Programming for Performance

2013, Lecture 1

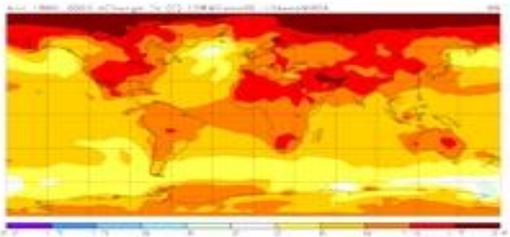
Sri Sathya Sai Institute of Higher Learning

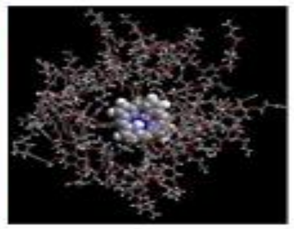
Picture: www.tapety-na-pulpit.org

Today

- Motivation for this course
- Organization of this course

Scientific Computing



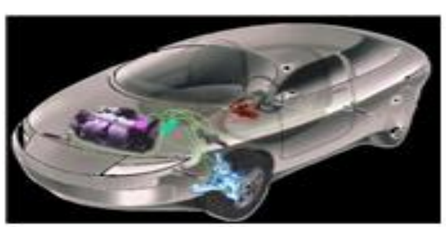


Physics/biology simulations

Consumer Computing

Audio/image/video processing

Embedded Computing



Signal processing, communication, control

Computing

- **Unlimited need for performance**
- **Large set of applications, but …**
- **Relatively small set of critical components (100s to 1000s)**
  - Matrix multiplication
  - Discrete Fourier transform (DFT)
  - Viterbi decoder
  - Shortest path computation
  - Stencils
  - Solving linear system
  - ….

Consumer Computing (Desktop, Phone, …)

Photo/video processing

Audio coding

Security

Original **JPEG** **JPEG2000**

Image compression

Methods:

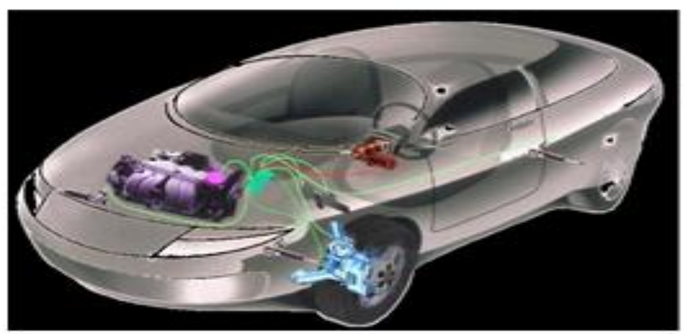
- Linear algebra
- Transforms
- Filters
- Others

Embedded Computing (Low-Power Processors)

www.dei.unipd.it

Sensor networks

www.ece.drexel.edu



Cars

www.microway.com.au

Robotics

Computation needed:

- Signal processing
- Control
- Communication

Methods:

- Linear algebra
- Transforms, Filters
- Coding

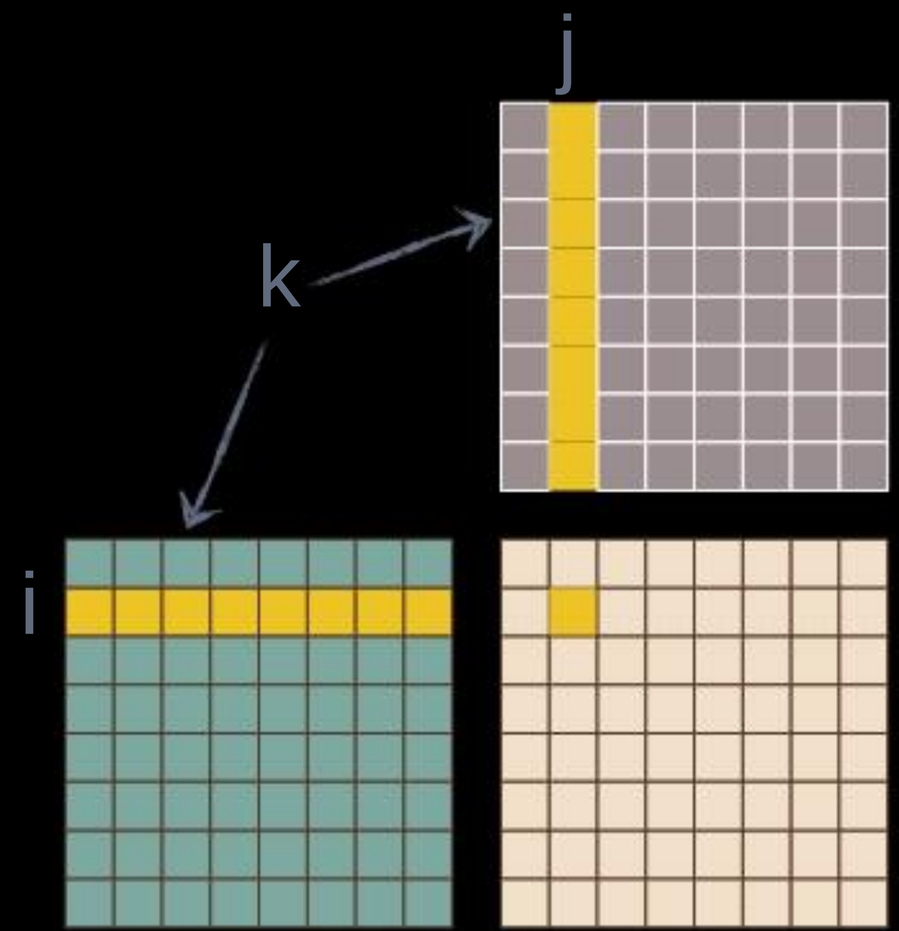
Classes of Performance-Critical Functions

- **Transforms**
- **Filters/correlation/convolution/stencils/interpolators**
- **Dense linear algebra functions**
- **Sparse linear algebra functions**
- **Coder/decoders**
- **Graph algorithms**
- ***… several others***

See also the 13 dwarfs/motifs in
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

An Example: Matrix Multiplication

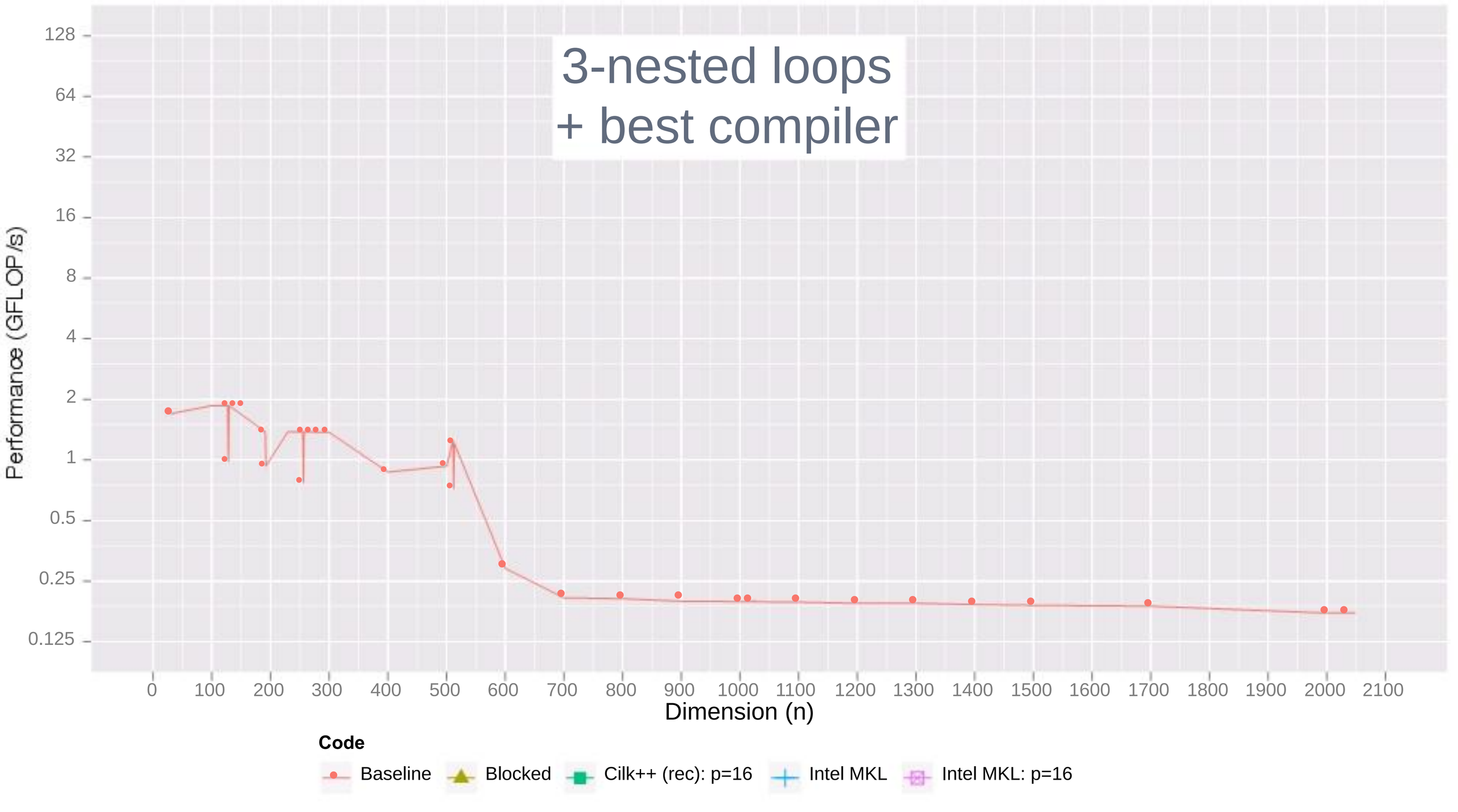
Example: **Matrix multiply** (non-Strassen)

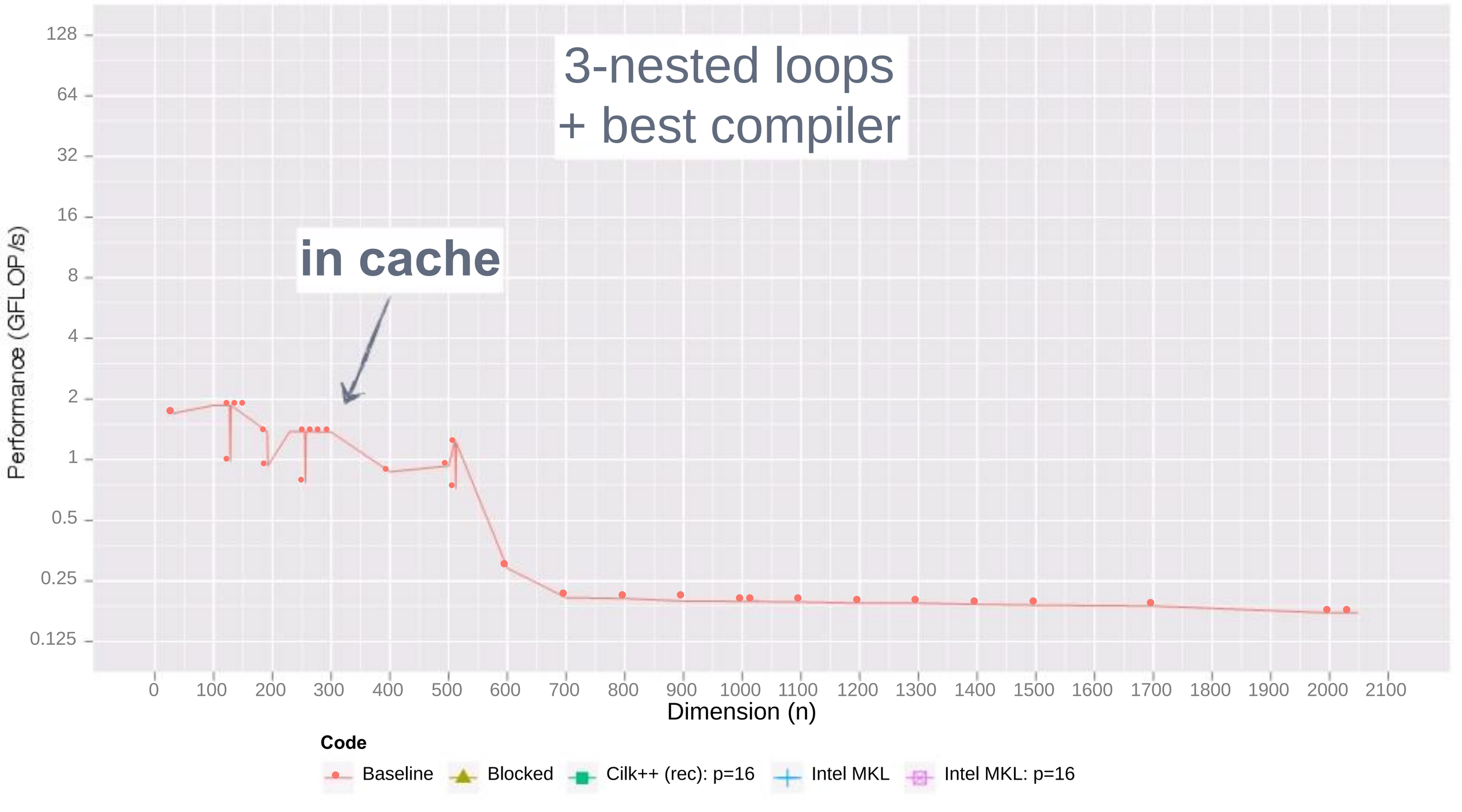


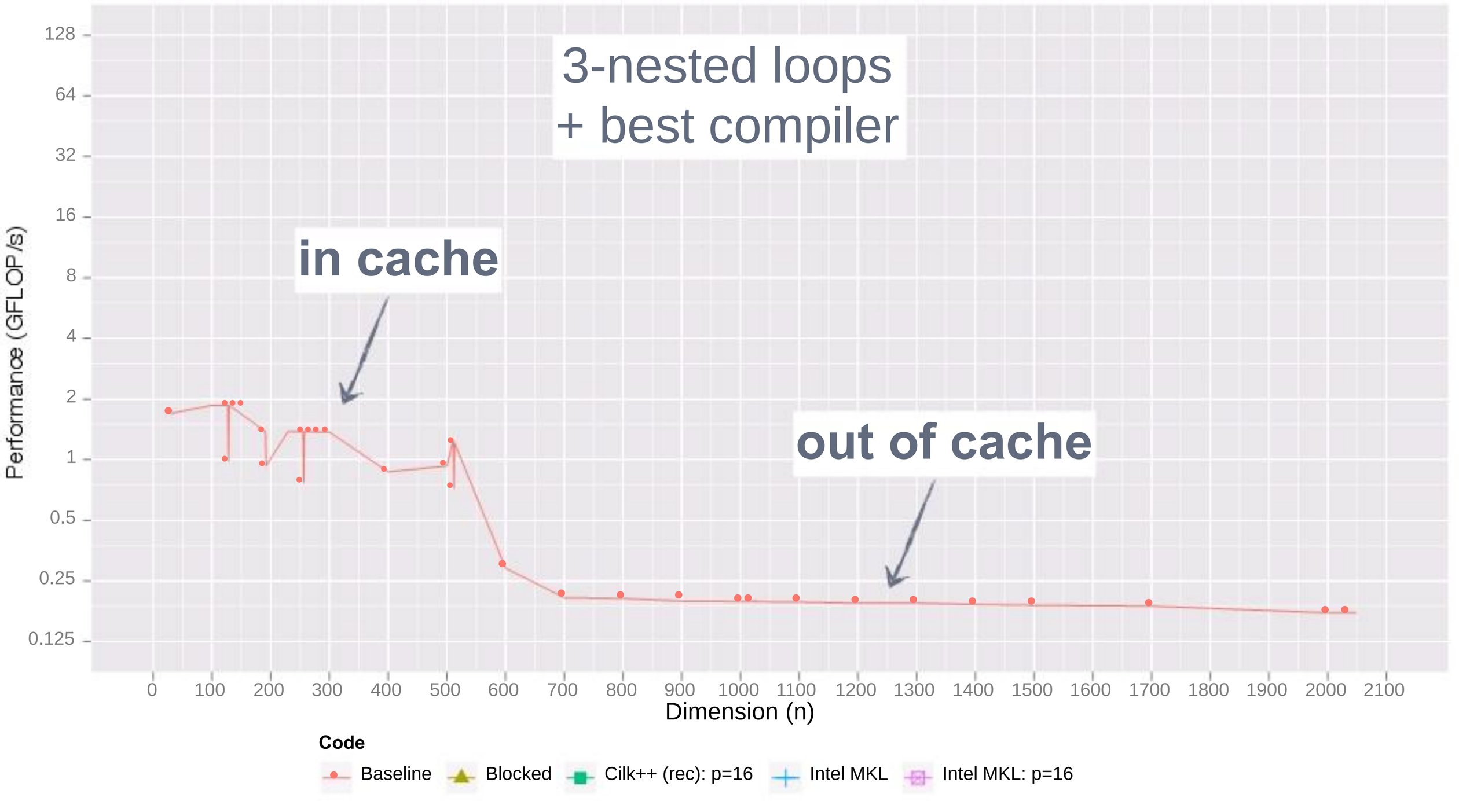

```
for i = 1 to n do
  for   j = 1    to  n     do
    for k = 1   to  n  do
```

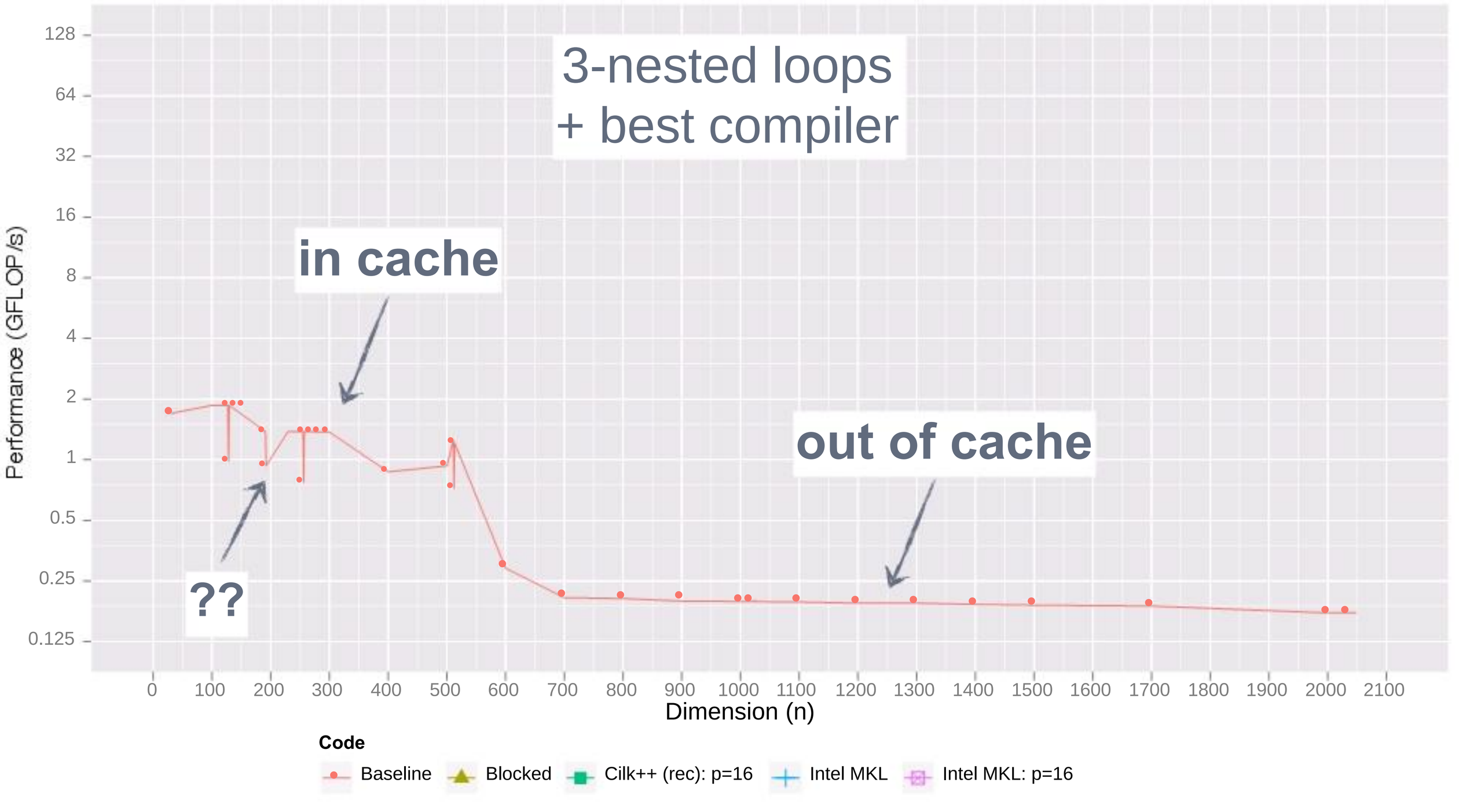
C[i,j] ← **C**[i,j] + **A**[i,k] · **B**[k,j]

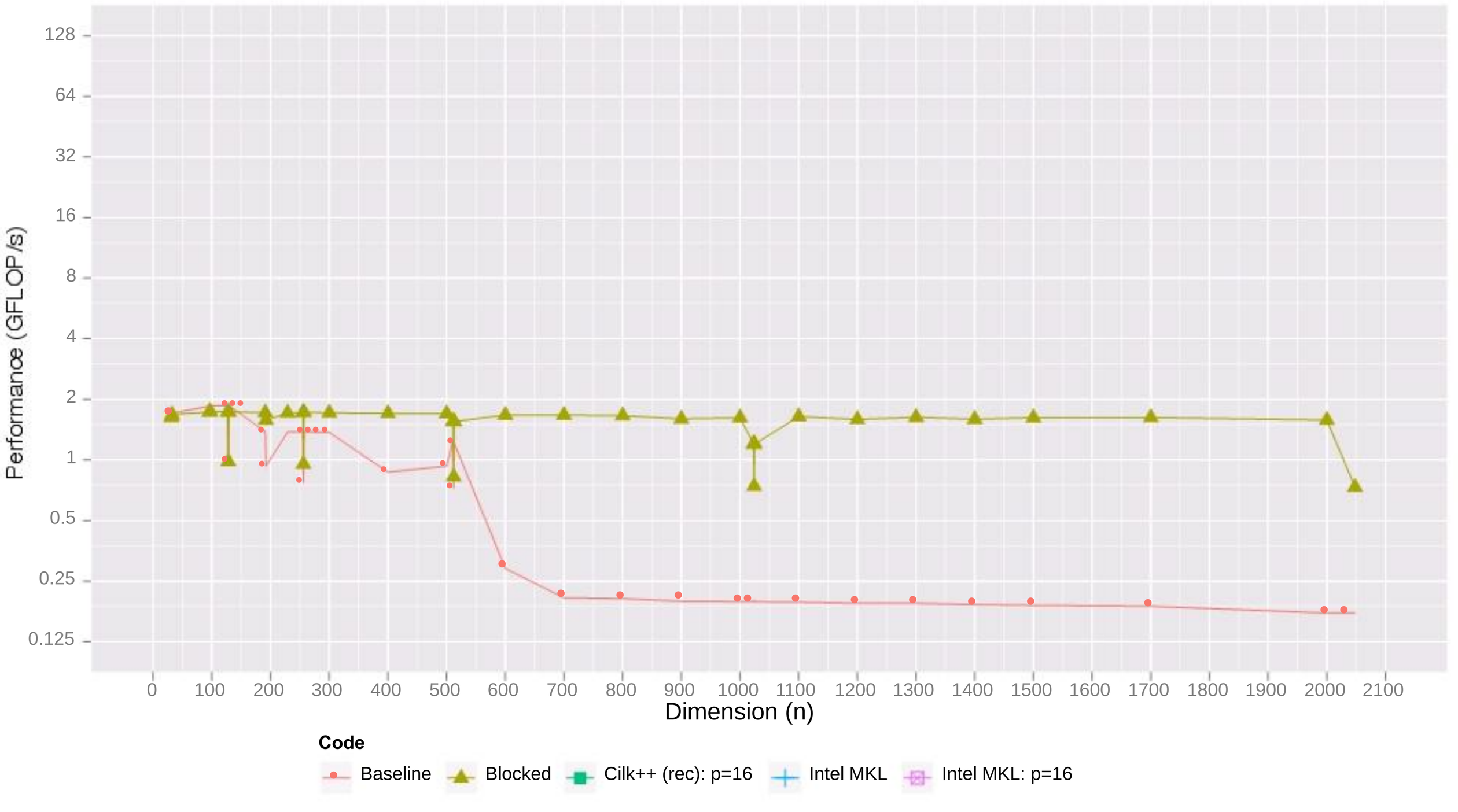
C C + A * B!

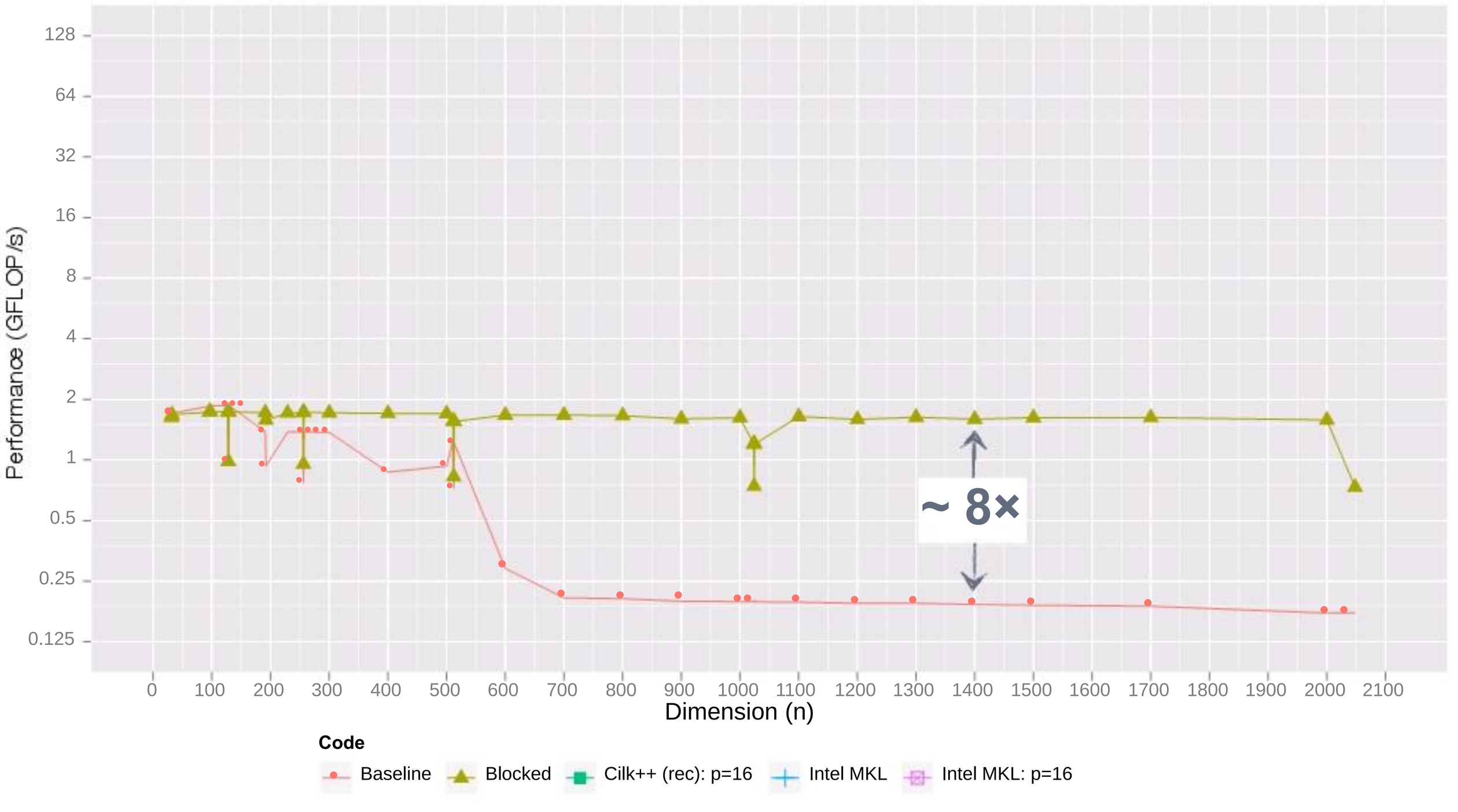

3-nested loops
+ best compiler
Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16

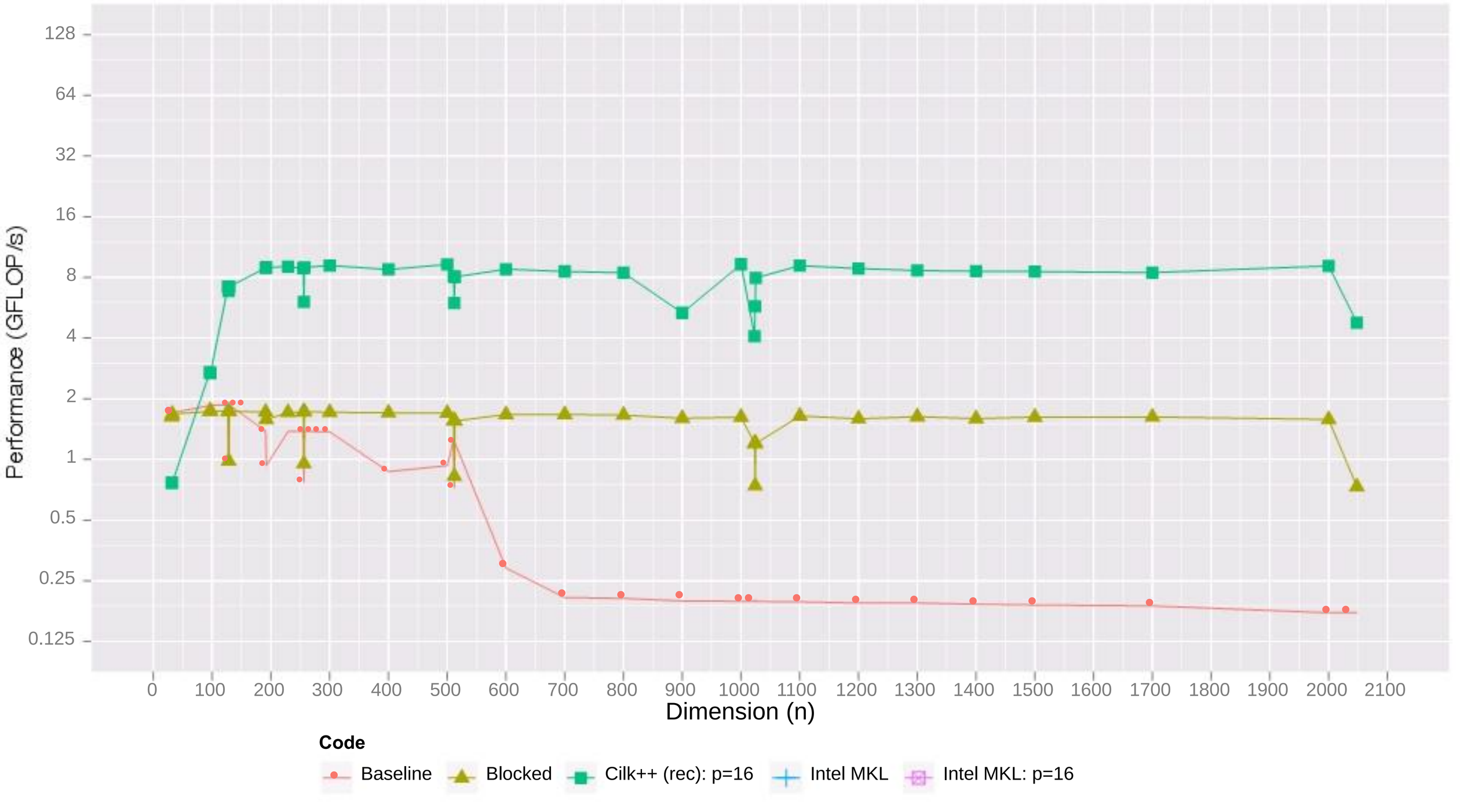

3-nested loops
+ best compiler
in cache
Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16

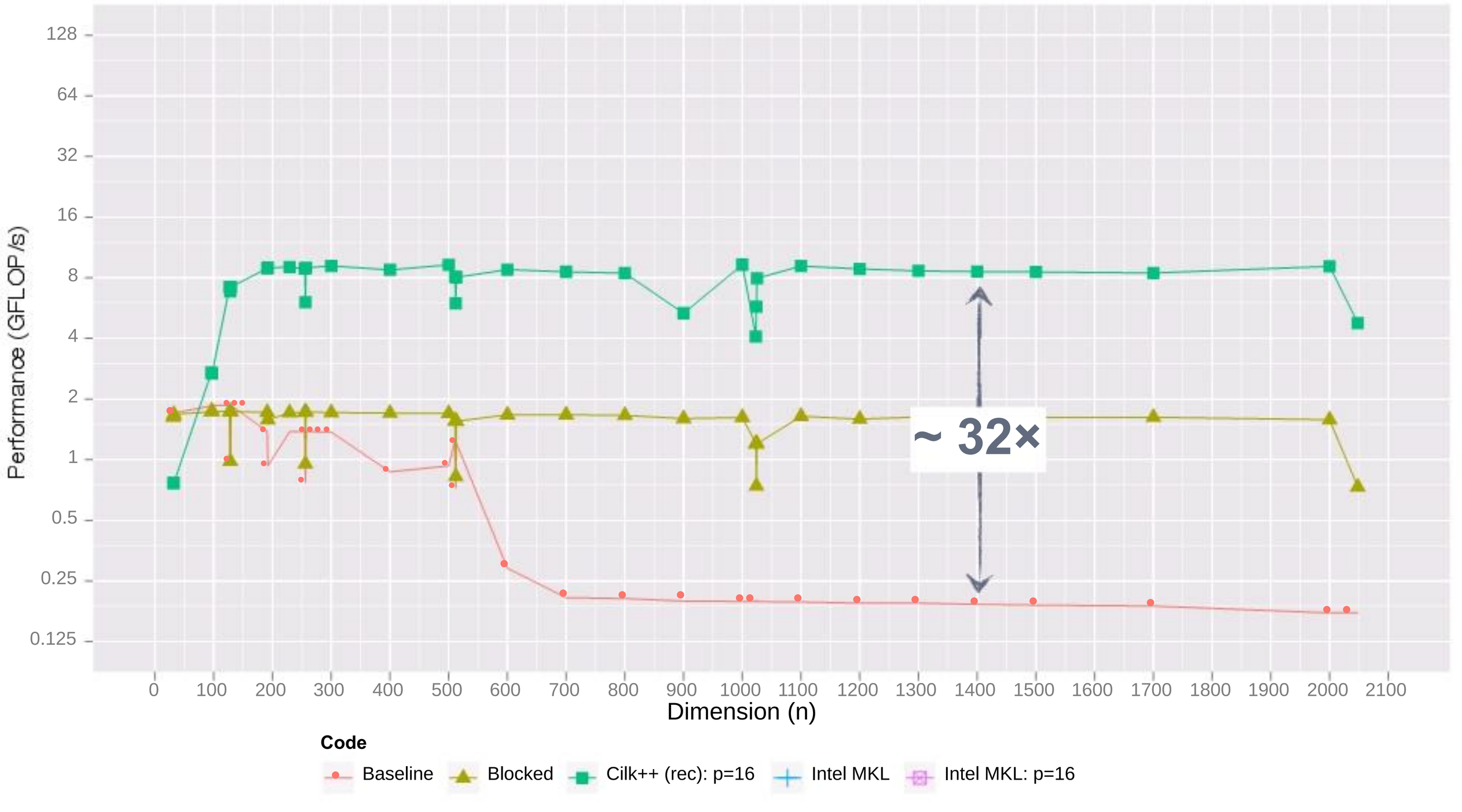

3-nested loops
+ best compiler
in cache
out of cache
Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16

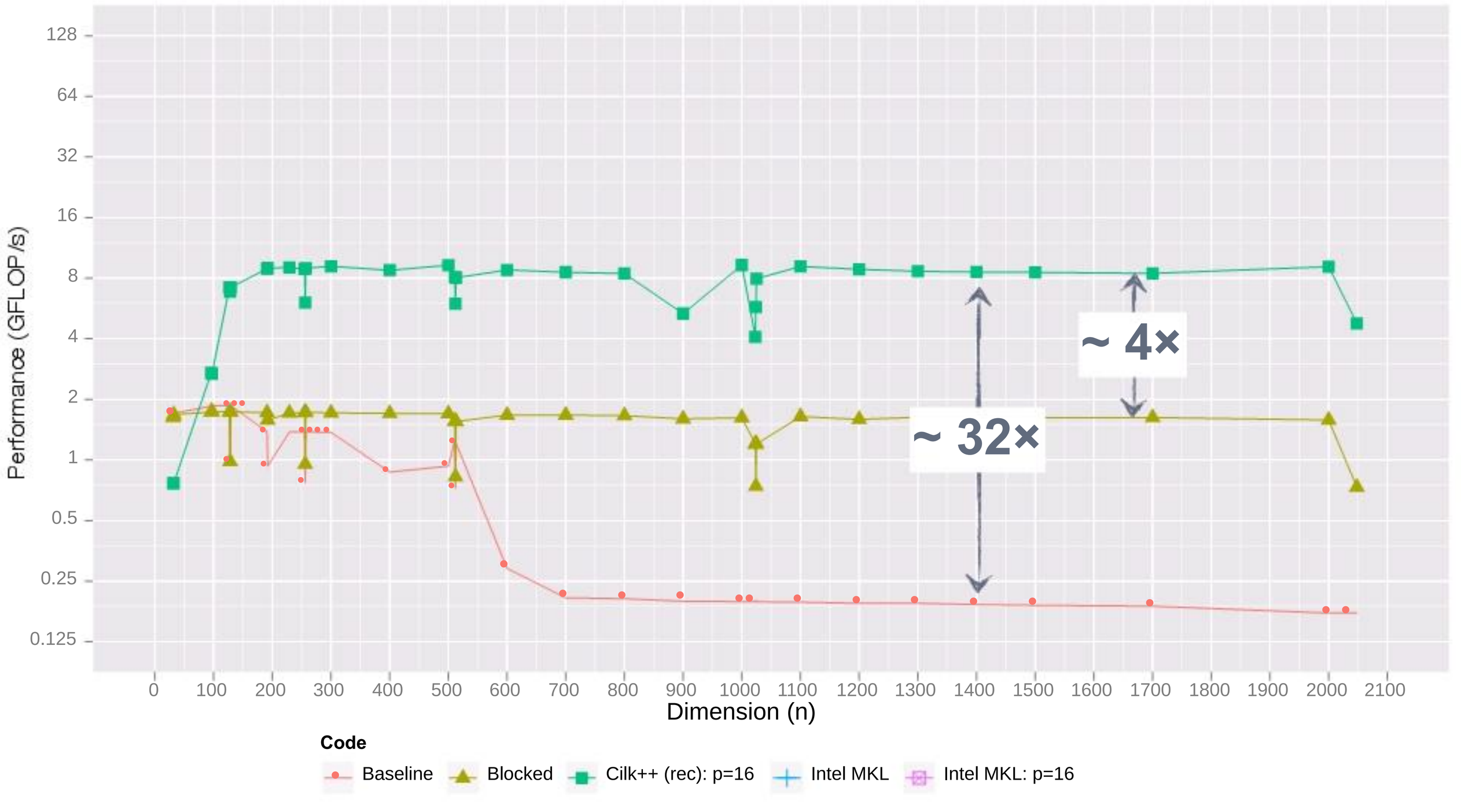

3-nested loops
+ best compiler
in cache
out of cache
??
Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16

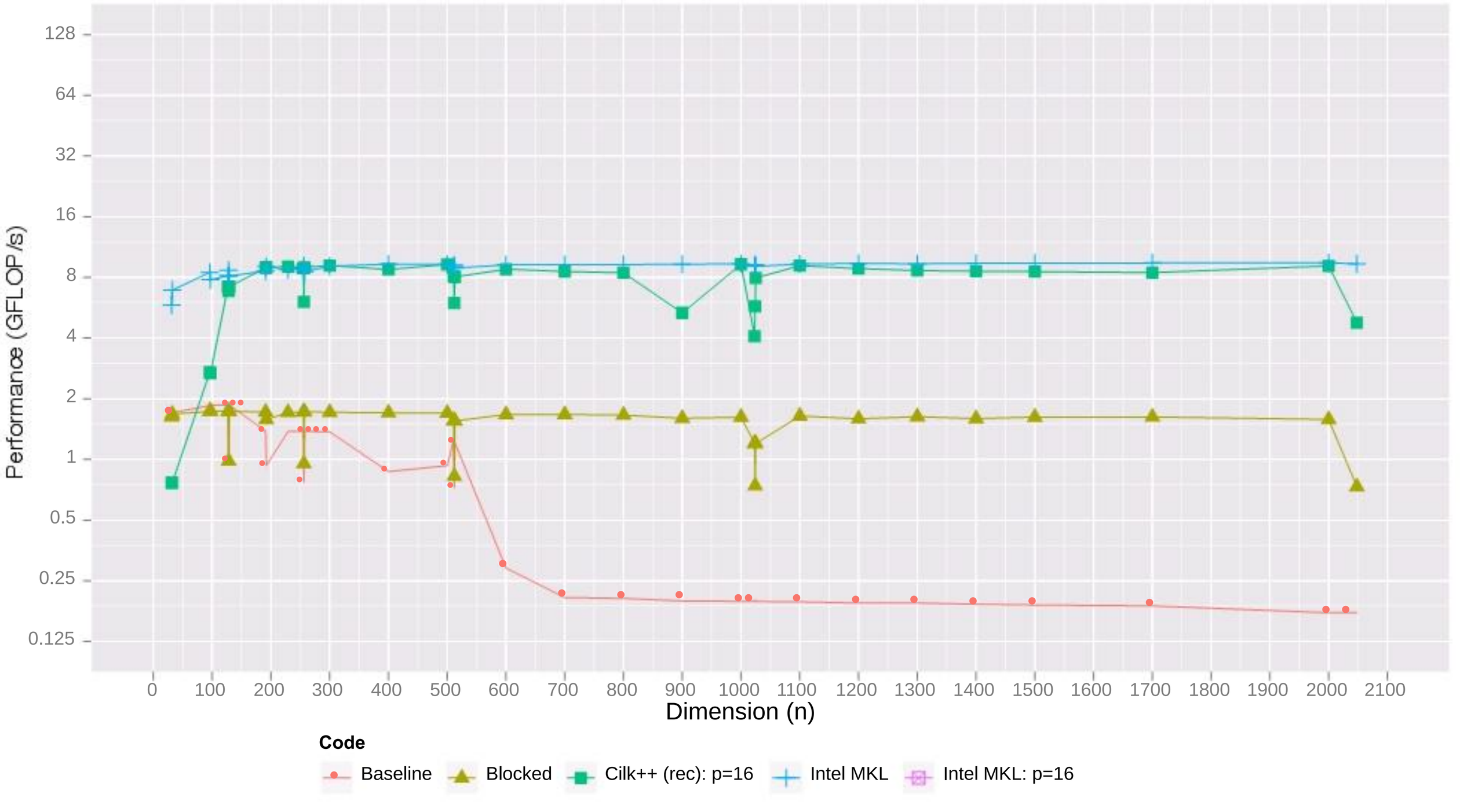

Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16

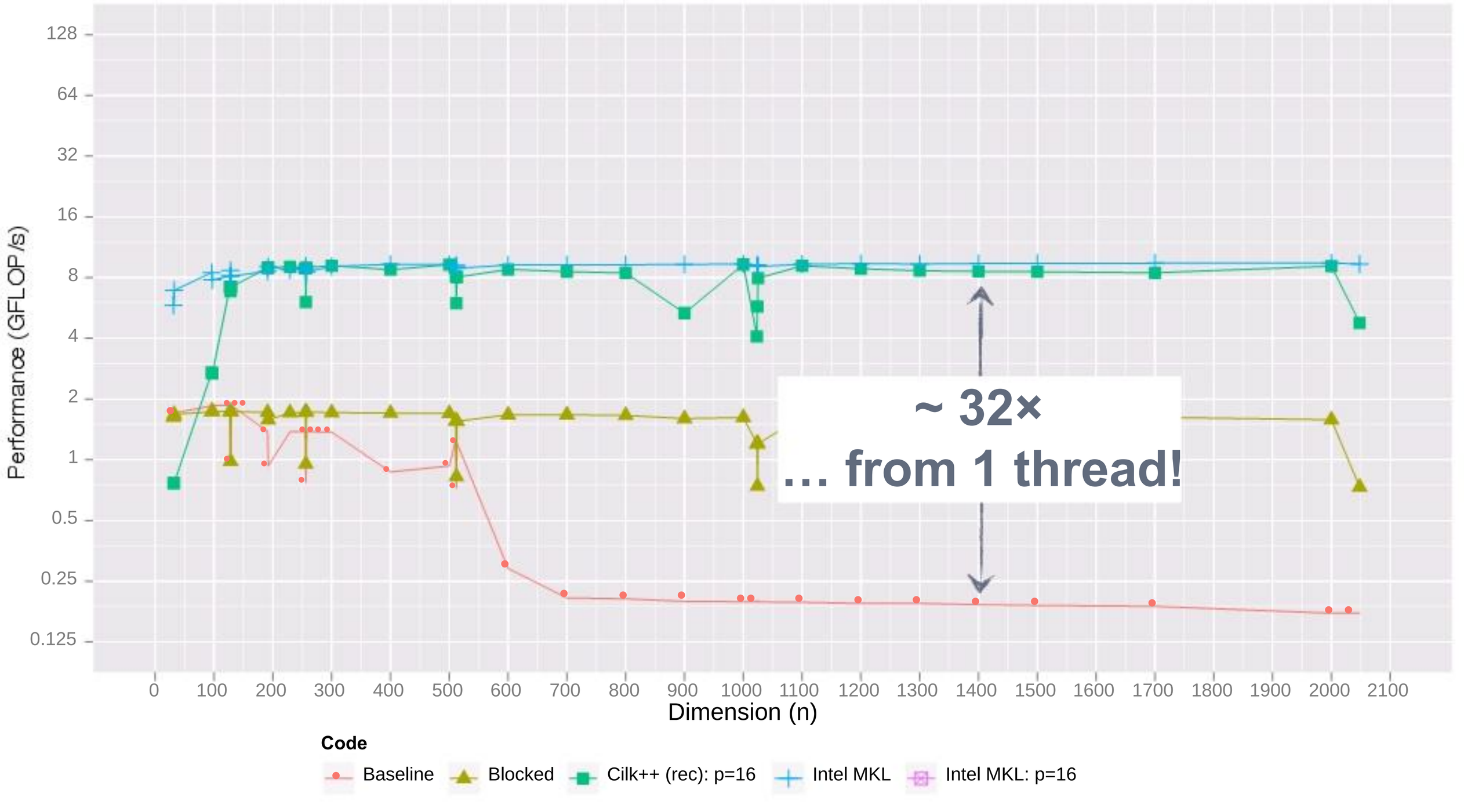

Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
~ 8×
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16

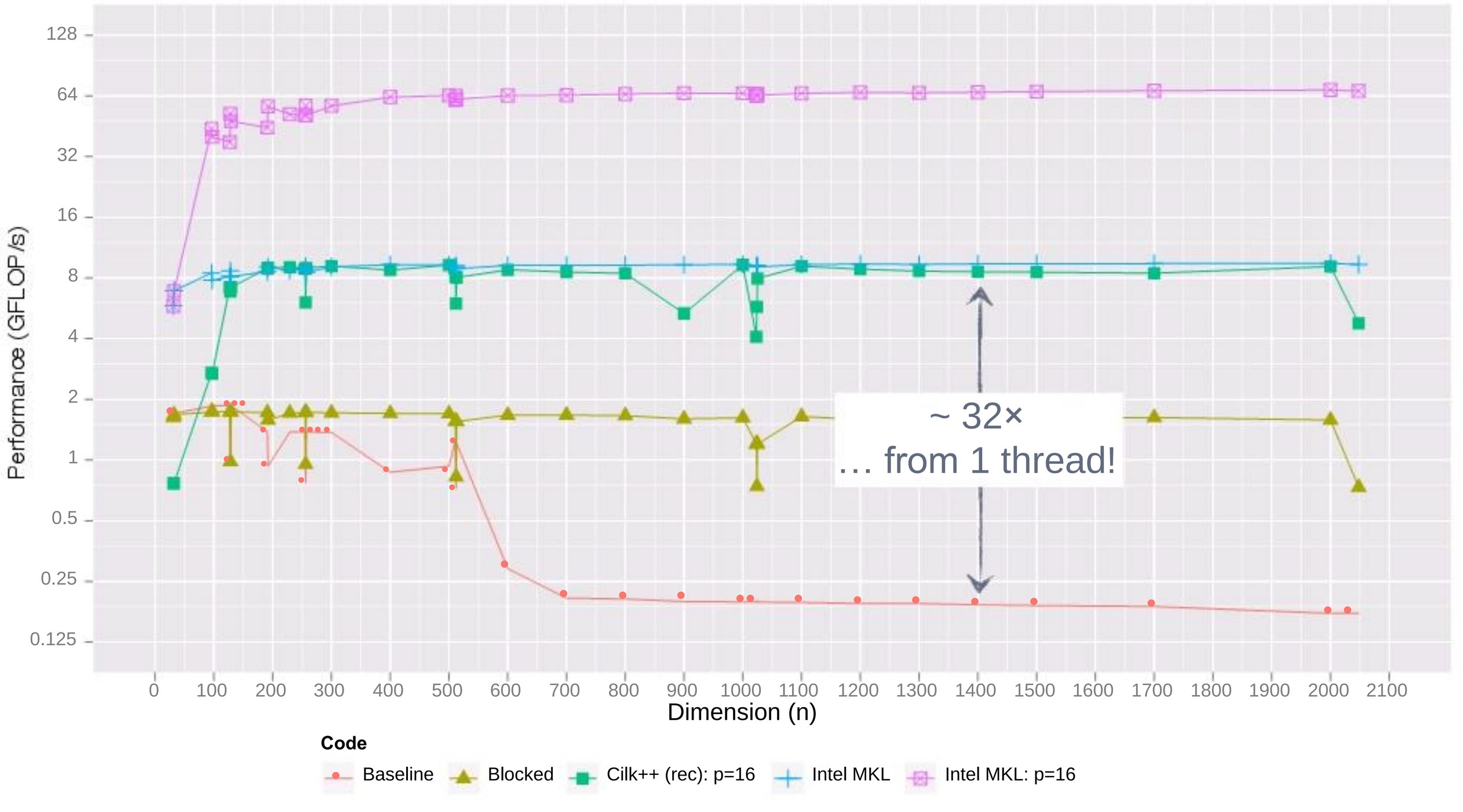

Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16

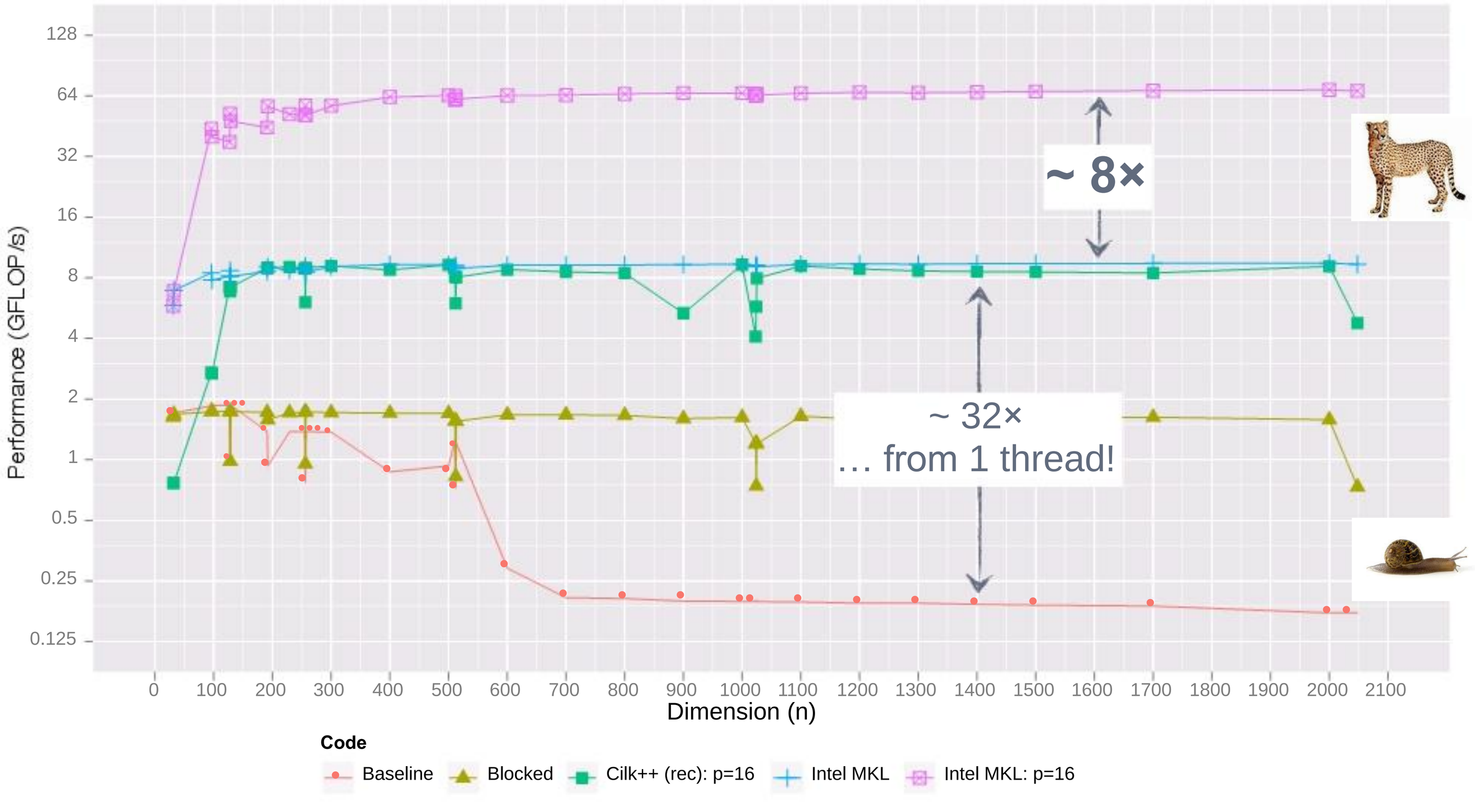

Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
~ 32×
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16


Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
~ 4×
~ 32×
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16


Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16


Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
~ 32×
… from 1 thread!
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16


Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
~ 32×
… from 1 thread!
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16


Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
~ 8×
~ 32×
… from 1 thread!
Code
Baseline
Blocked
Cilk++ (rec): p=16
Intel MKL
Intel MKL: p=16

Large, Complex Tuning Spaces

:: 78% of peak!

```
/** M. Dukhan and M. Schornak: CSE 6230 Project 1D, Fall 2011 **/

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <malloc.h>

#include "matmult.h"

#define CACHE_LINE 64

//  Kc   Mc   GFLOPS
// 256640     98.2
// 256612     99.6
// 256576     97.2
// 240640     98.9

#define K_C 256
#define M_C 612
#define N_R 4
#define M_R 4

#define MIN(a, b) (a < b) ? a : b
#include <pmmintrin.h>

#define USE_ASM

#ifndef USE_ASM
void matmult_dgedot4x4(double *  C, const double *  A, const double *  B, long stride, long kc_size) {
    register long i;
    register const double * B0 = B;
    register const double * B1 = B + stride;
    register const double * B2 = B + stride * 2;
    register const double * B3 = B + stride * 3;
    register __m128d c0 = _mm_load_pd(C);
    register __m128d c1 = _mm_load_pd(C + 2);
    register __m128d c2 = _mm_load_pd(C + stride);
    register __m128d c3 = _mm_load_pd(C + stride + 2);
    register __m128d c4 = _mm_load_pd(C + stride * 2);
    register __m128d c5 = _mm_load_pd(C + stride * 2 + 2);
    register __m128d c6 = _mm_load_pd(C + stride * 3);
    register __m128d c7 = _mm_load_pd(C + stride * 3 + 2);
    for (i = 0; i < kc_size; i++) {
        register __m128d a0, a1;
        register __m128d b0, b1, b2, b3;
        register __m128d b0_tmp, b1_tmp, b2_tmp, b3_tmp;

        a0 = _mm_load_pd(A + i * 4);
        a1 = _mm_load_pd(A + i * 4 + 2);

        b0 = _mm_loaddup_pd(B0 + i);
        b1 = _mm_loaddup_pd(B1 + i);
        b2 = _mm_loaddup_pd(B2 + i);
        b3 = _mm_loaddup_pd(B3 + i);

        c0 = _mm_add_pd(c0,  _mm_mul_pd(b0, a0));
        c1 = _mm_add_pd(c1,  _mm_mul_pd(b0, a1));
        c2 = _mm_add_pd(c2,  _mm_mul_pd(b1, a0));
        c3 = _mm_add_pd(c3,  _mm_mul_pd(b1, a1));
        c4 = _mm_add_pd(c4,  _mm_mul_pd(b2, a0));
        c5 = _mm_add_pd(c5,  _mm_mul_pd(b2, a1));
        c6 = _mm_add_pd(c6,  _mm_mul_pd(b3, a0));
        c7 = _mm_add_pd(c7,  _mm_mul_pd(b3, a1));
    }
    _mm_store_pd(C, c0);
    _mm_store_pd(C + 2, c1);
    _mm_store_pd(C + stride, c2);
    _mm_store_pd(C + stride + 2, c3);
    _mm_store_pd(C + stride * 2, c4);
    _mm_store_pd(C + stride * 2 + 2, c5);
    _mm_store_pd(C + stride * 3, c6);
    _mm_store_pd(C + stride * 3 + 2, c7);
}
#else
extern void matmult_dgedot4x4_asm(double *  C, const double *  A, const double * B, long stride, long kc_size);
#endif

#ifndef USE_ASM
// Input:
//  Matrix C of size 2 x 2
//  Matrix A of size 2 x Kc
//  Matrix B of size Kc x 2
void matmult_dgedot2x2(double * C, const double * A, const double * B, int stride, int kc_size) {
    register int n = kc_size;
    register const double * B0 = B;
    register const double * B1 = B + stride;
    // c0 and c1 have dependency chain due to addition
    // ADDPD latency on Nehalem is 5 clocks, so unrolling at least by 3 is required to hide latency
    register __m128d c0 = _mm_load_pd(C);
    register __m128d c1 = _mm_load_pd(C + stride);
    register __m128d c0_tmp0 = _mm_setzero_pd();
    register __m128d c1_tmp0 = _mm_setzero_pd();
    register __m128d c0_tmp1 = _mm_setzero_pd();
    register __m128d c1_tmp1 = _mm_setzero_pd();
    register __m128d c0_tmp2 = _mm_setzero_pd();
    register __m128d c1_tmp2 = _mm_setzero_pd();
    while (n > 0) {
        register __m128d a, b0, b1;

        a = _mm_load_pd(A);
        b0 = _mm_loaddup_pd(B0);
        b1 = _mm_loaddup_pd(B1);
        c0 = _mm_add_pd(c0,  _mm_mul_pd(b0, a));
        c1 = _mm_add_pd(c1,  _mm_mul_pd(b1, a));

        a = _mm_load_pd(A + 2);
        b0 = _mm_loaddup_pd(B0 + 1);
        b1 = _mm_loaddup_pd(B1 + 1);
        c0_tmp0 = _mm_add_pd(c0_tmp0,  _mm_mul_pd(b0, a));
        c1_tmp0 = _mm_add_pd(c1_tmp0,  _mm_mul_pd(b1, a));

        a = _mm_load_pd(A + 4);
        b0 = _mm_loaddup_pd(B0 + 2);
        b1 = _mm_loaddup_pd(B1 + 2);
        c0_tmp1 = _mm_add_pd(c0_tmp1,  _mm_mul_pd(b0, a));
        c1_tmp1 = _mm_add_pd(c1_tmp1,  _mm_mul_pd(b1, a));

        a = _mm_load_pd(A + 6);
        b0 = _mm_loaddup_pd(B0 + 3);
        b1 = _mm_loaddup_pd(B1 + 3);
        c0_tmp2 = _mm_add_pd(c0_tmp2,  _mm_mul_pd(b0, a));
        c1_tmp2 = _mm_add_pd(c1_tmp2,  _mm_mul_pd(b1, a));

        A += 8;
        B0 += 4;
        B1 += 4;
        n -= 4;
    }
    c0 = _mm_add_pd(_mm_add_pd(c0, c0_tmp0), _mm_add_pd(c0_tmp1, c0_tmp2));
    c1 = _mm_add_pd(_mm_add_pd(c1, c1_tmp0), _mm_add_pd(c1_tmp1, c1_tmp2));
    while (n > 0) {
        register __m128d a = _mm_load_pd(A);
        register __m128d b0 = _mm_loaddup_pd(B0);
        register __m128d b1 = _mm_loaddup_pd(B1);
        c0 = _mm_add_pd(c0,  _mm_mul_pd(b0, a));
        c1 = _mm_add_pd(c1,  _mm_mul_pd(b1, a));
        A += 2;
        B0 += 1;
        B1 += 1;
        n -= 1;
    }
    _mm_store_pd(C, c0);
    _mm_store_pd(C + stride, c1);
}
#endif

// Input:
//  Matrix C of size Mr x Nr
//  Matrix A of size Mr x Kc stored with stride mr_size
//  Matrix B of size Kc x Nr stored with stride kc_size
void matmult_dgedot(double * C, const double *  A, const double *  B, long stride, long kc_size, long nr_size, long
mr_size) {
    long i, j, k;
    for (j = 0; j < nr_size; j++) {
        for (i = 0; i < mr_size; i++) {
            double dotprod = 0.0;
            for (k = 0; k < kc_size; k++) {
                dotprod += A[i + k * mr_size] * B[k + j * stride];
            }
            C[i + j * stride] += dotprod;
        }
    }
}

// Input:
//  Matrix C of size Mc x K
//  Matrix A of size Mc x Kc
//  Matrix B of size Kc x K stored with stride Kc
void matmult_dgebp(double *  C, const double *  A, const double *  B, long stride, long k_size, long kc_size, long mc_size)
{
    long i, j, k;
    long j_max = k_size & (-N_R);
    long i_max = mc_size & (-M_R);
    for (j = 0; j < j_max; j += N_R) {
        long nr_size = N_R;
        for (i = 0; i < i_max; i += M_R) {
#ifdef USE_ASM
            matmult_dgedot4x4_asm(C + j * stride + i, A + i * kc_size, B + j * stride, stride, kc_size);
#else
            matmult_dgedot4x4(C + j * stride + i, A + i * kc_size, B + j * stride, stride, kc_size);
#endif
        }
        if (i < mc_size) {
            long mr_size = mc_size - i;
            matmult_dgedot(C + j * stride + i, A + i * kc_size, B + j * stride, stride, kc_size, nr_size, mr_size);
        }
    }
    if (j < k_size) {
        long nr_size = k_size - j;
        for (i = 0; i < mc_size; i += M_R) {
            long mr_size = MIN(M_R, mc_size - i);
            matmult_dgedot(C + j * stride + i, A + i * kc_size, B + j * stride, stride, kc_size, nr_size, mr_size);
        }
    }
}

// Input:
//  Matrix C of size M x K
//  Matrix A of size M x Kc
//  Matrix B of size Kc x K
void matmult_dgepp(double *  C, const double *  A, const double *  B, long stride, long m_size, long k_size, long kc_size)
{
    long i;

    for (i = 0; i < m_size; i += M_C) {
        int mc_size = MIN(M_C, m_size - i);
        matmult_dgebp(C + i, A + i * kc_size, B, stride, k_size, kc_size, mc_size);
    }
}

// Input:
//  Matrix C of size M x K
//  Matrix A of size M x N
//  Matrix B of size N x K
// Note: typically N = K, but there are border cases when this does not hold
void matmult_dgemm(double *  C, const double *  A, const double *  B, long stride, long m_size, long k_size, long n_size)
{
    long i;
    for (i = 0; i < n_size; i += K_C) {
        long kc_size = MIN(K_C, n_size - i);
        matmult_dgepp(C, A + i * m_size, B + i, stride, m_size, k_size, kc_size);
    }
}

// Input:
//  Matrix A of size Mc x Kc
// Output:
//  Matrix A_copy of size Mc x Kc stored with stride Kc
void matmult_repack_a_submatrix(const double *  A, double *  A_copy, long stride, long m_size, long k_size, long kc_size,
long mc_size) {
    long i, j, k;
    long i_max = mc_size & (-4);
    for (i = 0; i < i_max; i += M_R) {
        long mr_size = M_R;
        for (j = 0; j < kc_size; j++) {
            //~ A_copy[i * kc_size + j * mr_size] = A[j * stride + i];
            //~ A_copy[i * kc_size + j * mr_size + 1] = A[j * stride + i + 1];
            //~ A_copy[i * kc_size + j * mr_size + 2] = A[j * stride + i + 2];
            //~ A_copy[i * kc_size + j * mr_size + 3] = A[j * stride + i + 3];
            register __m128d tmp0 = _mm_load_pd(A + j * stride + i);
            register __m128d tmp1 = _mm_load_pd(A + j * stride + i + 2);
            _mm_store_pd(A_copy + i * kc_size + j * mr_size, tmp0);
            _mm_store_pd(A_copy + i * kc_size + j * mr_size + 2, tmp1);
        }
    }
    if (i < mc_size) {
        long mr_size = mc_size - i;
        for (j = 0; j < kc_size; j++) {
            for (k = 0; k < mr_size; k++) {
                A_copy[i * kc_size + j * mr_size + k] = A[j * stride + i + k];
            }
        }
    }
}

// Input:
//  Matrix A of size M x K
void matmult_repack_a(const double *  A, double *  A_copy, long stride, long m_size, long k_size) {
    long i, j;
    for (i = 0; i < m_size; i += M_C) {
        long mc_size = MIN(M_C, m_size - i);
        for (j = 0; j < k_size; j += K_C) {
            long kc_size = MIN(K_C, k_size - j);
            matmult_repack_a_submatrix(A + j * stride + i, A_copy + i * kc_size + j * m_size, stride, m_size, k_size, kc_size,
mc_size);
        }
    }
}

void matmult(const int lda, const double *A, const double *B, double *C) {
// K_C = 256, M_C = 192, N_R = 4, M_R = 4
// i_max j_max GFLOPS
//1     12    93.8
//  2    6    95.1
//  3    4    94.6
//  4    3    93.2
//  6    2    96.3
//  12   1    94.1
//  16   16   93.6

    int i_max = 3;
    int j_max = 4;

    int i_step = (lda + i_max - 1) / i_max;
    int j_step = (lda + j_max - 1) / j_max;
    int t;

    i_step = (i_step + 3) & (-4);
    j_step = (j_step + 3) & (-4);


    int A_submatrix_size = i_step * j_step + 24; // Anti-aliasing number
    double * A_copy = memalign(64, A_submatrix_size * i_max * j_max * sizeof(double));
    #pragma omp parallel for shared (lda, A, A_copy, B, C, i_step, j_step, i_max, j_max, A_submatrix_size) private (t)
schedule(dynamic, 1)
    for (t = 0; t < i_max * j_max; t++) {
        int i, j, k;
        // k = i * 4 + j
        i= t / j_max;
        j= t % j_max;

        int m_size, k_size, n_size;
        m_size = MIN(i_step, lda - i * i_step);
        k_size = MIN(j_step, lda - j * j_step);

        matmult_repack_a(A + i * i_step + j * j_step * lda, A_copy + (i * j_max + j) * A_submatrix_size, lda, m_size, k_size);
    }

    #pragma omp parallel for shared (lda, A_copy, B, C, i_step, j_step, i_max, j_max, A_submatrix_size) private (t)
schedule(dynamic, 1)
    for (t = 0; t < i_max * j_max; t++) {
        int i, j, k;
        i= t / j_max;
        j= t % j_max;

        int m_size, k_size, n_size;
        m_size = MIN(i_step, lda - i * i_step);
        k_size = MIN(j_step, lda - j * j_step);

        for (k = 0; k < j_max; k++) {
            n_size = MIN(j_step, lda - k * j_step);
            matmult_dgemm(C + i * i_step + j * j_step * lda, A_copy + (i * j_max + k) * A_submatrix_size, B + k * j_step + j *
j_step * lda, lda, m_size, k_size, n_size);
        }
    }
    free(A_copy);
}

//////////////////////
// Assembly section follows
//////////////////////
```

```
section .text
    global matmult_dgedot4x4_asm

align 64
matmult_dgedot4x4_asm:
; double * C: %rdi
; const double * A: %rsi
; const double * B: %rdx
; long stride: %rcx
; long kc_size: %r8
    push    rbx
    movaps xmm0, [rdi]
    shl rcx, 3 ; rcx = stride * 8
    mov r10, rdx ; r10 = B

    movaps xmm2, [rdi + rcx * 1]
    add r10, rcx ; r10 = B1
    mov rax, rcx ; rax = stride * 8
    add rcx, rcx ; rcx = (stride * 8) * 2

    movaps xmm4, [rdi + rcx * 1]
    add rcx, rax ; rcx = (stride * 8) * 3
    mov r9, rdx ; r9 = B
    mov r11, rdx ; r11 = B

    movaps xmm6, [rdi + rcx * 1]
    add r9, rax ; r9 = B1
    add r10, rax ; r10 = B2
    add r11, rcx ; r11 = B3

    movaps xmm1, [rdi + 16]
    movaps xmm3, [rdi + rax * 1 + 16]
    movaps xmm5, [rdi + rax * 2 + 16]
    movaps xmm7, [rdi + rcx * 1 + 16]
    xor rbx, rbx
    shl r8, 3

    ; c0 is xmm0
    ; c1 is xmm1
    ; c2 is xmm2
    ; c3 is xmm3
    ; c4 is xmm4
    ; c5 is xmm5
    ; c6 is xmm6
    ; c7 is xmm7

    ; B0 is rdx
    ; B1 is r9
    ; B2 is r10
    ; B3 is r11

    ; stride is rax
    ; stride * 3 is rcx

    align 32
.loop:
    movddup xmm10, [rdx + rbx * 1]

    movaps xmm14, xmm10
    movaps xmm8, [rsi + rbx * 4]
    mulpd xmm10, xmm8
    addpd xmm0, xmm10

    movaps xmm9, [rsi + rbx * 4 + 16]
    mulpd xmm14, xmm9
    addpd xmm1, xmm14

    movddup xmm11, [r9 + rbx * 1]
    movaps xmm14, xmm11
    mulpd xmm11, xmm8
    addpd xmm2, xmm11

    movddup xmm12, [r10 + rbx * 1]
    movaps xmm15, xmm12
    mulpd xmm14, xmm9
    addpd xmm3, xmm14

    mulpd xmm12, xmm8
    addpd xmm4, xmm12

~ 450 LOC  mulpd xmm15, xmm9

    addpd xmm5, xmm15
    add rbx, 8

    mulpd xmm8, xmm13
    addpd xmm6, xmm8

    mulpd xmm13, xmm9
    addpd xmm7, xmm13
    cmp rbx, r8
    jl .loop

    movaps [rdi], xmm0
    movaps [rdi + 16], xmm1
    movaps [rdi + rax * 1], xmm2
    movaps [rdi + rax * 1 + 16], xmm3
    movaps [rdi + rax * 2], xmm4
    movaps [rdi + rax * 2 + 16], xmm5
    movaps [rdi + rcx * 1], xmm6
    movaps [rdi + rcx * 1 + 16], xmm7

    pop rbx
        ret
```

Model predictive control

Eigenvalues

LU factorization

Optimal binary search organization

Image color conversions

Image geometry transformations

Enclosing ball of points

Metropolis algorithm, Monte Carlo

Seam carving

SURF feature detection

Submodular function optimization

Graph cuts, Edmond-Karps Algorithm

Gaussian filter

Black Scholes option pricing

Disparity map refinement

Singular-value decomposition

Mean shift algorithm for segmentation

Stencil computations

Displacement based algorithms

Motion estimation

Multiresolution classifier

Kalman filter

Object detection

IIR filters

Arithmetic for large numbers

Optimal binary search organization

Software defined radio

Shortest path problem

Feature set for biomedical imaging

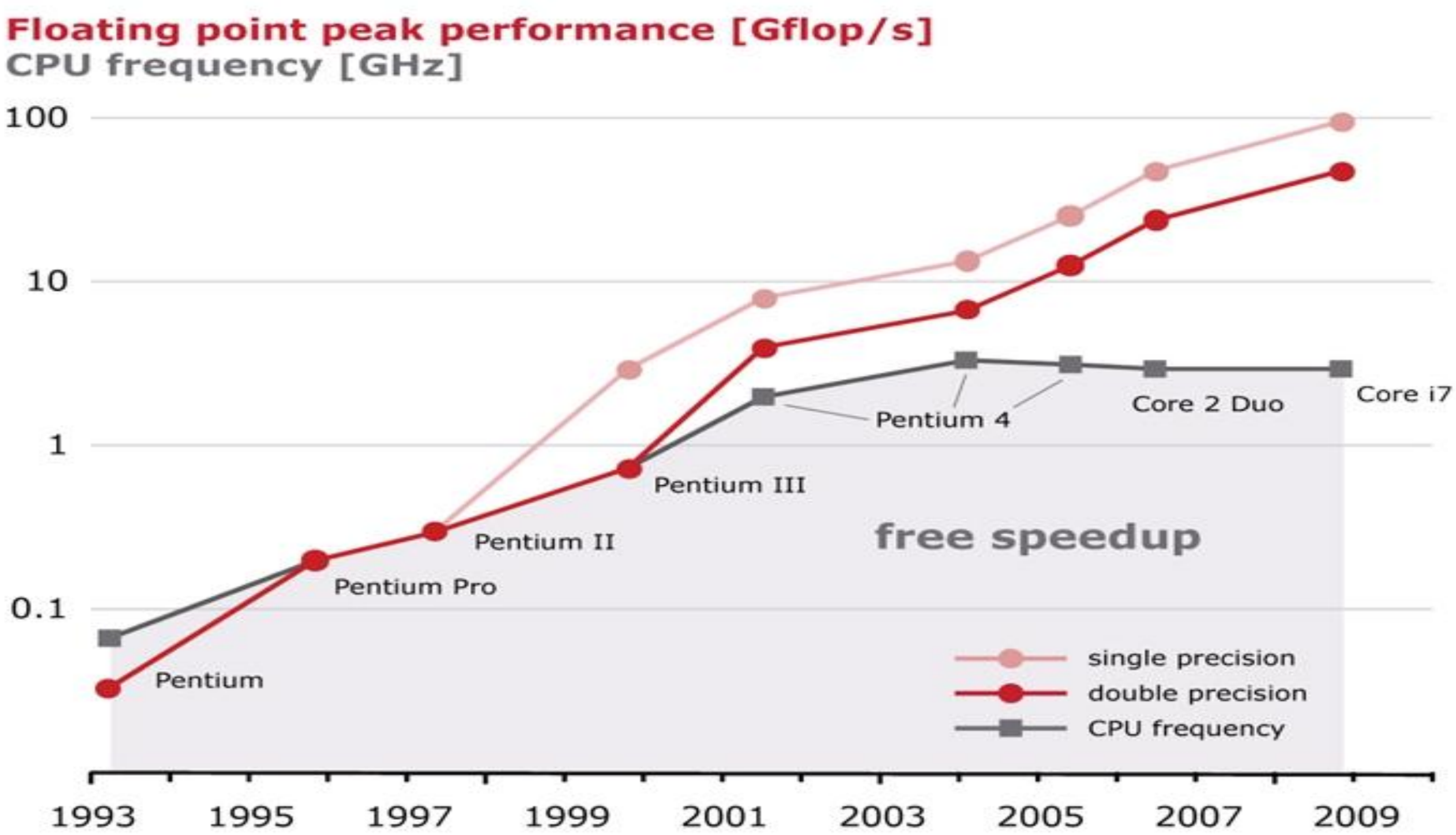
Biometrics identification

“Theorem:”

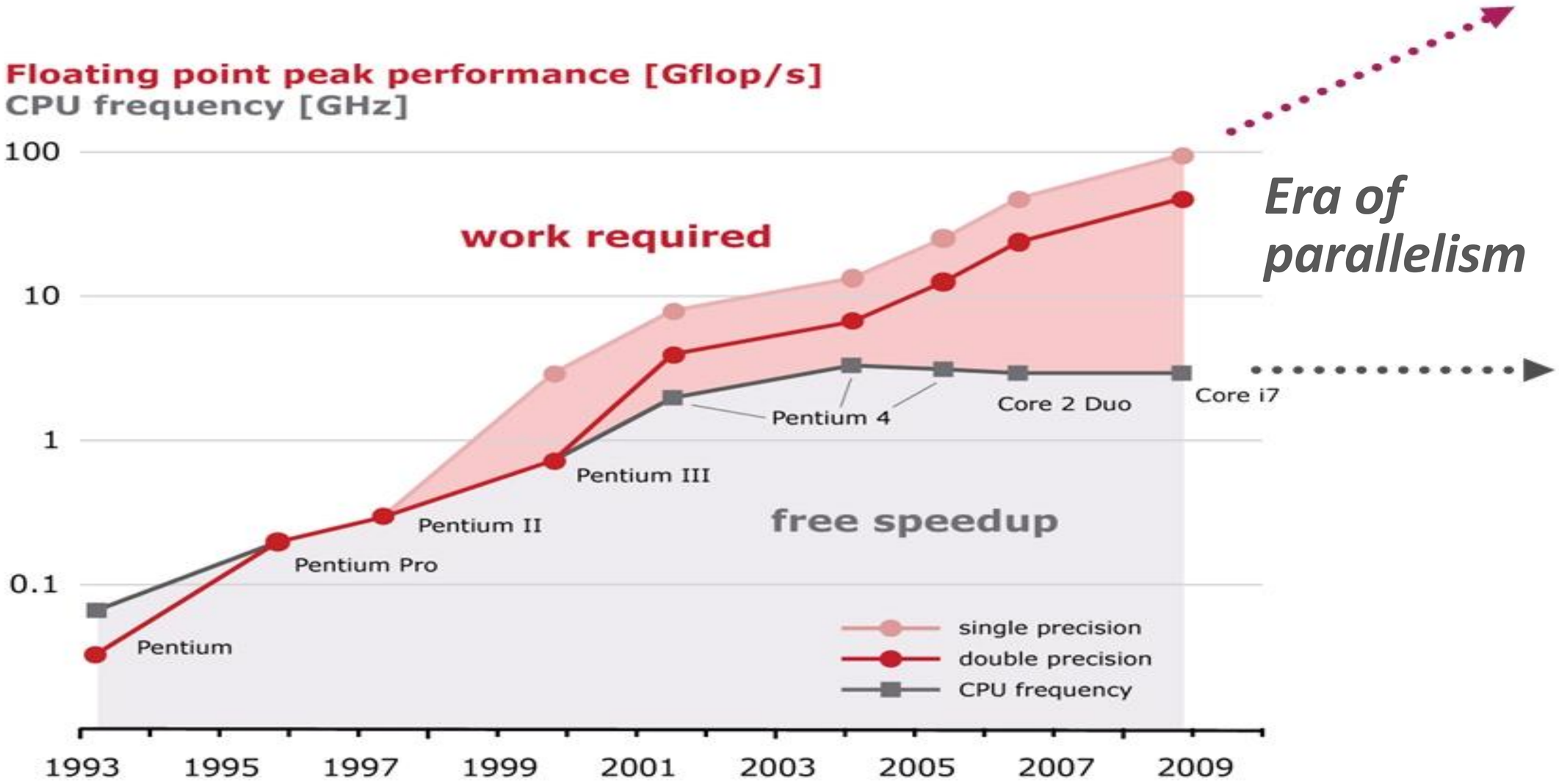
Let f be a mathematical function to be implemented on a state-of-the-art processor. Then

$$\frac{\textit{Performance of optimal implementation of } f}{\textit{Performance of straightforward implementation of } f} \approx \textit{10-100}$$

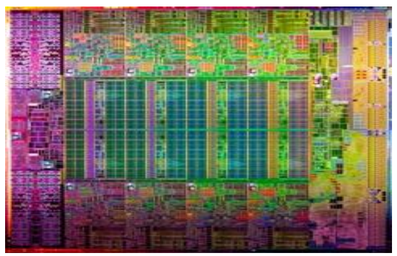
Evolution of Processors (Intel)



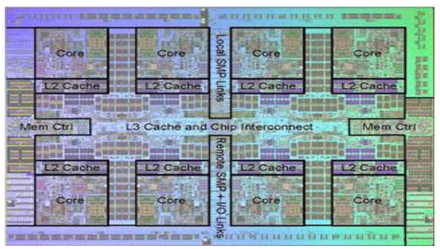
Evolution of Processors (Intel)



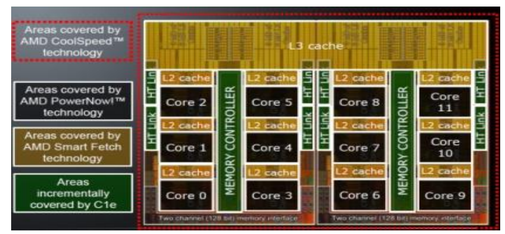
And There Will Be Variety …



Intel Sandy Bridge
8 cores

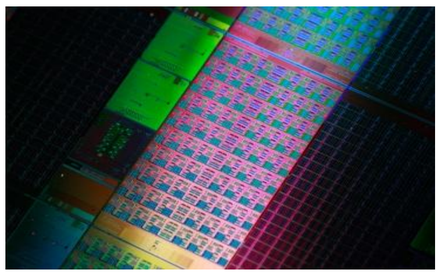


IBM Power 7
8 cores



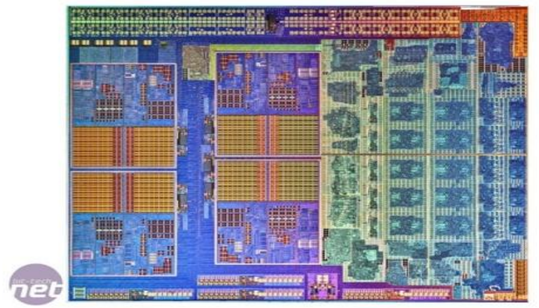
AMD MAGNY-COURS
12 cores

NVIDIA Fermi
512 cores



Intel MIC
61 cores

NVIDIA Kepler
1536 cores



AMD APU
4 CPU cores
400 GPU cores

By 2020:
Laptop ~ 100 cores (low power)
Typical server node chip ~400 cores

Note: All the chips have SMT

Summary and Facts I

- **Implementations with same operations count can have vastly different performance (up to 100x and more)**
  - A cache miss can be 100x more expensive than an operation
  - Vector instructions
  - Multiple cores = processors on one die
- **Minimizing operations count ≠ maximizing performance**
- **End of free speed-up for legacy code**
  - Future performance gains through increasing parallelism

Summary and Facts II

- **It is very difficult to write the fastest code**
  - Tuning for memory hierarchy
  - Vector instructions
  - Efficient parallelization (multiple threads)
  - Requires expert knowledge in algorithms, coding, and architecture
- **Fast code can be large**
  - Can violate “good” software engineering practices
- **Compilers often can’t do the job**
  - Often intricate changes in the algorithm required
  - Parallelization/vectorization still unsolved
- **Highest performance is in general non-portable**

Software Properties

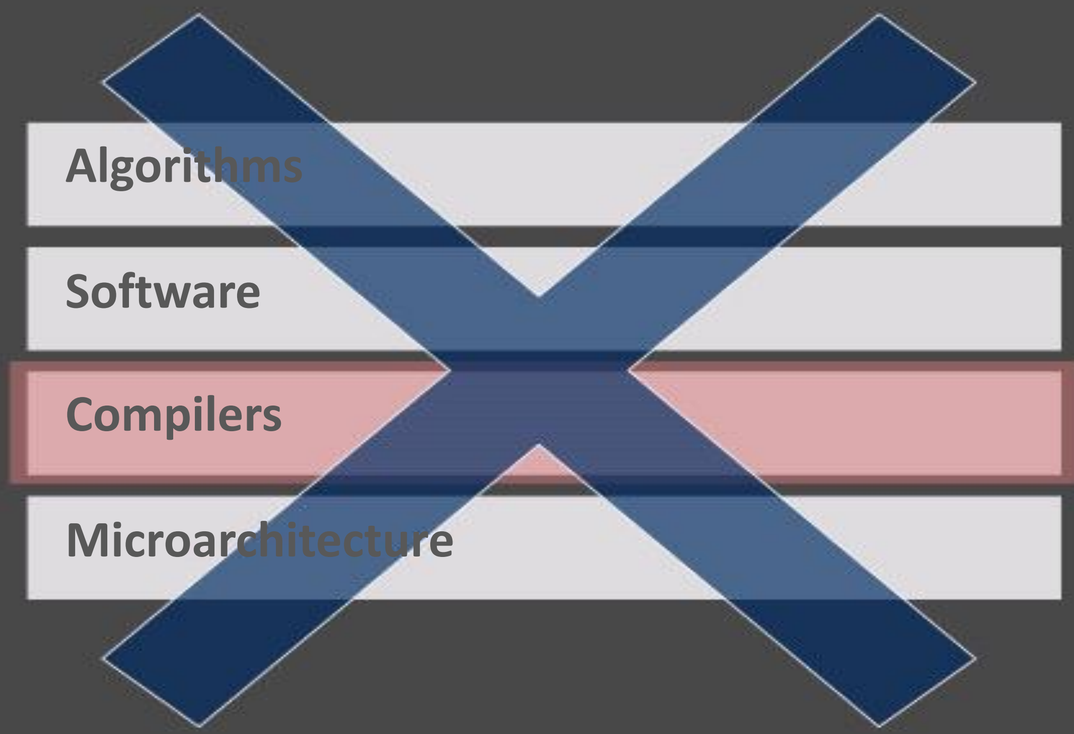
What software properties are more important than performance?

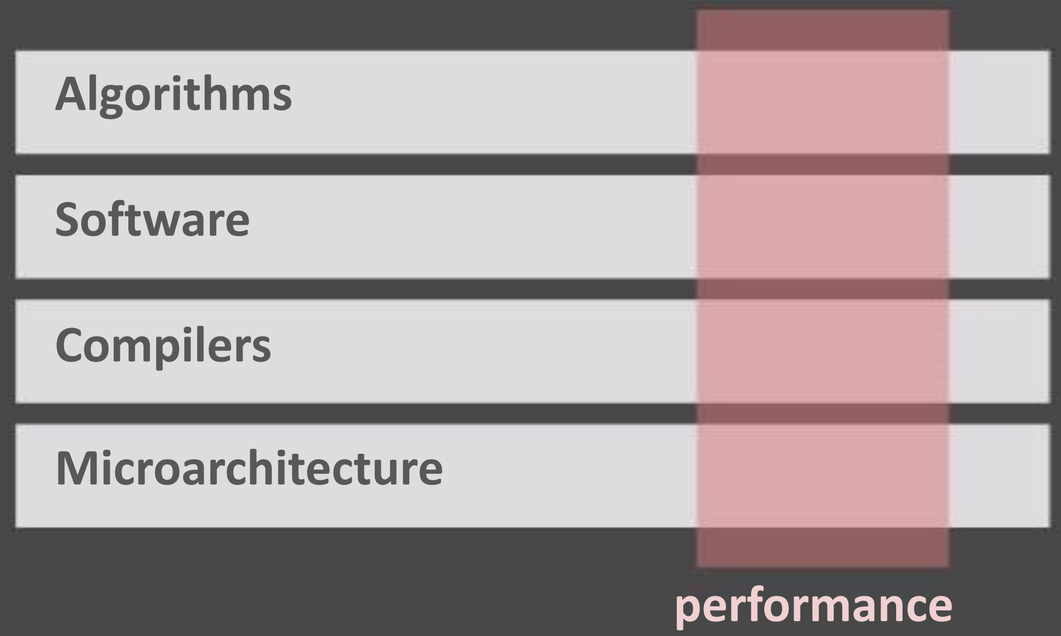
- Compatibility
- Correctness
- Clarity
- Debuggability
- Functionality
- Maintainability
- Modularity
- Portability
- Reliability
- Robustness
- Testability
- Usability

… and more.

If programmers are willing to sacrifice performance for these properties, why study performance?

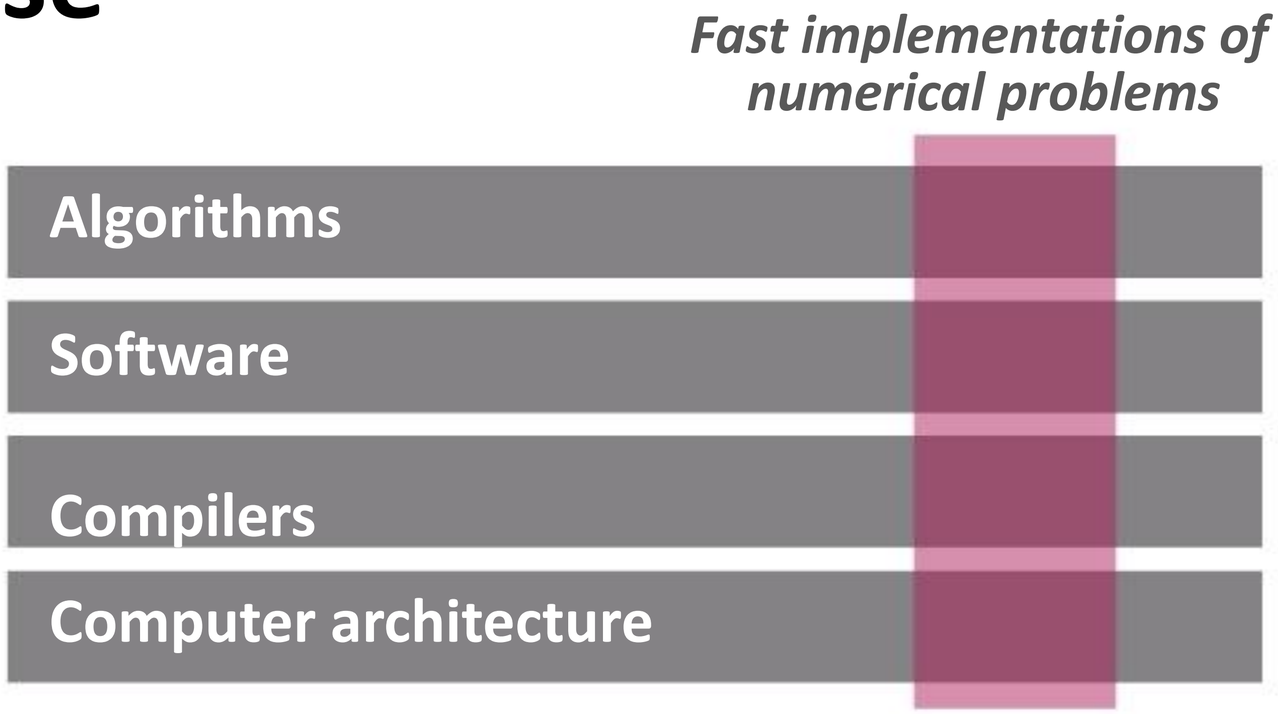
Performance is the currency of computing. You can often "buy" these properties with performance.





Performance is different than other software quality features

This Course



- **Obtain an understanding of performance (runtime)**
- **Learn how to write *fast code* for numerical problems**
  - Focus: Memory hierarchy and vector instructions
  - Principles studied using important examples
  - ***Applied in homeworks and a semester-long research project***
- **Learn about autotuning**

Organization of this course

Course Logistics and Topics

CSE 6230 Schedule (web) : Fall 2012

Week	Topic	Tu	Th	Reading and other assignments
1: Aug 20-24	* Motivation * Basic "laws" and principles: Moore, Amdahl, Little, balance * Multithreaded DAG model (work-depth) * Brent's Theorem * Basic algorithms			Calibration quiz; Ch 5; Ch 3.1; Ch 4.1
2: Aug 27-31	* Benchmarking * Cilk Plus programming model		Lab 1: Cilk Plus	PeerWise Q1
3: Sep 3-7	* GPUs * CUDA programming model		Lab 2: CUDA	
4: Sep 10-14	* GPU tuning * Memory hierarchy basics		Lab 3: Blocked algorithms in CUDA	Rest of Ch 3
5: Sep 17-21	* Polyhedral model		Lab 4: PolyOpt	PeerWise Q2
6: Sep 24-28	* SIMD vectorization		Lab 5: Optimization	Ch 1 & 2
7: Oct 1-5	* Assembly 101 * Microarchitecture 101		Lab 6: Advanced optimization; tools	Project proposals; Ch 4.1 & 4.2; Ch 8
8: Oct 8-12	* Memory optimization * Optimization (III)		Lab 7: More optimization & tools	
9: Oct 15-19	* OpenMP programming	*No class --- Fall break*		PeerWise Q3; Ch 6 & 7
10: Oct 22-26	* Distributed memory (I) * MPI programming		Lab 8: MPI	Rest of Ch 4; Ch 9 & 10
11: Oct 29-Nov 2	* Communication lower bounds		Lab 9: MPI + X	Ch 11
12: Nov 5-9	* Tasking-based runtimes		Lab 10: Tasking	Project checkpoint PeerWise Q4
13: Nov 12-16	*(No class)*	*No class --- Supercomputing 2012*		
14: Nov 19-23	* Parallel I/O -- Guest lecture by Fang Zheng (GT)		*No class --- Thanksgiving*	
15: Nov 26-30	* Emerging programming models (TBD)			PeerWise Q5
16: Dec 3-7	* Future trends in HPC		**Final project poster session**	
Dec 13, 2:50-5:40pm			**Final exam**	Final report & code

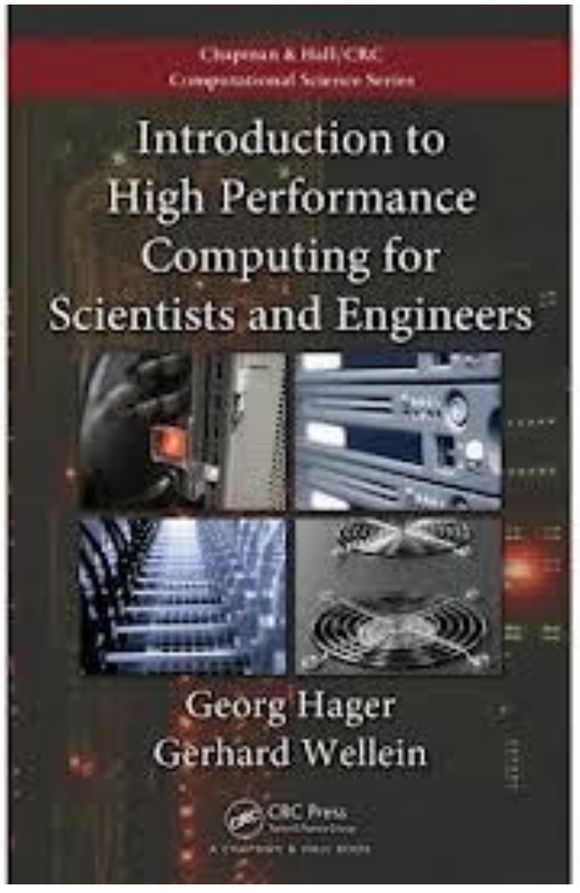
Key
No class
Exam

- Website: http://progforperf.github.com
- Class forum on Piazza: https://piazza.com/class#spring2013/cs1207

Format, Topics, Grading

- Goal: “Hands-on” experience
  - [40%] Labs: In-class and out-of-class, programming-focused
  - [35%] Project: Your choice
- Evaluation
  - [20%] Final
  - [5%] Make up five “test” questions (post on Piazza)
  - Extra credit: Participation on Piazza, extra stuff on labs

Textbook



Introduction to High Performance Computing for Computational Scientists and Engineers, by Georg Hager and Gerhard Wellein. ISBN: 978-1-4398-1192-4, CRC Press, 2010.

http://www.crcpress.com/product/isbn/9781439811924

CASE STUDY:

MATRIX MULTIPLICATION

Square-Matrix Multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

C A B

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$$

Assume for simplicity that $n = 2^k$.

Intel Xeon Computer System

Feature	Specification
Microarchitecture	Sandy Bridge
Clock frequency	2.4 GHz
Processor chips	2
Processing cores	8 per processor chip
Hyperthreading	2 way
Floating-point unit	8 double-precision operations per core per cycle
Cache-line size	64 B
L1-icache	32 KB private 8-way set associative
L1-dcache	32 KB private 8-way set associative
L2-cache	256 KB private 8-way set associative
L3-cache	20 MB shared 20-way set associative
DRAM	32 GB

$$\text{Peak} = 2 \times 8 \times 8 \times 2.4 \times 10_9 = 307 \text{ GFLOPS}$$

1. Triply Nested Loops in Python

```
import  sys,  random
from  time  import  *

n  =  4096

A  =  [[1.0*random.random()
        for      row    in       xrange   (n)]
       for  col  in  xrange(n)]
B  =
c  =
star
for
end

print  '%0.6f'  %  (end  ---  start)
```

Running time
= 34,962 seconds
≈ 9.75 hours

Is this fast?

Back-of-the-envelope calculation

$2n^3 = 2^{37}$ floating-point operations

Running time $\approx 2^{15}$ seconds

$\therefore$ Python gets $2^{37}/2^{15} = 2^{22} \approx 4$ MFLOPS

Peak = 307 GFLOPS

Python gets $\approx 0.0013\%$ of peak

2. Let's Try Java

Running time = 2,531 seconds
≈ 42 minutes
… about 14× faster than Python!
Still only 0.0177% of peak.

```
for (int i=0; i<n; i++) {
  for (int j=0; j<n; j++) {
    for (int k=0; k<n; k++)
      C[i][j] += A[i][k] * B[k][j];
  }
}
```

```
for (int i=0; i<n; ++i) {
  for (int j=0; j<n; ++j) {
    for (int k=0; k<n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

3. Why Not C?

Using the GCC compiler

Running time = 1,463 seconds

$\approx$ 24 minutes

… about 1.7× faster than Java.

```
for (int i=0; i<n; ++i) {
  for (int j=0; j<n; ++j) {
    for (int k=0; k<n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```


```
for (int i=0; i<n; ++i) {
  for (int j=0; j<n; ++j) {
    for (int k=0; k<n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Where We Stand So Far

Version	Implementation	Time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
1	Python	34,962.21	0.004	1		0.00%
2	Java	2,530.65	0.054	14	13.8	0.00%
3	C, using GCC	1,462.50	0.094	24	1.7	0.00%

Why is Python so slow and C so fast?

- Python is interpreted.
- Java is compiled to byte-code, which is then interpreted and just-in-time (JIT) compiled.
- C is compiled directly to machine code.

4. Optimization Switches

GCC provides a collection of optimization switches. Without touching the C code, we can just specify a switch to the compiler to ask it to optimize.

Opt. level	Meaning	Time (s)
`---O0`	Do not optimize	1463
`---O1`	Optimize	856
`---O2`	Optimize even more	851
`---O3`	Optimize yet more	427

5. Intel Compiler

GCC is not the only compiler in the world. Let's try the Intel compiler ICC with the `---03` optimization switch.

Version	Implementation	Time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
1	Python	34,962.21	0.004	1		0.00%
2	Java	2,530.65	0.054	14	13.8	0.00%
3	C, using GCC	1,462.50	0.094	24	1.7	0.00%
4	+ switches	426.79	0.322	82	3.4	0.10%
5	C, using ICC + switches	41.44	3.317	844	10.3	1.10%

Wow! We're now running in less than a minute and are over 800 times faster than the original Python! Why is ICC so much faster?

Vectorization

Each core of our computer has 8 vector units which can initiate 8 floating-point operations on each cycle using a single vector instruction, as long as the operations are independent. Most compilers can be induced to produce a vectorization report:

```
$ icc ---O3 ---std=c99 mm_c.c ---o mm_c_icc_O3 ---vec---report2

mm_c.c(42): (col. 5) remark: PERMUTED LOOP WAS VECTORIZED.
mm_c.c(43): (col. 7) remark: loop was not vectorized: not inner loop.
mm_c.c(41): (col. 3) remark: loop was not vectorized: not inner loop.
```

```
for (int i=0; i<n; ++i) {
  for (int j=0; j<n; ++j) {
    for (int k=0; k<n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Interchange these two loops.

6. Parallel Loops

We're running on only one of our 16 cores, leaving 15 idle. Let's use all of them!

```
cilk_for (int i=0; i<n; ++i) {
  cilk_for (int j=0; j<n; ++j) {
    for (int k=0; k<n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

The `cilk_for` keyword, which is supported by both ICC and GCC (not the main branch, however), indicates that all the iterations of the loop may execute in parallel.

Parallel-Loops Performance

Running time

- 18 seconds, the fastest so far!
- But wait, it's only 2.3 times faster than the previous version, which used only 1 core, and we're now using 16!

Hardware Performance Counters

Diagnose the problem with hardware performance counters

```
$ perf stat ---e instructions ---e cache---misses ------ ./mm_c_icc_03
38.946693
 Performance counter stats for './mm_c_icc_O3':
   174,398,127,129 instructions                     #     0.00    insns per cycle
         19,491,135 cache---misses
      39.422211009 seconds time elapsed

$ perf stat ---e instructions ---e cache---misses ------ ./mm_ploops
21.164532
 Performance counter stats for './mm_ploops':
   373,875,833,663 instructions                     #     0.00    insns per cycle
      9,682,355,938 cache---misses
      21.653572167 seconds time elapsed
```

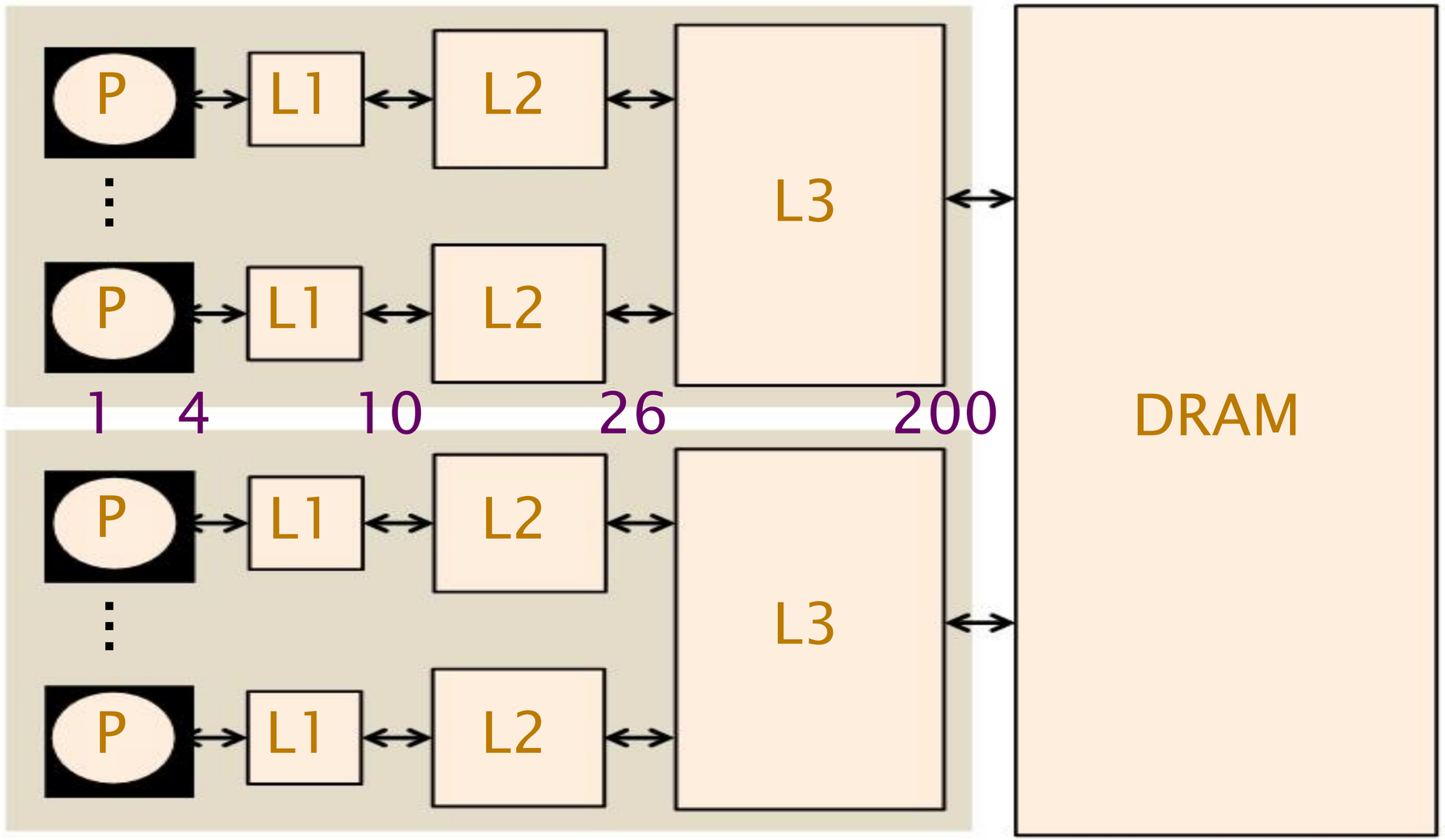
The parallel code is executing more than twice the number of instructions and incurring about 500 times the number of cache misses.

Compiler Bug!

Inspection of the assembly language output from the compiler reveals that `icc -O3` tiles the matrix — an effective optimization — but it fails to do so in the context of `cilk_for`.

What is tiling?

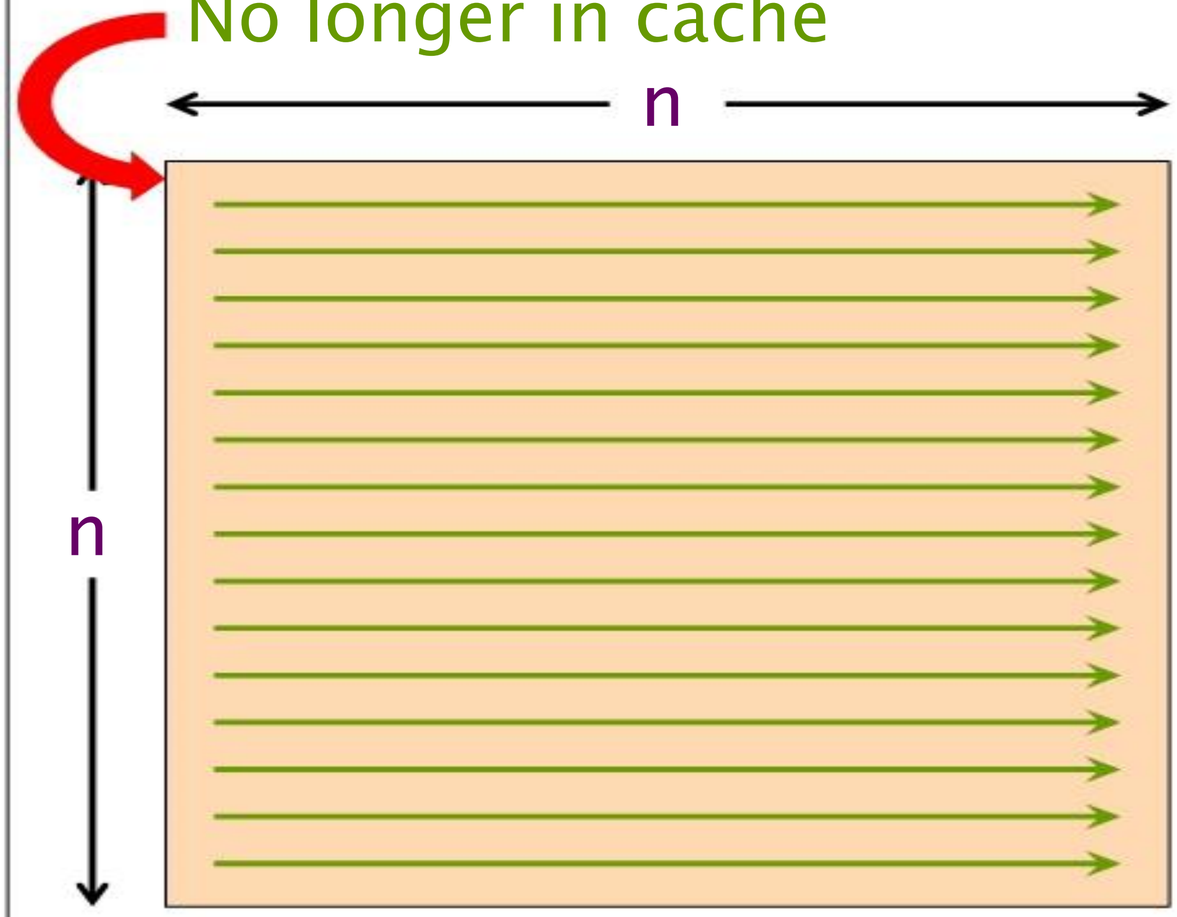
"Sandy Bridge" Memory Hierarchy



Latency in clock cycles

1 4 10 26 200

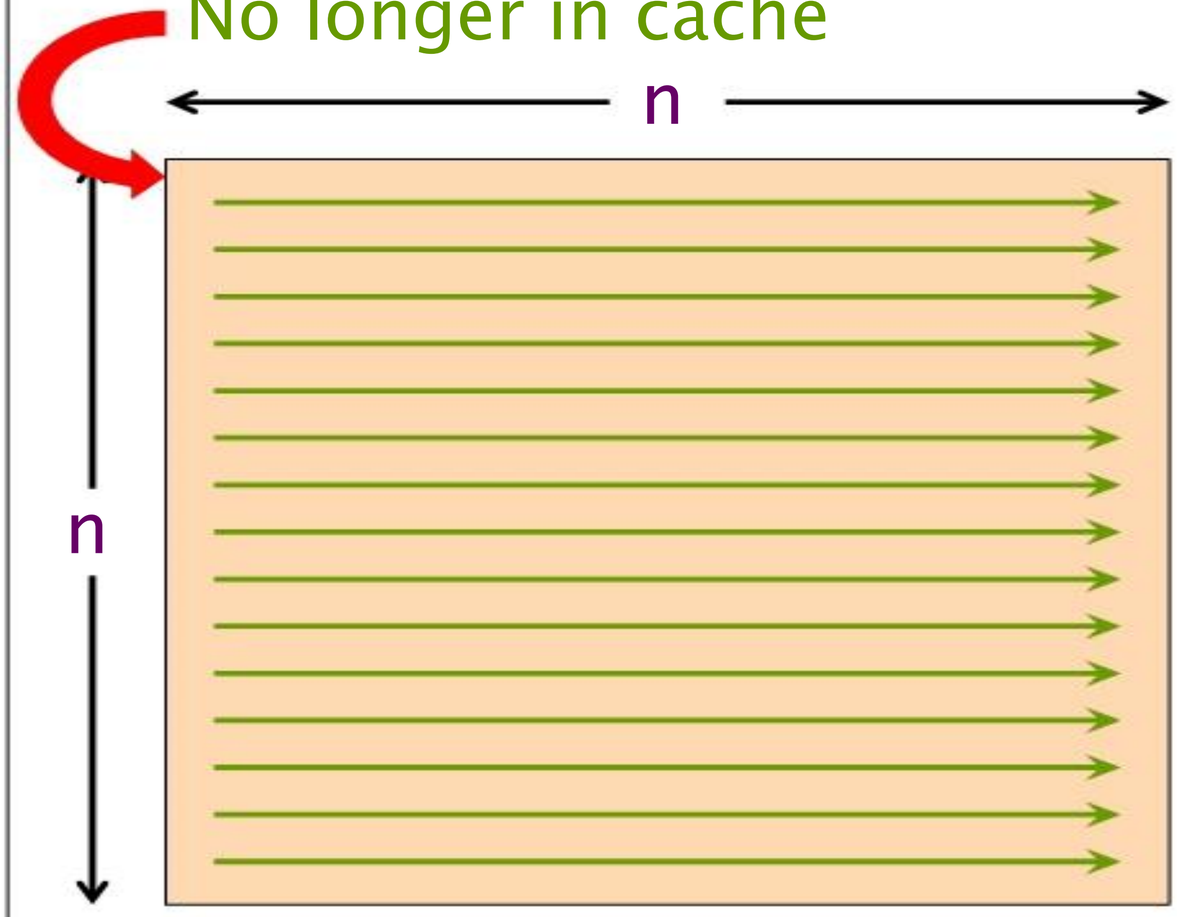
Cache Misses for Matrix Multiply



Cache misses

- Consider the code after the two inner loops have been permuted.
- As the code sequences through matrix B it incurs $\Theta(n^2)$ cold cache misses for new data elements it encounters
- By the time the code sequences through the matrix again, the first elements have been evicted.

Cache Misses for Matrix Multiply

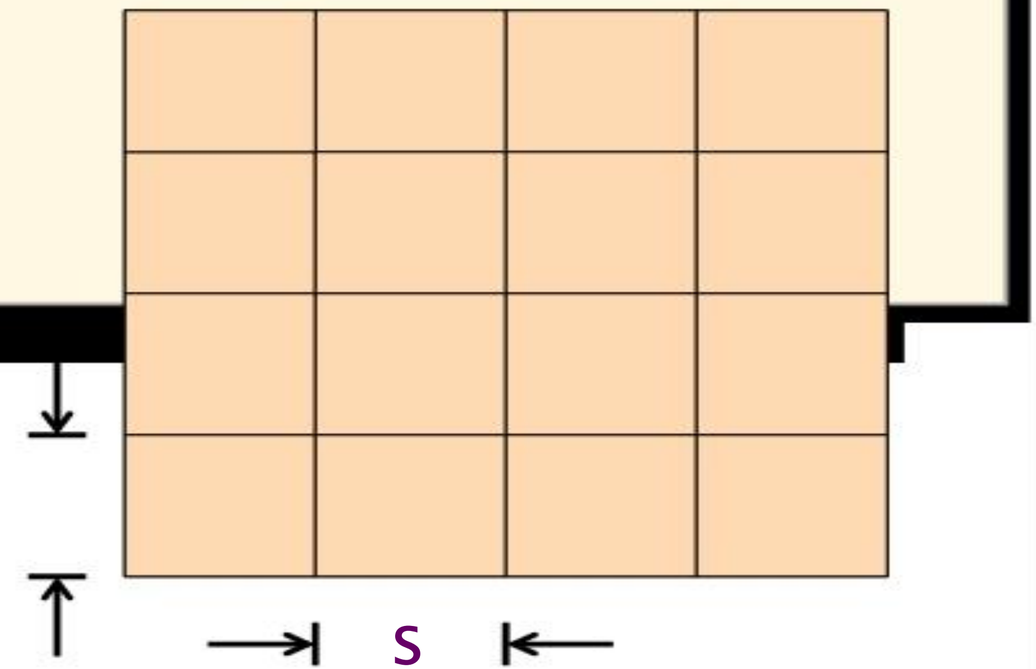


Cache misses

- As the code sequences through matrix B, it incurs $\Theta(n^2)$ cold cache misses for new data elements it encounters
- By the time the code sequences through the matrix again, the first elements have been evicted.
- $\Theta(n^3)$ cache misses.

7. Tiling

```
cilk_for (int ih = 0; ih < n; ih += s) {
  cilk_for (int jh = 0; jh < n; jh += s) {
    for ( int kh = 0 ; kh < n; kh += s) {
      for (int il = 0; il < s; il++) {
        for (int jl = 0; jl < s; jl++) {
          for (int kl = 0; kl < s; ++kl) {
            C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jh+jl];
          }
        }
      }
    }
  }
}
```



Cache misses

- If s_2 is sufficiently smaller than the size of the cache, the tiled loops incur only $\Theta(n_3/s)$ cache misses.

Performance of Tiling

Tile size	1 core (s)	16 cores (s)
1	859.64	65.30
2	309.60	26.32
4	178.17	9.19
8	93.14	6.56
16	76.65	5.09
32	60.85	3.49
64	54.17	3.24
128	44.79	2.76
256	40.87	4.73
512	42.77	6.79
1024	69.00	8.39
2048	65.96	26.13

Tile size

- For 16 cores, a 128×128 tile gives the best performance.
- For 1 core, however, 256×256 tile works best.
- If tile size is not properly tuned — either too large or too small — the code may perform poorly.
- Tiling is fragile.

Cache Behavior of Tiling

Version	Implementation	Tiling	Cores	Instructions	Cache misses	Time (s)
5	Serial optimized loops	no	1	165,416,672,245	1,980,866,303	64.1
5	Serial optimized loops	compiler	1	174,397,675,927	19,635,564	39.4
6	Parallel loops	no	1	246,711,574,370	4,363,182,438	82.9
6	Parallel loops	no	16	307,981,194,383	6,743,511,892	17.9
7	Parallel loops	128	1	177,072,223,809	69,526,210	44.2
7	Parallel loops	128	16	179,009,006,797	98,549,525	2.9

Version	Implementation	Time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
1	Python	34,962.21	0.004	1		0.00%
2	Java	2,530.65	0.054	14	13.8	0.00%
3	C, using GCC	1,462.50	0.094	24	1.7	0.00%
4	+ switches	426.79	0.322	82	3.4	0.10%
5	C, using ICC + switches	41.44	3.317	844	10.3	1.10%
6	Parallel loops	17.95	7.658	1,948	2.3	2.50%
7	+ tiling	2.76	49.825	12,675	6.5	16.20%

Divide and Conquer

IDEA: Tile for every power of 2.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$$

- 8 multiplications of $n/2 \times n/2$ matrices.
- 1 addition of $n \times n$ matrices.

8. Recursive Parallel Matrix Multiply

```
void mmdac (double *C, double *A, double *B, int size) {
  if (size <= 1) {
    * C += * A * * B;
  } else {
    int s11 = 0;
    int s12 = size/2;
    int s21 = (size/2)*n;
    int s22 = (size/2)*(n+1);
    cilk_spawn mmdac(C+s11, A+s11, B+s11, size/2);
    cilk_spawn mmdac(C+s12, A+s11, B+s12, size/2);
    cilk_spawn mmdac(C+s21, A+s21, B+s11, size/2);
               mmdac(C+s22, A+s21, B+s12, size/2);
    cilk_sync;
    cilk_spawn mmdac(C+s11, A+s12, B+s21, size/2);
    cilk_spawn mmdac(C+s12, A+s12, B+s22, size/2);
    cilk_spawn mmdac(C+s21, A+s22, B+s21, size/2);
               mmdac(C+s22, A+s22,
    cilk_sync;
  }
}
```

The named child function may execute in parallel with the parent caller.

Control may not pass this point until all spawned children have returned.

Performance of D&C

Version	Implementation	Time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
1	Python	34,962.21	0.004	1		0.00%
2	Java	2,530.65	0.054	14	13.8	0.00%
3	C, using GCC	1,462.50	0.094	24	1.7	0.00%
4	+ switches	426.79	0.322	82	3.4	0.10%
5	C, using ICC + switches	41.44	3.317	844	10.3	1.10%
6	Parallel loops	17.95	7.658	1,948	2.3	2.50%
7	+ tiling	2.76	49.825	12,675	6.5	16.20%
8	Parallel divide-and-conquer	142.61	0.964	245	0.0	0.30%

Uh, oh! A big step backwards!

Function-Call Overhead

```
void mmdac (double *C, double *A, double *B, int size) {
  if (size <= 1 ) {
    *C += *A * *B;
  } else {
    int s11 = 0;
    int s12 = size/2;
    int s21 = (size/2)*n;
    int s22 = (size/2)*(n+1);
    cilk_spawn mmdac(C+s11, A+s11, B+s11, size/2);
    cilk_spawn mmdac(C+s12, A+s11, B+s12, size/2);
    cilk_spawn mmdac(C+s21, A+s21, B+s11, size/2);
               mmdac(C+s22, A+s21, B+s12, size/2);
    cilk_sync;
    cilk_spawn mmdac(C+s11, A+s12, B+s21, size/2);
    cilk_spawn mmdac(C+s12, A+s12, B+s22, size/2);
    cilk_spawn mmdac(C+s21, A+s22, B+s21, size/2);
               mmdac(C+s22, A+s22, B+s22, size/2);
    cilk_sync;
  }
}
```

The base case is too small. We must coarsen the recursion to avoid function-call overhead.

Versions 9 &10

Version	Implementation	Time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
1	Python	34,962.21	0.004	1		0.00%
2	Java	2,530.65	0.054	14	13.8	0.00%
3	C, using GCC	1,462.50	0.094	24	1.7	0.00%
4	+ switches	426.79	0.322	82	3.4	0.10%
5	C, using ICC + switches	41.44	3.317	844	10.3	1.10%
6	Parallel loops	17.95	7.658	1,948	2.3	2.50%
7	+ tiling	2.76	49.825	12,675	6.5	16.20%
8	Parallel divide-and-conquer	142.61	0.964	245	0	0.30%
9	+ coarsening	5.44	25.255	6,425	26.2	8.20%
10	+ transpose	1.72	79.698	20,274	3.2	25.90%

- Coarsening does a good job.
- Transposing the B matrix does a great job!

Unportable Performance

- Use the `-xHost` ICC compiler switch to generate modern AVX vector instructions, but the code won't run on older machines.
- Use the `-axAVX` ICC compiler switch, which generates multiple clones of the code, one of which uses the AVX instructions. A test is made at runtime as to which version to use.
- Portable code, but the performance is not portable.
- Use compiler intrinsics (assembly-language directives) to access the AVX instructions directly. Highly unportable, but great performance!

Final Reckoning

Version	Implementation	Time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
1	Python	34,962.21	0.004	1		0.00%
2	Java	2,530.65	0.054	14	13.8	0.00%
3	C, using GCC	1,462.50	0.094	24	1.7	0.00%
4	+ switches	426.79	0.322	82	3.4	0.10%
5	C, using ICC + switches	41.44	3.317	844	10.3	1.10%
6	Parallel loops	17.95	7.658	1,948	2.3	2.50%
7	+ tiling	2.76	49.825	12,675	6.5	16.20%
8	Parallel divide-and-conquer	142.61	0.964	245	0.0	0.30%
9	+ coarsening	5.44	25.255	6,425	26.2	8.20%
10	+ transpose	1.72	79.698	20,274	3.2	25.90%
11	+ machine-specific compilation	1.58	86.725	22,061	1.1	28.20%
12	+ AVX intrinsics	0.76	180.741	45,978	2.1	58.80%
13	Intel MKL	0.63	218.096	55,480	1.2	71.00%

Our sub-1-second Version 12 rivals the experts who coded the Intel Math Kernel Library! We're over 4.6 orders of magnitude better than the original Python.

Performance Engineering

Gas economy

45,978 ×

- You won't generally see the kind of performance improvement we obtained for matrix multiplication.
- But in this course, you will learn how to print the currency of performance all by yourself.