CIS 431/531
Intro to Parallel Computing

Lecture 1

Why do we care about parallelism?

What? (Goal)

Learn practical parallel programming (i.e., how to write ***fast*** code).

Why?

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

How?

Parallelism

Data Locality

Specialization

Motivation

Currently, parallelism is driven by ***power***, ***memory***, and ***physics***.

Performance & Power

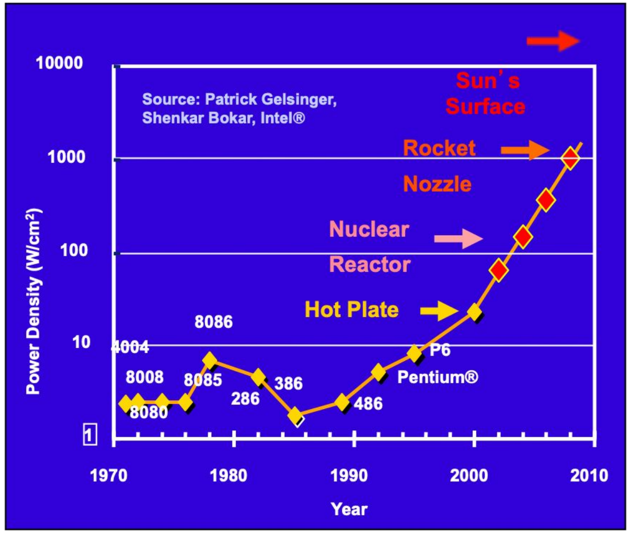
Performance $\propto$?

Performance & Power

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

Power $\propto$?

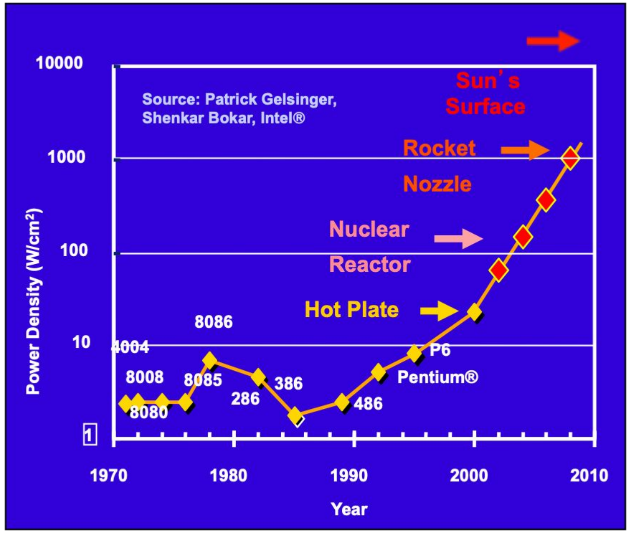
Performance & Power



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

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

Performance & Power

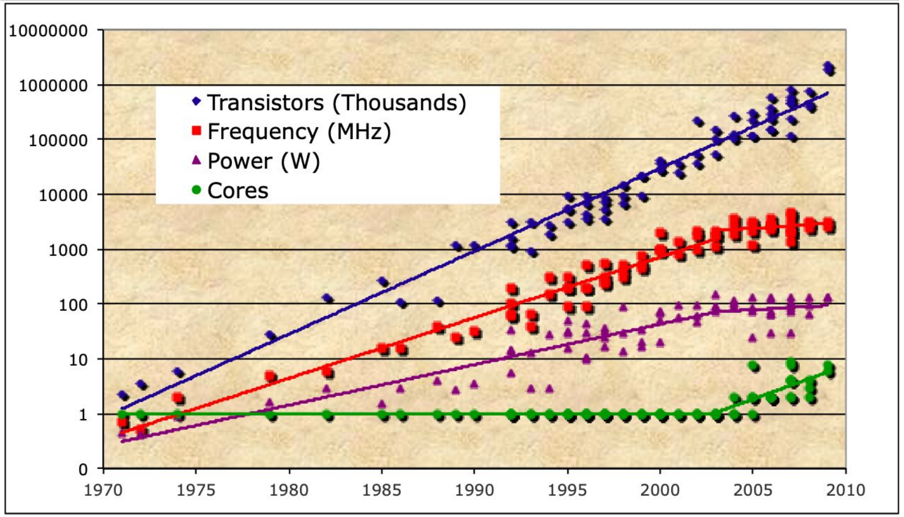


Performance ∝ (cores) × (freq)

Power ∝ (cores) × ($\text{freq}^{2.5}$)

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

Parallelism & Power



Notice the transition around **2004**.

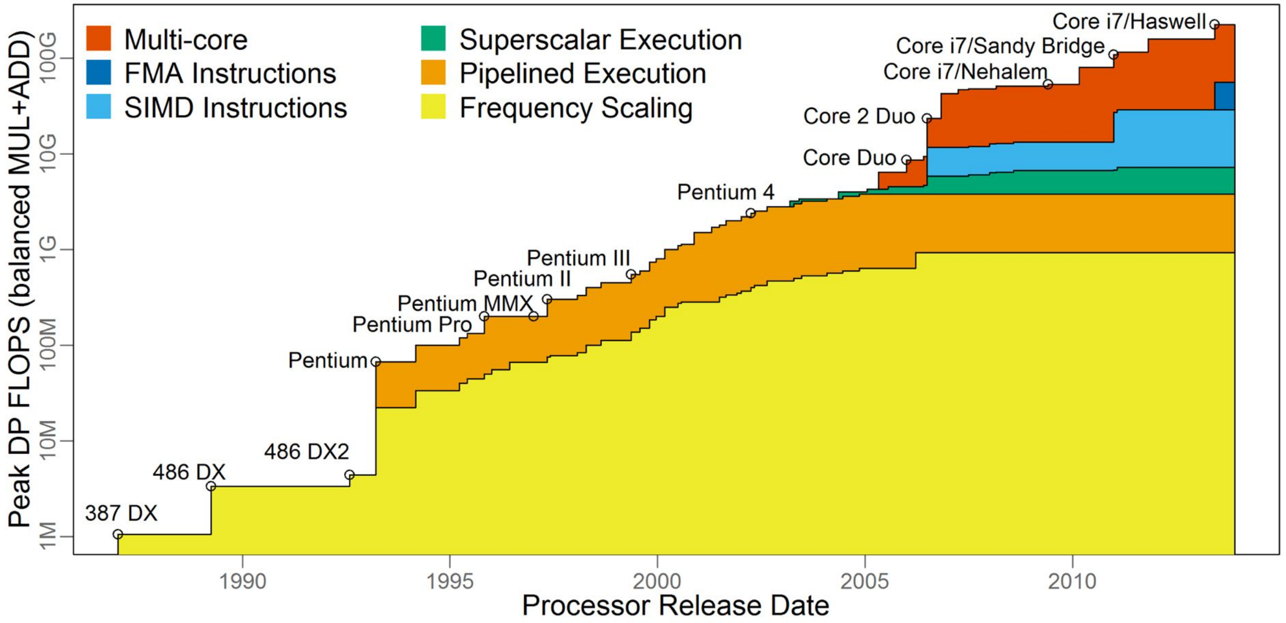
End of ***Dennard Scaling***.

Parallelism & Power

Conclusion

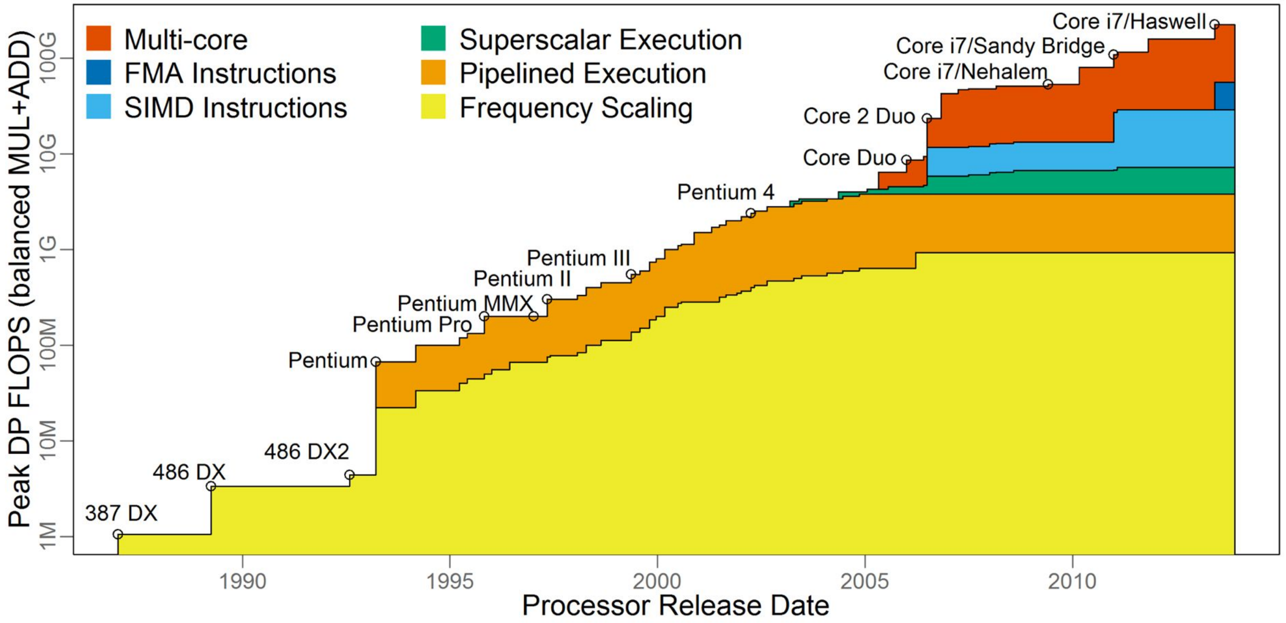
- We **HAVE** to rely on increasing the number of cores to increase the performance of a processor
- This means that **programmers** have to write code that can take advantage of parallelism to better utilize processors

Performance Factors



What else has been done to increase performance?

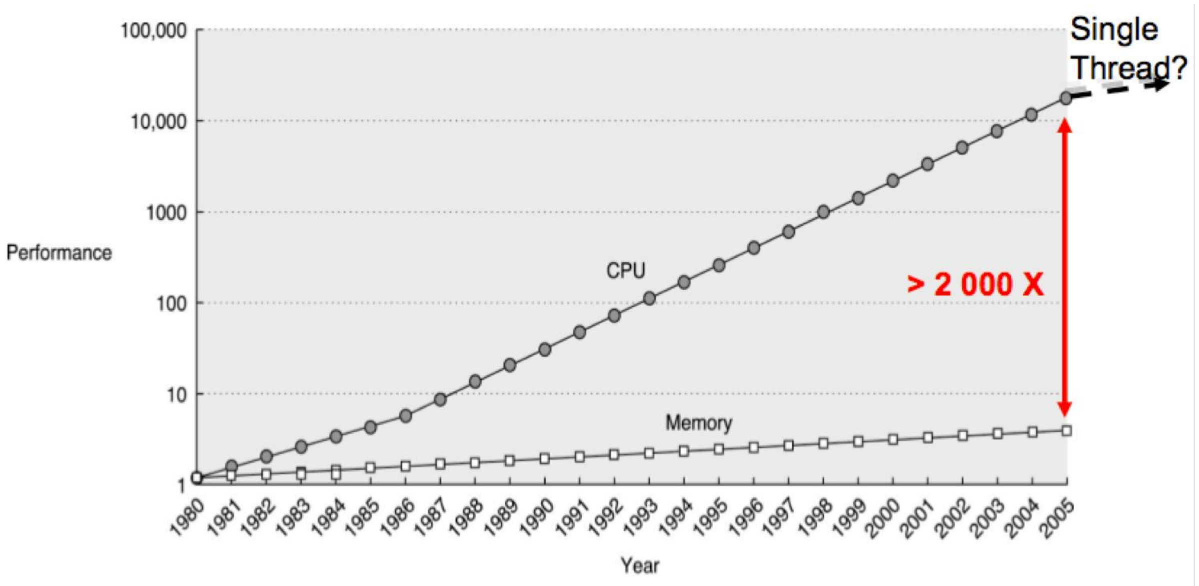
Performance Factors



What else has been done to increase performance?

Low-precision operations, tensor cores, neural engines, etc.

Memory



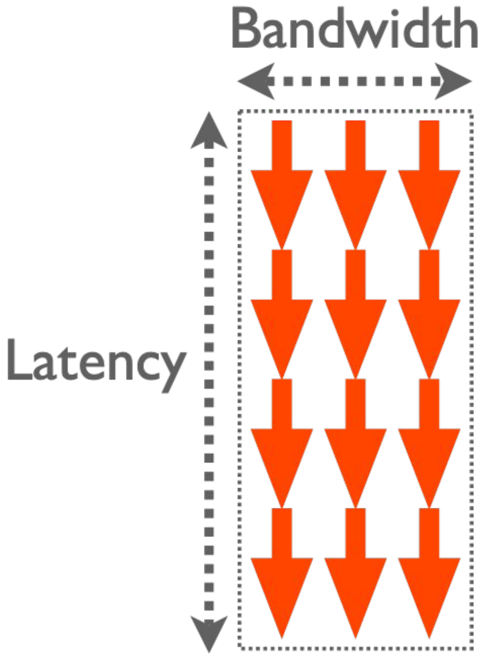
Memory performance has scaled much more slowly than instruction performance (i.e., FLOPS)

Historically, memory

latency halves every ~9 years

bandwidth doubles every ~ 3 years

Memory



Little's Law (queuing theory)

$$L = \lambda W$$

Equivalently,

$$\text{Concurrency} = \text{Latency} \times \text{Bandwidth}$$

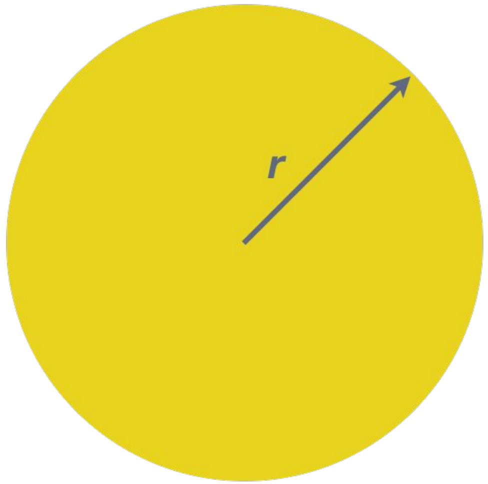
That is, larger the latency and bandwidth, more concurrency (i.e., threads) is required to fully utilize the memory bandwidth.

Memory

Conclusion

- To keep the increased number of processing units busy, we must deliver data at a higher rate (i.e., higher bandwidth).
- However, higher bandwidth requires having more memory requests in-flight, which can only happen if you can find more independent threads of execution (i.e., increased parallelism).

Physical Limits



Size of a modern processor - 40mm x 40mm -> r = 20mm

Speed of light = 3 x 10^8 m/s

Time to go from one end of a chip to the other t = (40x10^{-3} m)/(3 x 10^8 m/s) = 1.33 x 10^{-10} seconds

One operation CANNOT take more than 1.33 x 10^{-10} seconds.

Equivalently, clock frequency cannot exceed 1/(1.33 x 10^{-10}) = 7.5 GHz

Latest Intel I9-11900K runs at 5.3 GHz

Physical Limits

1. Quantum mechanics (e.g., quantum tunneling effect)
2. EUV (extreme ultraviolet) lithography

Conclusion

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

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

Apple A17 Pro System-on-chip (SoC)

- ***3nm*** *process*
- **19 Billion** transistors
- **2x** performance cores (up to 3.78 GHz)
- **4x** power-efficiency cores (up to 2.11 GHz)
- **6x** GPU cores (up to 1.4 GHz)
- **16x** Neural Engine cores
- **~8 Watts**

Intel Core i9-13900KS

10nm *process*

8x performance cores (up to 5.4 GHz)

16x power-efficiency cores (up to 4.3 GHz)

6 GHz single core boost frequency

32x GPU cores (up to 1.65 GHz)

Gaussian & Neural accelerator

~125 Watts

FRONTIER
OAK RIDGE National Laboratory
U.S. DEPARTMENT OF ENERGY
Hewlett Packard Enterprise
AMD

Frontier Supercomputer (Department of Energy)

- ***1.102 exaFLOPs*** *(quintillion or 10^18)*
- **9,472x** AMD Epyc 7453 CPUs
  - 606,208 cores @ 2GHz
- **37,888x** AMD Radeon Instinct MI250x GPUs
  - 8,335,360 cores @ up to 1.7 GHz
- **21 MWatts**

Beyond Parallelism

To achieve extreme performance, more is needed - **data locality** and **specialization**.

- Data locality is necessary since bandwidth is much slower than processor performance
- What about specialization?

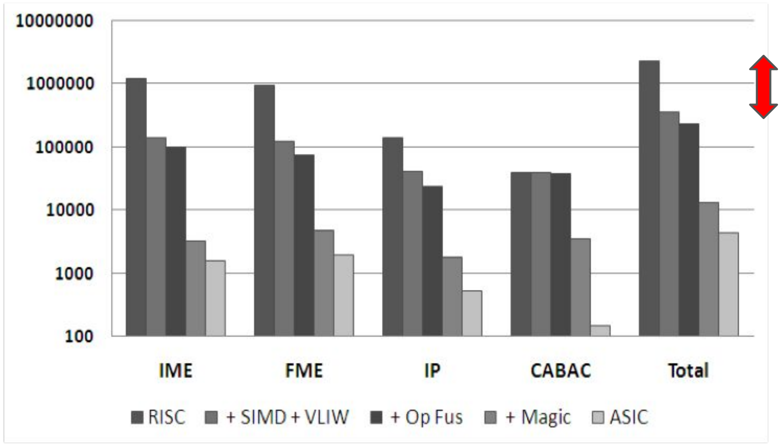


Figure 2. Each set of bar graphs represents energy consumption (μJ) at each stage of optimization for IME, FME, IP and CABAC respectively. Each optimization builds on the ones in the previous stage with the first bar in each set representing RISC energy dissipation followed by generic optimizations such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

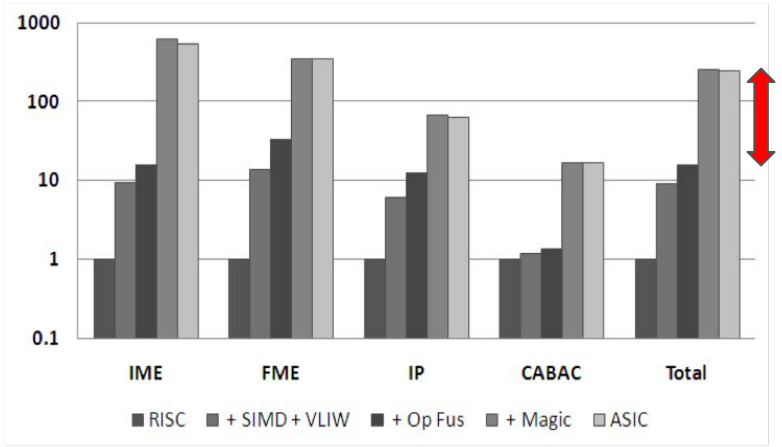


Figure 3. Each set of bar graphs represents speedup at each stage of optimization. Each optimization builds on those of the previous stage with the first bar in each set representing RISC speedup, followed by generic optimizations such as SIMD and VLIW, then operation fusion and finally "magic" instructions

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

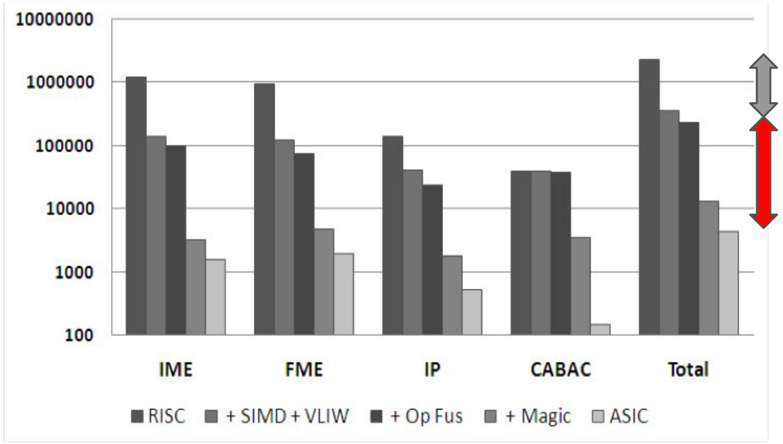


Figure 2. Each set of bar graphs represents energy consumption (μJ) at each stage of optimization for IME, FME, IP and CABAC respectively. Each optimization builds on the ones in the previous stage with the first bar in each set representing RISC energy dissipation followed by generic optimizations such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

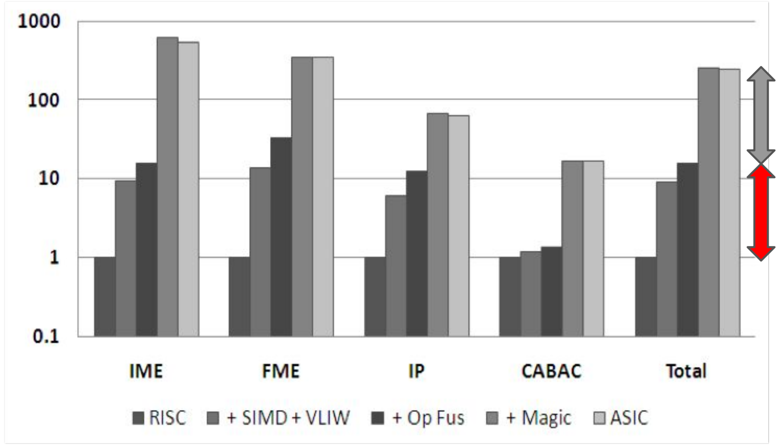


Figure 3. Each set of bar graphs represents speedup at each stage of optimization. Each optimization builds on those of the previous stage with the first bar in each set representing RISC speedup, followed by generic optimizations such as SIMD and VLIW, then operation fusion and finally "magic" instructions

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

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

Specialization

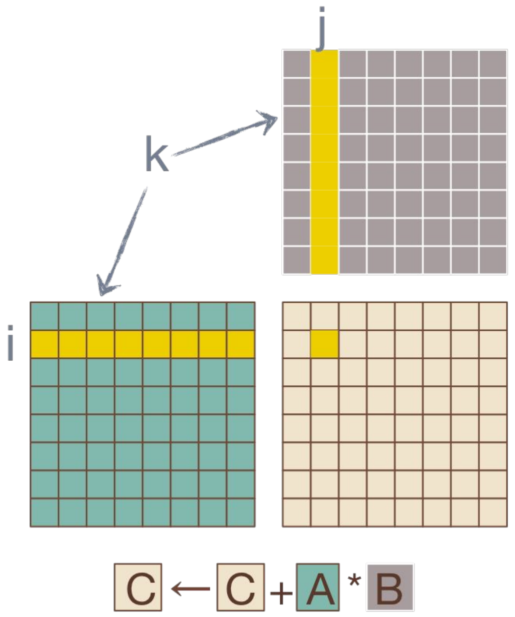
1. Application-specific integrated circuit (ASIC) is **500x** more energy efficient than general-purpose chip multi-processors (CMP)
2. Over **90%** of energy is "**overhead**" (e.g., instruction fetch/decode, etc.) because the cost of actual FLOP is very cheap.

Conclusion

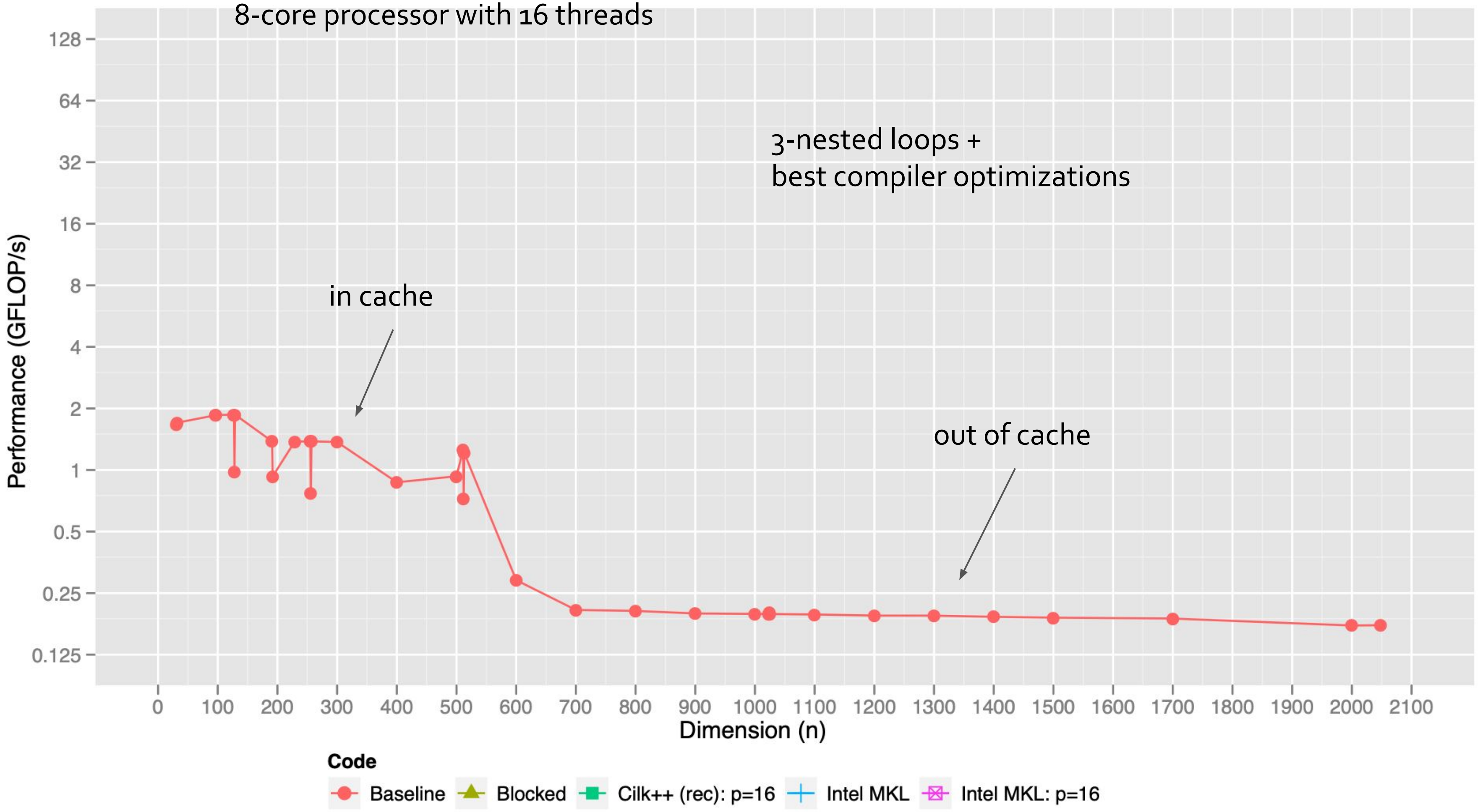
These observations imply we need to pay attention to **data movement** and exploit **custom units** (e.g., GPUs) to improve energy efficiency and performance (in addition to parallelization)

Theory vs. Practice

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

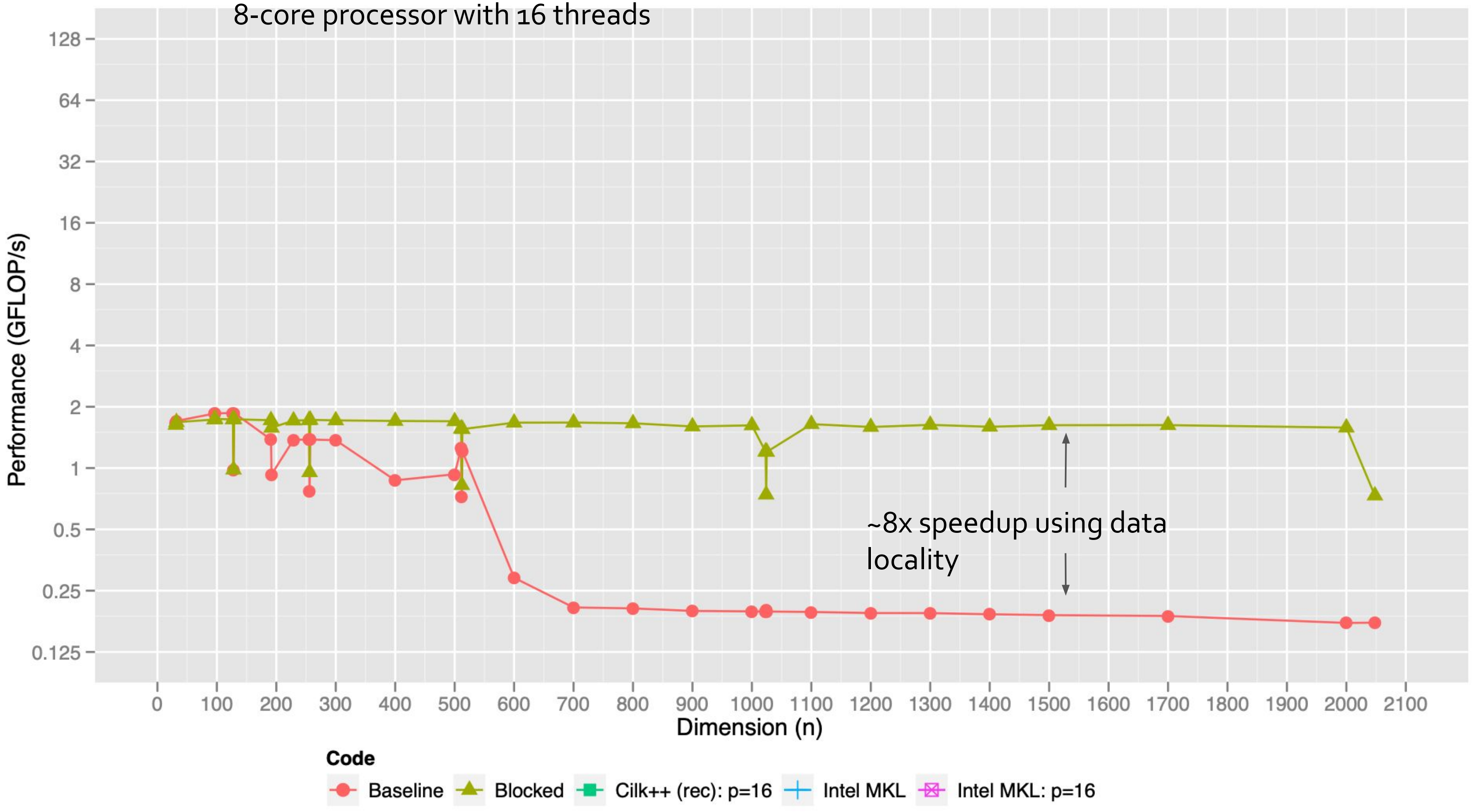


```
for i = 1 to n do
  for j = 1 to n do
    for k = 1 to n do
      C[i,j] ← C[i,j] + A[i,k] · B[k,j]
```

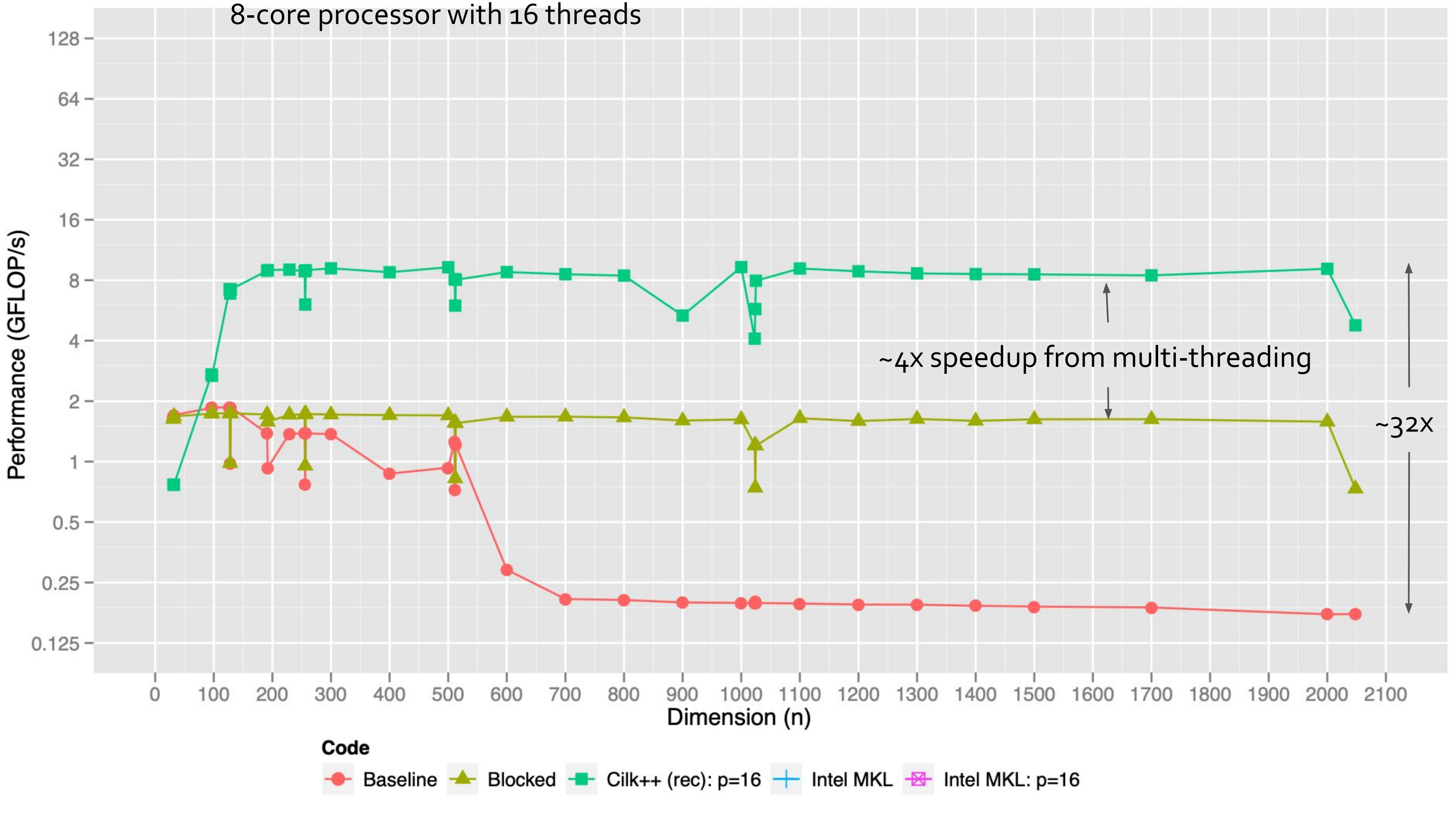


8-core processor with 16 threads
3-nested loops +
best compiler optimizations
in cache
out of cache
Performance (GFLOP/s)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
Dimension (n)
Code
Baseline
Blocked
Cilk++ (rec):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

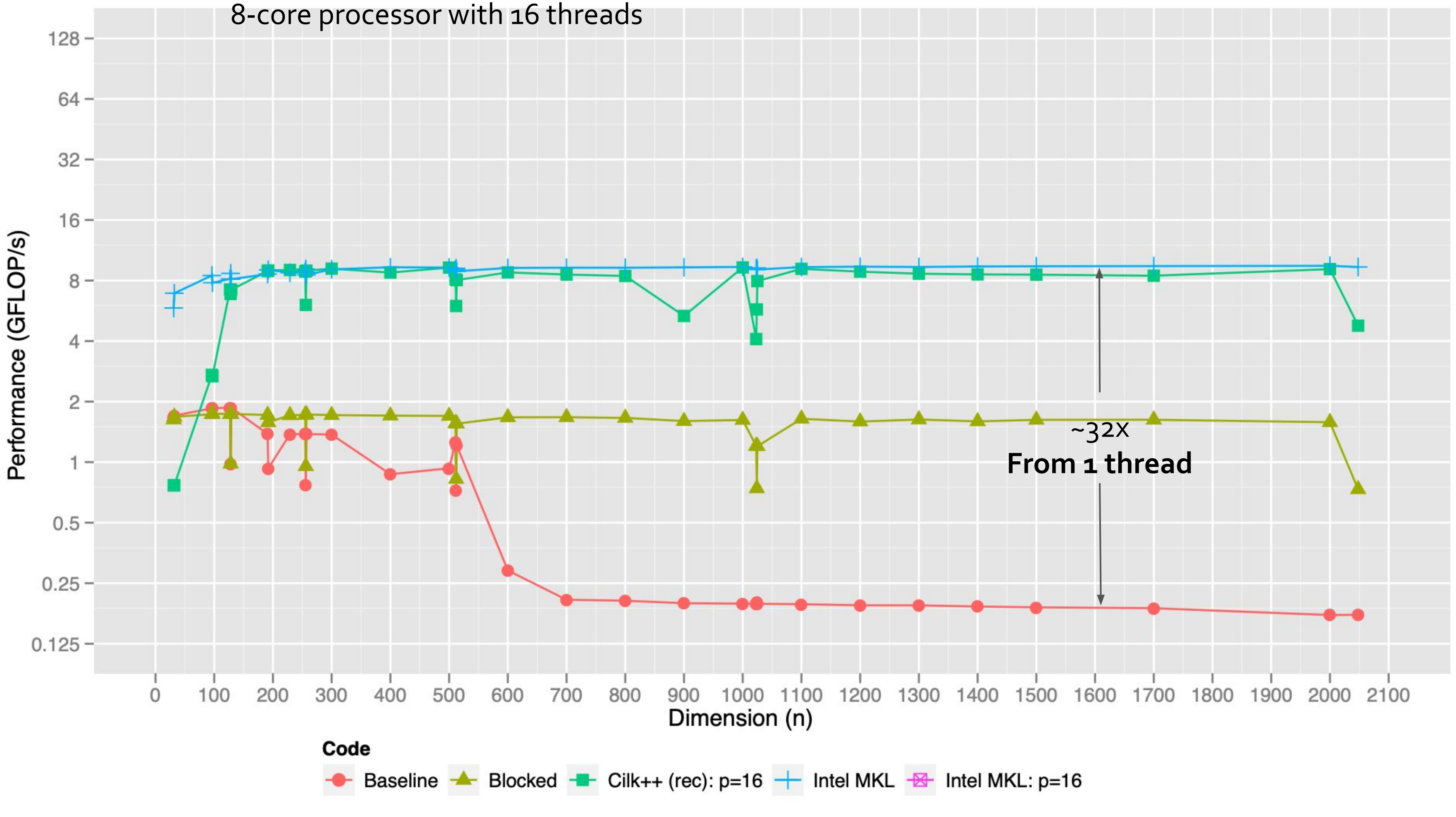
8-core processor with 16 threads

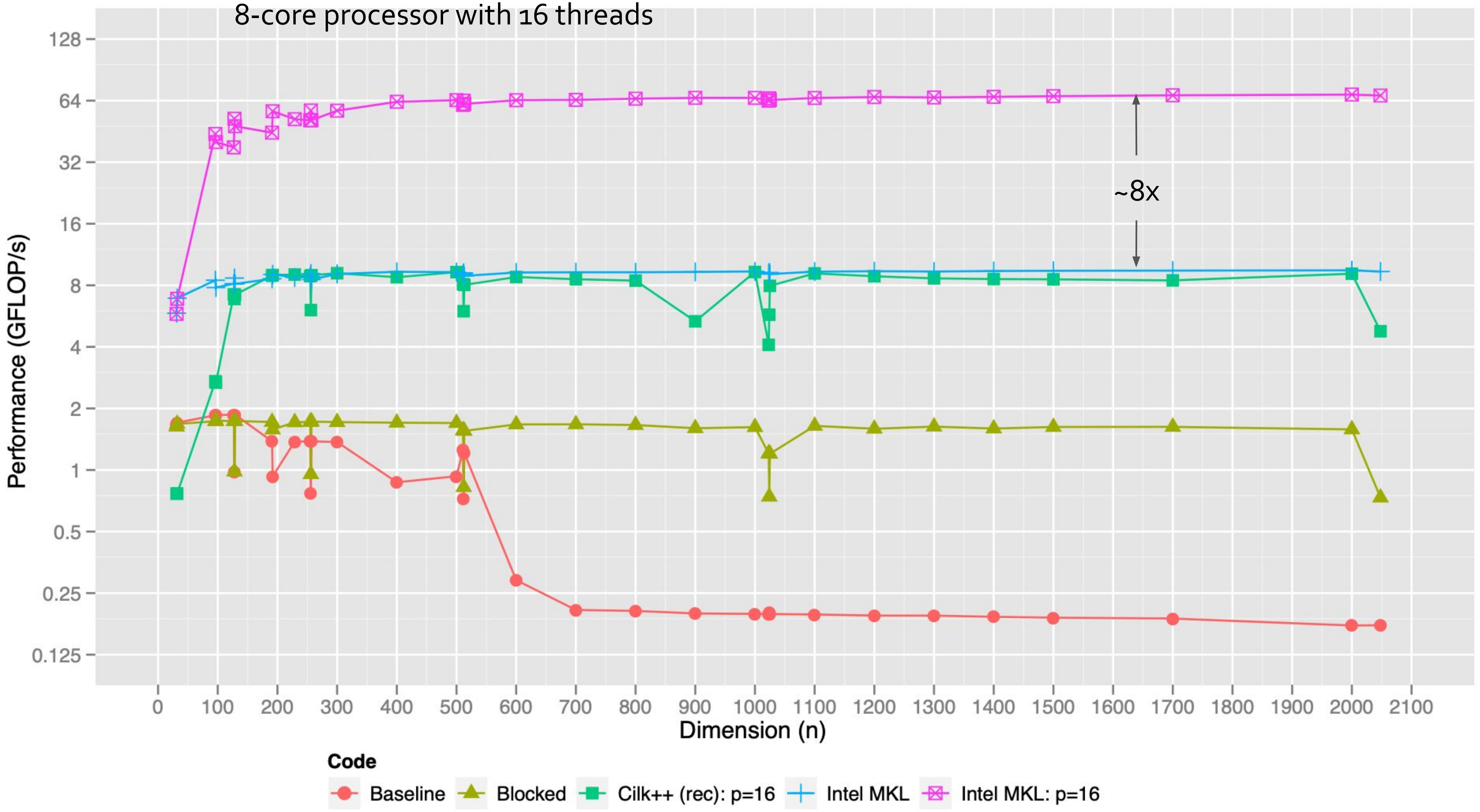


~8x speedup using data locality


8-core processor with 16 threads
~4x speedup from multi-threading
~32x
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

8-core processor with 16 threads




8-core processor with 16 threads
~8x
128
64
32
16
8
4
2
1
0.5
0.25
0.125
Performance (GFLOP/s)
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

```c
/** 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 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

   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 transistors (Moore's Law) and **3 years** for memory

Questions

Misc.

- All coding **must be** version controlled using **git** on **BitBucket**.
  - **Commit your code frequently!**
- All reports must be written using **Latex**.

Syllabus