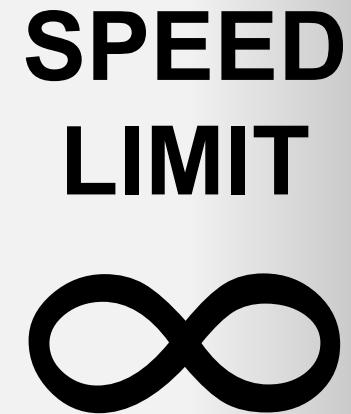


6.172 Performance Engineering of Software Systems



LECTURE 1 Introduction & Matrix Multiplication Charles E. Leiserson

WHY PERFORMANCE ENGINEERING?



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” needed properties with performance.

Computer Programming in the Early Days

Software performance engineering was common, because machine resources were limited.

IBM System/360



Courtesy of [alihodza](#) on Flickr.
Used under CC-BY-NC.

Launched: 1964

Clock rate: 33 KHz

Data path: 32 bits

Memory: 524 Kbytes

Cost: \$5,000/month

DEC PDP-11



Courtesy of [jonrb](#) on Flickr.
Used under CC-BY-NC.

Launched: 1970

Clock rate: 1.25 MHz

Data path: 16 bits

Memory: 56 Kbytes

Cost: \$20,000

Apple II



Courtesy of [mwichary](#) on Flickr.
Used under CC-BY.

Launched: 1977

Clock rate: 1 MHz

Data path: 8 bits

Memory: 48 Kbytes

Cost: \$1,395

Many programs strained the machine's resources.

- Programs had to be planned around the machine.
- Many programs would not “fit” without intense performance engineering.

Lessons Learned from the 70's and 80's

Premature optimization is
the root of all evil. [K79]

过早的优化是万恶之源

Donald Knuth

比起任何其他单一原因（包括盲目愚蠢），以效率为名（不一定要达到效率）所犯下的计算犯罪更多。

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity. [W79]

The First Rule of Program Optimization: Don't do it.
The Second Rule of Program Optimization — For experts only: Don't do it yet. [J88]

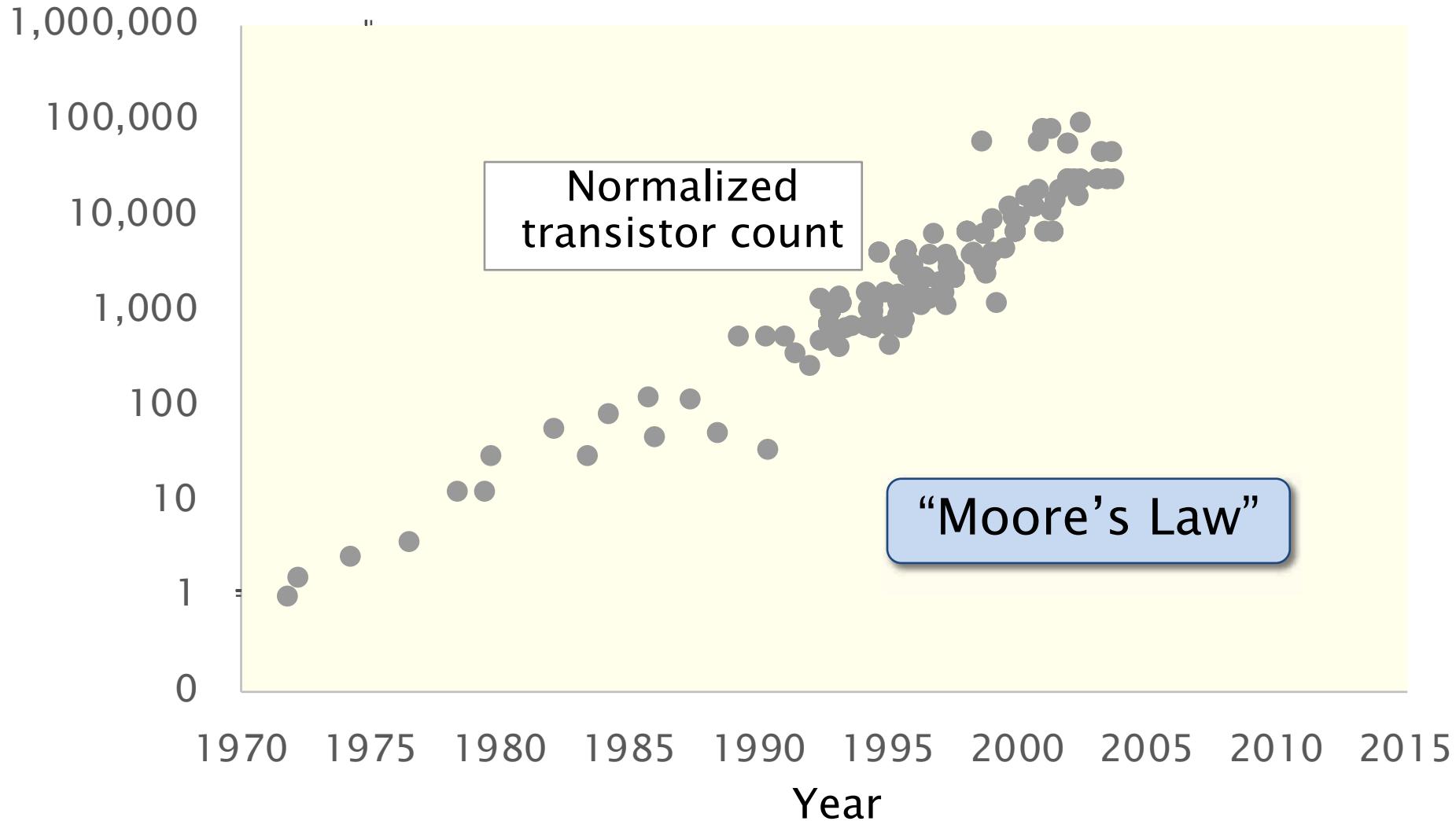
William Wulf

Michael Jackson

程序优化的第一条规则：不要这样做。

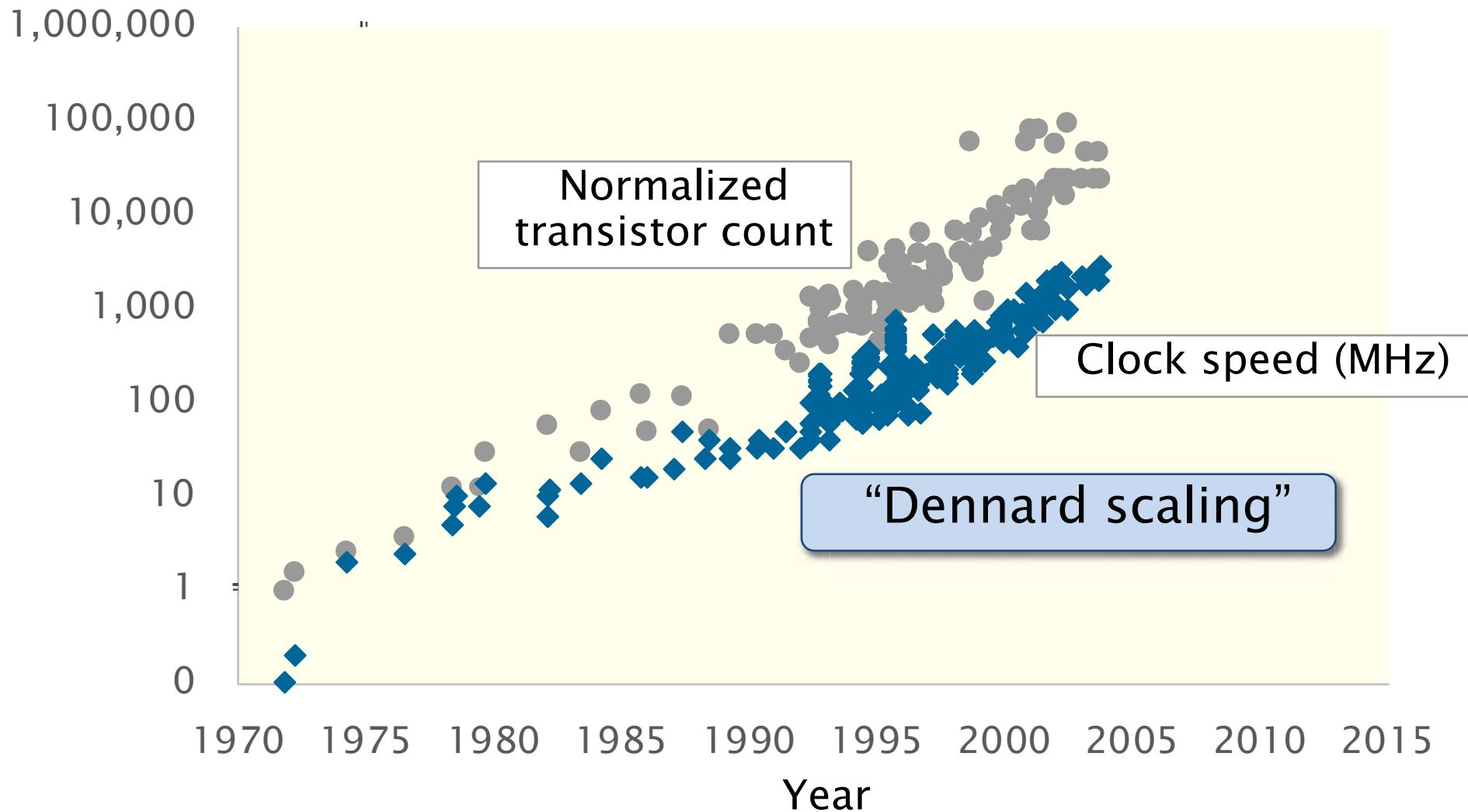
程序优化的第二条规则-仅适用于专家：请不要这样做。

Technology Scaling Until 2004



Processor data from Stanford's CPU DB [DKM12].

Technology Scaling Until 2004



Processor data from Stanford's CPU DB [DKM12].

Advances in Hardware

Apple computers with similar prices from 1977 to 2004



Courtesy of [mwichary](#) on Flickr.
Used under CC-BY.

Apple II

Launched: 1977
Clock rate: 1 MHz
Data path: 8 bits
Memory: 48 KB
Cost: \$1,395



Courtesy of [compuudemano](#) on Flickr. Used under CC-BY.

Power Macintosh G4

Launched: 2000
Clock rate: 400 MHz
Data path: 32 bits
Memory: 64 MB
Cost: \$1,599



Courtesy of [Bernie Kohl](#) on Wikipedia.
Used under CC0.

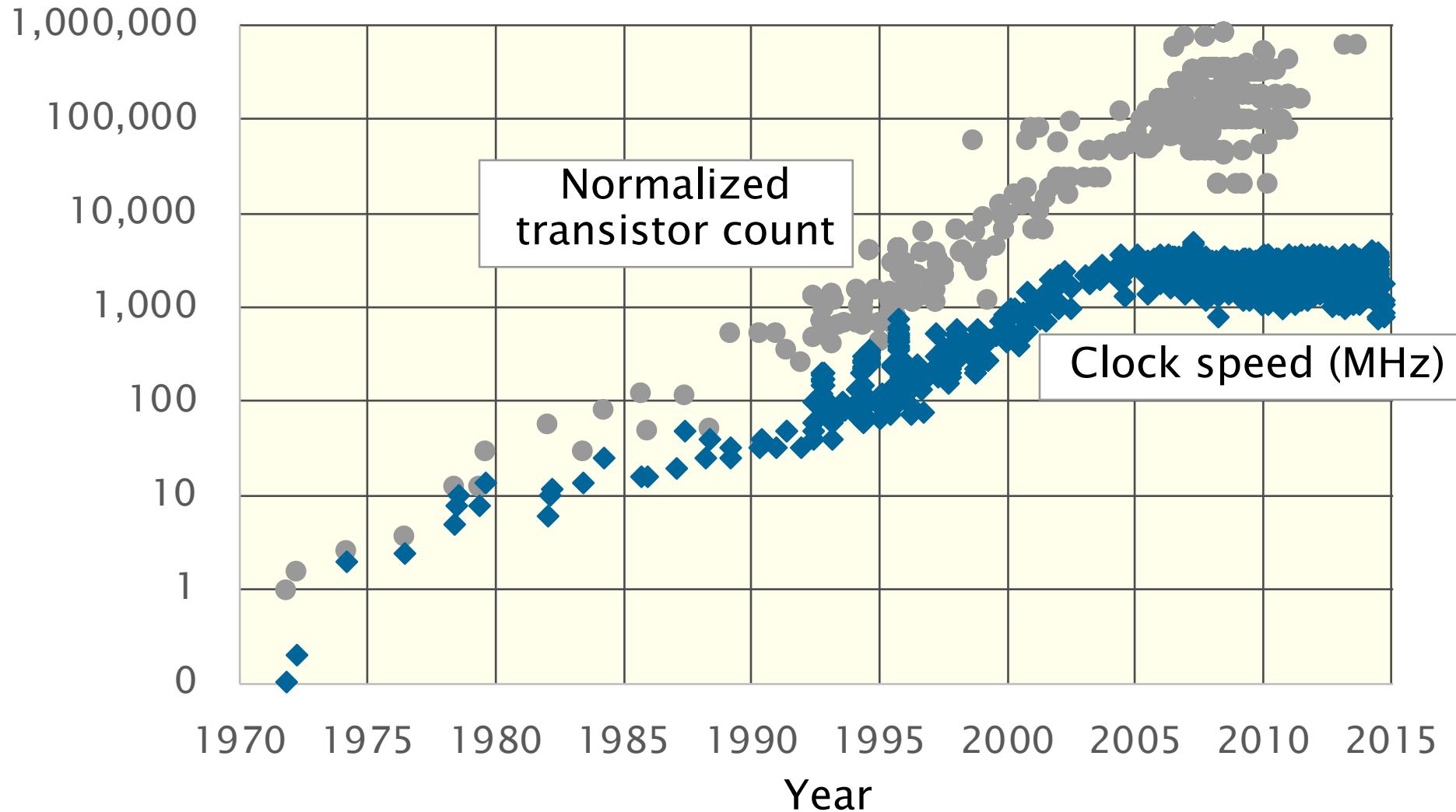
Power Macintosh G5

Launched: 2004
Clock rate: 1.8 GHz
Data path: 64 bits
Memory: 256 MB
Cost: \$1,499

Until 2004

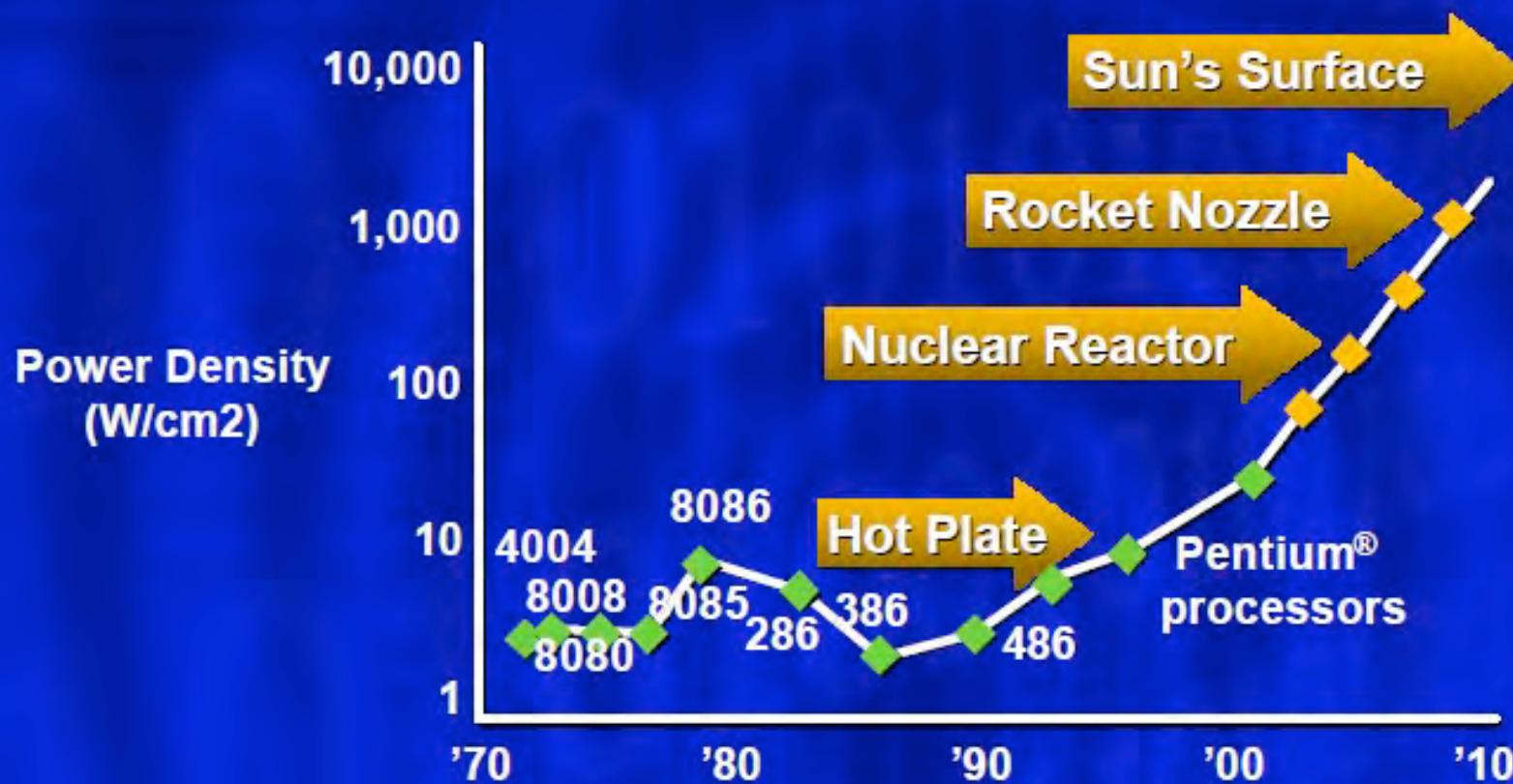
Moore's Law and the scaling of clock frequency
= printing press for the currency of performance.

Technology Scaling After 2004



Processor data from Stanford's CPU DB [DKM12].

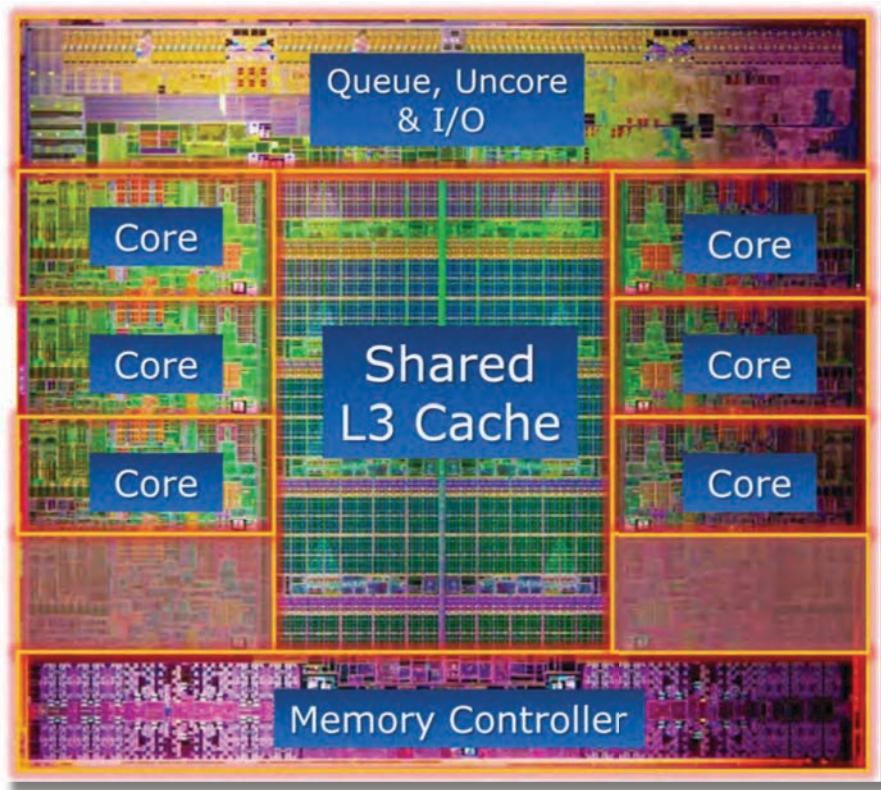
Power Density



Source: Patrick Gelsinger, *Intel Developer's Forum*, Intel Corporation, 2004.

The growth of power density, as seen in 2004, if the scaling of clock frequency had continued its trend of 25%–30% increase per year.

Vendor Solution: Multicore

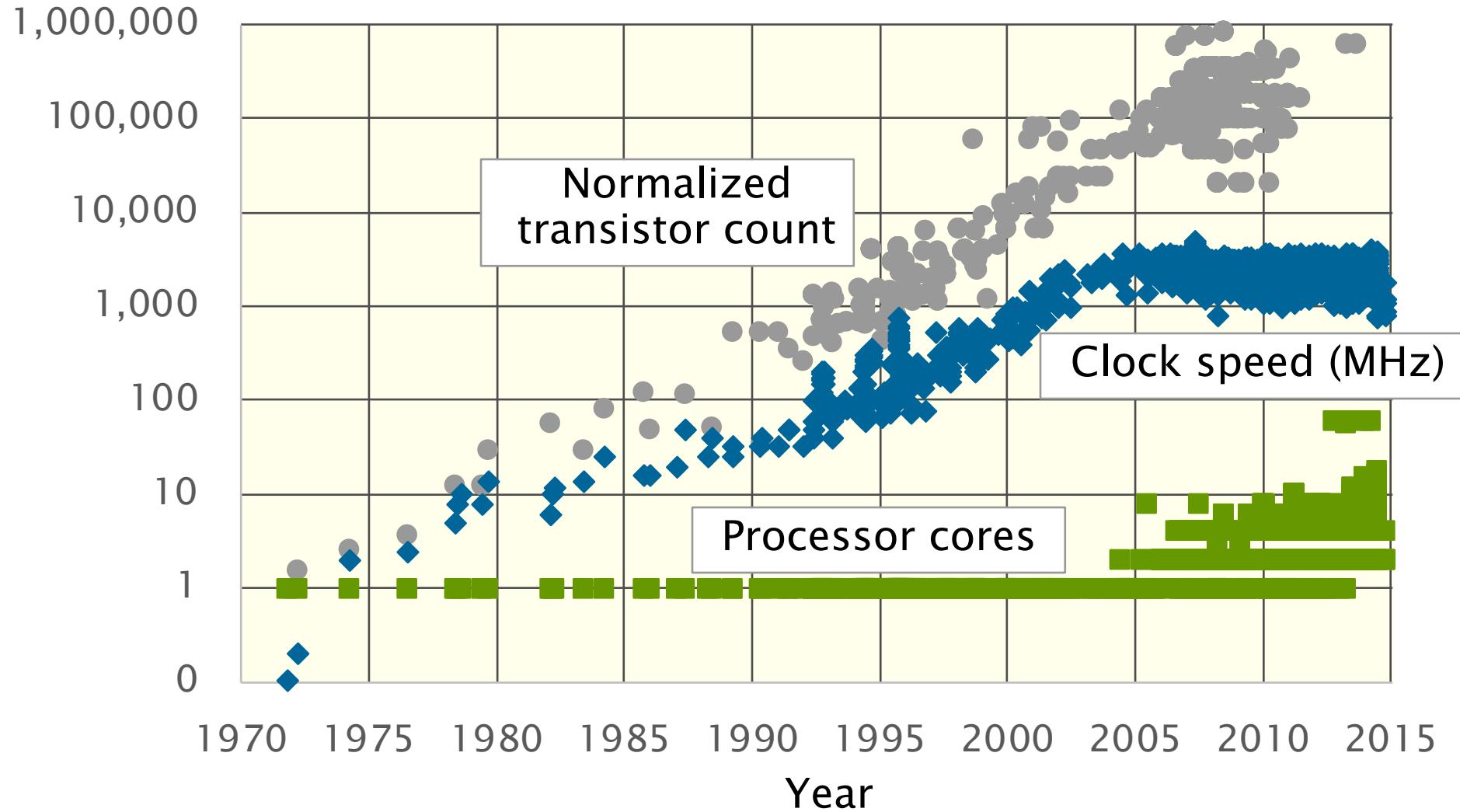


Intel Core i7 3960X
(Sandy Bridge E), 2011

- 6 cores
- 3.3 GHz
- 15-MB L3 cache

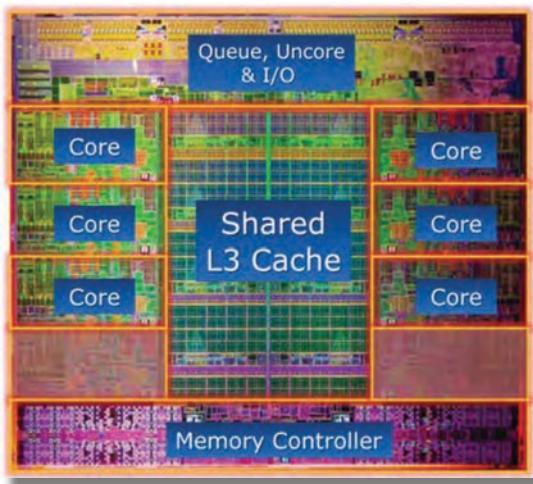
- To scale performance, processor manufacturers put many processing cores on the microprocessor chip.
- Each generation of Moore's Law potentially doubles the number of cores.

Technology Scaling



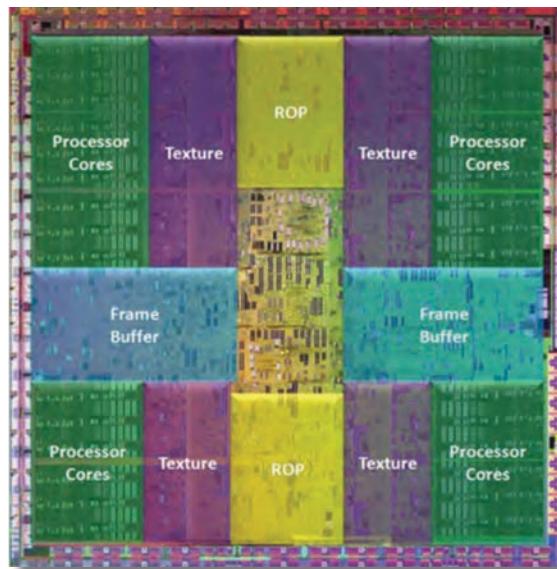
Processor data from Stanford's CPU DB [DKM12].

Performance Is No Longer Free



2011 Intel
Skylake
processor

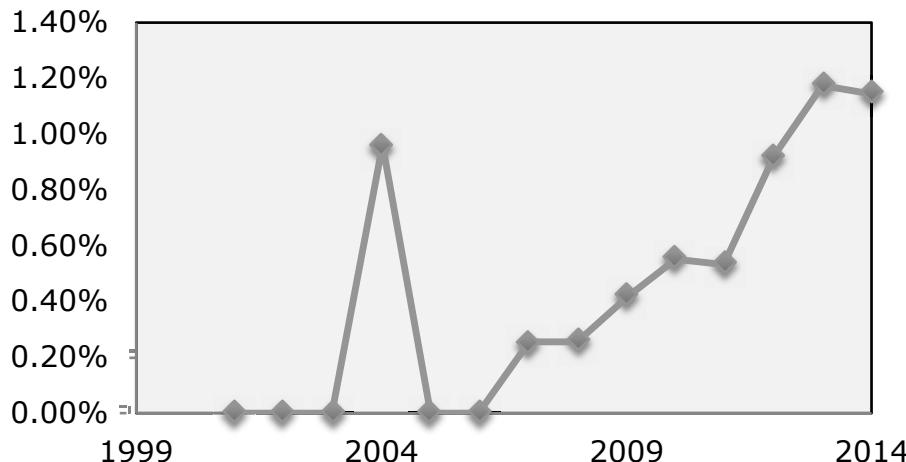
2008
NVIDIA
GT200
GPU



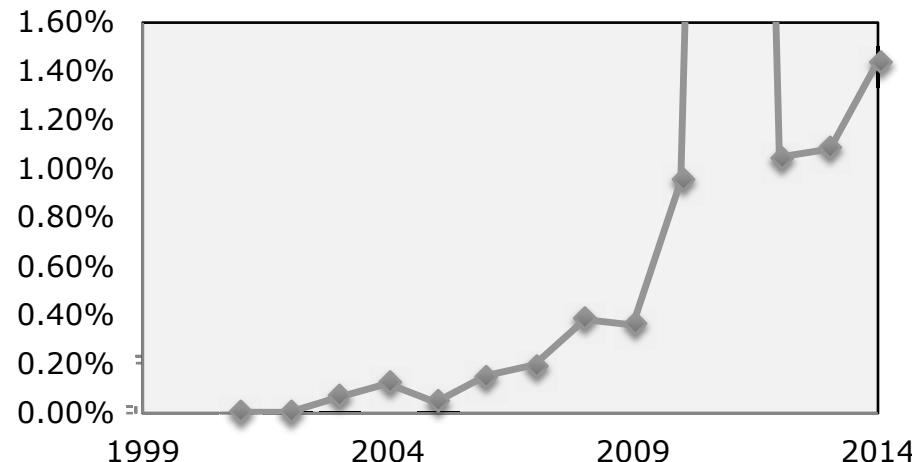
Moore's Law continues to increase computer performance. But now that performance looks like big multicore processors with complex cache hierarchies, wide vector units, GPU's, FPGA's, etc. Generally, software must be adapted to utilize this hardware efficiently!

Software Bugs Mentioning “Performance”

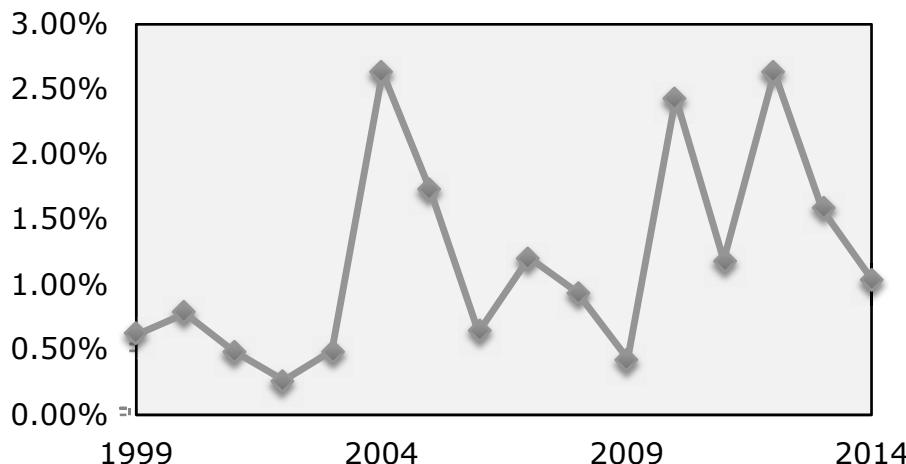
Bug reports for Mozilla “Core”



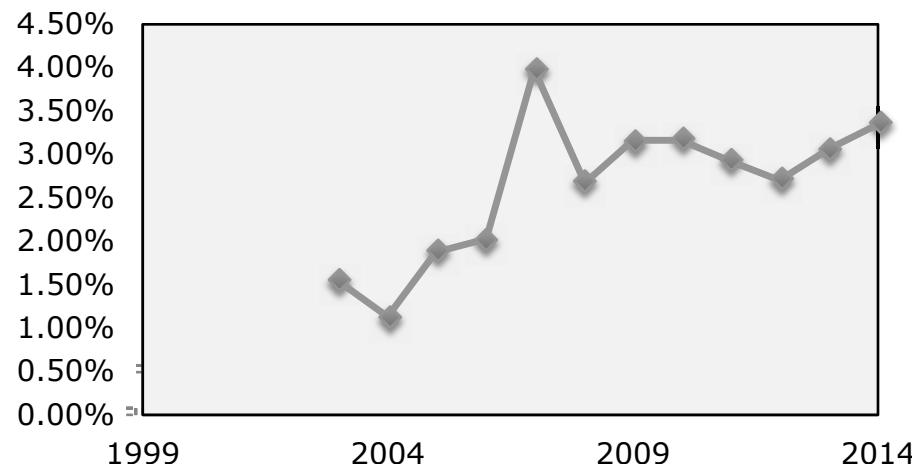
Commit messages for MySQL



Commit messages for OpenSSL

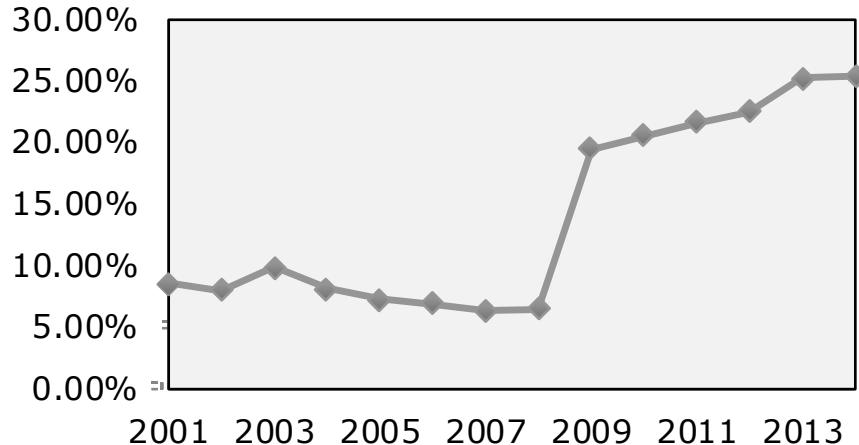


Bug reports for the Eclipse IDE

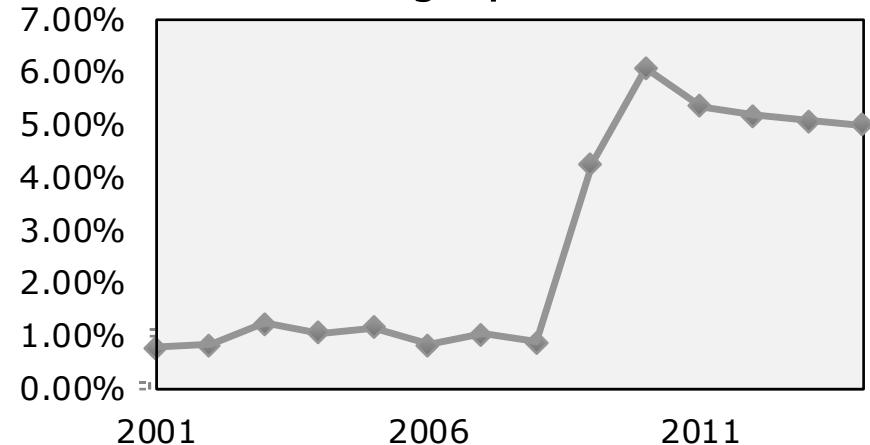


Software Developer Jobs

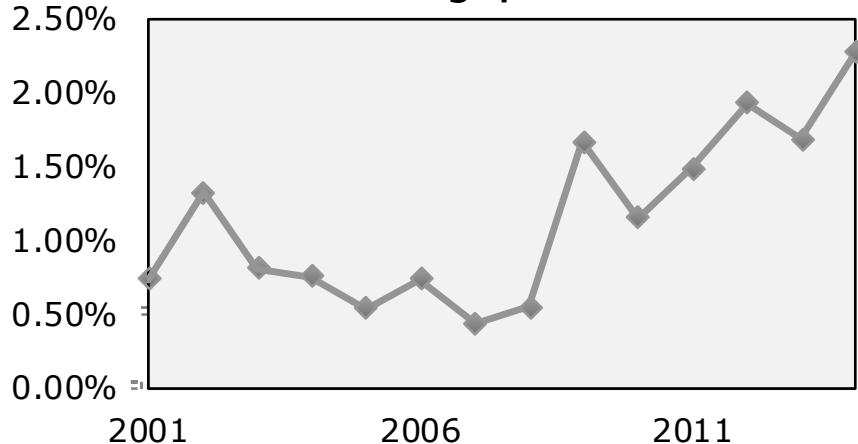
Mentioning “performance”



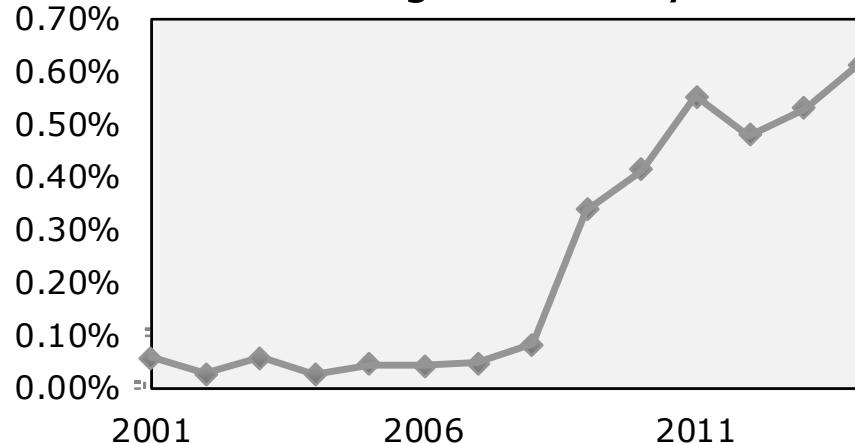
Mentioning “optimization”



Mentioning “parallel”



Mentioning “concurrency”

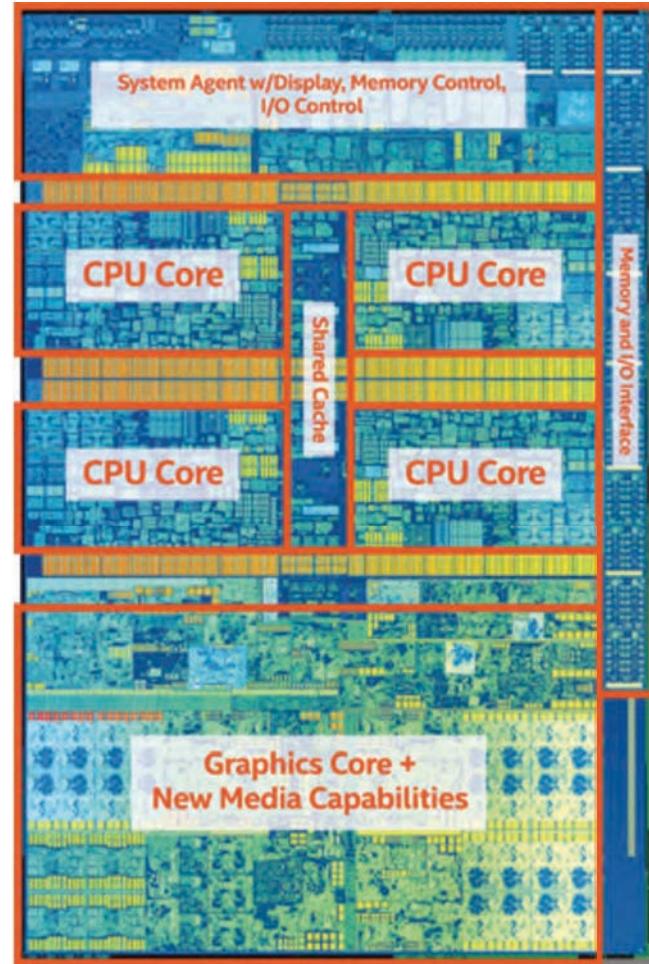


Source: Monster.com

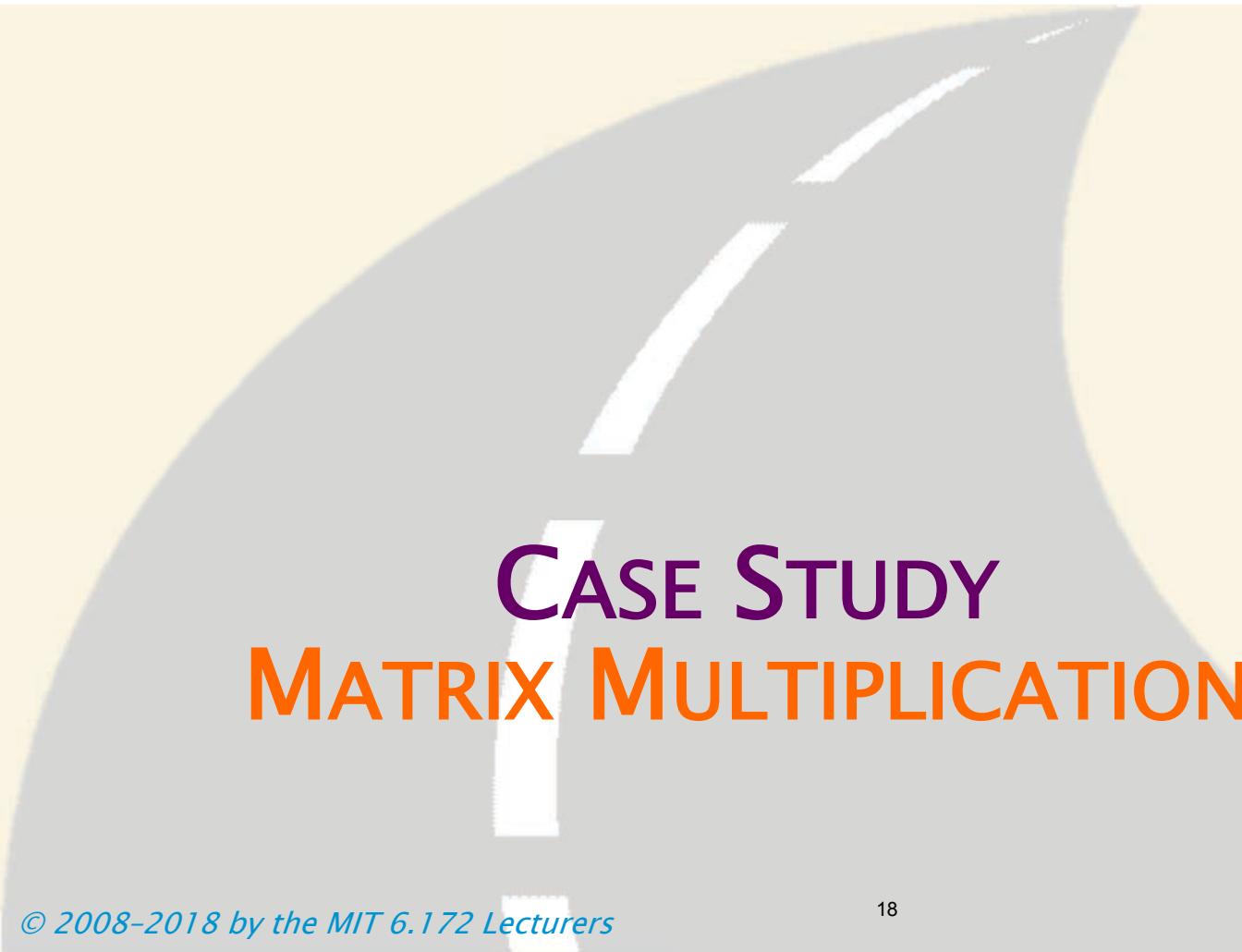
Performance Engineering Is Still Hard

A modern multicore desktop processor contains parallel-processing cores, vector units, caches, prefetchers, GPU's, hyperthreading, dynamic frequency scaling, etc., etc.

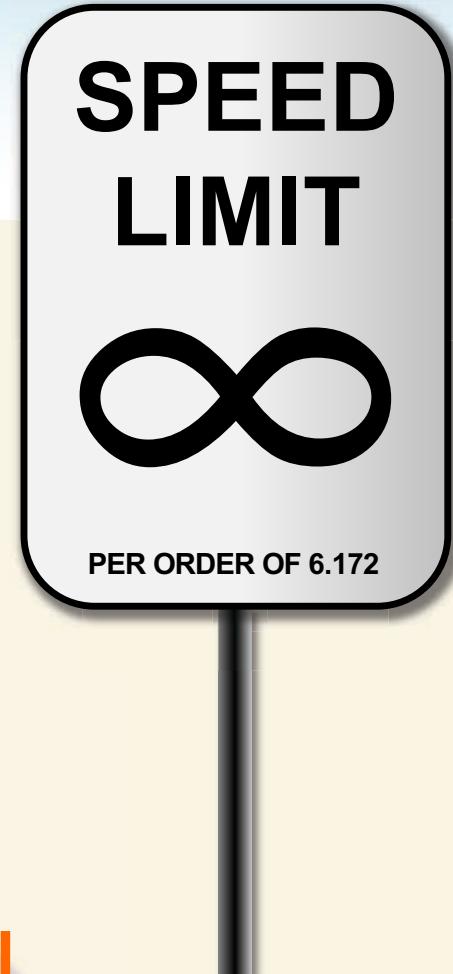
How can we write software to utilize modern hardware efficiently?



2017 Intel 7th-generation desktop processor



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 $n = 2^k$.

AWS c4.8xlarge Machine Specs

| Feature | Specification |
|---------------------|---|
| Microarchitecture | Haswell (Intel Xeon E5-2666 v3) |
| Clock frequency | 2.9 GHz |
| Processor chips | 2 |
| Processing cores | 9 per processor chip |
| Hyperthreading | 2 way |
| Floating-point unit | 8 double-precision operations, including fused-multiply-add, 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 (LLC) | 25 MB shared 20-way set associative |
| DRAM | 60 GB |

$$\text{Peak} = (2.9 \times 10^9) \times 2 \times 9 \times 16 = 836 \text{ GFLOPS}$$

Version 1: Nested Loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
      for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
      for col in xrange(n)]
C = [[0 for row in xrange(n)]
      for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

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

Running time
= 21042 seconds
 \approx 6 hours

Is this fast?

Should we expect
more from our
machine?

Version 1: Nested Loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
      for col in xrange(n)]

B =
C =
start
for
end

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

Running time
= 21042 seconds
 \approx 6 hours

Is this fast?

Back-of-the-envelope calculation

back-of-envelope calculation 猜量推算，粗略计算，保守计算

$$C = 2n^3 = 2(2^{12})^3 = 2^{37} \text{ floating-point operations}$$

Running time = 21042 seconds

\therefore Python gets $2^{37}/21042 \approx 6.25$ MFLOPS

Peak \approx 836 GFLOPS

Python gets $\approx 0.00075\%$ of peak

Version 2: Java

```
import java.util.Random;

public class mm_java {
    static int n = 4096;
    static double[][] A = new double[n][n];
    static double[][] B = new double[n][n];
    static double[][] C = new double[n][n];

    public static void main(String[] args) {
        Random r = new Random();

        for (int i=0; i<n; i++) {
            for (int j=0; j<n; j++) {
                A[i][j] = r.nextDouble();
                B[i][j] = r.nextDouble();
                C[i][j] = 0;
            }
        }

        long start = System.nanoTime();

        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];
                }
            }
        }

        long stop = System.nanoTime();

        double tdiff = (stop - start) * 1e-9;
        System.out.println(tdiff);
    }
}
```

Running time = 2,738 seconds
≈ 46 minutes
... about 8.8× faster than Python.

```
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];
        }
    }
}
```

Version 3: C

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#define n 4096
double A[n][n];
double B[n][n];
double C[n][n];

float tdiff(struct timeval *start,
            struct timeval *end) {
    return (end->tv_sec-start->tv_sec) +
        1e-6*(end->tv_usec-start->tv_usec);
}

int main(int argc, const char *argv[]) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            A[i][j] = (double)rand() / (double)RAND_MAX;
            B[i][j] = (double)rand() / (double)RAND_MAX;
            C[i][j] = 0;
        }
    }

    struct timeval start, end;
    gettimeofday(&start, NULL);

    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];
            }
        }
    }

    gettimeofday(&end, NULL);
    printf("%0.6f\n", tdiff(&start, &end));
    return 0;
}
```

Using the Clang/LLVM 5.0 compiler

Running time = 1,156 seconds
≈ 19 minutes,
or about 2× faster than Java and
about 18× faster than Python.

```
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

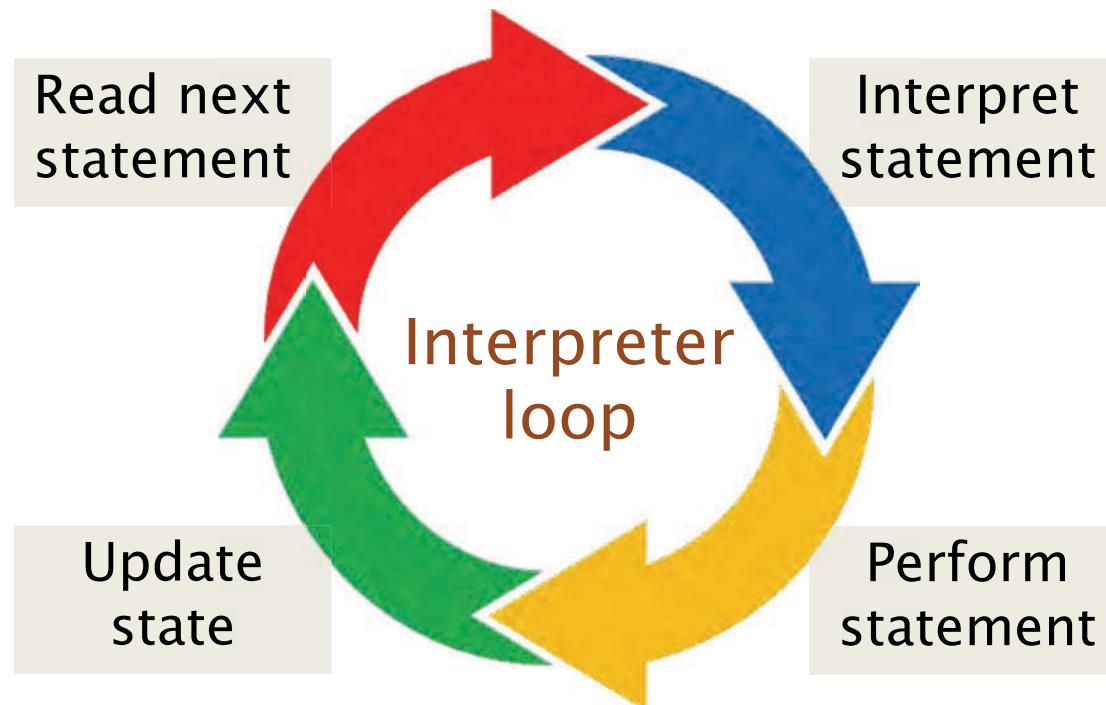
| Version | Implementation | Running time (s) | Relative speedup | Absolute Speedup | GFLOPS | Percent of peak |
|---------|----------------|------------------|------------------|------------------|--------|-----------------|
| 1 | Python | 21041.67 | 1.00 | 1 | 0.007 | 0.001 |
| 2 | Java | 2387.32 | 8.81 | 9 | 0.058 | 0.007 |
| 3 | C | 1155.77 | 2.07 | 18 | 0.119 | 0.014 |

Why is Python so slow and C so fast?

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

Interpreters are versatile, but slow

- The interpreter reads, interprets, and performs each program statement and updates the machine state.
- Interpreters can easily support high-level programming features — such as dynamic code alteration — at the cost of performance.



JIT Compilation

- JIT compilers can recover some of the performance lost by interpretation.
- When code is first executed, it is interpreted.
- The runtime system keeps track of how often the various pieces of code are executed.
- Whenever some piece of code executes sufficiently frequently, it gets compiled to machine code in real time. 每当某段代码执行得足够频繁时，它就会被实时编译为机器代码。
- Future executions of that code use the more-efficient compiled version.

Loop Order

We can change the order of the loops in this program without affecting its correctness.

```
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];  
        }  
    }  
}
```

Loop Order

We can change the order of the loops in this program without affecting its correctness.

j作为最内层循环，按列递增；因为数组按行存储，因此B在第K行进行列 j++，相当于在一列上操作，有效利用了缓存

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

Does the order of loops matter for performance?

Performance of Different Orders

| Loop order (outer to inner) | Running time (s) |
|--------------------------------|---------------------|
| i, j, k | 1155.77 |
| i, k, j | 177.68 |
| j, i, k | 1080.61 |
| j, k, i | 3056.63 |
| k, i, j | 179.21 |
| k, j, i | 3032.82 |

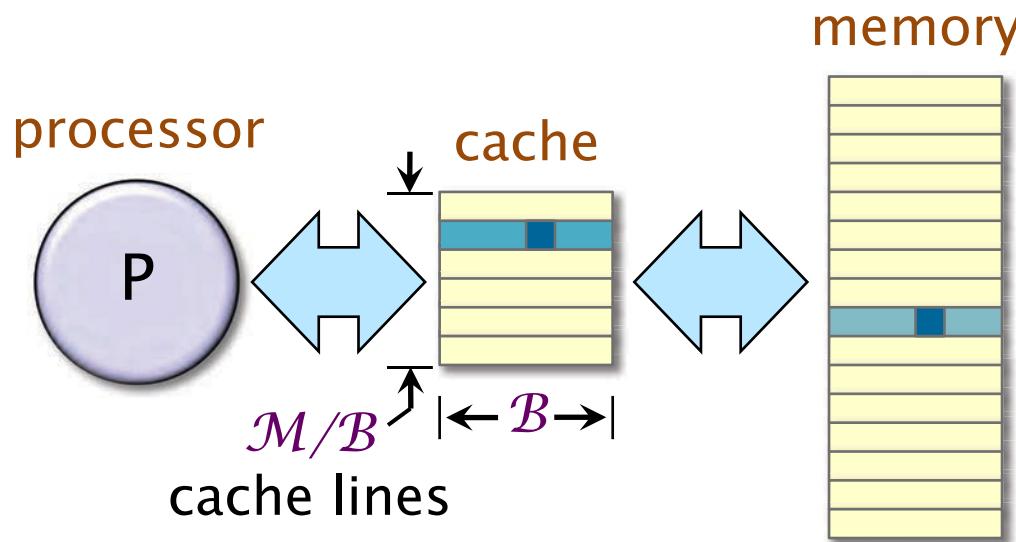
Loop order affects running time by a factor of 18!

What's going on!?

Hardware Caches

Each processor reads and writes main memory in contiguous blocks, called *cache lines*.

- Previously accessed cache lines are stored in a smaller memory, called a *cache*, that sits near the processor.
- *Cache hits* — accesses to data in cache — are fast.
- *Cache misses* — accesses to data not in cache — are slow.



Memory Layout of Matrices

In this matrix-multiplication code, matrices are laid out in memory in *row-major order*.

Matrix

| |
|-------|
| Row 1 |
| Row 2 |
| Row 3 |
| Row 4 |
| Row 5 |
| Row 6 |
| Row 7 |
| Row 8 |

What does this layout imply about the performance of different loop orders?

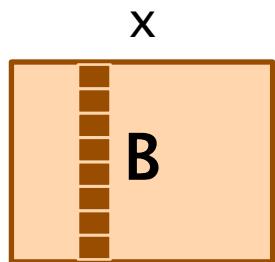
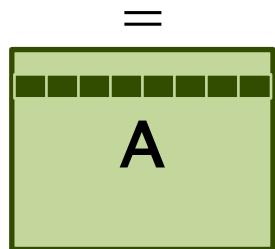
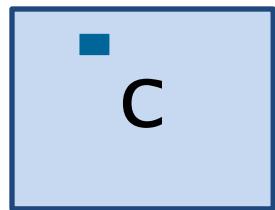
Memory



Access Pattern for Order i, j, k

```
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];
```

Running time:
1155.77s



In-memory layout



Excellent spatial locality



Good spatial locality



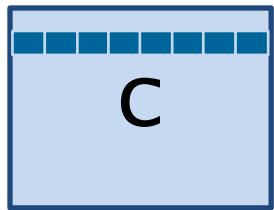
Poor spatial locality

4096 elements apart

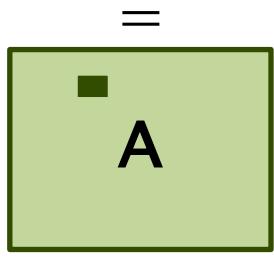
Access Pattern for Order i, k, j

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

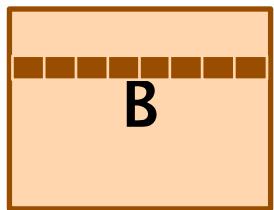
Running time:
177.68s



In-memory layout



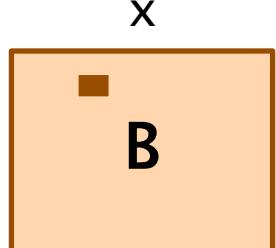
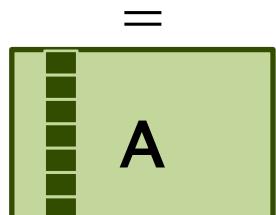
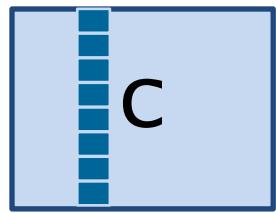
x



Access Pattern for Order j, k, i

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

Running time:
3056.63s



In-memory layout



Performance of Different Orders

We can measure the effect of different access patterns using the Cachegrind cache simulator:

```
$ valgrind --tool=cachegrind ./mm
```

| Loop order (outer to inner) | Running time (s) | Last-level-cache miss rate |
|--------------------------------|---------------------|-------------------------------|
| i, j, k | 1155.77 | 7.7% |
| i, k, j | 177.68 | 1.0% |
| j, i, k | 1080.61 | 8.6% |
| j, k, i | 3056.63 | 15.4% |
| k, i, j | 179.21 | 1.0% |
| k, j, i | 3032.82 | 15.4% |

Version 4: Interchange Loops

| Version | Implementation | Running time (s) | Relative speedup | Absolute Speedup | GFLOPS | Percent of peak |
|---------|---------------------|------------------|------------------|------------------|--------|-----------------|
| 1 | Python | 21041.67 | 1.00 | 1 | 0.006 | 0.001 |
| 2 | Java | 2387.32 | 8.81 | 9 | 0.058 | 0.007 |
| 3 | C | 1155.77 | 2.07 | 18 | 0.118 | 0.014 |
| 4 | + interchange loops | 177.68 | 6.50 | 118 | 0.774 | 0.093 |

What other simple changes we can try?

Compiler Optimization

Clang provides a collection of optimization switches. You can specify a switch to the compiler to ask it to optimize.

| Opt. level | Meaning | Time (s) |
|------------|--------------------|----------|
| -00 | Do not optimize | 177.54 |
| -01 | Optimize | 66.24 |
| -02 | Optimize even more | 54.63 |
| -03 | Optimize yet more | 55.58 |

Clang also supports optimization levels for special purposes, such as **-Os**, which aims to limit code size, and **-Og**, for debugging purposes.

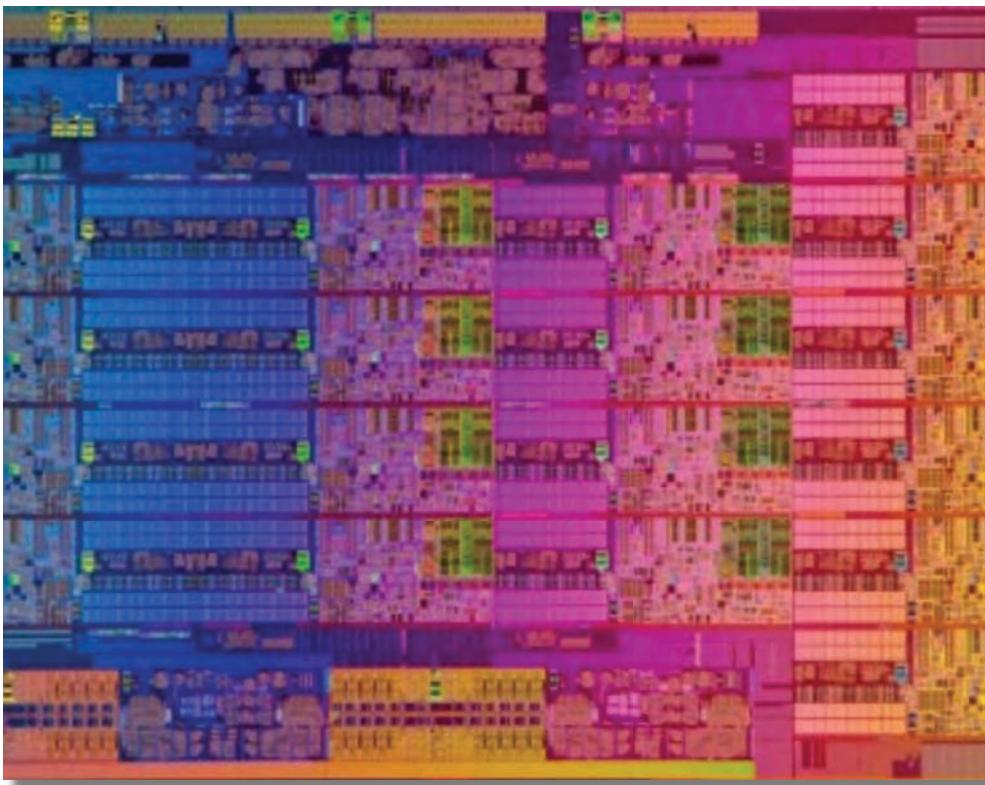
Version 5: Optimization Flags

| Version | Implementation | Running time (s) | Relative speedup | Absolute Speedup | GFLOPS | Percent of peak |
|---------|----------------------|------------------|------------------|------------------|--------|-----------------|
| 1 | Python | 21041.67 | 1.00 | 1 | 0.006 | 0.001 |
| 2 | Java | 2387.32 | 8.81 | 9 | 0.058 | 0.007 |
| 3 | C | 1155.77 | 2.07 | 18 | 0.118 | 0.014 |
| 4 | + interchange loops | 177.68 | 6.50 | 118 | 0.774 | 0.093 |
| 5 | + optimization flags | 54.63 | 3.25 | 385 | 2.516 | 0.301 |

With simple code and compiler technology, we can achieve 0.3% of the peak performance of the machine.

What's causing the low performance?

Multicore Parallelism



Intel Haswell E5:
9 cores per chip

The AWS test
machine has 2 of
these chips.

We're running on just 1 of the 18 parallel-processing
cores on this system. Let's use them all!

© [Intel](#). All rights reserved. This content is excluded from our Creative Commons license.
For more information, see <https://ocw.mit.edu/help/faq-fair-use/>

Parallel Loops

The `cilk_for` loop allows all iterations of the loop to execute in parallel.

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

These loops can be (easily) parallelized.

Which parallel version works best?

Experimenting with Parallel Loops

Parallel i loop

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

Running time: 3.18s

Parallel j loop

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

Running time: 531.71s

Rule of Thumb
Parallelize outer
loops rather than
inner loops.

并行化外层循环而不是内存循环

Parallel i and j

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

Running time: 10.64s

Version 6: Parallel Loops

| Version | Implementation | Running time (s) | Relative speedup | Absolute Speedup | GFLOPS | Percent of peak |
|---------|----------------------|------------------|------------------|------------------|--------|-----------------|
| 1 | Python | 21041.67 | 1.00 | 1 | 0.006 | 0.001 |
| 2 | Java | 2387.32 | 8.81 | 9 | 0.058 | 0.007 |
| 3 | C | 1155.77 | 2.07 | 18 | 0.118 | 0.014 |
| 4 | + interchange loops | 177.68 | 6.50 | 118 | 0.774 | 0.093 |
| 5 | + optimization flags | 54.63 | 3.25 | 385 | 2.516 | 0.301 |
| 6 | Parallel loops | 3.04 | 17.97 | 6,921 | 45.211 | 5.408 |

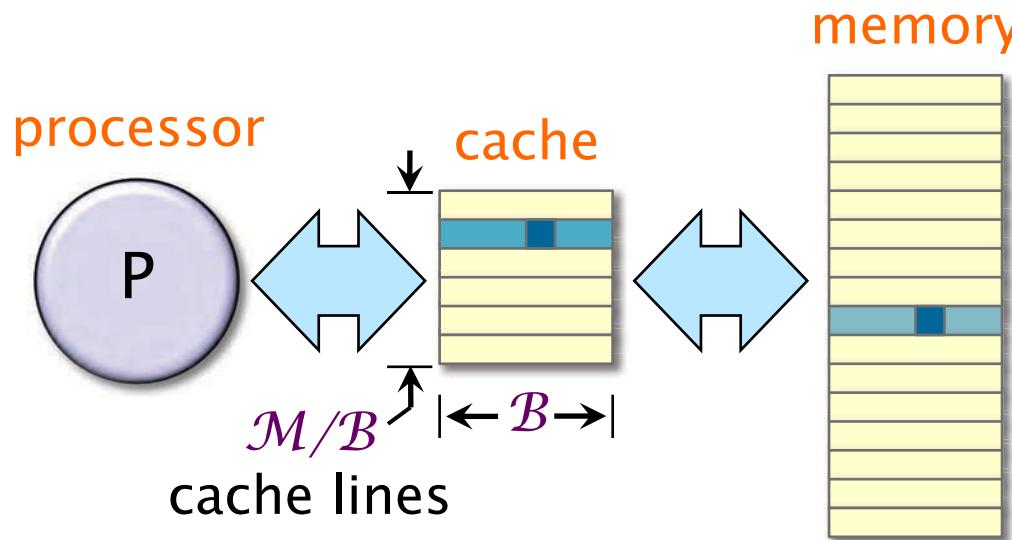
Using parallel loops gets us almost $18\times$ speedup on 18 cores! (Disclaimer: Not all code is so easy to parallelize effectively.)

Why are we still getting just 5% of peak?

Hardware Caches, Revisited

IDEA: Restructure the computation to reuse data in the cache as much as possible.

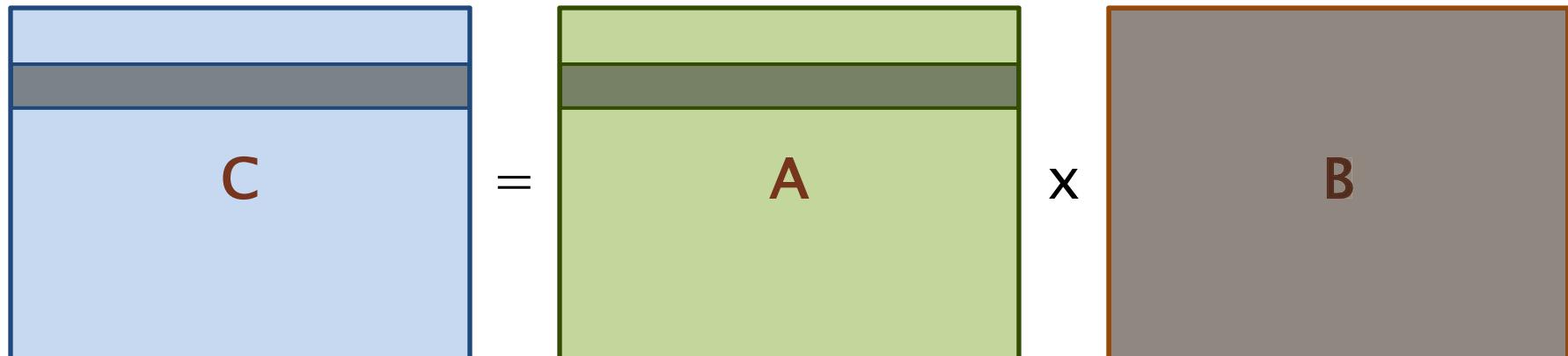
- Cache misses are slow, and cache hits are fast.
- Try to make the most of the cache by reusing the data that's already there.



Data Reuse: Loops

How many memory accesses must the looping code perform to fully compute 1 row of C?

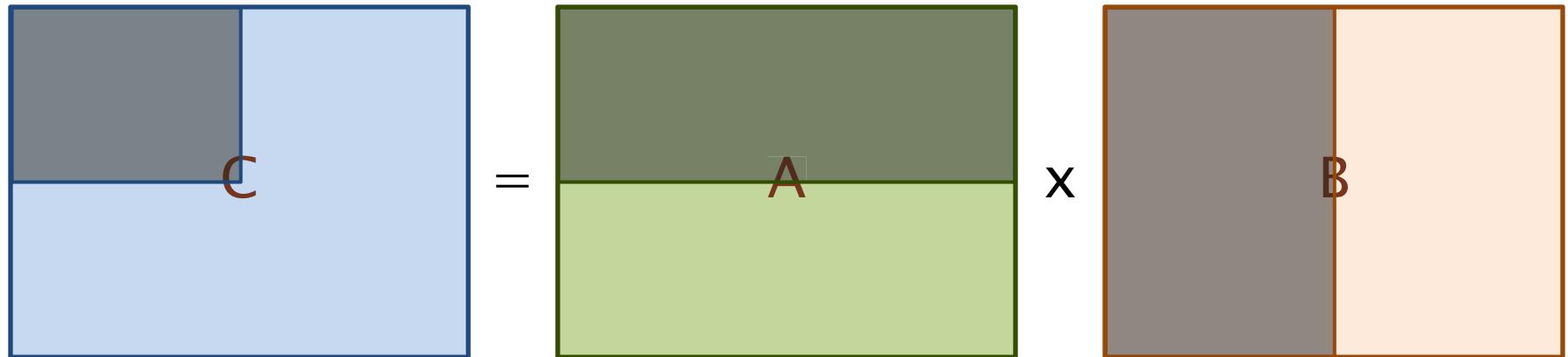
- $4096 * 1 = 4096$ writes to C,
- $4096 * 1 = 4096$ reads from A, and
- $4096 * 4096 = 16,777,216$ reads from B, which is
- 16,785,408 memory accesses total.



Data Reuse: Blocks

How about to compute a 64×64 block of **C**?

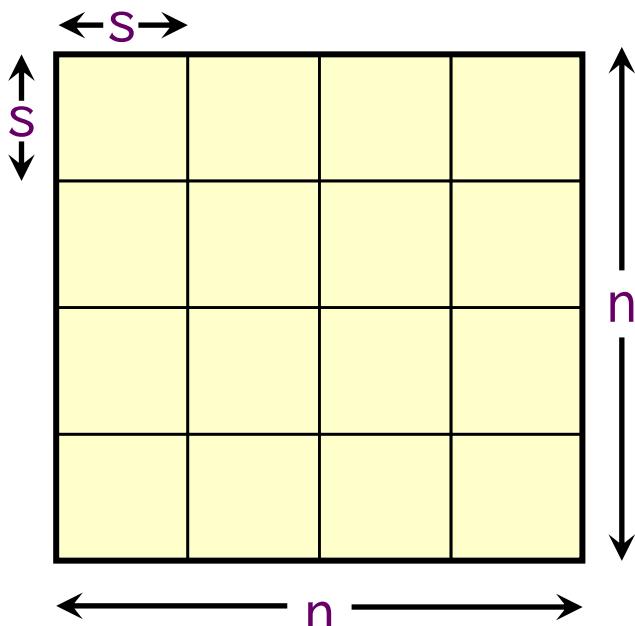
- $64 \cdot 64 = 4096$ writes to **C**,
- $64 \cdot 4096 = 262,144$ reads from **A**, and
- $4096 \cdot 64 = 262,144$ reads from **B**, or
- 528,384 memory accesses total.



Tiled Matrix Multiplication

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

Tuning parameter
How do we find the
right value of s ?
Experiment!



| Tile size | Running time (s) |
|-----------|------------------|
| 4 | 6.74 |
| 8 | 2.76 |
| 16 | 2.49 |
| 32 | 1.74 |
| 64 | 2.33 |
| 128 | 2.13 |

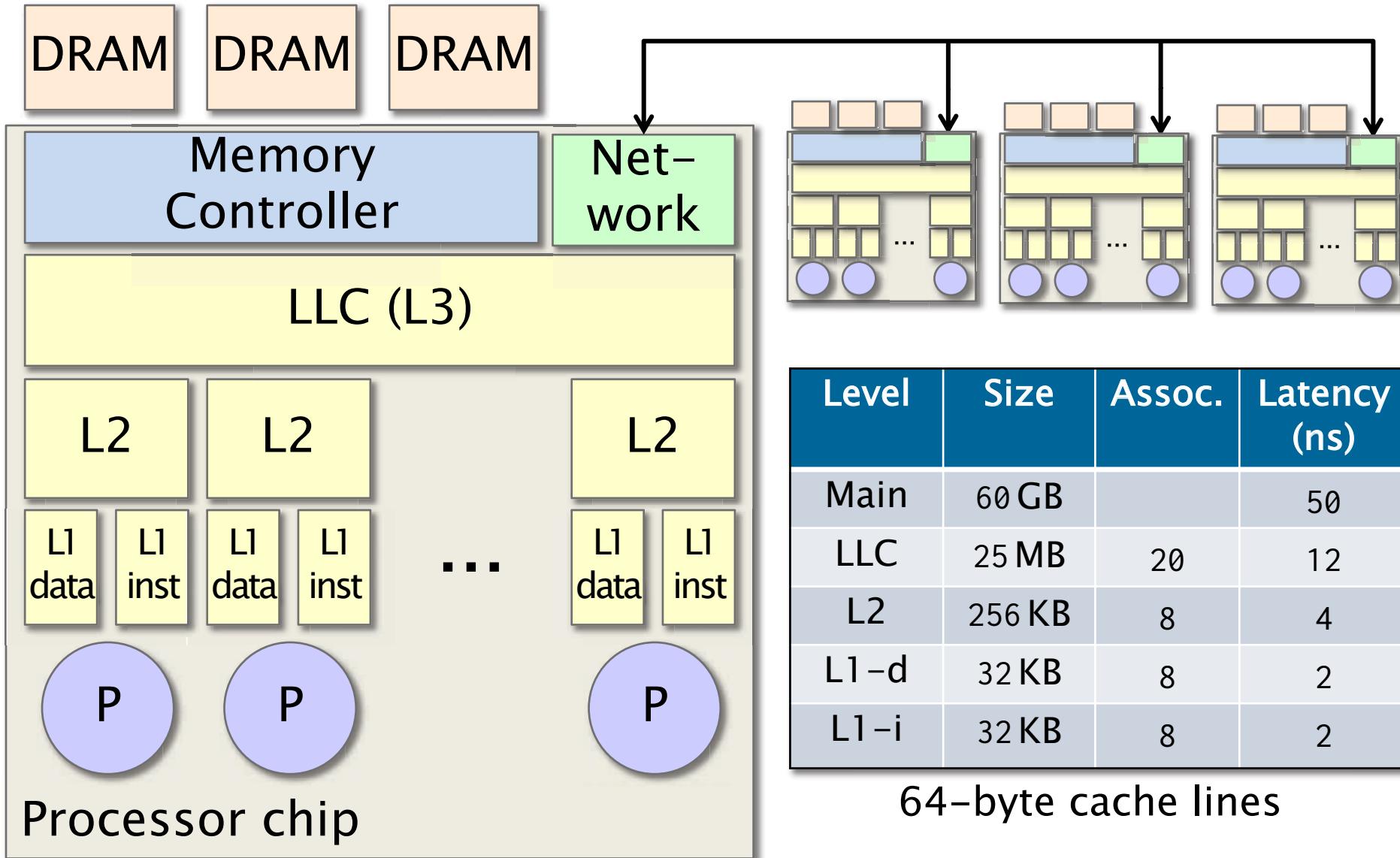
Version 7: Tiling

| Version | Implementation | Running time (s) | Relative speedup | Absolute Speedup | GFLOPS | Percent of peak |
|---------|----------------------|------------------|------------------|------------------|--------|-----------------|
| 1 | Python | 21041.67 | 1.00 | 1 | 0.006 | 0.001 |
| 2 | Java | 2387.32 | 8.81 | 9 | 0.058 | 0.007 |
| 3 | C | 1155.77 | 2.07 | 18 | 0.118 | 0.014 |
| 4 | + interchange loops | 177.68 | 6.50 | 118 | 0.774 | 0.093 |
| 5 | + optimization flags | 54.63 | 3.25 | 385 | 2.516 | 0.301 |
| 6 | Parallel loops | 3.04 | 17.97 | 6,921 | 45.211 | 5.408 |
| 7 | + tiling | 1.79 | 1.70 | 11,772 | 76.782 | 9.184 |

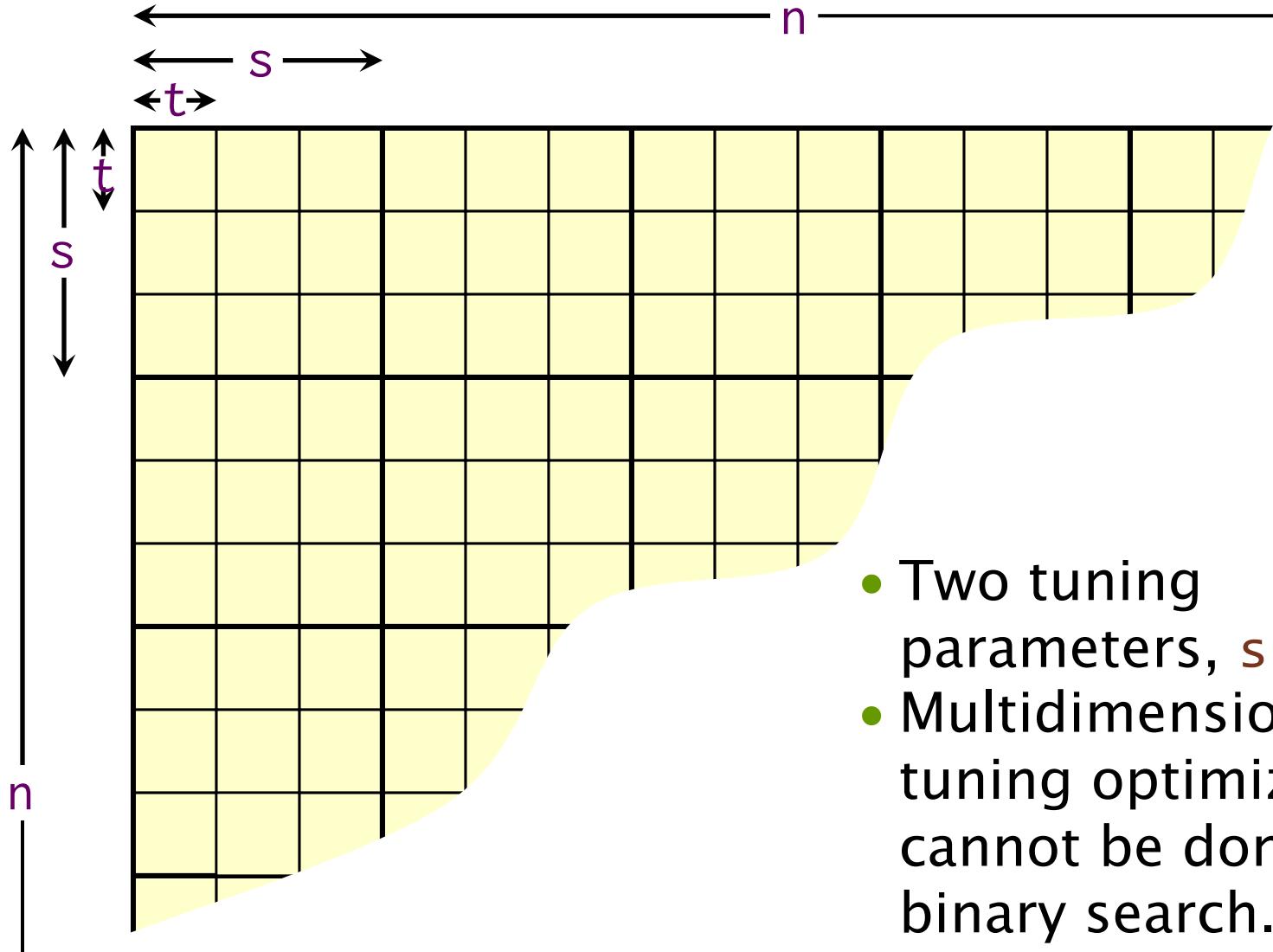
| Implementation | Cache references (millions) | L1-d cache misses (millions) | Last-level cache misses (millions) |
|----------------|--------------------------------|---------------------------------|---------------------------------------|
| Parallel loops | 104,090 | 17,220 | 8,600 |
| + tiling | 64,690 | 11,777 | 416 |

The tiled implementation performs about 62% fewer cache references and incurs 68% fewer cache misses.

Multicore Cache Hierarchy

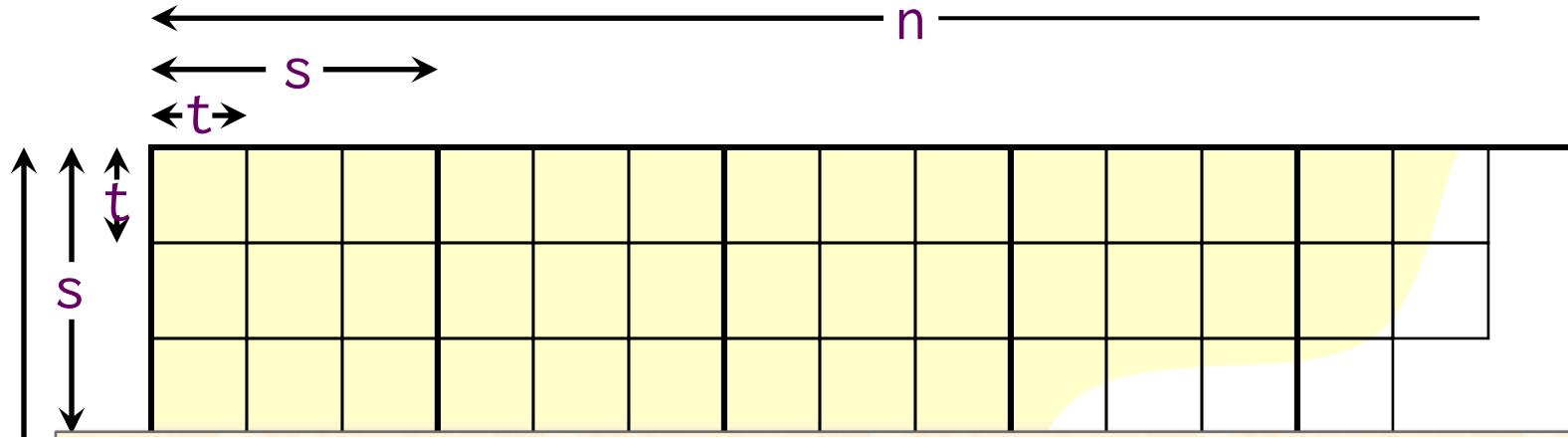


Tiling for a Two-Level Cache



- Two tuning parameters, s and t .
- Multidimensional tuning optimization cannot be done with binary search.

Tiling for a Two-Level Cache



```
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 im = 0; im < s; im += t)
            for (int jm = 0; jm < s; jm += t)
                for (int km = 0; km < s; km += t)
                    for (int il = 0; il < t; ++il)
                        for (int kl = 0; kl < t; ++kl)
                            for (int jl = 0; jl < t; ++jl)
                                C[ih+im+il][jh+jm+jl] +=
                                    A[ih+im+il][kh+km+kl] * B[kh+km+kl][jh+jm+jl];
```

Recursive Matrix Multiplication

IDEA: Tile for **every** power of 2 simultaneously.

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$
$$= \begin{bmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{bmatrix} + \begin{bmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{bmatrix}$$

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

Recursive Parallel Matrix Multiply

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == n);
  if (n <= 1) {
    *C += *A * *B;
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);

    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);

    cilk_sync;
  }
}
```

The child function call is **spawned**, meaning it may execute in parallel with the parent caller.

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

Recursive Parallel Matrix Multiply

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == n);
  if (n <= 1) { // Base case
    *C += *A * *B;
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,0,1), n_B, n/2);
    cilk_sync;
  }
}
```

The base case is too small. We must **coarsen** the recursion to overcome function-call overheads.

Running time: 93.93s
... about 50 \times slower than the last version!

Coarsening The Recursion

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
    assert((n & (-n)) == n);
    if [n <= THRESHOLD] {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
                    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        cilk_sync;
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
                    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        cilk_sync;
    }
}
```

Just one tuning parameter, for the size of the base case.

Coarsening The Recursion

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
    assert((n & (-n)) == n);
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,0,1), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,1), n_C, X(A,0,1), n_A, X(B,0,1), n_B, n/2);
        cilk_sync;
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,0,1), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,1), n_C, X(A,0,1), n_A, X(B,0,1), n_B, n/2);
        cilk_sync;
    }
}
}

void mm_base(double *restrict C, int n_C,
             double *restrict A, int n_A,
             double *restrict B, int n_B,
             int n)
{ // C = A * B
    for (int i = 0; i < n; ++i)
        for (int k = 0; k < n; ++k)
            for (int j = 0; j < n; ++j)
                C[i*n_C+j] += A[i*n_A+k] * B[k*n_B+j];
}
```

Coarsening The Recursion

| Base-case size | Running time (s) |
|----------------|------------------|
| 4 | 3.00 |
| 8 | 1.34 |
| 16 | 1.34 |
| 32 | 1.30 |
| 64 | 1.95 |
| 128 | 2.08 |

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
    assert((n & (-n)) == n);
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, mm_dac(X(C,1,1), n_C, X(A,1,0), n_A,
        cilk_sync;
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, mm_dac(X(C,1,1), n_C, X(A,1,1), n_A,
        cilk_sync;
    }
}
```

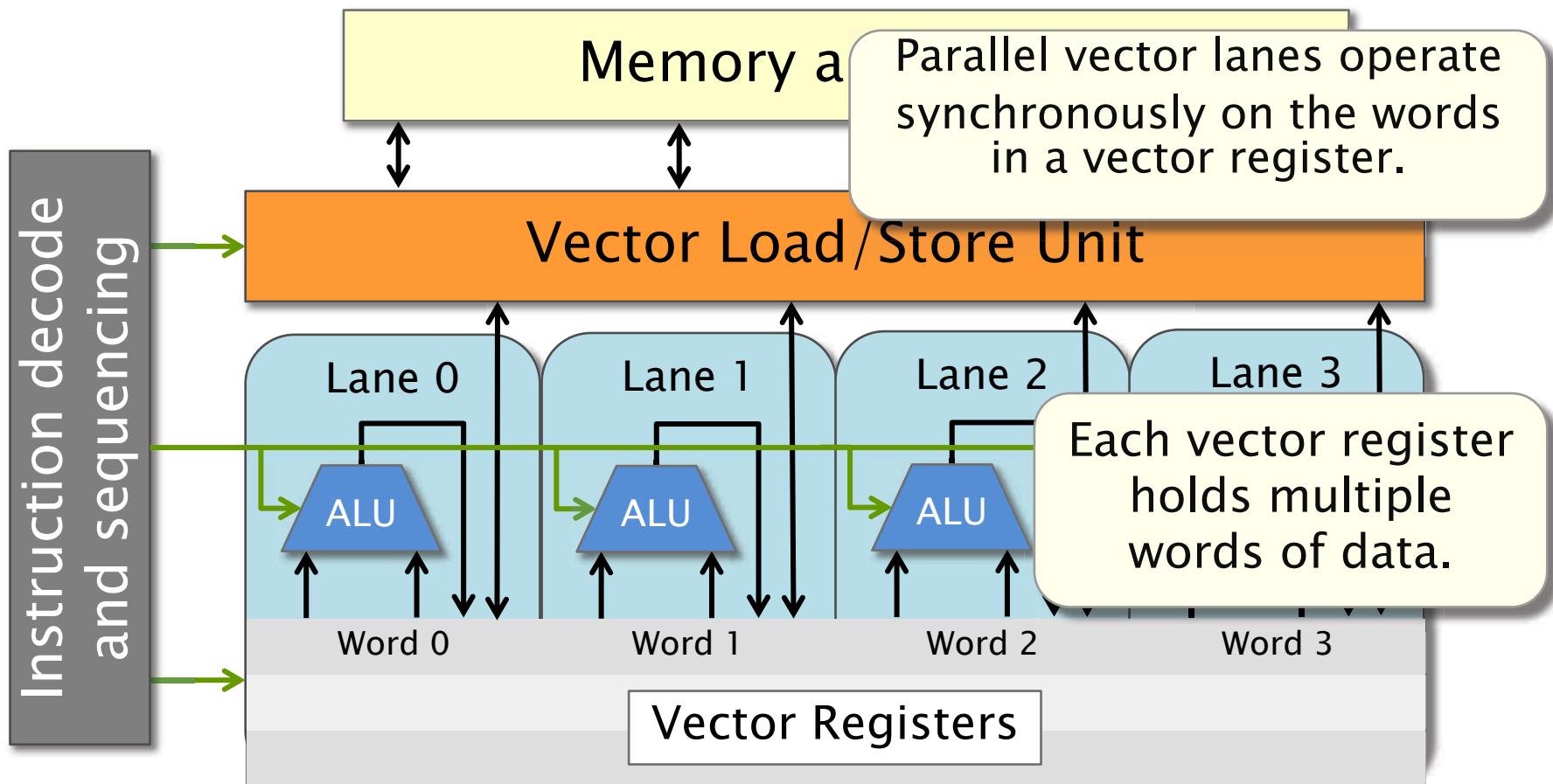
8. Divide-and-Conquer

| Version | Implementation | Running time (s) | Relative speedup | Absolute Speedup | GFLOPS | Percent of peak |
|---------|-----------------------------|------------------|------------------|------------------|---------|-----------------|
| 1 | Python | 21041.67 | 1.00 | 1 | 0.006 | 0.001 |
| 2 | Java | 2387.32 | 8.81 | 9 | 0.058 | 0.007 |
| 3 | C | 1155.77 | 2.07 | 18 | 0.118 | 0.014 |
| 4 | + interchange loops | 177.68 | 6.50 | 118 | 0.774 | 0.093 |
| 5 | + optimization flags | 54.63 | 3.25 | 385 | 2.516 | 0.301 |
| 6 | Parallel loops | 3.04 | 17.97 | 6,921 | 45.211 | 5.408 |
| 7 | + tiling | 1.79 | 1.70 | 11,772 | 76.782 | 9.184 |
| 8 | Parallel divide-and-conquer | 1.30 | 1.38 | 16,197 | 105.722 | 12.646 |

| Implementation | Cache references (millions) | L1-d cache misses (millions) | Last-level cache misses (millions) |
|-----------------------------|--------------------------------|---------------------------------|---------------------------------------|
| Parallel loops | 104,090 | 17,220 | 8,600 |
| + tiling | 64,690 | 11,777 | 416 |
| Parallel divide-and-conquer | 58,230 | 9,407 | 64 |

Vector Hardware

Modern microprocessors incorporate vector hardware to process data in **single-instruction stream, multiple-data stream (SIMD)** fashion.



Compiler Vectorization

Clang/LLVM uses vector instructions automatically when compiling at optimization level **-O2** or higher. Clang/LLVM can be induced to produce a *vectorization report* as follows:

```
$ clang -O3 -std=c99 mm.c -o mm -Rpass=vector
mm.c:42:7: remark: vectorized loop (vectorization width: 2,
interleaved count: 2) [-Rpass=loop-vectorize]
    for (int j = 0; j < n; ++j) {
        ^
```

Many machines don't support the newest set of vector instructions, however, so the compiler uses vector instructions conservatively by default.

Vectorization Flags

Programmers can direct the compiler to use modern vector instructions using **compiler flags** such as the following:

- **-mavx**: Use Intel AVX vector instructions.
- **-mavx2**: Use Intel AVX2 vector instructions.
- **-mfma**: Use fused multiply-add vector instructions.
- **-march=<string>**: Use whatever instructions are available on the specified architecture.
- **-march=native**: Use whatever instructions are available on the architecture of the machine doing compilation.

Due to restrictions on floating-point arithmetic, additional flags, such as **-ffast-math**, might be needed for these vectorization flags to have an effect.

Version 9: Compiler Vectorization

| Version | Implementation | Running time (s) | Relative speedup | Absolute Speedup | GFLOPS | Percent of peak |
|---------|-----------------------------|------------------|------------------|------------------|---------|-----------------|
| 1 | Python | 21041.67 | 1.00 | 1 | 0.006 | 0.001 |
| 2 | Java | 2387.32 | 8.81 | 9 | 0.058 | 0.007 |
| 3 | C | 1155.77 | 2.07 | 18 | 0.118 | 0.014 |
| 4 | + interchange loops | 177.68 | 6.50 | 118 | 0.774 | 0.093 |
| 5 | + optimization flags | 54.63 | 3.25 | 385 | 2.516 | 0.301 |
| 6 | Parallel loops | 3.04 | 17.97 | 6,921 | 45.211 | 5.408 |
| 7 | + tiling | 1.79 | 1.70 | 11,772 | 76.782 | 9.184 |
| 8 | Parallel divide-and-conquer | 1.30 | 1.38 | 16,197 | 105.722 | 12.646 |
| 9 | + compiler vectorization | 0.70 | 1.87 | 30,272 | 196.341 | 23.486 |

Using the flags `-march=native -ffast-math` nearly doubles the program's performance!

Can we be smarter than the compiler?

AVX Intrinsic Instructions

Intel provides C-style functions, called *intrinsic instructions*, that provide direct access to hardware vector operations:

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

The screenshot shows the Intel Intrinsics Guide interface. On the left, there's a sidebar with a 'Technologies' section containing checkboxes for various instruction sets: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX (which is checked), AVX2 (which is checked), FMA (which is checked), AVX-512, KNC, SVML, and Other. Below that is a 'Categories' section with a checkbox for Application-Targeted. The main area has a search bar with 'mm_search' typed in. To the right of the search bar is a help icon (?). A yellow callout box highlights the search results, stating: 'The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C-style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.' The search results list several intrinsic functions starting with '_mm256_':

| Intrinsic Function | Equivalent Assembly |
|--|-----------------------|
| <code>_mm256i _mm256_abs_epi16 (_m256i a)</code> | <code>vpabsw</code> |
| <code>_m256i _mm256_abs_epi32 (_m256i a)</code> | <code>vpabsd</code> |
| <code>_m256i _mm256_abs_epi8 (_m256i a)</code> | <code>vpabsb</code> |
| <code>_m256i _mm256_add_epi16 (_m256i a, _m256i b)</code> | <code>vpaddw</code> |
| <code>_m256i _mm256_add_epi32 (_m256i a, _m256i b)</code> | <code>vpaddsd</code> |
| <code>_m256i _mm256_add_epi64 (_m256i a, _m256i b)</code> | <code>vpaddq</code> |
| <code>_m256i _mm256_add_epi8 (_m256i a, _m256i b)</code> | <code>vpaddb</code> |
| <code>_m256d _mm256_add_pd (_m256d a, _m256d b)</code> | <code>vaddpd</code> |
| <code>_m256 _mm256_add_ps (_m256 a, _m256 b)</code> | <code>vaddps</code> |
| <code>_m256i _mm256_adds_epi16 (_m256i a, _m256i b)</code> | <code>vpaddsw</code> |
| <code>_m256i _mm256_adds_epi8 (_m256i a, _m256i b)</code> | <code>vpaddsb</code> |
| <code>_m256i _mm256_adds_epu16 (_m256i a, _m256i b)</code> | <code>vpaddusw</code> |

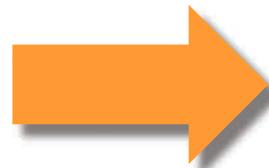
Plus More Optimizations

We can apply several more insights and performance-engineering tricks to make this code run faster, including:

- Preprocessing
- Matrix transposition
- Data alignment
- Memory-management optimizations
- A clever algorithm for the base case that uses AVX intrinsic instructions explicitly

Plus Performance Engineering

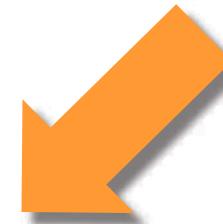
Think,



code,



run, run, run...



...to test and measure many
different implementations

Version 10: AVX Intrinsics

| Version | Implementation | Running time (s) | Relative speedup | Absolute Speedup | GFLOPS | Percent of peak |
|---------|-----------------------------|------------------|------------------|------------------|---------|-----------------|
| 1 | Python | 21041.67 | 1.00 | 1 | 0.006 | 0.001 |
| 2 | Java | 2387.32 | 8.81 | 9 | 0.058 | 0.007 |
| 3 | C | 1155.77 | 2.07 | 18 | 0.118 | 0.014 |
| 4 | + interchange loops | 177.68 | 6.50 | 118 | 0.774 | 0.093 |
| 5 | + optimization flags | 54.63 | 3.25 | 385 | 2.516 | 0.301 |
| 6 | Parallel loops | 3.04 | 17.97 | 6,921 | 45.211 | 5.408 |
| 7 | + tiling | 1.79 | 1.70 | 11,772 | 76.782 | 9.184 |
| 8 | Parallel divide-and-conquer | 1.30 | 1.38 | 16,197 | 105.722 | 12.646 |
| 9 | + compiler vectorization | 0.70 | 1.87 | 30,272 | 196.341 | 23.486 |
| 10 | + AVX intrinsics | 0.39 | 1.76 | 53,292 | 352.408 | 41.677 |

Version 11: Final Reckoning

| Version | Implementation | Running time (s) | Relative speedup | Absolute Speedup | GFLOPS | Percent of peak |
|---------|-----------------------------|------------------|------------------|------------------|---------|-----------------|
| 1 | Python | 21041.67 | 1.00 | 1 | 0.006 | 0.001 |
| 2 | Java | 2387.32 | 8.81 | 9 | 0.058 | 0.007 |
| 3 | C | 1155.77 | 2.07 | 18 | 0.118 | 0.014 |
| 4 | + interchange loops | 177.68 | 6.50 | 118 | 0.774 | 0.093 |
| 5 | + optimization flags | 54.63 | 3.25 | 385 | 2.516 | 0.301 |
| 6 | Parallel loops | 3.04 | 17.97 | 6,921 | 45.211 | 5.408 |
| 7 | + tiling | 1.79 | 1.70 | 11,772 | 76.782 | 9.184 |
| 8 | Parallel divide-and-conquer | 1.30 | 1.38 | 16,197 | 105.722 | 12.646 |
| 9 | + compiler vectorization | 0.70 | 1.87 | 30,272 | 196.341 | 23.486 |
| 10 | + AVX intrinsics | 0.39 | 1.76 | 53,292 | 352.408 | 41.677 |
| 11 | Intel MKL | 0.41 | 0.97 | 51,497 | 335.217 | 40.098 |

Version 10 is competitive with Intel's professionally engineered Math Kernel Library!

Performance Engineering

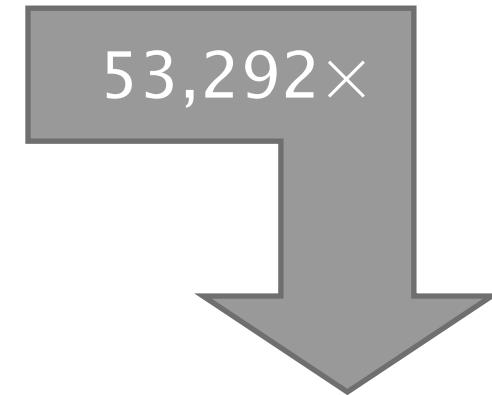


Courtesy of [stevepj2009](#) on Flickr. Used under CC-BY.

- You won't generally see the magnitude of performance improvement we obtained for matrix multiplication.
- But in 6.172, you will learn how to print the currency of performance all by yourself.

Gas economy
MPG

53,292×



Courtesy of [pngimg](#).
Used under CC-BY-NC.

MIT OpenCourseWare
<https://ocw.mit.edu>

6.172 Performance Engineering of Software Systems
Fall 2018

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.