LGen: Program Generator for Linear Algebra

Daniele Spampinato
Markus Püschel

Computer Science
ETH Zürich

Kalman Filter

Predict

\[ x_k = Ax_{k-1} + Bu_k \]
\[ P_k = AP_{k-1}A^T + Q \]

Update

\[ x_k = x_k + P_kH^T(HP_kH^T + R)^{-1}(y_k - Hx_k) \]
\[ P_k = P_k - P_kH^T(HP_kH^T + R)^{-1}HP_k \]

For example, commonly used in robotics
Let’s assume 13 states

Fast code needed
Three Performance Plots

Matrix multiplication
Gflop/s

Fast Fourier transform
Gflop/s

WiFi Receiver
Gflop/s

input size
input size
input size

fastest code
straightforward C

Same op count
Best compiler + optimization flags

Evolutions of Processors (Intel)

CPU Frequency [GHz]

1993 1995 1997 1999 2001 2003 2005 2007 2009 2011 2013 2015

Year

Pentium

Pentium III

Pentium Pro

Pentium 4

Core

Nehalem

Haswell

Sandy Bridge

free speedup

And there is Processor Variety ...

Domain-specific (here: Tile)

FPGA accelerators

Fast code = good algorithm + code style + locality + vectorization + parallelization

LTE Viterbi Decoder
(scalar code)

LTE Viterbi Decoder
(vector code)
**Current practice:** Thousands of programmers tune performance-critical code to processors. This is redone for every new processor and for every new processor generation.

---

**Goal:**

**Program Generation for Linear Algebra**

Generate *highest performance code* for linear algebra computations directly from a mathematical description

**Approach**

- Mathematical DSLs
- Rewriting systems for difficult optimizations
- Compiler
- Learning and search for fine-tuning

![Program Generation in Spiral](image)

Example: Linear Transforms

Start: Basic linear algebra

---

LGel: Program Generator for Linear Algebra
SRL Workshop, Zürich, October 2016
© Markus Püschel, Computer Science, ETH Zürich
Library performance sgemm (C += AB)
Intel Core i7-2600 CPU @ 3.40GHz

91% of peak = problem solved

What happens for smaller sizes?

A closer look at small problem sizes
Intel Core i7-2600 CPU @ 3.40GHz

28% of peak

[30 x n] times [n x 30] matrix
Are small problems so important?

Required by many performance-critical applications:
- Optimization algorithms
- Kalman filters
- Geometric transformations
- Real-time localization and mapping

Often for specific input sizes

Do not necessarily comply with standard interface (e.g., BLAS)

Of special interest for a variety of embedded systems
- Reduced HW and SW resources

Basic Linear Algebra Computations (BLACs)

Examples:

\[ y = Ax \]

\[ C = \alpha AB^T + \beta C \]

\[ \gamma = x^T (A + B)y + \delta \]

Composed of:
- Scalars, vectors, and matrices
- Operators:
  - Addition
  - Scalar multiplication
  - Matrix multiplication
  - Transposition

Assumption: All input and output vectors and matrices have a fixed size
void f(double const * A, double const * x, double * y) {
    double t0, ..;
    t0 = x[0];
    t1 = x[1];
    ...
    t9 = t3 * t0;
    t10 = t6 * t0;
    t11 = t4 * t1;
    t12 = t9 + t11;
    ...
    y[0] = t16;
    y[1] = t18;
}
**Architecture of LGen**

*Design similar to Spiral*

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

\[
y = Ax
\]

\[
[y = Ax]_{2,1}
\]

\[
y = \sum_{i,j} S_i(G_i \cdots)
\]

```
...Mov(mmMulPs A[0,0], x[0,0], t[0,0]
...
```

**Scalar code generation**

\[
4
\]

\[
y = Ax + y
\]
Tiling in LL

\[ y = Ax + y \]

Task: Tiling decision for equation → tiling decision for operands

Tiling decision for equation

\( r = 2, \ c = 1 \)

Tiling in LL

\[ y = Ax + y \]

\[ y_{2,1} = [Ax + y]_{2,1} \]
Tiling in LL

\[ y = Ax + y_{2,1} \]

\[ y_{2,1} = [Ax + y]_{2,1} \]

\[ y_{2,1} = [Ax]_{2,1} + [y]_{2,1} \]

---

Tiling in LL

\[ y = Ax + y_{2,1} \]

\[ y_{2,1} = [Ax + y]_{2,1} \]

\[ y_{2,1} = [Ax]_{2,1} + [y]_{2,1} \]

\[ y_{2,1} = [A]_{2,1}[x]_{k,1} + [y]_{2,1} \]

Choice that can be used for search
Tiling in LL

\[
[y = Ax + y]_{2,1} \\
[y]_{2,1} = [Ax + y]_{2,1} \\
[y]_{2,1} = [Ax]_{2,1} + [y]_{2,1} \\
[y]_{2,1} = [A]_{2,2} [x]_{2,1} + [y]_{2,1}
\]

Σ-LL: Basics

**Extension of Sigma-SPL (Franchetti et al., PLDI 2005)**

**Gathers:**

\[
G_L = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix} \\
G_R = \begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\]

**Scatters:**

\[
S_L = G_R \\
S_R = G_L
\]

**Extracting a block**

\[
A = \begin{bmatrix}
B & 0 \\
0 & 0
\end{bmatrix} \\
B = A(0 : 1, 0 : 1) = G_L A G_R
\]

**Expanding a block**

\[
C = \begin{bmatrix}
B & 0 \\
0 & 0
\end{bmatrix} \\
C = S_L B S_R
\]

*Gathers and scatters make data accesses explicit*
LL to $\Sigma$-LL

\[ [y]_{2,1} = [A]_{2,2} [x]_{2,1} + [y]_{2,1} \]

\[
S_0 \left( G_0AG_0 \right) S_0 \cdot S_0 \left( G_0x \right) + \cdots + S_2 \left( G_2AG_2 \right) S_2 \cdot S_2 \left( G_2x \right) \\
= \sum_{\ell=0,2}^{3} \sum_{j=0,2}^{3} S_{\ell} \left( G_\ell AG_j \right) \left( G_jx \right) \\
= \sum_{\ell=0,2}^{3} \sum_{j=0,2}^{3} S_{\ell} \sum_{j'=0}^{1} \sum_{j''=0}^{1} S_{j'} \left( G_{\ell,j'} AG_jG_{j''} \right) \left( G_{j'',j'}x \right) 
\]

LL to $\Sigma$-LL

\[ [y]_{2,1} = [A]_{2,2} [x]_{2,1} + [y]_{2,1} \]

\[
S_0 \left( G_0AG_0 \right) S_0 \cdot S_0 \left( G_0x \right) + \cdots + S_2 \left( G_2AG_2 \right) S_2 \cdot S_2 \left( G_2x \right) \\
= \sum_{\ell=0,2}^{3} \sum_{j=0,2}^{3} S_{\ell} \left( G_\ell AG_j \right) \left( G_jx \right) \\
= \sum_{\ell,j,j',j''} S_{\ell+j} \left( G_{\ell+j'} AG_{j+j''} \right) \left( G_{j+j'',j'}x \right) 
\]
**LL to Σ-LL: Loop fusion**

\[
[y]_{2,1} = [A]_{2,2}[x]_{2,1} + [y]_{2,1}
\]

\[
t = \sum_{i,j,j'} S_{t+i} (G_{t+i} AG_{j+j'})(G_{j+j'} x)
\]

\[
y = \sum_{i,i'} S_{t+i} (G_{t+i} t + G_{t+i} y)
\]

\[
y = \sum_{i,i',j,j'} S_{t+i} (G_{t+i} AG_{j+j'})(G_{j+j'} x) + G_{t+i} y
\]

---

**Σ-LL to C-IR**

\[
y = \sum_{i,i',j,j'} S_{t+i} (G_{t+i} AG_{j+j'})(G_{j+j'} x) + G_{t+i} y
\]

Loop unrolling

\[
\text{Mov (Mul A[0,0], x[0,0]), t[0,0]}
\]

\[
\text{Mov (Mul A[0,1], x[1,0]), t[1,0]}
\]

\[
\text{Mov (Mul A[0,2], x[2,0]), t[2,0]}
\]

Scalar replacement

SSA normalization

Peephole optimizations
Vector code generation: Basic Idea

\[ y = Ax + y \]_{r,c} \\
\[ [y]_{\nu,1} = [A]_{\nu,\nu}[x]_{\nu,1} + [y]_{\nu,1} \]

\[ y = \sum_{i,j} S_{i,j}^{\nu,4} \left( G_{i}^{\nu,4} A G_{j}^{\nu,4} \right) \left( G_{j}^{\nu,4} x + G_{i}^{\nu,4} y \right) \]

Computation expressed in terms of \( \nu \)-BLACS

\nu-BLACs

Addition (3 \nu-BLACs)

Scalar Multiplication (7 \nu-BLACs)

Transposition (3 \nu-BLACs)

Matrix Multiplication (5 \nu-BLACs)

18 cases implemented once for every vector ISA
Performance evaluation & search

Search on tiling strategies

Other degrees of freedom: currently model

Current search methods:
- exhaustive search
- random search (in the following: 10 samples)

Experiments

Hardware details
- Intel Xeon X5680 (Westmere EP) @ 3.3 GHz
- 32 kB L1 D-cache
- SSE 4.2 (theoretical peak 8 flops/cycle)
- Intel’s SpeedStep and Turbo Boost disabled

Software details
- RHEL Server 6 – kernel v. 2.6.32
- icc v. 13.1 with flags: -03 -xHost -fargument-noalias -fno-alias -lp -ipo

Comparisons
- Handwritten naïve code: Fixed and general size
- Libraries: Intel MKL v. 11, Intel IPP v. 7.1
- Generators: Eigen v.3.1.3, BTO v.1.3
Plotting

Case 1: Simple BLACs

\[ y = Ax \]
Case 2: BLACs closely matching BLAS

\[ C = \alpha AB + \beta C \]

\[
\begin{align*}
A & \in \mathbb{R}^{n \times 4} \\
B & \in \mathbb{R}^{4 \times 4}
\end{align*}
\]

Case 3: More than one BLAS call

\[ C = \alpha (A_0 + A_1)^T B + \beta C \]

\[
\begin{align*}
A_0 & \in \mathbb{R}^{4 \times n} \\
B & \in \mathbb{R}^{4 \times 4}
\end{align*}
\]
Case 4: Micro BLACs

\[ y = Ax \]
\[ C = AB \]
\[ \alpha = x^T Ay \]

On Embedded Processors

Work by Nikos Kyratsas

\[ C = \alpha(A_0 + A_1)^T B + \beta C \]
**Challenge: Alignment Analysis**

**unaligned load/stores only**

```c
for (size_t i2 = 0; i2 < 400; i2+=16) {
    for (size_t j3 = 0; j3 < 112; j3+=4) {
        for (size_t ii4 = 0; ii4 < 16; ii4+=4) {
            t0_7_0 = _mm_loadu_ps(A + 115*i2 + 115*ii4 + j3);
            t0_6_0 = _mm_loadu_ps(A + 115*i2 + 115*ii4 + j3 + 115);
            t0_5_0 = _mm_loadu_ps(A + 115*i2 + 115*ii4 + j3 + 230);
            t0_4_0 = _mm_loadu_ps(A + 115*i2 + 115*ii4 + j3 + 345);
            t0_3_0 = _mm_loadu_ps(B + 115*i2 + 115*ii4 + j3);
            t0_2_0 = _mm_loadu_ps(B + 115*i2 + 115*ii4 + j3 + 115);
            t0_1_0 = _mm_loadu_ps(B + 115*i2 + 115*ii4 + j3 + 230);
            t0_0_0 = _mm_loadu_ps(B + 115*i2 + 115*ii4 + j3 + 345);

            // 4x4 BLAC: 4x4 + 4x4
            t0_8_0 = _mm_add_ps(t0_7_0, t0_3_0);
            t0_9_0 = _mm_add_ps(t0_6_0, t0_2_0);
            t0_10_0 = _mm_add_ps(t0_5_0, t0_1_0);
            t0_11_0 = _mm_add_ps(t0_4_0, t0_0_0);

            // 4x4 > 4x4 Incompact
            t0_8_1 = t0_8_0;
            t0_9_1 = t0_9_0;
            t0_10_1 = t0_10_0;
            t0_11_1 = t0_11_0;

            _mm_storeu_ps(C + 115*i2 + 115*ii4 + j3, t0_8_1);
            _mm_storeu_ps(C + 115*i2 + 115*ii4 + j3 + 115, t0_9_1);
            _mm_storeu_ps(C + 115*i2 + 115*ii4 + j3 + 230, t0_10_1);
            _mm_storeu_ps(C + 115*i2 + 115*ii4 + j3 + 345, t0_11_1);
        }
    }
}
```

**with aligned load/stores**

```c
for (size_t i2 = 0; i2 < 400; i2+=16) {
    for (size_t j3 = 0; j3 < 112; j3+=4) {
        for (size_t ii4 = 0; ii4 < 16; ii4+=4) {
            t0_7_0 = _mm_loadu_ps(A + 115*i2 + 115*ii4 + j3);
            t0_6_0 = _mm_loadu_ps(A + 115*i2 + 115*ii4 + j3 + 115);
            t0_5_0 = _mm_loadu_ps(A + 115*i2 + 115*ii4 + j3 + 230);
            t0_4_0 = _mm_loadu_ps(A + 115*i2 + 115*ii4 + j3 + 345);
            t0_3_0 = _mm_loadu_ps(B + 115*i2 + 115*ii4 + j3);
            t0_2_0 = _mm_loadu_ps(B + 115*i2 + 115*ii4 + j3 + 115);
            t0_1_0 = _mm_loadu_ps(B + 115*i2 + 115*ii4 + j3 + 230);
            t0_0_0 = _mm_loadu_ps(B + 115*i2 + 115*ii4 + j3 + 345);

            // 4x4 BLAC: 4x4 + 4x4
            t0_8_0 = _mm_add_ps(t0_7_0, t0_3_0);
            t0_9_0 = _mm_add_ps(t0_6_0, t0_2_0);
            t0_10_0 = _mm_add_ps(t0_5_0, t0_1_0);
            t0_11_0 = _mm_add_ps(t0_4_0, t0_0_0);

            // 4x4 > 4x4 Incompact
            t0_8_1 = t0_8_0;
            t0_9_1 = t0_9_0;
            t0_10_1 = t0_10_0;
            t0_11_1 = t0_11_0;

            _mm_storeu_ps(C + 115*i2 + 115*ii4 + j3, t0_8_1);
            _mm_storeu_ps(C + 115*i2 + 115*ii4 + j3 + 115, t0_9_1);
            _mm_storeu_ps(C + 115*i2 + 115*ii4 + j3 + 230, t0_10_1);
            _mm_storeu_ps(C + 115*i2 + 115*ii4 + j3 + 345, t0_11_1);
        }
    }
}
```

---

**Solution: Abstract interpretation**

Assume \(B[i]\) is aligned:

```c
for( size_t j5 = 0; j5 < 80; j5+=16 ) {
    for( size_t k4 = 8; k4 < 48; k4+=8 ) {
        for( size_t kk7 = 0; kk7 < 8; kk7+=4 ) {
            for( size_t jj8 = 0; jj8 < 16; jj8+=4 ) {
                // Eval(\(B + j5 + jj8 + \begin{array}{l} 81*kk4 + 81*kk7) = \begin{array}{l} (-oo,+oo), 0+4Z) + (\begin{array}{l} [8,40), 0+16Z) + (\begin{array}{l} [0,12), 0+4Z) + (\begin{array}{l} [81,81), 81+0Z) * (\begin{array}{l} [8,40), 0+16Z) + (\begin{array}{l} [0,12), 0+4Z) + (\begin{array}{l} [81,81), 81+0Z) * (\begin{array}{l} [0,12), 0+4Z) = (\begin{array}{l} (-oo,+oo), 0+gcd(4,16,4,648,324)Z) = (\begin{array}{l} (-oo,+oo), 0+4Z)
            ...
            t2_19_0 = _mm_loadu_ps(B + j5 + jj8 + \begin{array}{l} 81*kk4 + 81*kk7); // Eval(\(B + j5 + jj8 + \begin{array}{l} 81*kk4 + 81*kk7) = (\begin{array}{l} (-oo,+oo), 0+4Z) + (\begin{array}{l} [8,40), 0+16Z) + (\begin{array}{l} [0,12), 0+4Z) + (\begin{array}{l} [81,81), 81+0Z) * (\begin{array}{l} [8,40), 0+16Z) + (\begin{array}{l} [0,12), 0+4Z) + (\begin{array}{l} [81,81), 81+0Z) * (\begin{array}{l} [0,12), 0+4Z) = (\begin{array}{l} (-oo,+oo), 0+gcd(4,16,4,648,324)Z) = (\begin{array}{l} (-oo,+oo), 0+4Z)
        ...
    }
}
```

---

**Analysis is sound and precise**
What’s Next?

Step 1: BLACs – [CGO 14, DATE 15]
Step 2: structured BLACs – [CGO 16]
Step 3: higher level linear algebra - collaboration
Step 4: applications [e.g., Kalman filter] – not yet

**Kalman Filter**

**Predict**

\[ x_k = A x_{k-1} + B u_k \]
\[ P_k = A P_{k-1} A^T + Q \]

**Update**

\[ x_k = x_k + P_k A^T (H P_k A^T + R)^{-1} (z_k - H x_k) \]
\[ P_k = P_k - P_k A^T (H P_k A^T + R)^{-1} H P_k \]

Fast code

---

**Cl1ck: Synthesis of Linear Algebra Algorithms**

**Cholesky factorization**

More on Cl1ck


---

LGen: Program Generator for Linear Algebra
SRL Workshop, Zürich, October 2016
© Markus Püschel, Computer Science, ETH Zürich
Preliminary results

Performance vs n

- Triangular, continuous-time Sylvester equation
  \[ LX + XU = C \]
- Triangular, continuous-time Lyapunov equation
  \[ L X + XU = C \]
- Cholesky decomposition
  \[ X^T X = S \]

Our generator - best of 4 variants

Intel MKL 11.2

Summary

Goal: Automatically from linear algebra to fast code

\[ \gamma = x^T (A + B)y + \delta \]

LGen: Program Generator for Linear Algebra
SRL Workshop, Zürich, October 2016
© Markus Püschel, Computer Science, ETH Zürich
Soon ...

**Kalman Filter**

Predict
\[ x_k = Ax_{k-1} + Bu_k \]
\[ P_k = AP_{k-1}A^T + Q \]

Update
\[ x_k = x_k + P_kH^T(HP_kH^T + R)^{-1}(y_k - Hx_k) \]
\[ P_k = P_k - P_kH^T(HP_kH^T + R)^{-1}HP_k \]

For example, commonly used in robotics
Let’s assume 13 states

More info: [http://spiral.net/software/lgen.html](http://spiral.net/software/lgen.html)