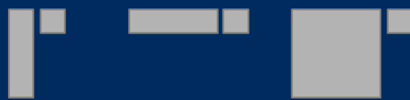
LGen: Program Generator for Linear Algebra

Daniele Spampinato
Markus Püschel

Computer Science
ETH zürich



SPIRAL
www.spiral.net

Kalman Filter

Predict

$x_k = Ax_{k-1} + Bu_k$

$P_k = AP_{k-1}A^T + Q$

Update

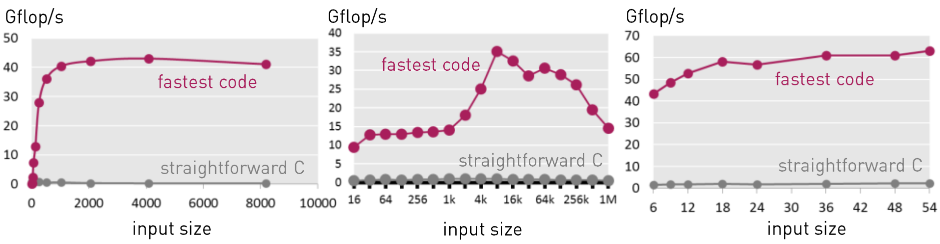
$x_k = x_k + P_kH^T(HP_kH^T + R)^{-1}(z_k - Hx_k)$

$P_k = P_k - P_kH^T(HP_kH^T + R)^{-1}HP_k$

Fast code needed

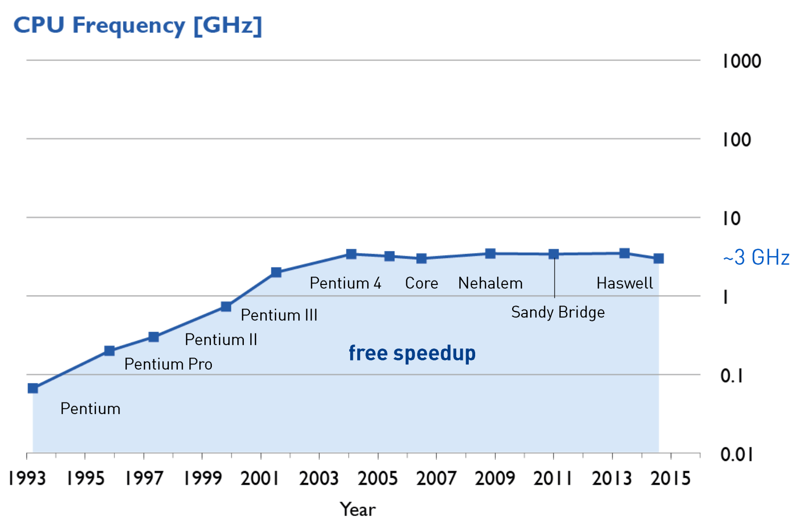
For example, commonly used in robotics
Let's assume 13 states

Three Performance Plots



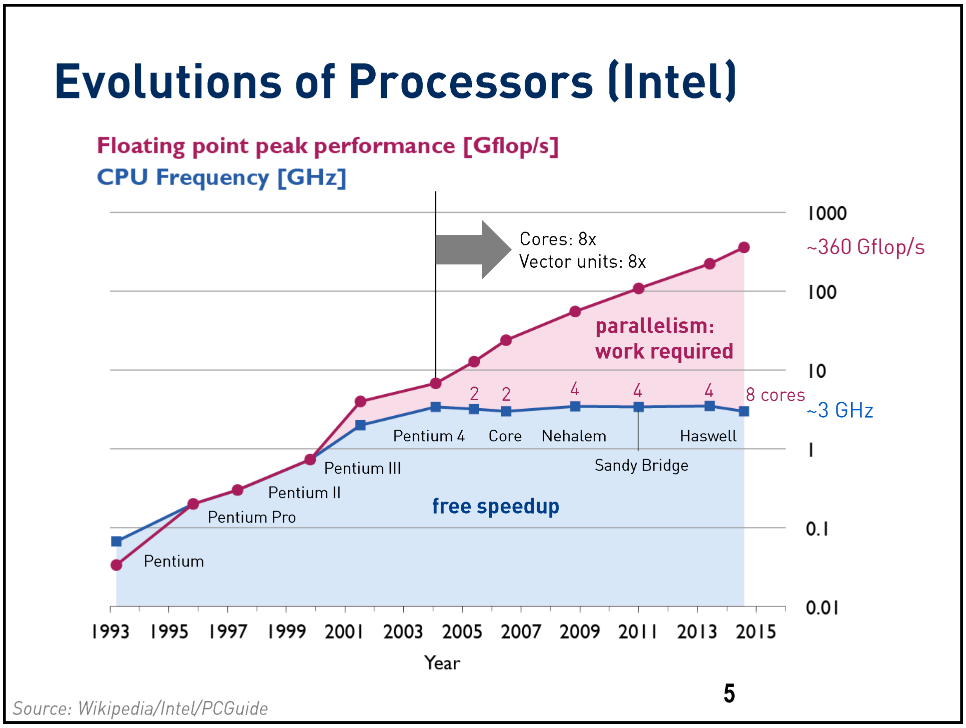
Same op count
Best compiler + optimization flags

Evolutions of Processors (Intel)

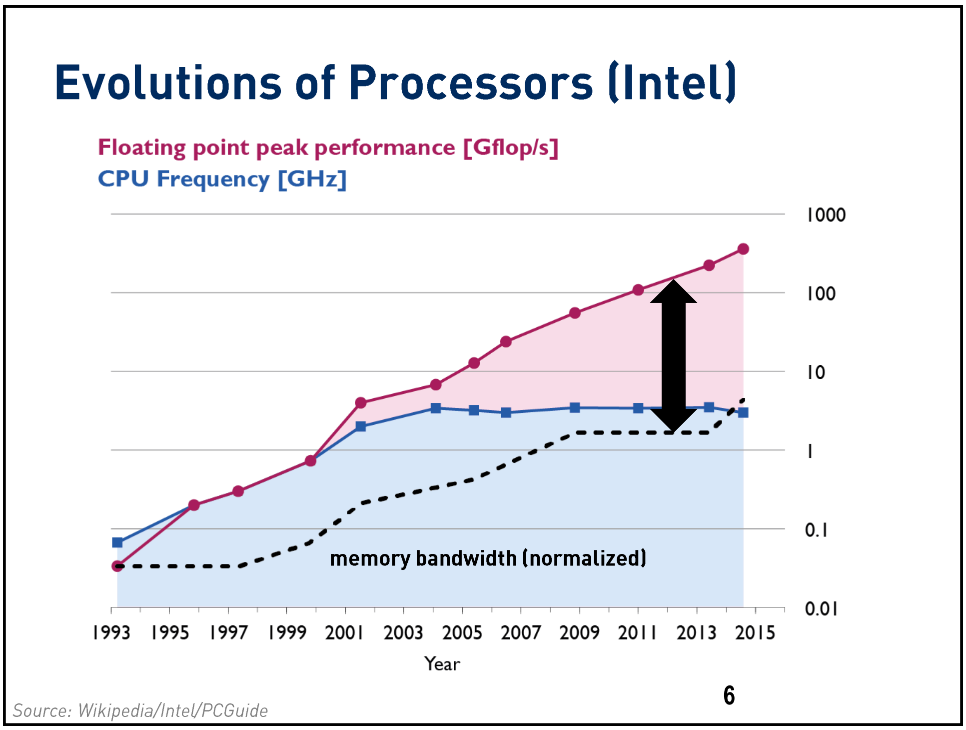


Source: Wikipedia/Intel/PCGuide

4

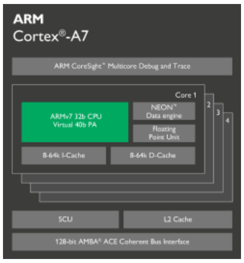


Evolutions of Processors (Intel)
Floating point peak performance [Gflop/s]
CPU Frequency [GHz]
Cores: 8x
Vector units: 8x
~360 Gflop/s
parallelism:
work required
2
2
4
4
4
8 cores
~3 GHz
Pentium 4
Core
Nehalem
Haswell
Sandy Bridge
Pentium III
Pentium II
Pentium Pro
Pentium
free speedup
1000
100
10
1
0.1
0.01
1993 1995 1997 1999 2001 2003 2005 2007 2009 2011 2013 2015
Year
Source: Wikipedia/Intel/PCGuide
5



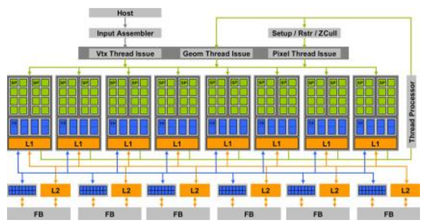
Evolutions of Processors (Intel)
Floating point peak performance [Gflop/s]
CPU Frequency [GHz]
memory bandwidth (normalized)
1000
100
10
1
0.1
0.01
1993 1995 1997 1999 2001 2003 2005 2007 2009 2011 2013 2015
Year
Source: Wikipedia/Intel/PCGuide
6

And there is Processor Variety …



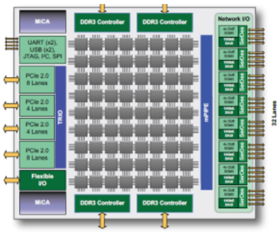
arm.com

Nvidia Tesla



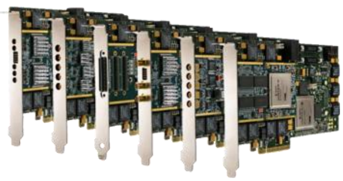
beyond3d.com

Domain-specific (here: Tile)



mellanox.com

FPGA accelerators



nallatech.com

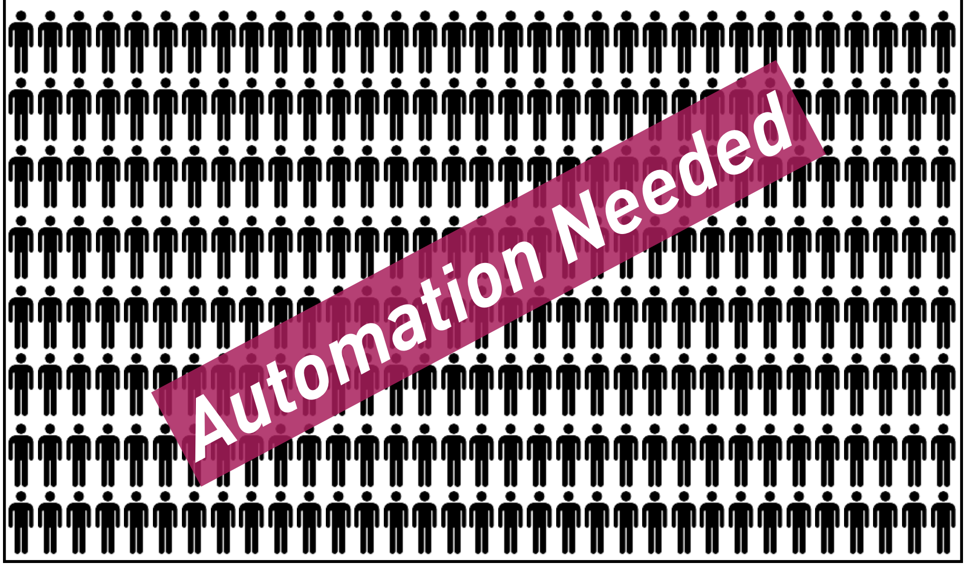
7

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.

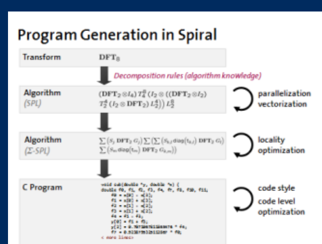


Goal:
Program Generation for Linear Algebra

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*

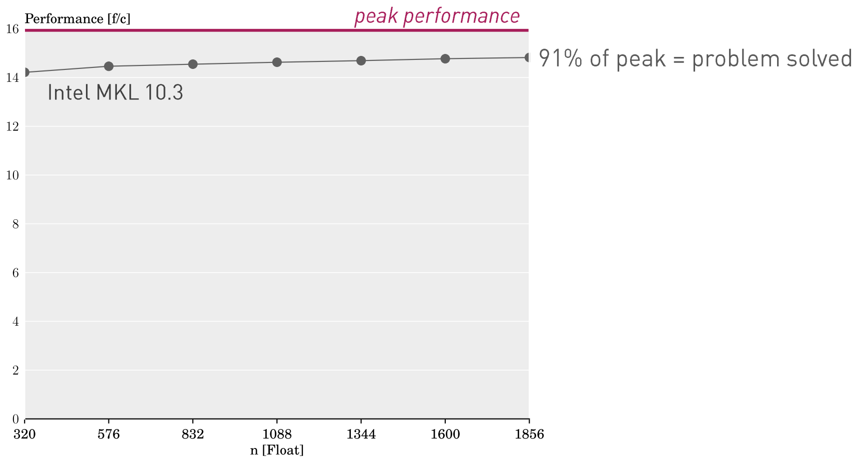


Example: Linear Transforms
www.spiral.net

Start: Basic linear algebra

Library performance sgemm (C += AB)

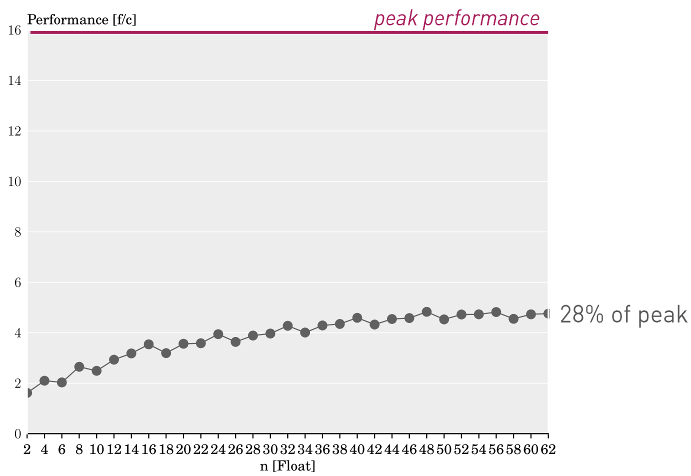
Intel Core i7-2600 CPU @ 3.40GHz



What happens for smaller sizes?

A closer look at small problem sizes

Intel Core i7-2600 CPU @ 3.40GHz



(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

Our Goal: From (any) BLAC to fast code

$$\gamma = x^T(A+B)y + \delta$$ ← **A** is 2x3, **x** is 3x1, …

↓

LGen

↓

```
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;
}
```

Our Goal: From (any) BLAC to fast code

$$\gamma = x^T(A+B)y + \delta$$ ← **A** is 2x3, **x** is 3x1, …

↓

LGen

↓

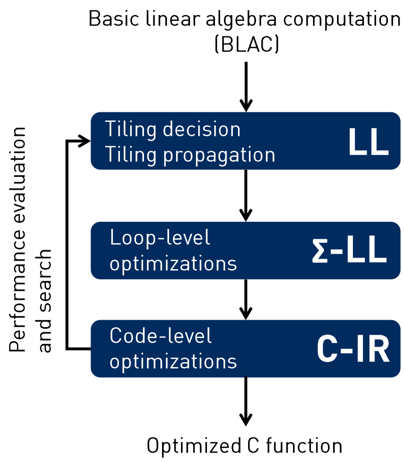
```
void f(double const * A, double const * x, double * y) {
  __m128d t0, …;

  t0 = _mm_loadu_pd(A);
  t1 = _mm_load_sd(A + 2);
  ...
  t6 = _mm_hadd_pd(_mm_mul_pd(t0, t4), _mm_mul_pd(t2, t4));
  t7 = _mm_shuffle_pd(t1, t3, 0);
  t8 = _mm_mul_pd(t7, _mm_shuffle_pd(t5, t5, 0));
  t9 = _mm_add_pd(t6, t8);

  _mm_storeu_pd(y, t9);
}
```

Architecture of LGen

Design similar to Spiral



$$y = Ax$$

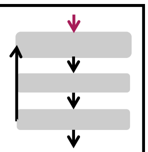
$$[y = Ax]_{2,1}$$

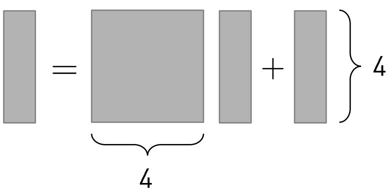
$$y = \sum_{i,j} S_i(G_i \cdots)$$

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

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

Scalar code generation





$$y = Ax + y$$

Tiling in LL

$$[y = Ax + y]_{r,c}$$

Tiling decision for equation

$$r = 2, c = 1$$

Task: *Tiling decision for equation → tiling decision for operands*

Tiling in LL

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

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

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

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

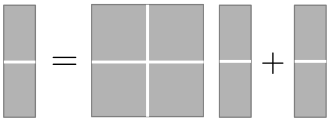
$$[y]_{2,1} = [Ax + y]_{2,1}$$

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

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

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: $G_L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$ $\quad G_R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$

Scatters: $S_L = G_R$ $\quad S_R = G_L$

Extracting a block

$A =$ 

$$B = A(0:1, 0:1) = G_L A G_R$$

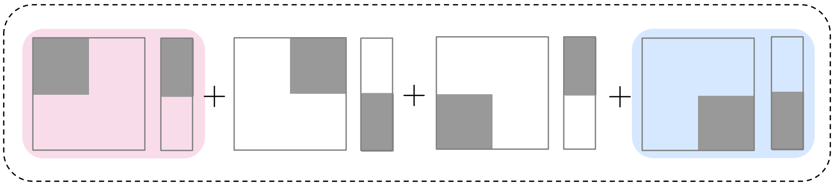
Expanding a block

$$C = \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix} \qquad C = S_L B S_R$$

Gathers *and* ***scatters*** *make data accesses explicit*

LL to Σ-LL

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



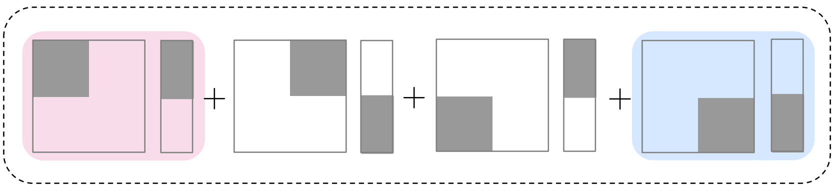
$$S_0\,(G_0 A G_0)\,S_0 \cdot S_0\,(G_0 x) + \cdots + S_2\,(G_2 A G_2)\,S_2 \cdot S_2\,(G_2 x)$$

$$= \sum_{\iota=0,2}^{3} \sum_{j=0,2}^{3} S_\iota\,(G_\iota A G_j)\,(G_j x)$$

$$= \sum_{\iota=0,2}^{3} \sum_{j=0,2}^{3} S_\iota \sum_{\iota'=0}^{1} \sum_{j'=0}^{1} S_{\iota'}\,(G_{\iota'} G_\iota A G_j G_{j'})\,(G_{j'} G_j x)$$

LL to Σ-LL

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



$$S_0\,(G_0 A G_0)\,S_0 \cdot S_0\,(G_0 x) + \cdots + S_2\,(G_2 A G_2)\,S_2 \cdot S_2\,(G_2 x)$$

$$= \sum_{\iota=0,2}^{3} \sum_{j=0,2}^{3} S_\iota\,(G_\iota A G_j)\,(G_j x)$$

$$= \sum_{\iota,j,\iota',j'} S_{\iota+\iota'}\,(G_{\iota+\iota'} A G_{j+j'})\,(G_{j+j'} x)$$

LL to Σ-LL: Loop fusion

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

$$\begin{cases} t = \sum_{\iota,j,\iota',j'} S_{\iota+\iota'}\left(G_{\iota+\iota'} A G_{j+j'}\right)\left(G_{j+j'} x\right) \\ y = \sum_{i,i'} S_{i+i'}\left(G_{i+i'} t + G_{i+i'} y\right) \end{cases}$$

$$G_{i+i'} \sum_{\iota,j,\iota',j'} S_{\iota+\iota'}(\cdots) = \left(G_{i+i'} A G_{j+j'}\right)\left(G_{j+j'} x\right)$$

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

Σ-LL to C-IR

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

```
...
Mov (Mul A[0,0], x[0,0]), t[0,0]
Mov (Mul A[0,1], x[1,0]), t[1,0]
Mov (Mul A[0,2], x[2,0]), t[2,0]
...
```

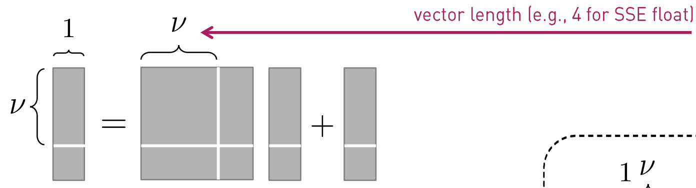
Loop unrolling

Scalar replacement

SSA normalization

Peephole optimizations

Vector code generation: Basic Idea

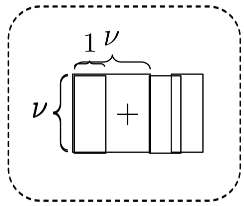


$$[y = Ax + y]_{r,c}$$

$$(r,c) \in \{(1,\nu), (\nu,1), (\nu,\nu)\}$$

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

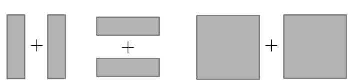
$$y = \sum_{i,j} S_i^{\nu,4}\left[\left(G_i^{\nu,4} A G_j^{\nu,4}\right)\left(G_j^{\nu,4} x\right) + G_i^{\nu,4} y\right]$$



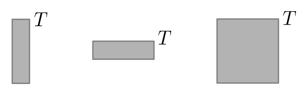
Computation expressed in terms of ν-BLACS

ν-BLACs

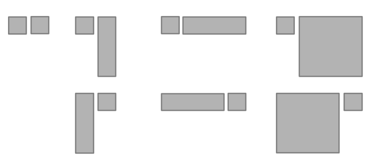
Addition (3 ν-BLACs)



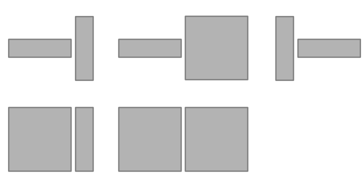
Transposition (3 ν-BLACs)



Scalar Multiplication (7 ν-BLACs)

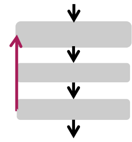


Matrix Multiplication (5 ν-BLACs)

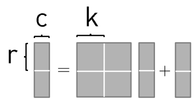


18 cases *implemented once for every vector ISA*

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: `-O3 -xHost -fargument-noalias -fno-alias -ip –ipo`

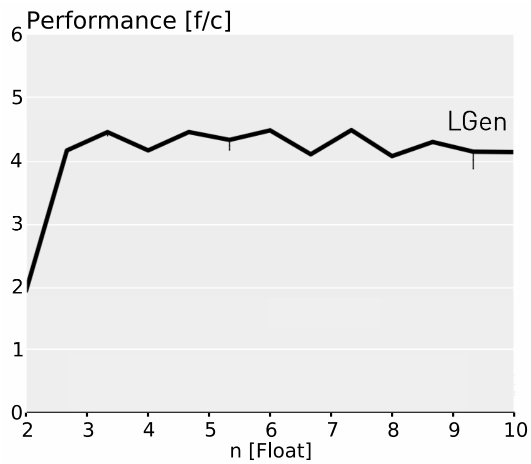
Comparisons

Handwritten naïve code: Fixed and general size

Libraries: Intel MKL v. 11, Intel IPP v. 7.1

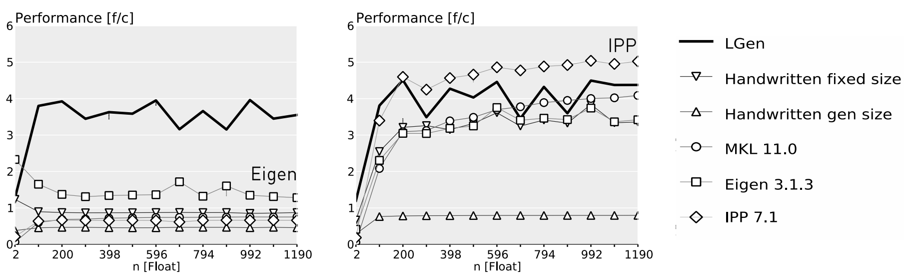
Generators: Eigen v.3.1.3, BTO v.1.3

Plotting



Case 1: Simple BLACs

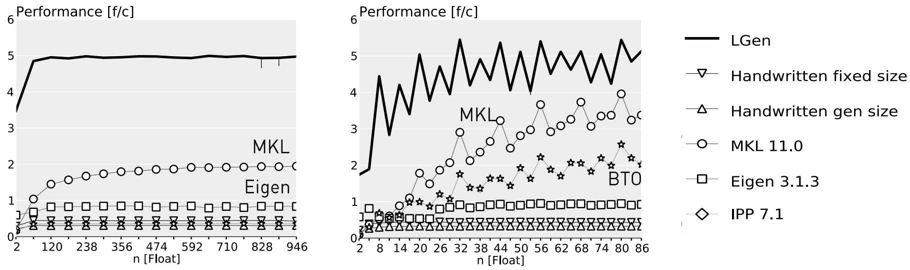
$$y = Ax$$



$A \in \mathbb{R}^{n\times 4}$ $A \in \mathbb{R}^{4\times n}$

Case 2: BLACs closely matching BLAS

$$C = \alpha AB + \beta C$$

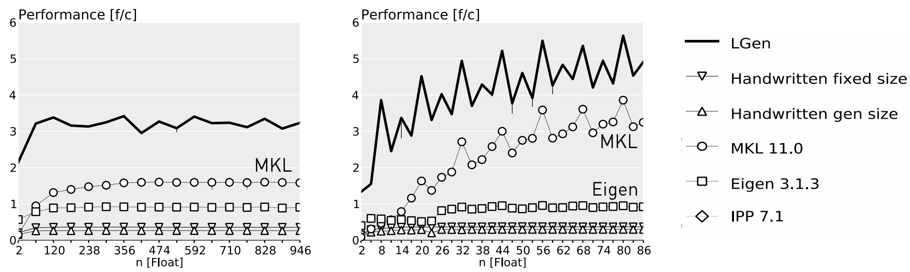


$A \in \mathbb{R}^{n \times 4}$ $\quad$ $A \in \mathbb{R}^{n \times 4}$

$B \in \mathbb{R}^{4 \times 4}$ $\quad$ $B \in \mathbb{R}^{4 \times n}$

Case 3: More than one BLAS call

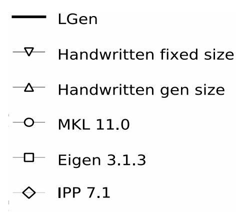
$$C = \alpha (A_0 + A_1)^T B + \beta C$$

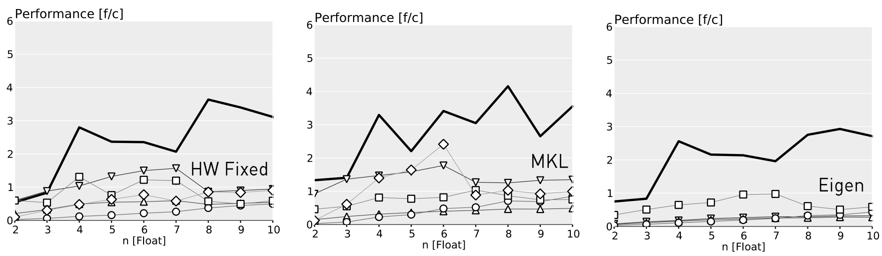


$A_0 \in \mathbb{R}^{4 \times n}$ $\quad$ $A_0 \in \mathbb{R}^{4 \times n}$

$B \in \mathbb{R}^{4 \times 4}$ $\quad$ $B \in \mathbb{R}^{4 \times n}$

Case 4: Micro BLACs



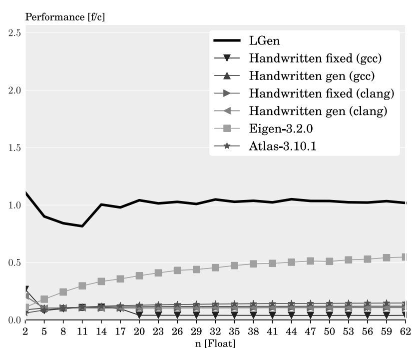


$y = Ax$ $\qquad C = AB$ $\qquad \alpha = x^T Ay$

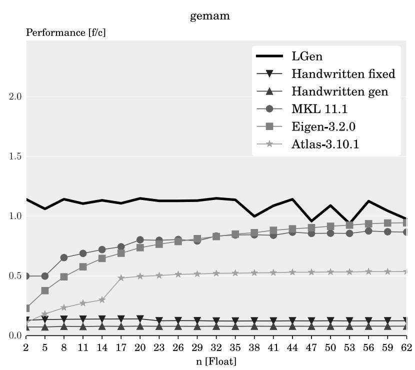
On Embedded Processors

Work by Nikos Kyrtatas

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



ARM Cortex-A8 @ 1 GHz



Intel Atom D2550 @ 1.86 GHz

Challenge: Alignment Analysis

unaligned load/stores only

```
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);

      // 4-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_store_ps(C + 115*i2 + 115*ii4 + j3 + 345, t0_11_1);
    }
  }
}
```

goal

with aligned load/stores

```
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_load_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_load_ps(B + 115*i2 + 115*ii4 + j3 + 230);
      t0_0_0 = _mm_loadu_ps(B + 115*i2 + 115*ii4 + j3 + 345);

      // 4-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_store_ps(C + 115*i2 + 115*ii4 + j3 + 345, t0_11_1);
    }
  }
}
```

Solution: Abstract interpretation

assume B[] is aligned:

interval congruence

```
                                               // B->([-oo,+oo], 0+4Z)
for( size_t j5 = 0; j5 < 80; j5+=16 ) {        // j5->([0,64], 0+16Z)
  ...
  for( size_t k4 = 8; k4 < 48; k4+=8 ) {       // k4->([8,40], 0+8Z)
    for( size_t kk7 = 0; kk7 < 8; kk7+=4 ) {   // kk7->([0,4], 0+4Z)
      for( size_t jj8 = 0; jj8 < 16; jj8+=4 ) {   // jj8->([0,12], 0+4Z)
        ...
        t0_19_0 = _mm_load_ps(B + j5 + jj8 + 81*k4 + 81*kk7);
// Eval(B + j5 + jj8 + 81*k4 + 81*kk7) = ([-oo,+oo], 0+4Z) + ([0,64], 0+16Z) +
// ([0,12], 0+4Z) + ([81,81], 81+0Z) * ([8,40], 0+8Z) + ([81,81], 81+0Z) * ([0,4], 0+4Z) =
// = ([-oo,+oo], 0+gcd(4,16,4,648,324)Z) = ([-oo,+oo], 0+4Z)
```

aligned

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**

Diego Fabregat

Paolo Bientinesi

RWTH Aachen

Kalman Filter

Predict

$x_k = Ax_{k-1} + Bu_k$

$P_k = AP_{k-1}A^T + Q$

Update

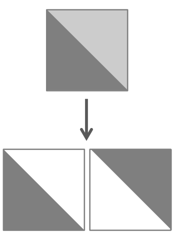
$x_k = x_k + P_k H^T (HP_k H^T + R)^{-1}(z_k - Hx_k)$

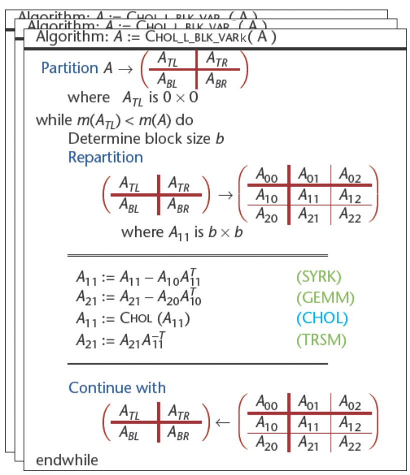
$P_k = P_k - P_k H^T (HP_k H^T + R)^{-1} HP_k$

→ fast code

Cl1ck: Synthesis of Linear Algebra Algorithms

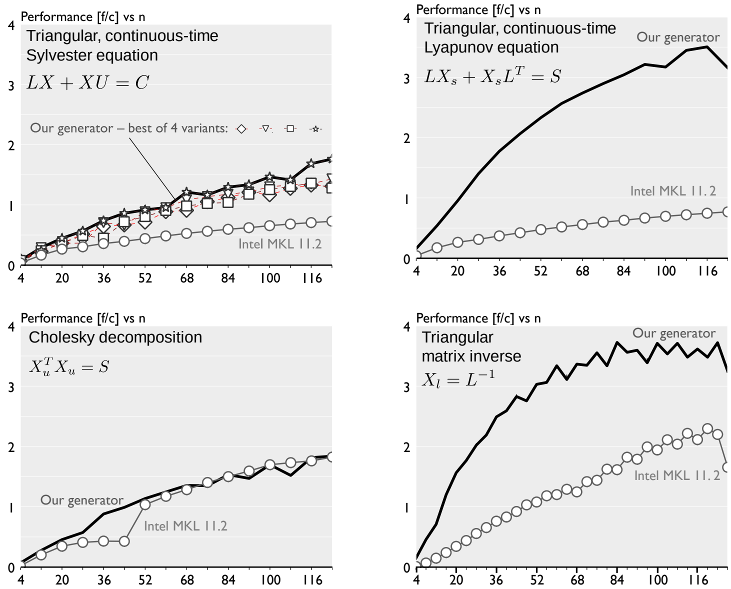
Cholesky factorization





More on Cl1ck
- D. Fabregat-Traver and P. Bientinesi. Knowledge-Based Automatic Generation of Partitioned Matrix Expressions. CASC, 2011.
- D. Fabregat-Traver and P. Bientinesi. Automatic Generation of Loop-invariants for Matrix Operations. ICCSA, 2011.

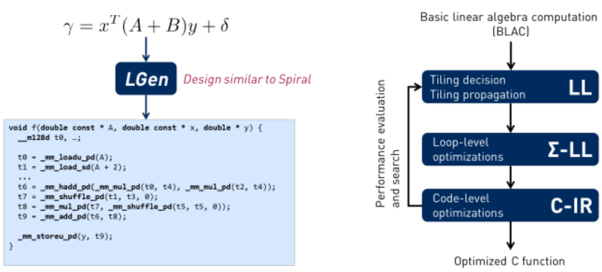
Preliminary results

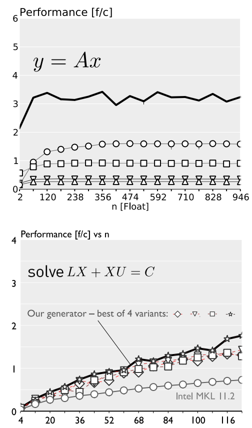


bug (2x too good)

Summary

Goal: Automatically from linear algebra to fast code





Soon …

Kalman Filter

Predict

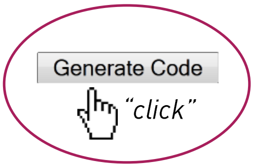
$x_k = Ax_{k-1} + Bu_k$

$P_k = AP_{k-1}A^T + Q$

Update

$x_k = x_k + P_kH^T(HP_kH^T + R)^{-1}(z_k - Hx_k)$

$P_k = P_k - P_kH^T(HP_kH^T + R)^{-1}HP_k$



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

More info: http://spiral.net/software/lgen.html