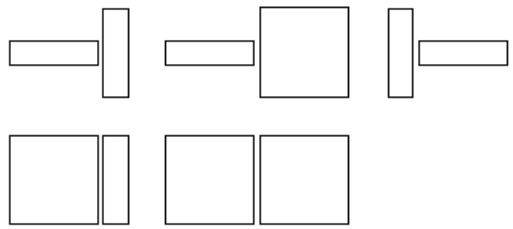
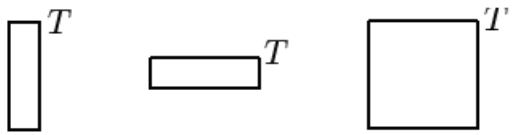
- decided by number of valid tiling strategies
- influenced by extra degrees of freedom introduced by certain BLACs
- Loop orderings are fixed
- exhaustive search or random search

Generated functions are executed and autotuned at runtime

Vector code generation: Overview

LGen requires 18 ν-BLACs to vectorize LL

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

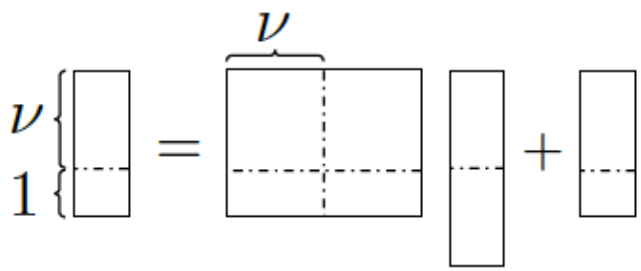
Operator	Required ν-BLACs
Addition (3 ν-BLACs)	
Scalar Multiplication (7 ν-BLACs)	
Matrix Multiplication (5 ν-BLACs)	
Transposition (3 ν-BLACs)	

Vector code generation: Extensions from scalar

Vectorized code generation requires extra steps:

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

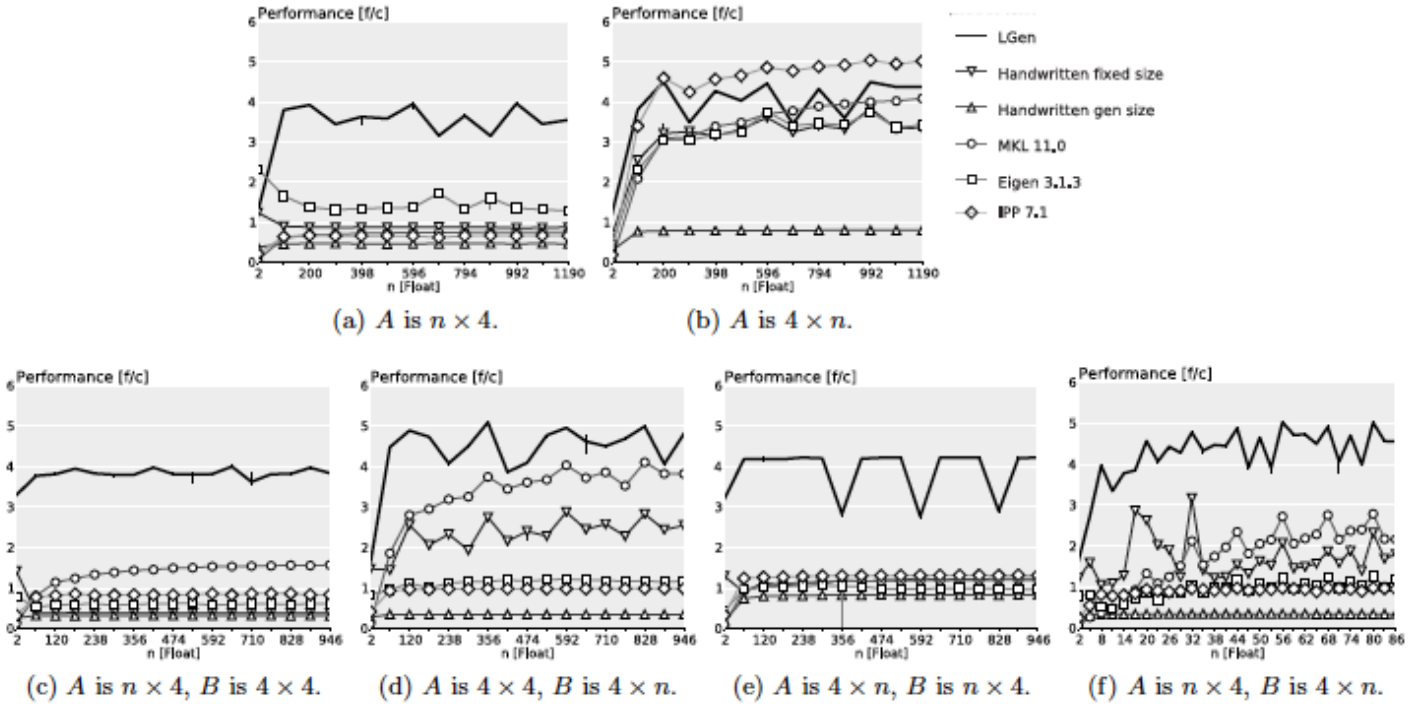
$y = A_{3\times4}x + y$ tiled with $(r, c) = (\nu, \nu)$:



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

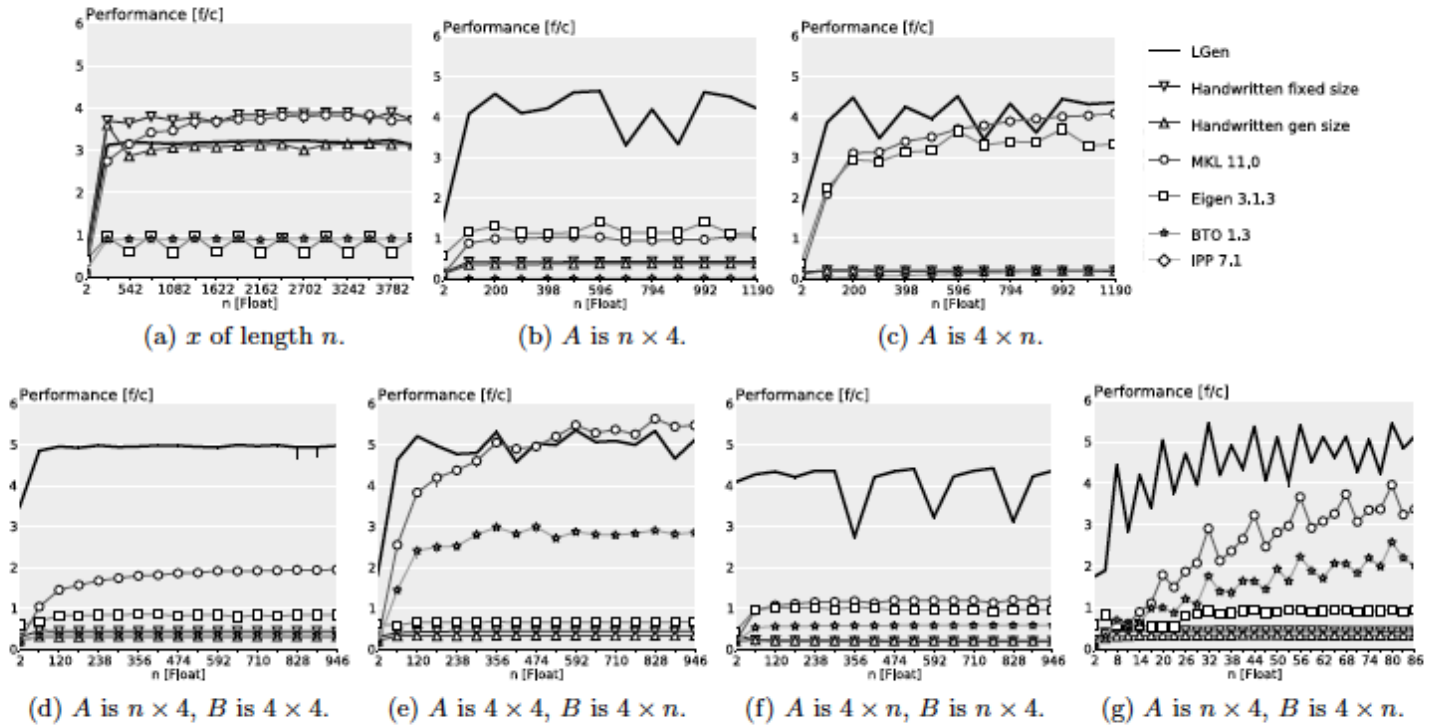


(a) A is $n \times 4$. (b) A is $4 \times n$.

(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): $C = AB$.

LGen performance: BLAS-like BLACs

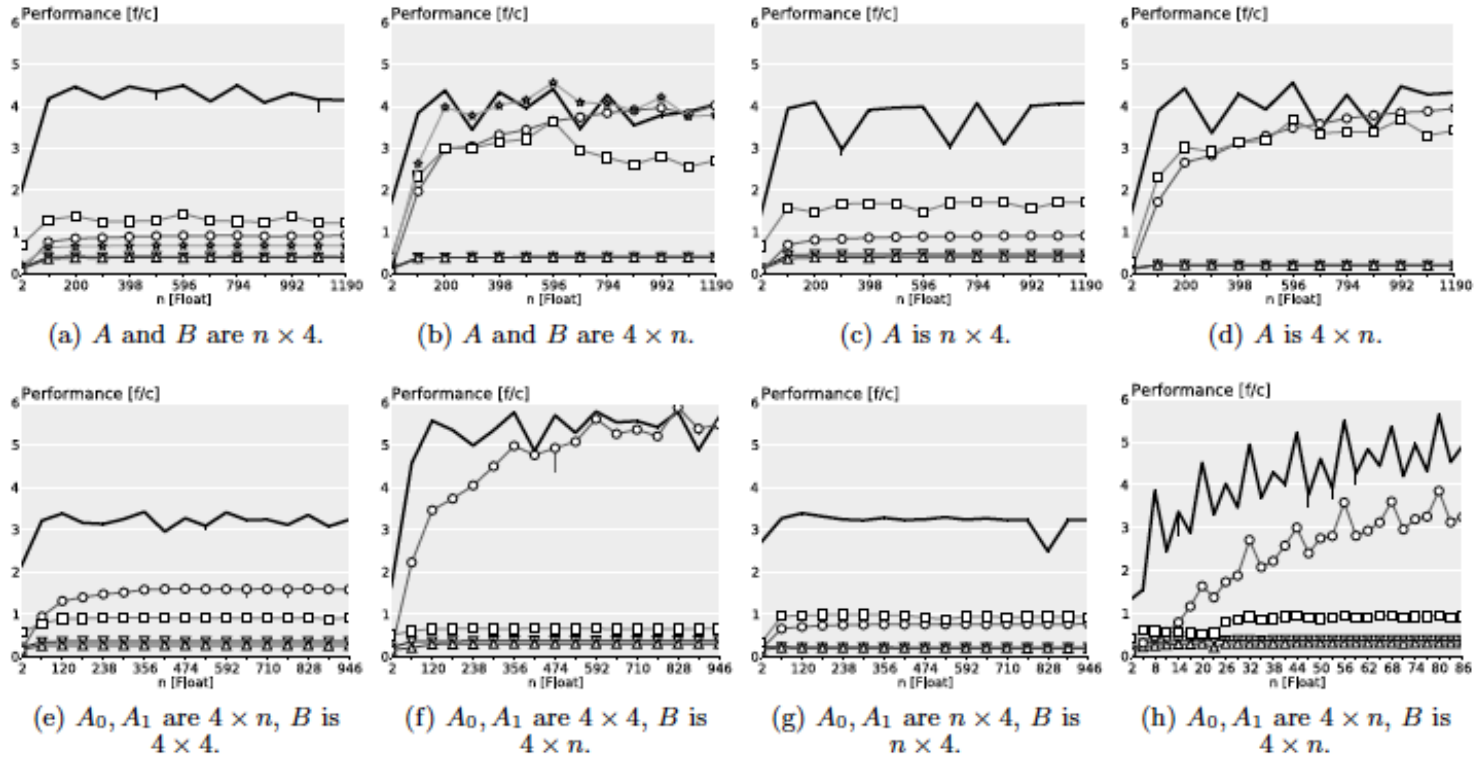


(a) x of length n. (b) A is $n \times 4$. (c) A is $4 \times n$.

(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): $C = \alpha AB + \beta C$.

LGen performance: Complicated BLACs

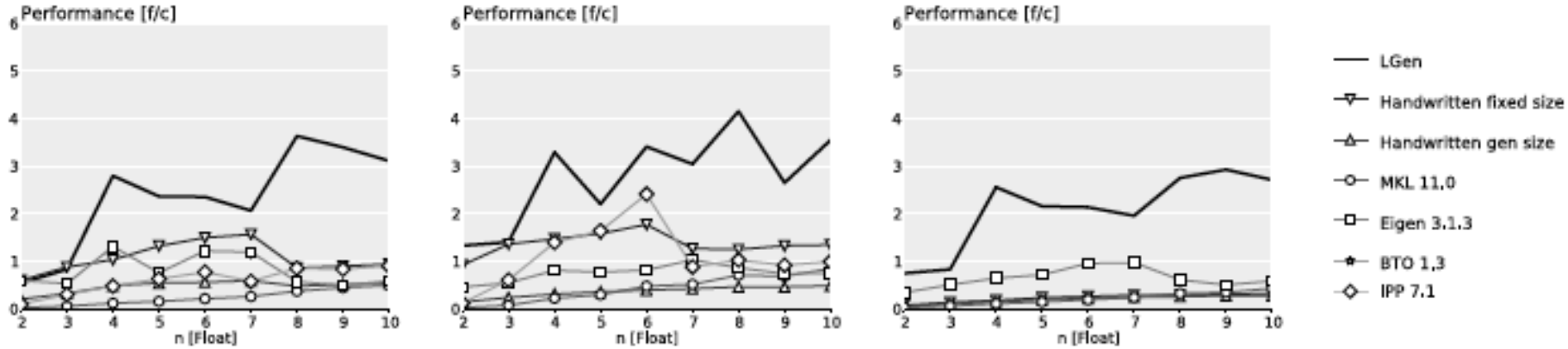


(a) A and B are $n \times 4$. (b) A and B are $4 \times n$. (c) A is $n \times 4$. (d) A is $4 \times n$.

(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):** $C = \alpha(A_0 + A_1)^T B + \beta C$.

LGen performance: Micro BLACs



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

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

Lets 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
  - $s = {}_{m,n}^{k,l}[i,j] : \mathbb{R}^{k \times l} \rightarrow \mathbb{R}^{m \times n}$
  - $A \rightarrow sA = A[i : i+k-1, j : j+l-1]$
- 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 introducted: L, U, S, Z, G

One additional operator introduced: $G \backslash v$

sLGen: Matrix structures

Lets do an example: $A = LU + S \qquad A, L, U, S \in \mathbb{R}^{4\times 4}$

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

Convert LL to Σ-LL

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

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

sLGen: Matrix structures

Translation from Σ-LL to C-IR doesn't change:

- matrix summations mapped to loops
- ν-BLACs mapped to codelets
- gathers/scatters mapped to data accesses

Stucture not exploited

sLGen: Internal representation of structures

Polyhedral sets and maps decompose a structured matrix in Σ-LL

Sets:

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

Maps:

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

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

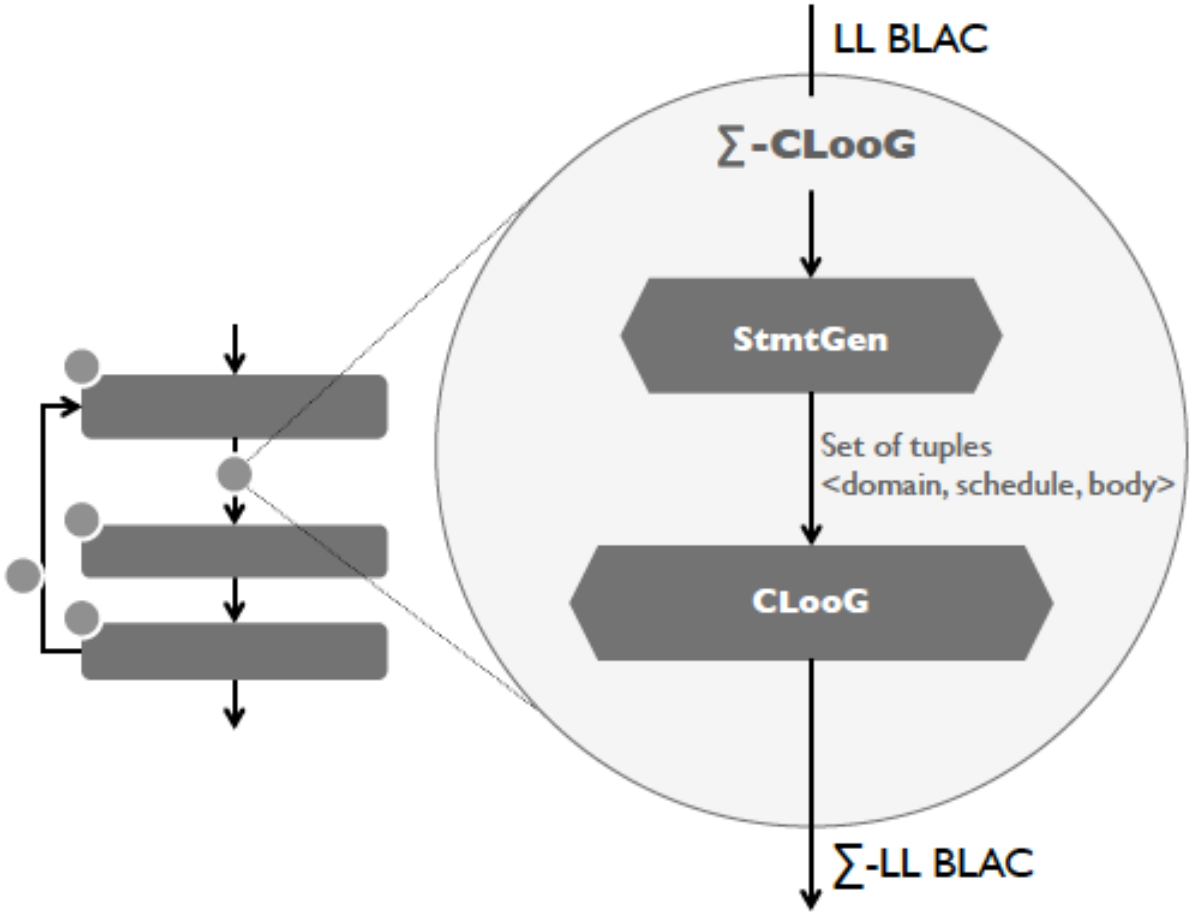
Example:

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

Structures propogated 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:

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

sLGen:

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

sLGen: Code generation

What are CLooG statements?

- <domain, schedule, body>
  - domain: polyhedral set σ representing the iteration space of statement (range of indices in the body)
  - schedule: polyhedral map ρ determining the traversal order of domain's tuples (order of indices in the body)
  - body: Σ-LL expression B
- generated by each structure's SInfo and AInfo

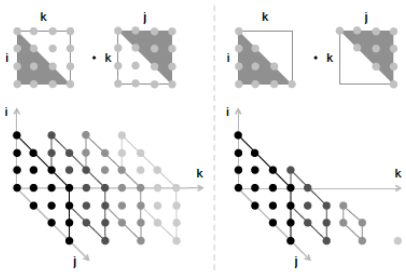
Example:

$$\begin{cases} \sigma = \{(i,k,j) | k = 0 \wedge 0 \leq i < 4 \wedge 0 \leq j \leq i\} \\ \rho = ((i,k,j),(k,i,j)) \\ B = {}^{1,1}_{4,4}[i,j] \left(L[i,k]^{4,4}_{1,1} U[k,j]^{4,4}_{1,1} + S[i,j]^{4,4}_{1,1} \right) \end{cases}$$

sLGen: Code generation

Role of StmtGen: create CLooG statements using sBLAC expression tree

1. create unique index space for input sBLAC
   - $A_{i,j} = L_{i,k}U_{k,j} + S_{i,j} \rightarrow$ 3 indices found
2. expand SInfo/AInfo dictionaries using new index space
   - L.SInfo
   $$= \begin{cases} G : \{(i,k,j) | 0 \leq i < 4 \wedge 0 \leq k \leq i\} \\ Z : \{(i,k,j) | 0 \leq i < 4 \wedge i < k < 4\} \end{cases}$$
   - L.AInfo
   $$= \left\{ \{(i,k,j) | 0 \leq 4 \wedge 0 \leq k \leq i\} : ([i,k]_{1,1}^{4,4}, id) \right.$$
   - U, A, S change as well
3. build CLooG statement for each operator in input sBLAC



Data: Inputs I_0, I_1.
Result: Iteration space (*iterSpace*) of $I_0 I_1$.
$iterSpace \leftarrow \emptyset$;
for $(\mathcal{M}_0 : \sigma_0) \in$ I$_0$.SInfo : $\mathcal{M}_0 \neq \mathcal{Z}$ **do**
  for $(\mathcal{M}_1 : \sigma_1) \in$ I$_1$.SInfo : $\mathcal{M}_1 \neq \mathcal{Z}$ **do**
    // iteration space based on all pairs
    // of input non-zero regions.
    $iterSpace = iterSpace \cup (\sigma_0 \cap \sigma_1)$;
  end
end
Algorithm 1: Computing the iteration space for matrix multiplication.

sLGen: Code generation

Iteration space of first leaf:

$$\begin{aligned} iterSpace_{LU} &= L.\mathsf{SInfo}[G] \cap U.\mathsf{SInfo}[G] \\ &= \{(i,k,j) | 0 \leq k < 4 \wedge k \leq i,j < 4\} \end{aligned}$$

Split iteration space: initial accesses and accumulations

$$\begin{aligned} iterSpace_{LU}^{init} &= \{(i,0,j) | 0 \leq i < 4 \wedge 0 \leq j < 4\} \\ iterSpace_{LU}^{acc} &= \{(i,k,j) | 1 \leq k < 4 \wedge k \leq i,j < 4\} \end{aligned}$$

Final domains need to be intersected with AInfo dictionaries

$$\begin{aligned} domain_{LU}^{init} &= iterSpace_{LU}^{init} \\ domain_{LU}^{acc} &= iterSpace_{LU}^{acc} \end{aligned}$$

sLGen: Code generation

Final bodies need to be intersected with AInfo dictionaries

$$B_{LU}^{init} = B_{LU}^{acc} = {}^{1,1}_{4,4}[i,j]\left(L[i,k]^{4,4}_{1,1}U[k,j]^{4,4}_{1,1}\right)$$

Finally, we get partial CLooG statements for LU:

$$s_{LU}^{init} =< dom_{LU}^{init}, \emptyset, B_{LU}^{init} >$$
$$s_{LU}^{acc} =< dom_{LU}^{acc}, \emptyset, B_{LU}^{acc} >$$

Remaining steps:

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

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

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

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

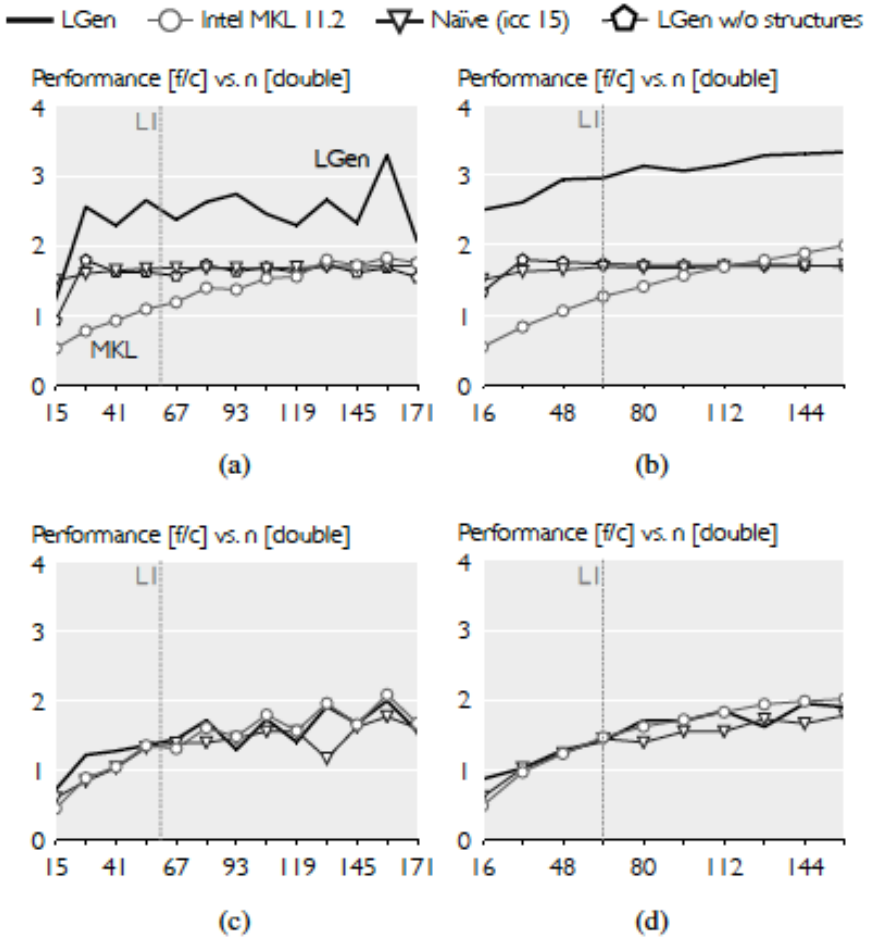


Figure 5. BLAS category: (a)–(b) *dsyrk* ($f = 4n^2 + 4n$) and (c)–(d) *dtrsv* ($f = n^2 + n$). In (b) and (d) all sizes are multiple of the vector length ($\nu = 4$). LGen w/o structures is missing in (c) and (d) as the triangular solve operator is not supported by such an approach.

sLGen performance: BLAS-like

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

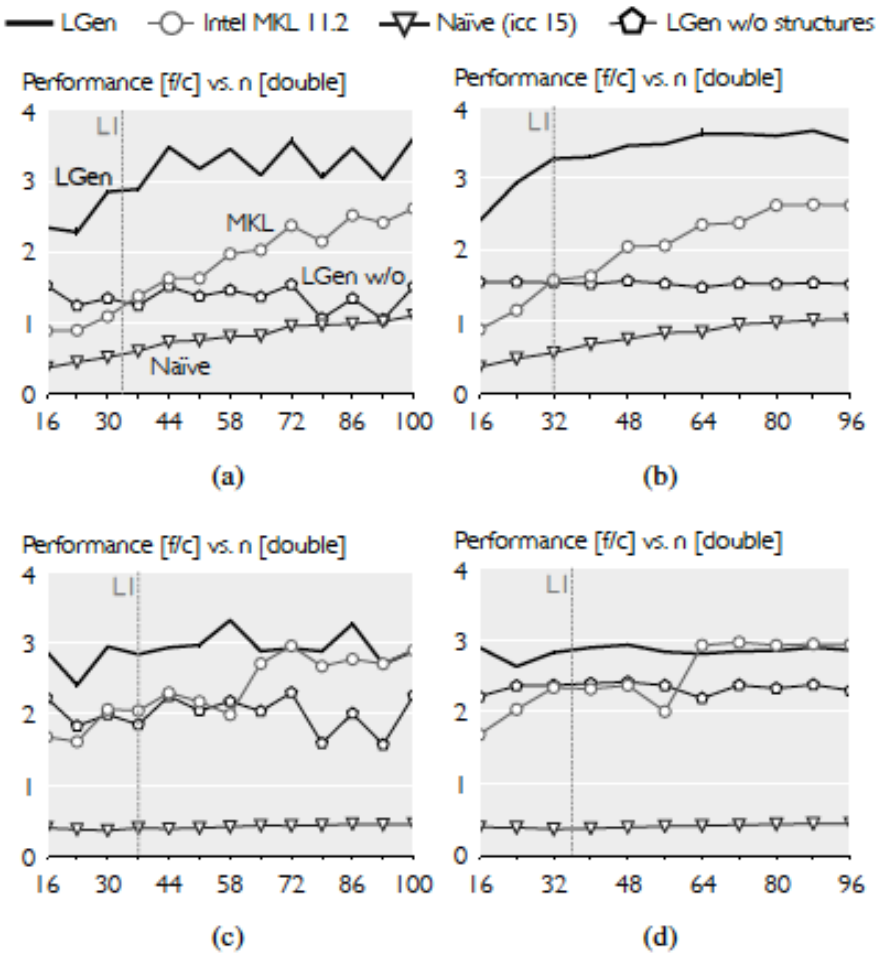


Figure 6. BLAS-like category: (a)–(b) *dlusmm* ($f = \frac{1}{3}(2n^3 + n) + n^2$) and (c)–(d) *dsylmm* ($f = n^3 + n^2$). In (b) and (d) all sizes are multiple of the vector length ($\nu = 4$).

sLGen performance: BLAS-like

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

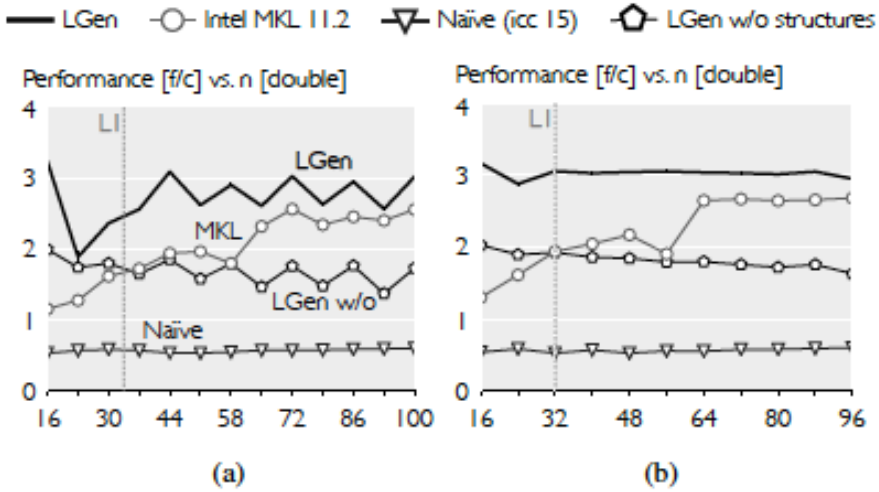


Figure 7. Non-BLAS category: (a)–(b) *composite* ($f = n^3 + \frac{5}{2}(n^2 + n)$). In (b) all sizes are multiple of the vector length ($\nu = 4$).

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

- ν-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)