High-Performance Computing

What? (Goal)

How to write ***fast*** code (not how to write code fast)

Why?

Fast code saves ***time*** and ***energy***

How?

Parallelism

Data Locality

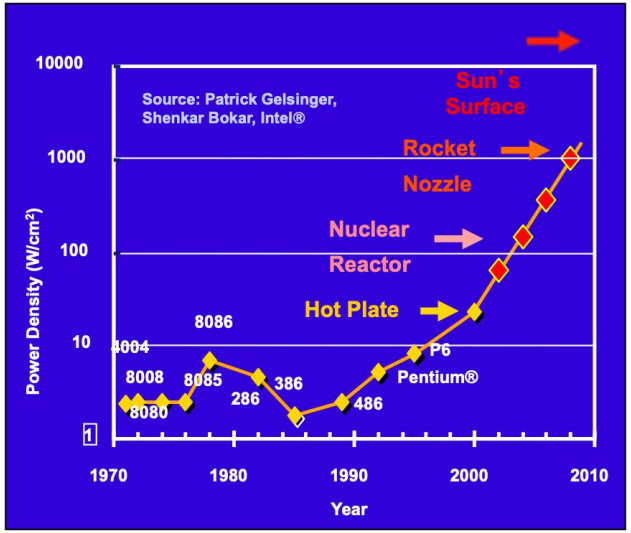
Specialization

Motivation

Currently, parallelism is driven by

power, ***memory***, and ***physics***.

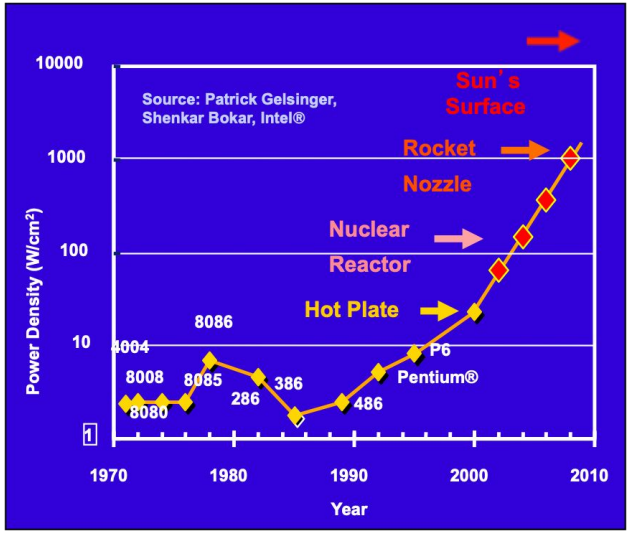
Performance & Power



Performance $\propto$ (cores) $\times$ (freq)

Power $\propto$ (cores) $\times$ (freq$^{2.5}$)

Performance & Power

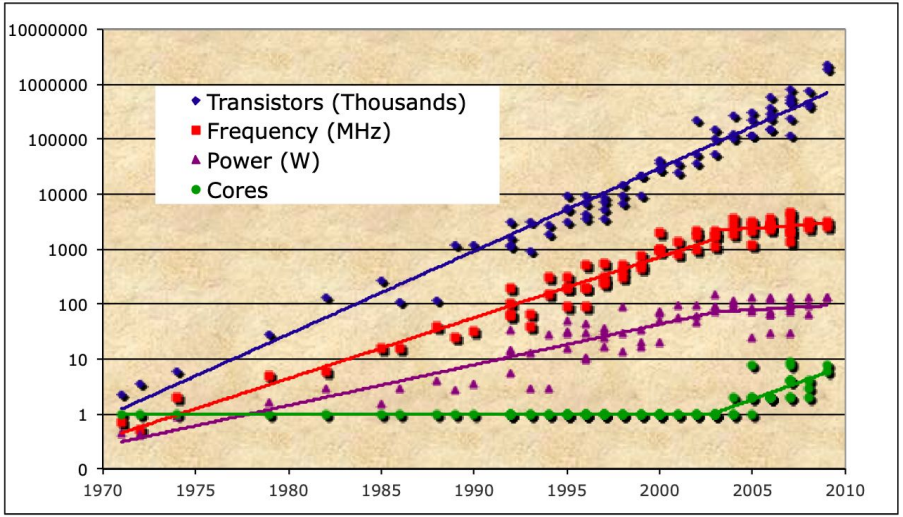


Performance $\propto$ (cores) $\times$ (freq)

Power $\propto$ (cores) $\times$ (freq$^{2.5}$)

*Is it better to increase performance by doubling **frequency** or the number **cores**?*

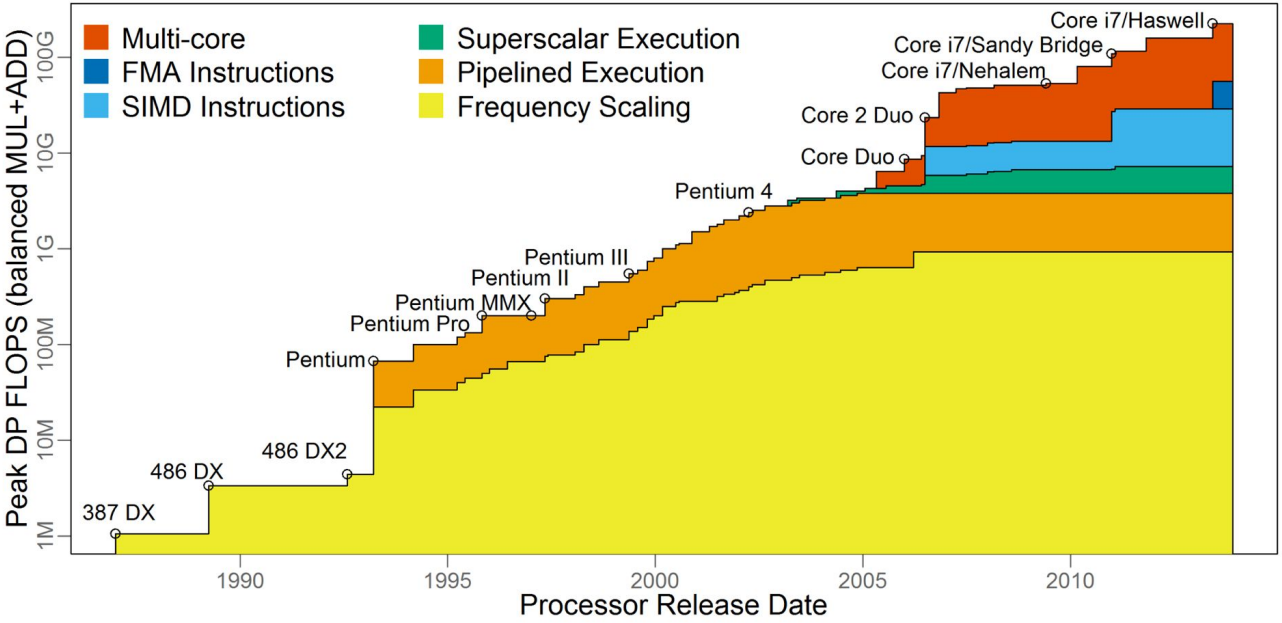
Parallelism & Power



Notice the transition around 2004.

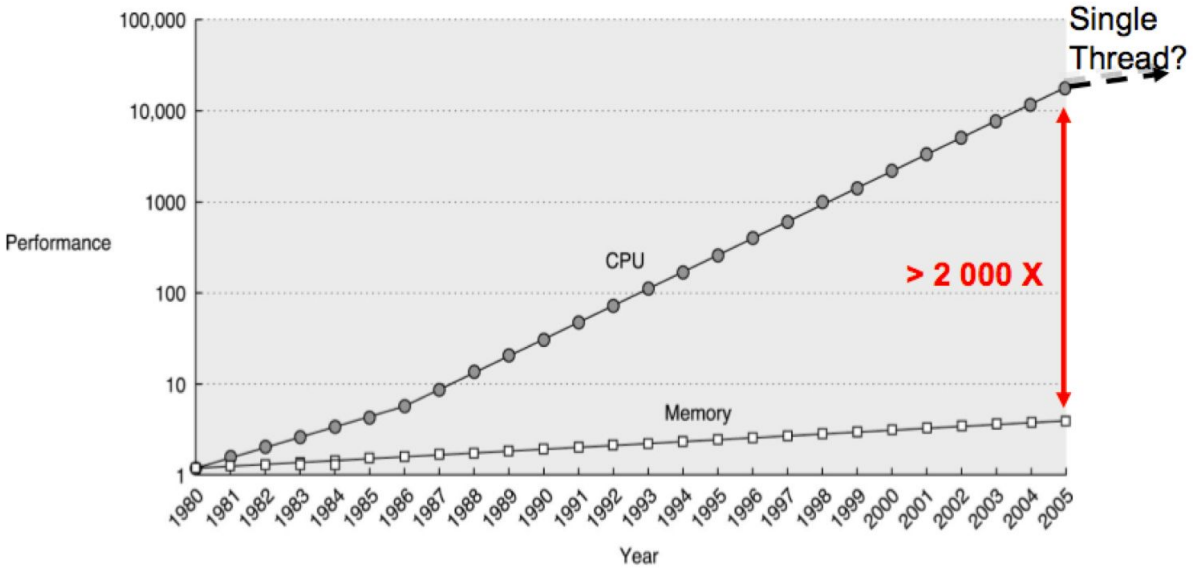
End of ***Dennard Scaling***.

Performance Factors



What else has been done to increase performance?

Memory



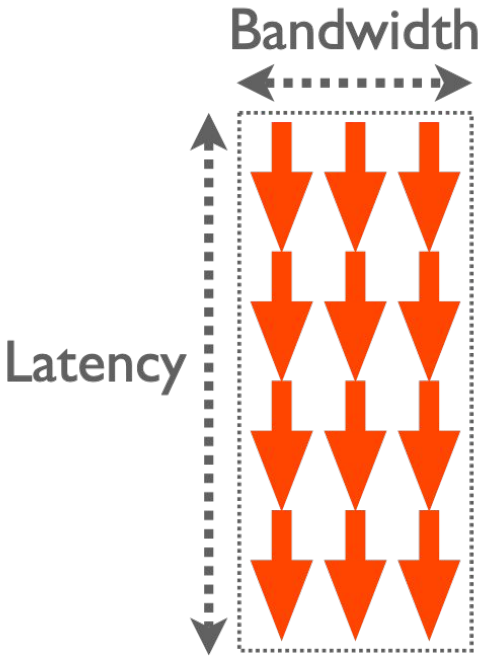
Memory performance has scaled much more slowly than instruction performance (i.e., FLOPS)

Historically, memory

latency halves every ~9 years

bandwidth doubles every ~ 3 years

Memory



Little's Law (queuing theory)

$$L = \lambda W$$

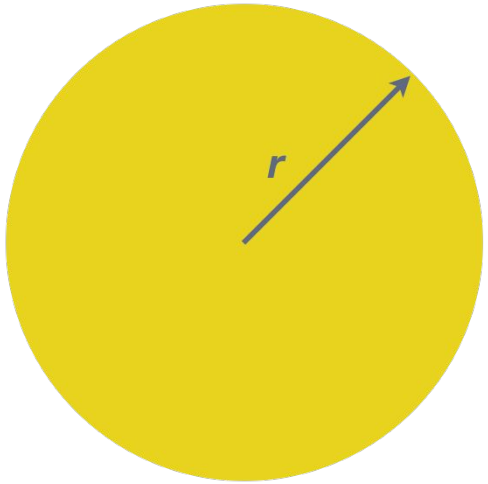
Equivalently,

Concurrency =

Latency × Bandwidth

That is, larger the latency and bandwidth, more concurrency (i.e., threads) is required to fully utilize the memory bandwidth.

Physical Limits



Speed of light $c = 3 \times 10^8$ m/s

1 "op" needs to travel distance 2r

1 Top/s ⟶ 1 op = 1×10^{-12} s

$2r / c = 1 \times 10^{-12}$ s ⟶

$r = (1 \times 10^{-12} \times c) / 2 = 0.15$ mm

Given this size, what about memory?

1TB ⟶ ~6 Å^2 / byte⟶ O(10) atoms

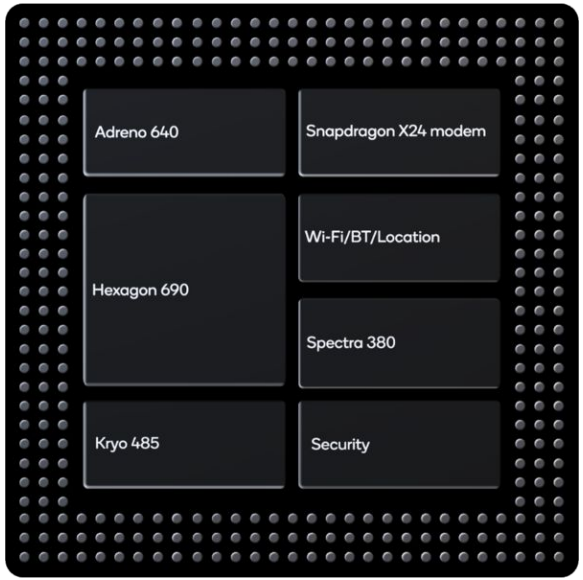
Physical Limits

1. Quantum mechanics (e.g., tunneling)
2. EUV (extreme ultraviolet) lithography
3. 3D transistors/chips

Conclusion

As a result of these trends, parallelism has become ubiquitous.

No matter what system scale you care about, parallelism affects you.



Qualcomm Snapdragon 855 Processor

1. **4x** Kyro 485 high-efficiency cores @ **1.8 GHz**
2. **3x** Kyro 485 high-performance cores @ **2.42 GHz**
3. **1x** Kyro 485 high-performance cores @ **2.84 GHz**
4. **1x** Adreno 640 GPU @ **600 MHz**
   a. **384x** ALU
5. **~10 Watts**

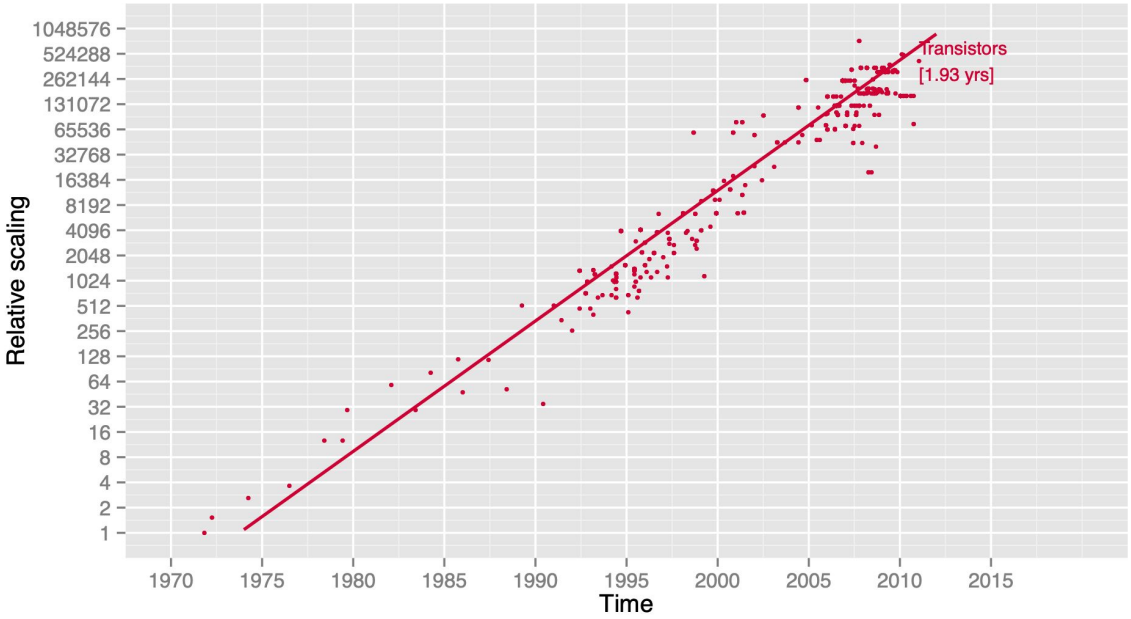
Intel Core i7-9700k

1. **8x** Cores @ **4.90 GHz**
2. **1x** Intel UHD Graphics 630 @ **1.20 GHz**
   a. **24x** execution units
3. **95 Watts**

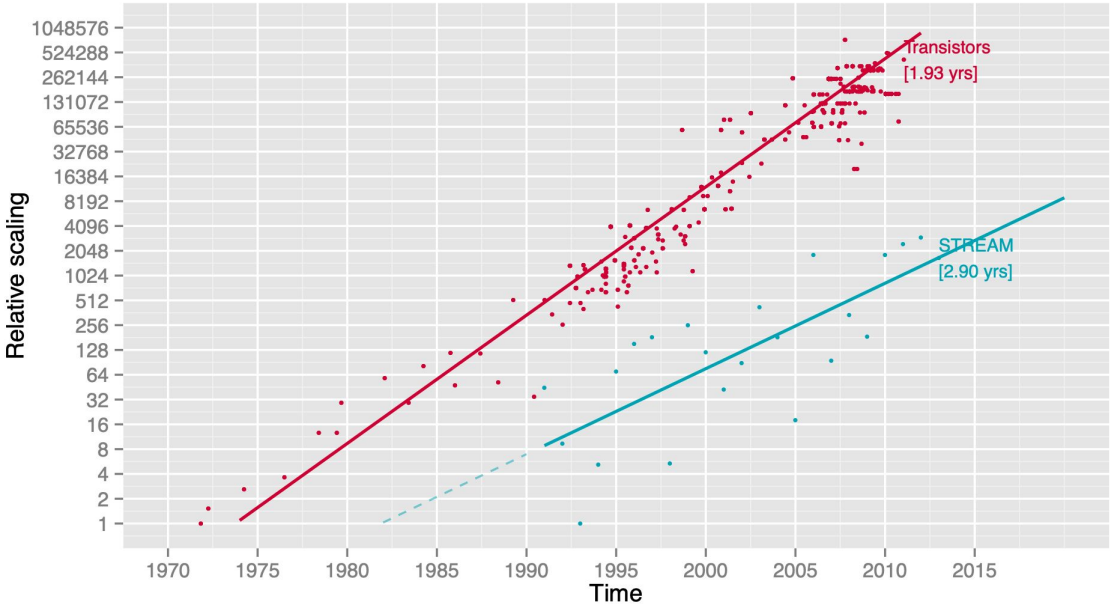
Summit Supercomputer (Department of Energy)

1. **4608x** POWER9 nodes
    a. **2x** POWER9 CPUs @ **4 GHz**
        i. **20x** Cores per CPU
    b. **6x** Nvidia Tesla V100 GPUs @ **1.53 GHz**
        i. **5120x** CUDA cores per GPU
2. **13 MWatts**

Data Movement



Data Movement



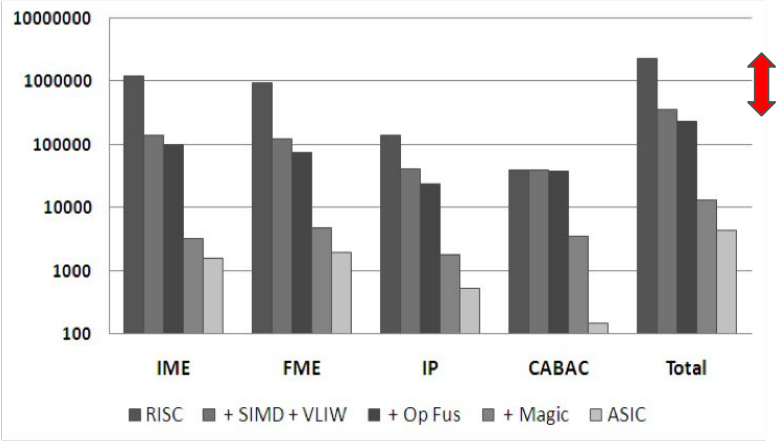


Figure 2. Each set of bar graphs represents energy consumption (μJ) at each stage of optimization for IME, FME, IP and CABAC respectively. Each optimization builds on the ones in the previous stage with the first bar in each set representing RISC energy dissipation followed by generic optimizations such as SIMD and VLIW, operation fusion and ending with "magic" instructions

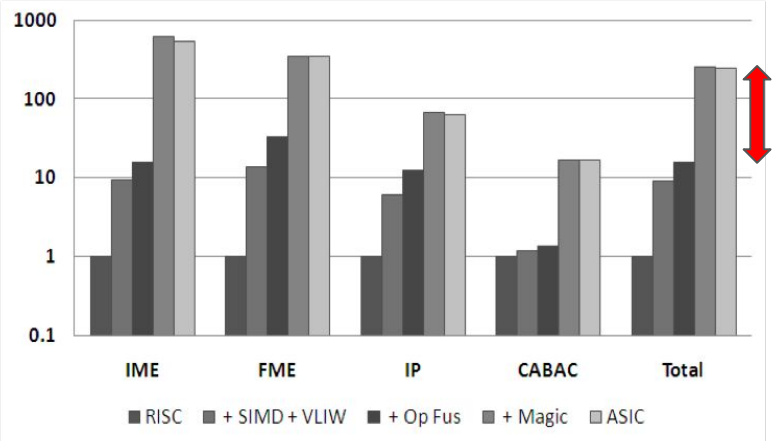


Figure 3. Each set of bar graphs represents speedup at each stage of optimization. Each optimization builds on those of the previous stage with the first bar in each set representing RISC speedup, followed by generic optimizations such as SIMD and VLIW, then operation fusion and finally "magic" instructions

~ 10x in energy (left) and time (right) from generic optimizations (hardware & software)

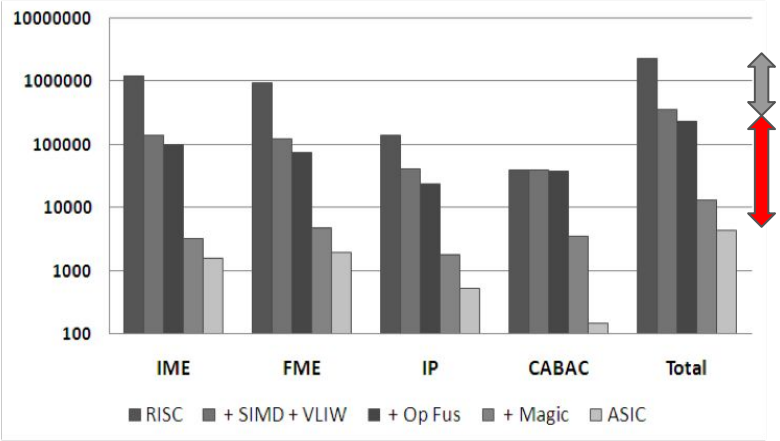


Figure 2. Each set of bar graphs represents energy consumption (μJ) at each stage of optimization for IME, FME, IP and CABAC respectively. Each optimization builds on the ones in the previous stage with the first bar in each set representing RISC energy dissipation followed by generic optimizations such as SIMD and VLIW, operation fusion and ending with "magic" instructions

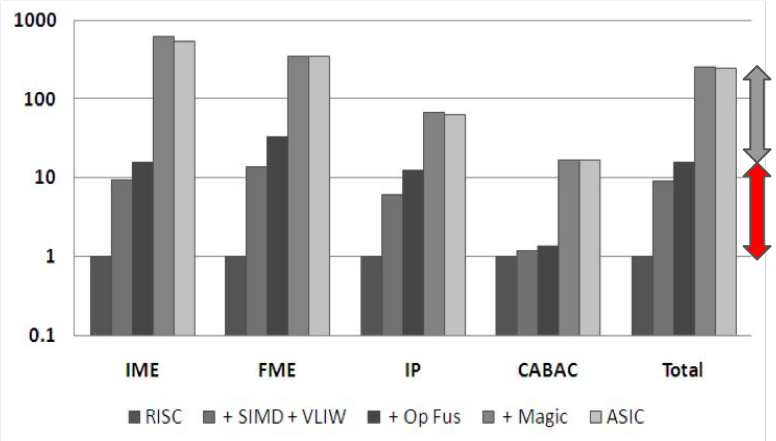


Figure 3. Each set of bar graphs represents speedup at each stage of optimization. Each optimization builds on those of the previous stage with the first bar in each set representing RISC speedup, followed by generic optimizations such as SIMD and VLIW, then operation fusion and finally "magic" instructions

~ 10x in energy (left) and time (right) from generic optimizations (hardware & software)

~ 10x+ in energy (left) and time (right) from application-specific customization (of the hardware)

Specialization

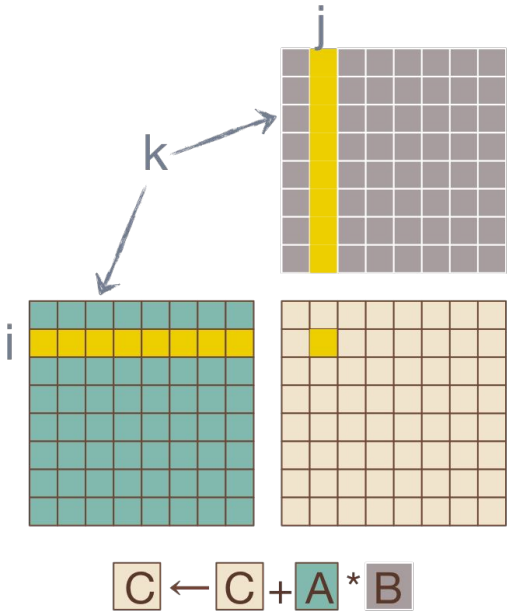
1. Application-specific integrated circuit (ASIC) is **500x** more energy efficient than general-purpose chip multi-processors (CMP)
2. Over **90%** of energy is "**overhead**" (e.g., instruction fetch/decode, etc.) because the cost of actual FLOP is very cheap.

Conclusion

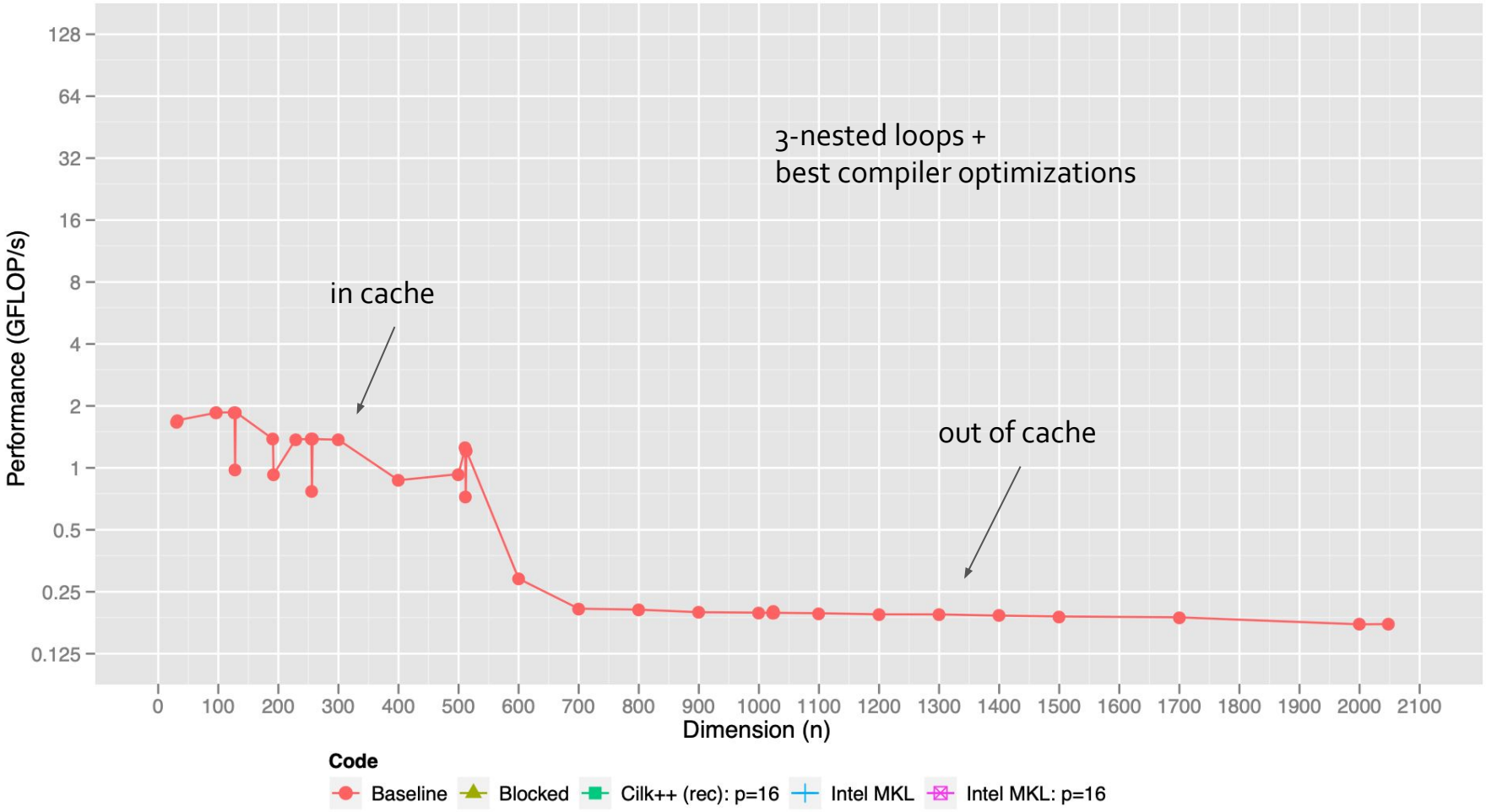
These observations imply we need to pay attention to data movement and exploit custom units (e.g., GPUs) to improve energy efficiency and performance (in addition to parallelization)

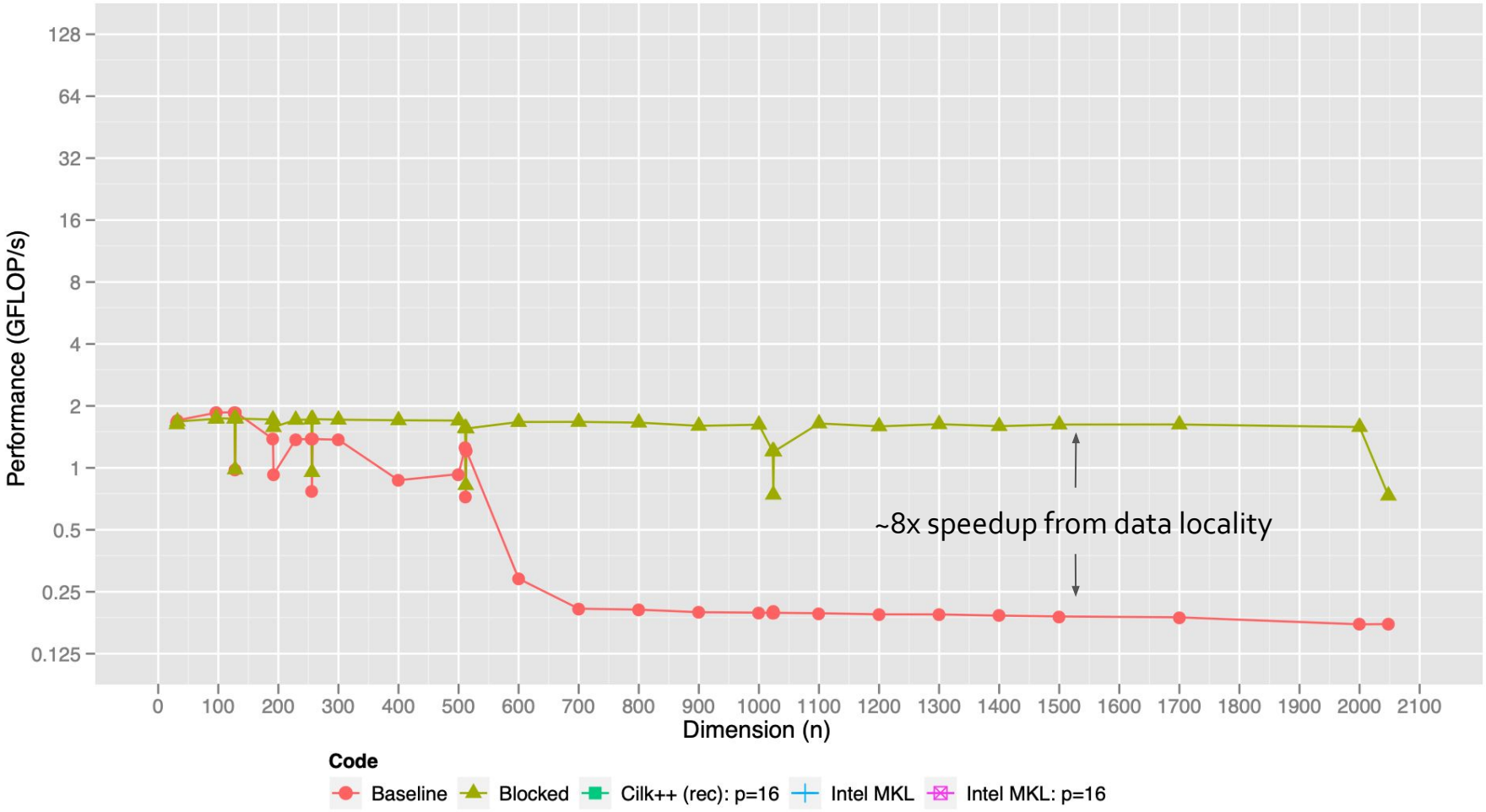
Theory vs. Practice

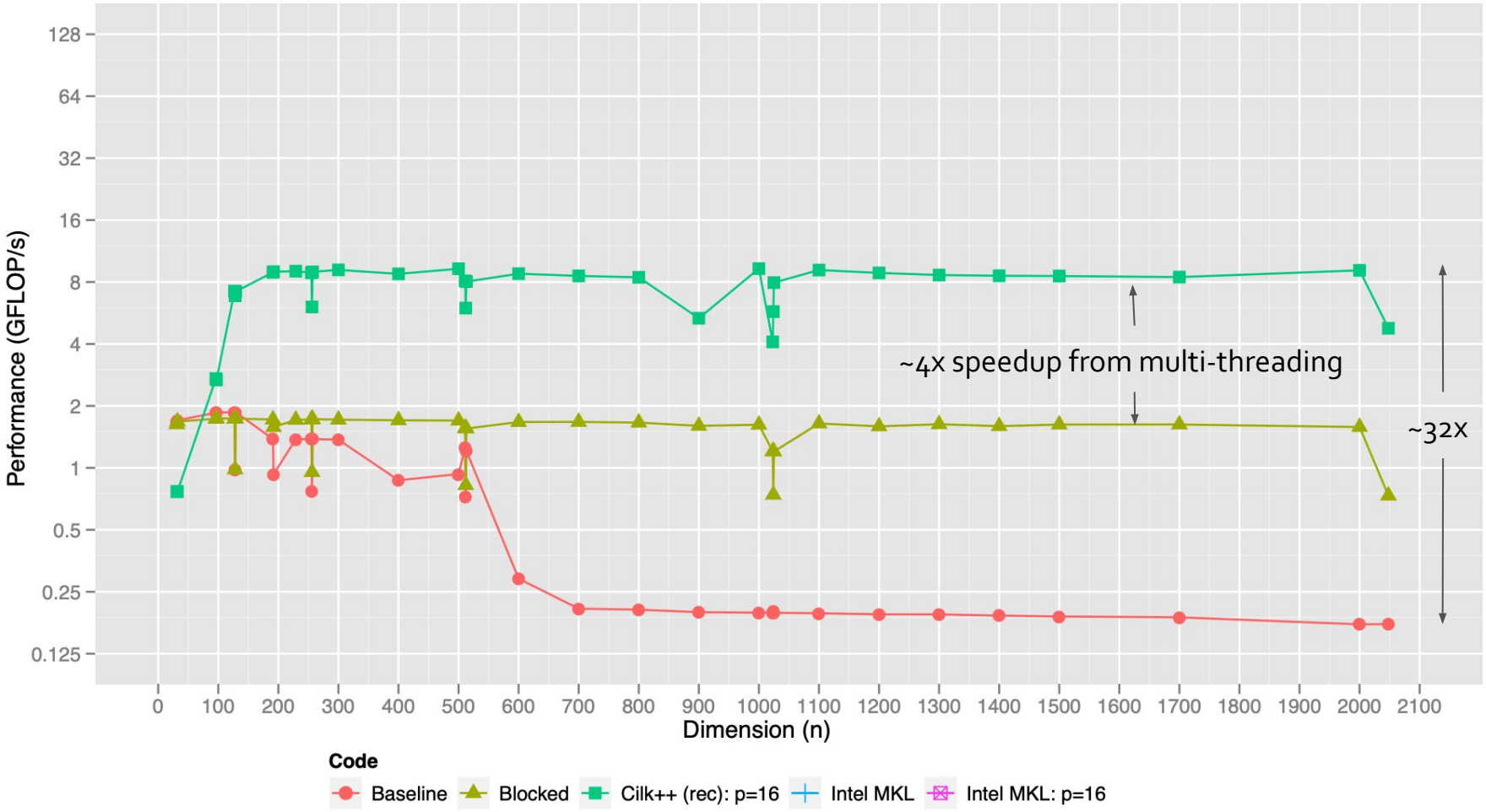
Example: **Matrix multiply** (non-Strassen)

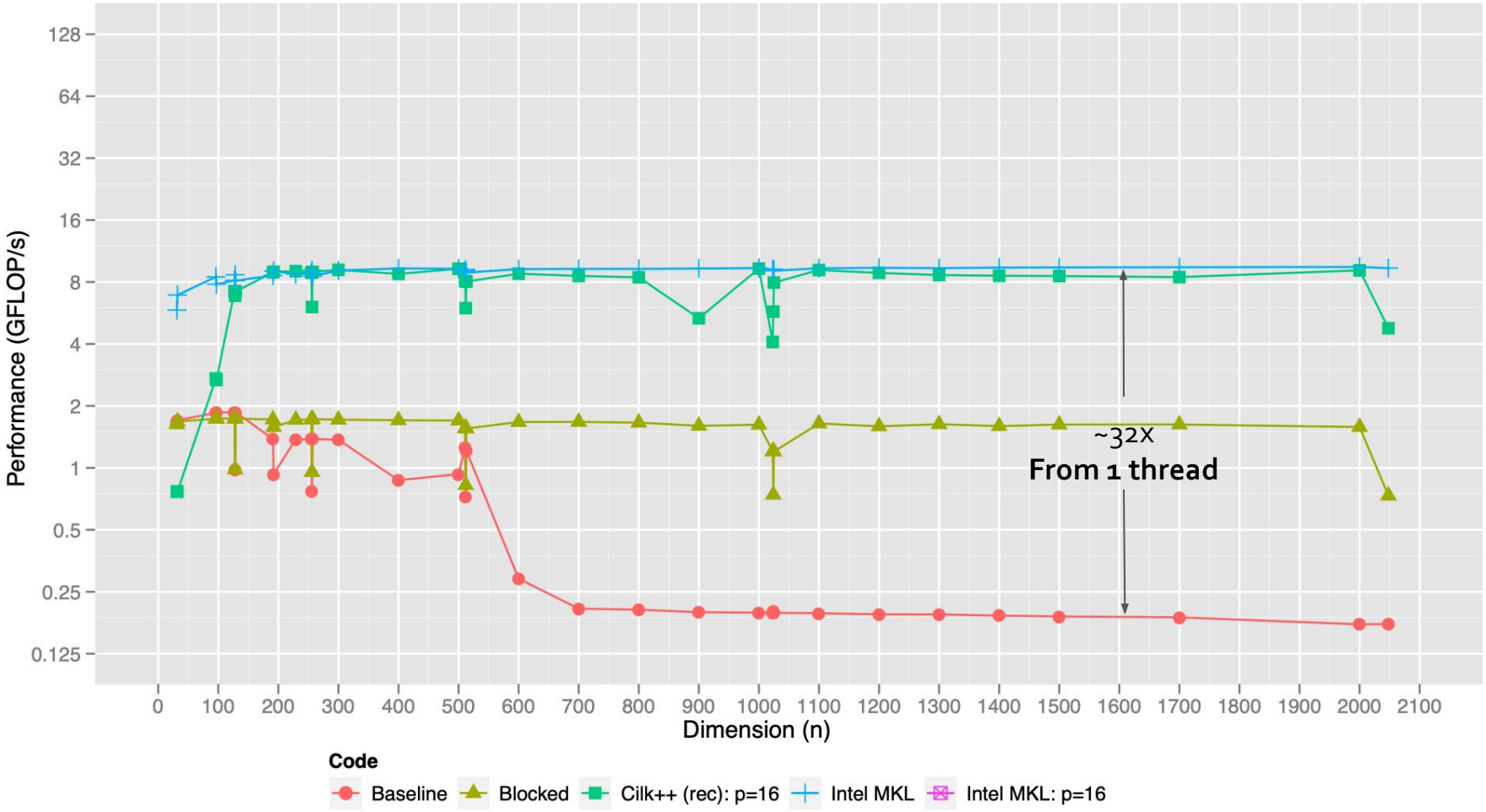


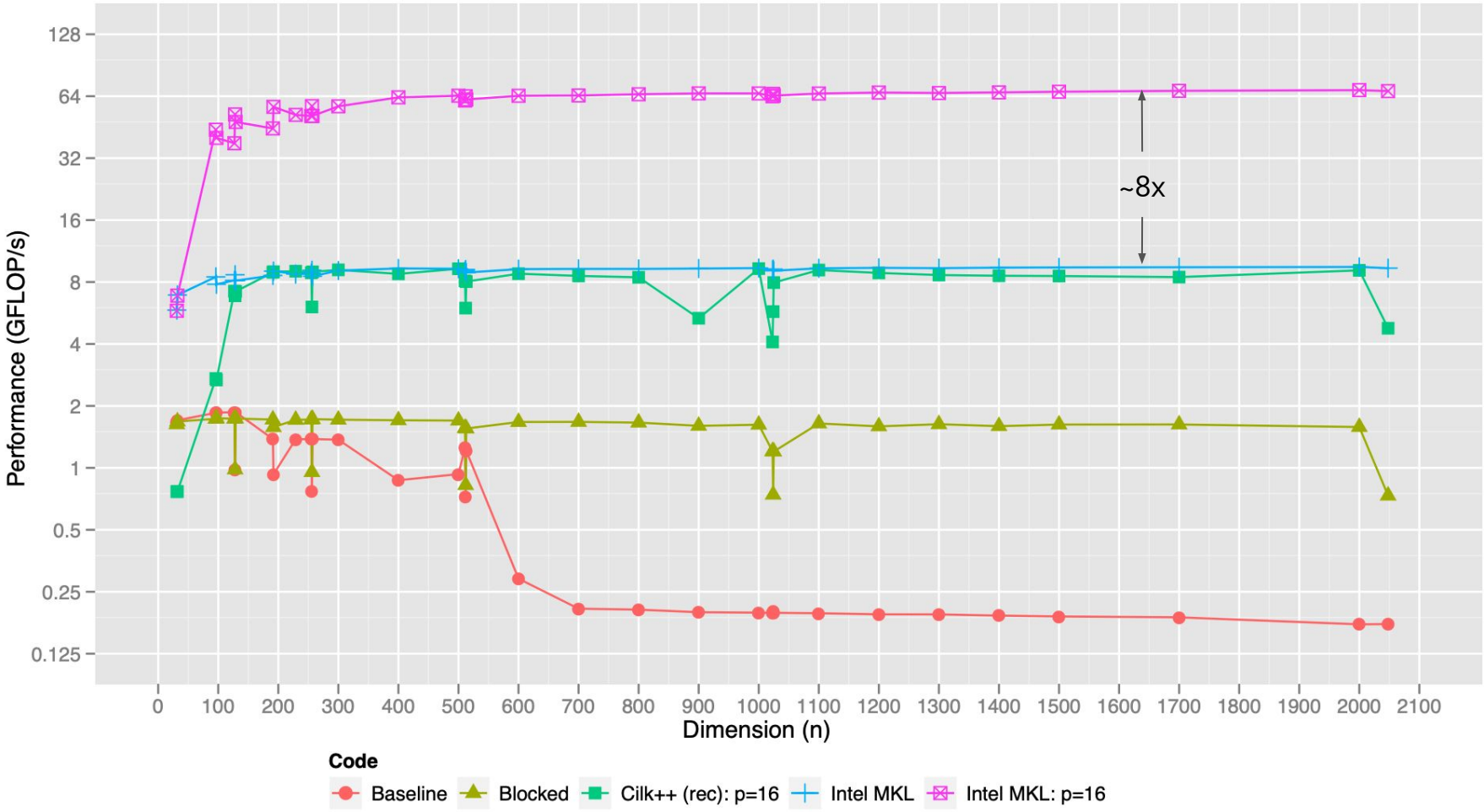

```
for i = 1 to n do
  for j = 1 to n do
    for k = 1 to n do
      C[i,j] ← C[i,j] + A[i,k] · B[k,j]
```


3-nested loops +
best compiler optimizations
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)
~8x speedup from data locality
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)
~4x speedup from multi-threading
~32x
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)
~32X
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
~8x
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

Large and complex tuning space

M. Dukhan & M Schornak [CSE 6230, Fall 2011] **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
// 256  640   98.2
// 256  612   99.6
// 256  576   97.2
// 240  640   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 i_max = k_size & (-N_R);
    long j_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 + * mr_size] = A[j * stride + i];
            //~ A_copy[i * kc_size + * mr_size + 1] = A[j * stride + i + 1];
            //~ A_copy[i * kc_size + * mr_size + 2] = A[j * stride + i + 2];
            //~ A_copy[i * kc_size + * 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 * j_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

    movddup xmm13, [r11 + rbx * 1]
    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
```

Can Compilers Do This?

Proebsting's Law

Compilers double code performance every **18 years**.

vs. **2 years** for Moore's Law and **3 years** for memory

OpenMP

OpenMP - Open Multi-Processing

API that supports multi-platform parallel shared-memory multiprocessing programming in C, C++, etc.

Uses #pragma to denote places that can be parallelized

OpenMP Syntax

Most OpenMP constructs are compiler directives

#pragma omp <directive> [clause …]

```
#pragma omp parallel default(shared) private(a, b)
```

Library Functions

Thread queries (number of threads, thread ID, etc.)

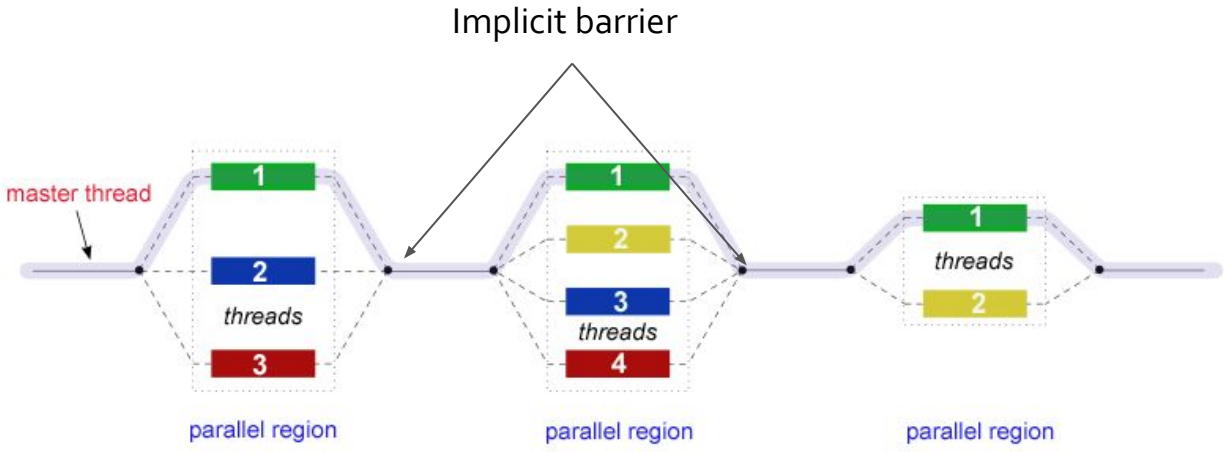
```
int omp_get_num_threads(void)
```

Environment Variables

Setting number of threads, affinity, etc.

```
export OMP_NUM_THREADS=8
```

OpenMP Model



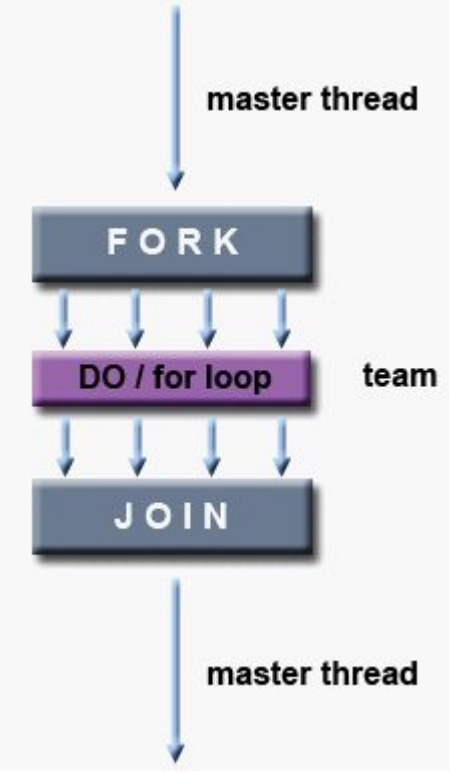
Work-Sharing - Do/For

Share iterations of the loop across the threads (i.e., data parallelism)

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < 100; i++) {
        x[i] = 1;
    }
}
```

OR

```
#pragma omp parallel for
for(int i = 0; i < ARR_SIZE; i++) {
        x[i]++;
}
```



Also an implicit barrier at the end of the loop

OpenMP Directives

`schedule (type, chunk size)`

Specify how the threads are assigned to the loop iterations

`type`

- `static`
- `dynamic`
- `guided`
- `runtime`
- `Auto`

`chunk size` – denotes how many loop iterations are assigned to each thread at a time

Kind	Description
static	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is loop_count/number_of_threads.Set chunk to 1 to interleave the iterations.
dynamic	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Be careful when using this scheduling type because of the extra overhead involved.
guided	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use. By default the chunk size is approximately loop_count/number_of_threads.
auto	When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.
runtime	Uses the OMP_schedule environment variable to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as would appear on the parallel construct.

Next Homework

Quicksort

- You should all be familiar with this
- Recursively…
  - **Partition** your list around a pivot - smaller numbers go on the left, larger numbers go on the right
  - Call quicksort on your left and right sides of the array

Is there a different way of partitioning the data?

Partitioning

***A*:**	9	5	7	11	1	3	8	14	4	21	***x* = 8**

Partitioning

Is 9 less than 8?
Yes = 1
No = 0

***A*:**	9	5	7	11	1	3	8	14	4	21	***x* = 8**

	0	1	2	3	4	5	6	7	8	9
***B*:**	9	5	7	11	1	3	8	14	4	21

	0	1	2	3	4	5	6	7	8	9
***lt*:**	0	1	1	0	1	1	0	0	1	0

	0	1	2	3	4	5	6	7	8	9
***gt*:**	1	0	0	1	0	0	0	1	0	1

Partitioning

***A*:**	9	5	7	11	1	3	8	14	4	21	***x* = 8**

	0	1	2	3	4	5	6	7	8	9
***B*:**	9	5	7	11	1	3	8	14	4	21

	0	1	2	3	4	5	6	7	8	9
***lt*:**	0	1	1	0	1	1	0	0	1	0
	0	1	2	2	3	4	4	4	5	5

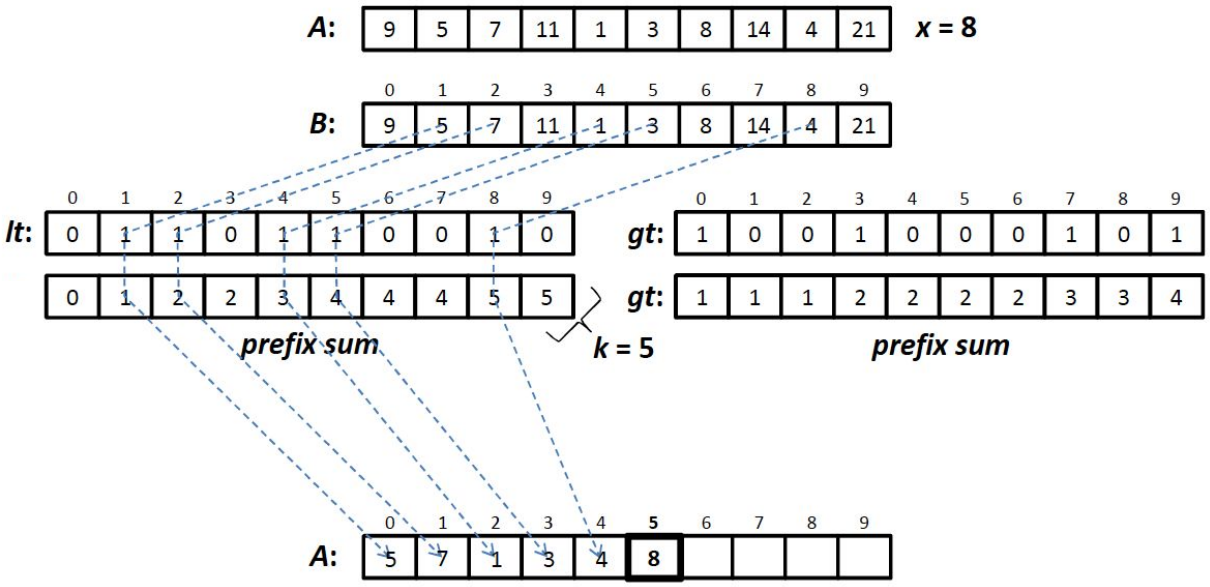
prefix sum

	0	1	2	3	4	5	6	7	8	9
***gt*:**	1	0	0	1	0	0	0	1	0	1
***gt*:**	1	1	1	2	2	2	2	3	3	4

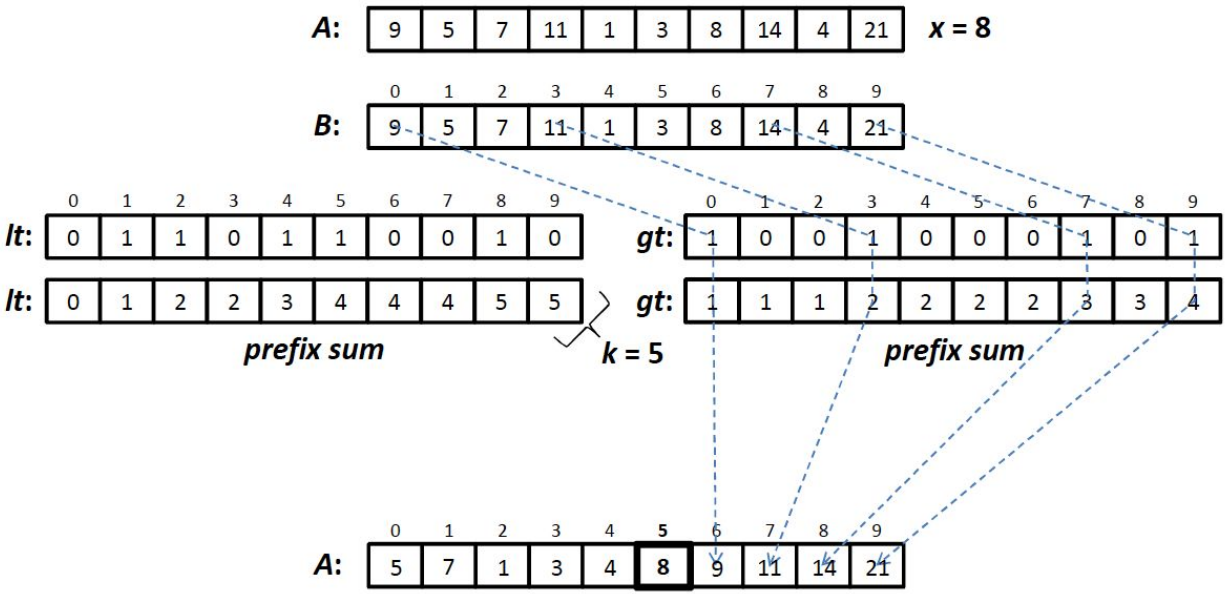
prefix sum

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$
$$\ldots$$

Partitioning



Partitioning



Partitioning

Homework 8 (last one)

- Implement naive quicksort
- Implement quick sort with this new partitioning method
  - Parallelize it with OpenMP
  - The code WILL NOT be faster on ix-dev, but it should work correctly regardless of the # of threads (i.e., OMP_NUM_THREADS)
  - Try up to 1M elements to make sure it works correctly