

# Performance Engineering of Software Systems



**LECTURER**  
**Xuhao Chen**  
*Fall 2024*



# Performance Engineering of Software Systems



**Slack:** [xxx.slack.com](https://xxx.slack.com)  
**Canvas:** [canvas.mit.edu/courses/16631](https://canvas.mit.edu/courses/16631)

→ Read Course Info  
→ HW0 — due tonight!  
→ Attend ANY recitation **TOMORROW:**

10am-12pm	@ 26-322
1-3pm	@ 34-301 or 34-302
3-5pm	@ 34-302 or 34-304

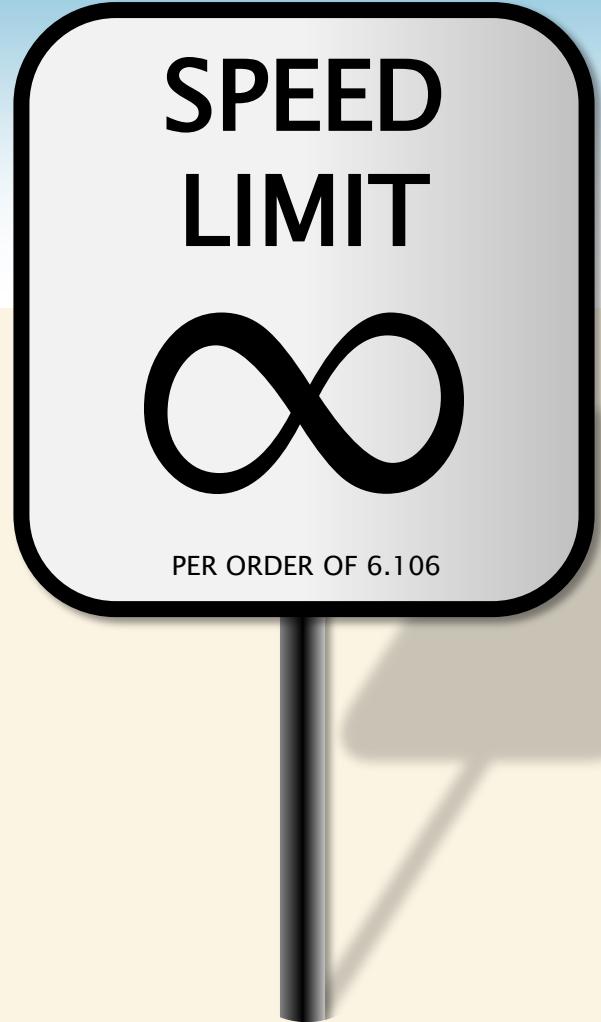


Performance  
Engineering of  
Software Systems

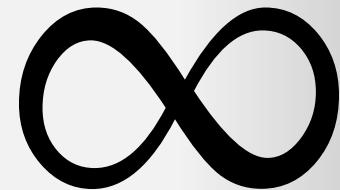


LECTURE 1  
Introduction &  
Matrix Multiplication  
Xuhao Chen





SPEED  
LIMIT



PER ORDER OF 6.106

# WHY SOFTWARE PERFORMANCE ENGINEERING?

# Is Performance Important?

What software properties are more important than performance?

- Functionality
- Correctness
- Security

# Is Performance Important?

What software properties are more important than performance?

- Compatibility
  - Correctness
  - Clarity
  - Debuggability
  - Functionality
  - Maintainability
  - Modularity
  - Portability
  - Reliability
  - Robustness
  - Security
  - Usability
- … and more.

# Is Performance Important?

What software properties are more important than performance?

- Compatibility
  - Correctness
  - Clarity
  - Debuggability
  - Functionality
  - Maintainability
  - Modularity
  - Portability
  - Reliability
  - Robustness
  - Security
  - Usability
- … and more.

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

# Is Performance Important?

What software properties are more important than performance?

- Compatibility
  - Correctness
  - Clarity
  - Debuggability
  - Functionality
  - Maintainability
  - Modularity
  - Portability
  - Reliability
  - Robustness
  - Security
  - Usability
- … and more.

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



# Analogy for Performance



# Is Performance Important?

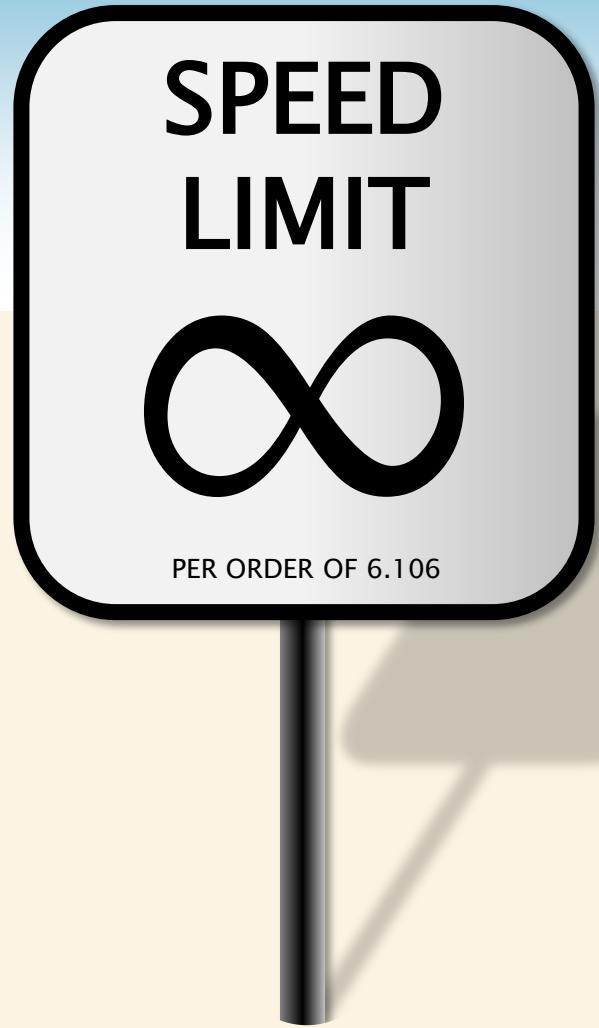
What software properties are more important than performance?

- Compatibility
  - Correctness
  - Clarity
  - Debuggability
  - Functionality
  - Maintainability
  - Modularity
  - Portability
  - Reliability
  - Robustness
  - Security
  - Usability
- … and more.

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

Performance is the **currency** of computing. You can often “buy” needed properties with performance.

# A BRIEF HISTORY OF PERFORMANCE ENGINEERING



# Computer Programming in the Early Days

Long ago, software performance engineering was common, because machine resources were limited.

IBM System/360



DEC PDP-11



Apple II



**Performance Engineering Ruled!**

Launched: 1964

Clock rate: 33 KHz

Data path: 32 bits

Memory: 524 Kbytes

Cost: \$250,000

Launched: 1970

Clock rate: 1.25 MHz

Data path: 16 bits

Memory: 56 Kbytes

Cost: \$20,000

Launched: 1977

Clock rate: 1 MHz

Data path: 8 bits

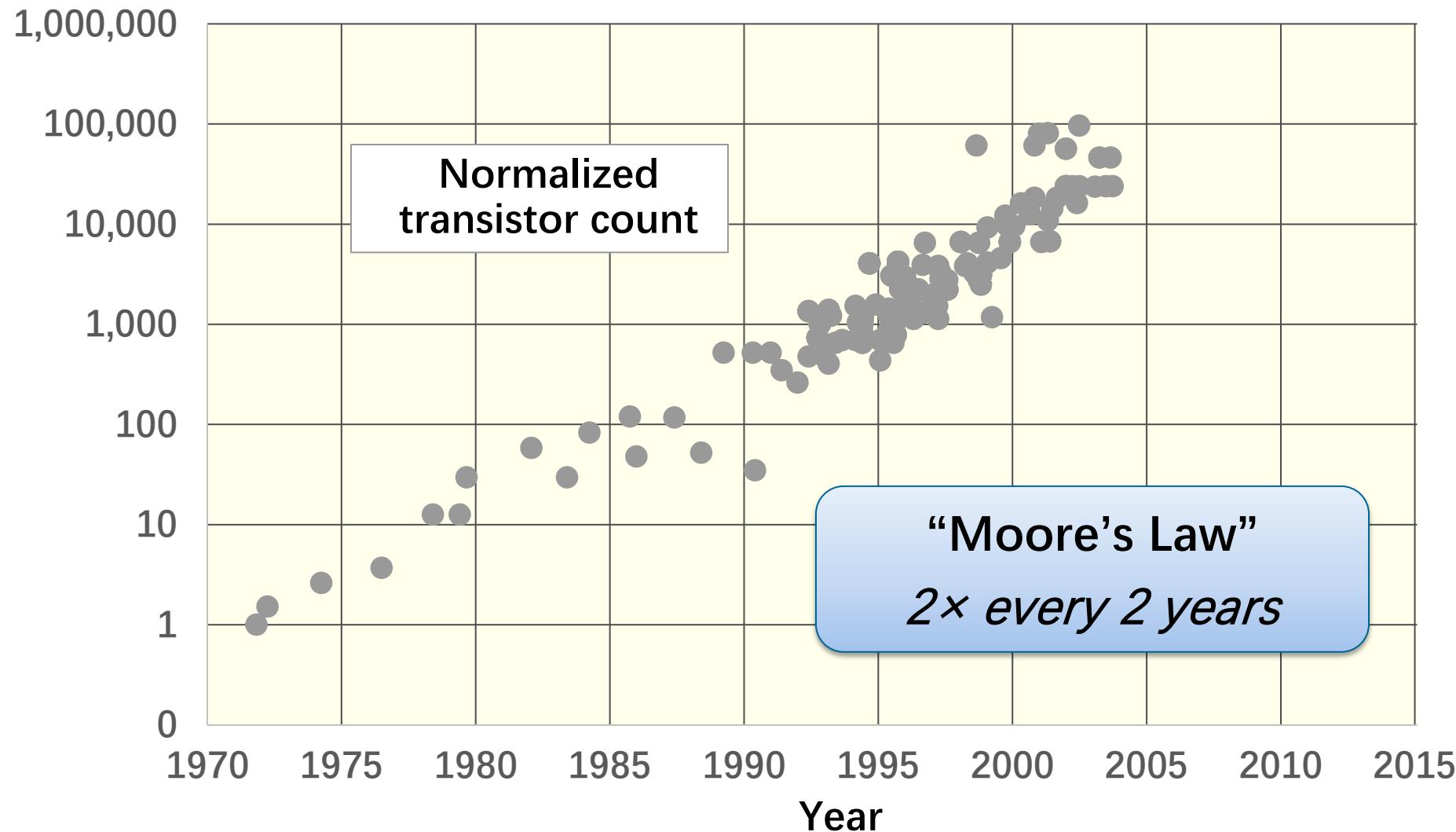
Memory: 48 Kbytes

Cost: \$1,395

**Many applications strained machine resources.**

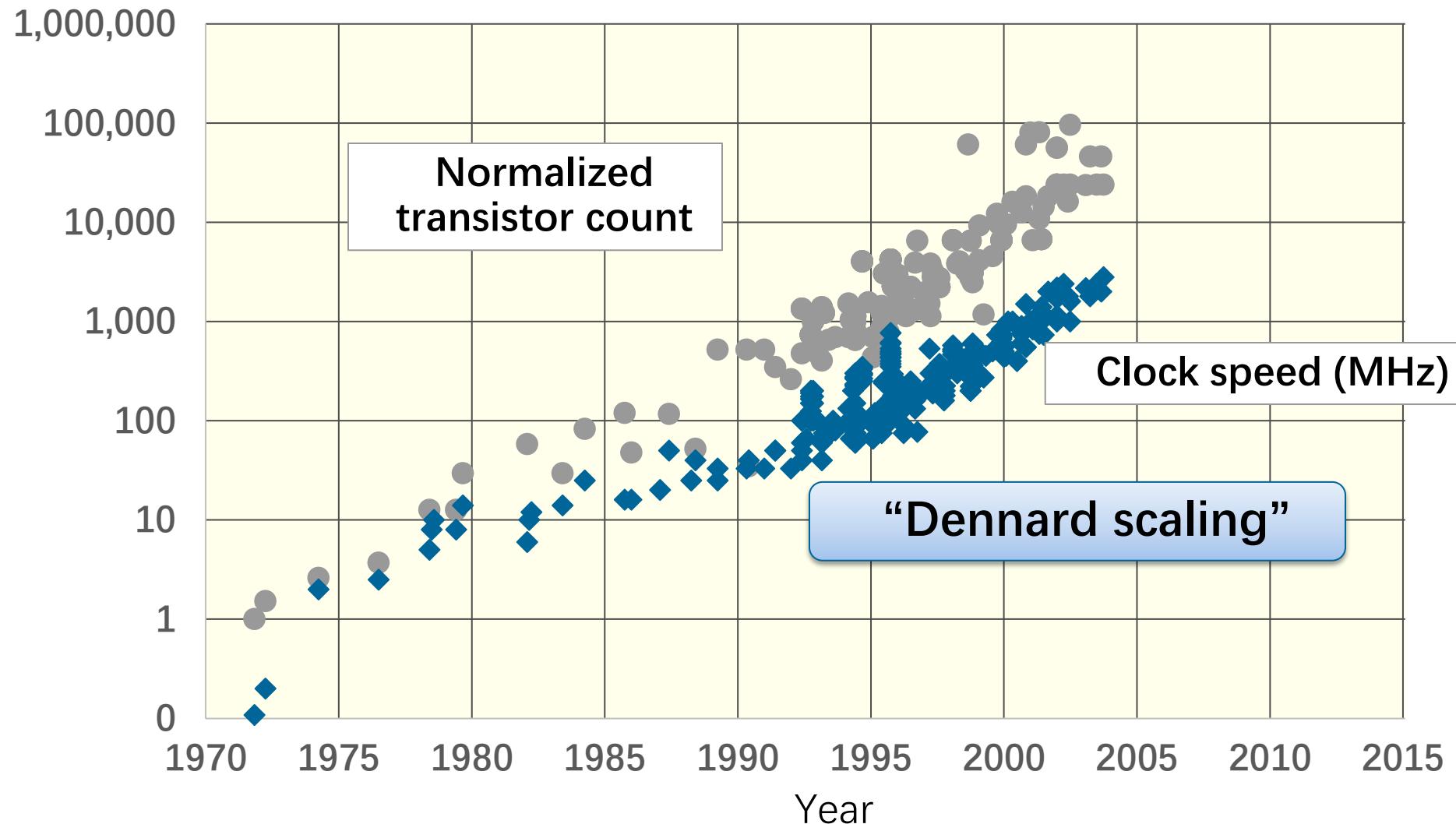
- Programs had to be planned around the machine.
- Many programs would not “fit” without intense performance engineering.

# Technology Scaling from 70's to 2004



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

# Technology Scaling from 70's to 2004



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

# Advances in Hardware

Apple computers with similar prices from 1977 to 2004



**Apple II**

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



**Power Macintosh G4**

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



**Power Macintosh G5**

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

# Lessons Learned in the Beginning of this Era



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

*William A. Wulf*

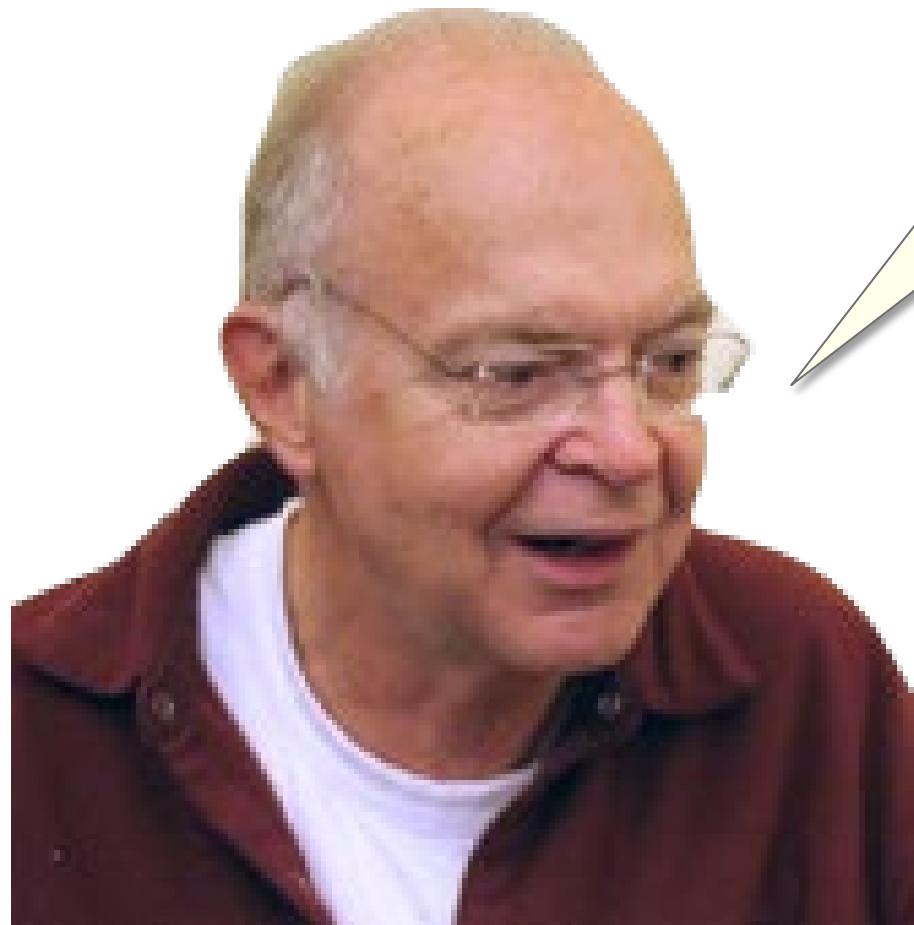
# Lessons Learned in the Beginning of this Era

The First Rule of Program Optimization: Don't do it.  
The Second Rule of Program Optimization — For experts only:  
Don't do it yet. []88]



*Michael A. Jackson*

# Lessons Learned in the Beginning of this Era



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

*Donald E. Knuth*

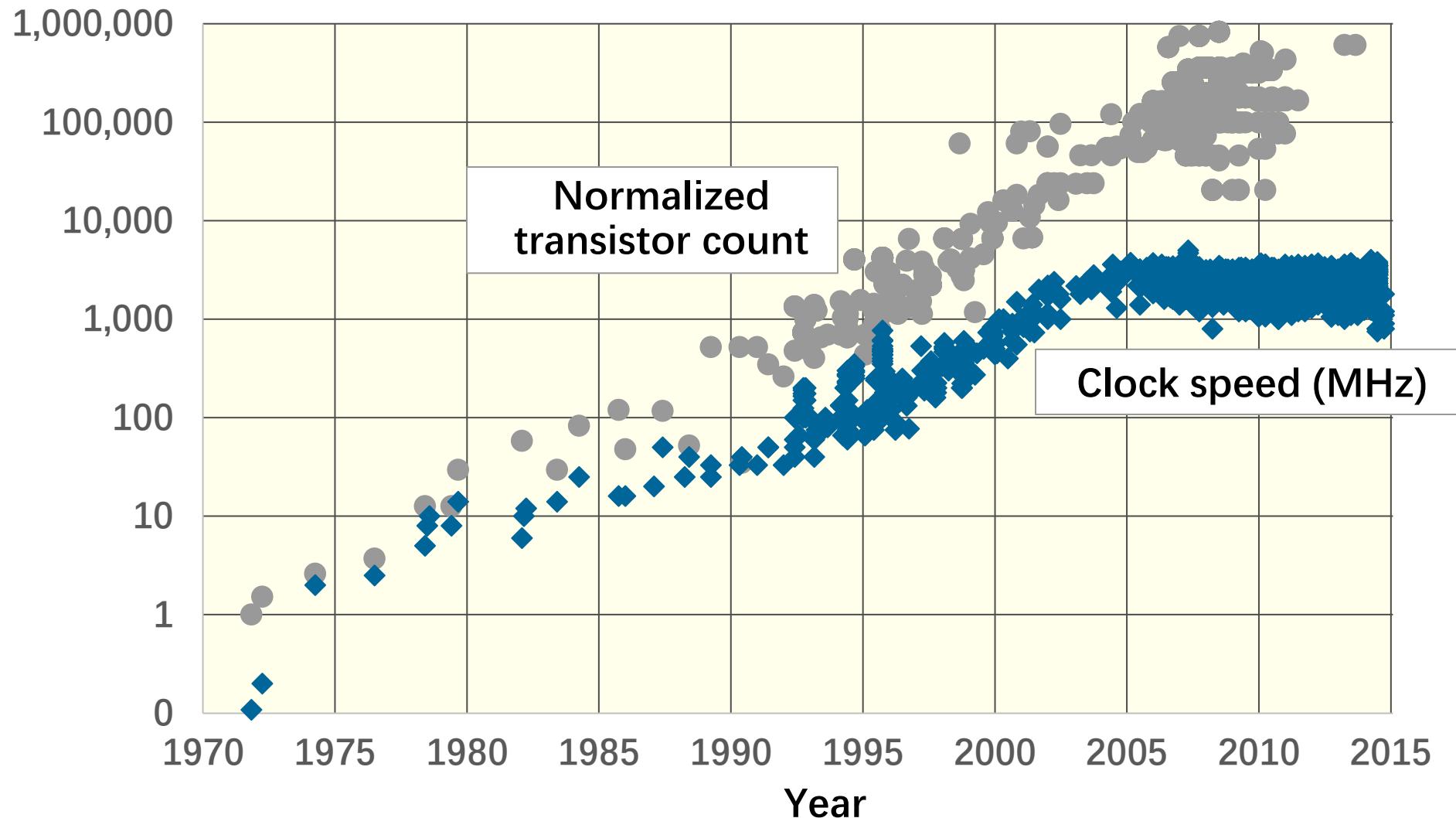
# Until 2004

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

## Performance Engineering Ruled!

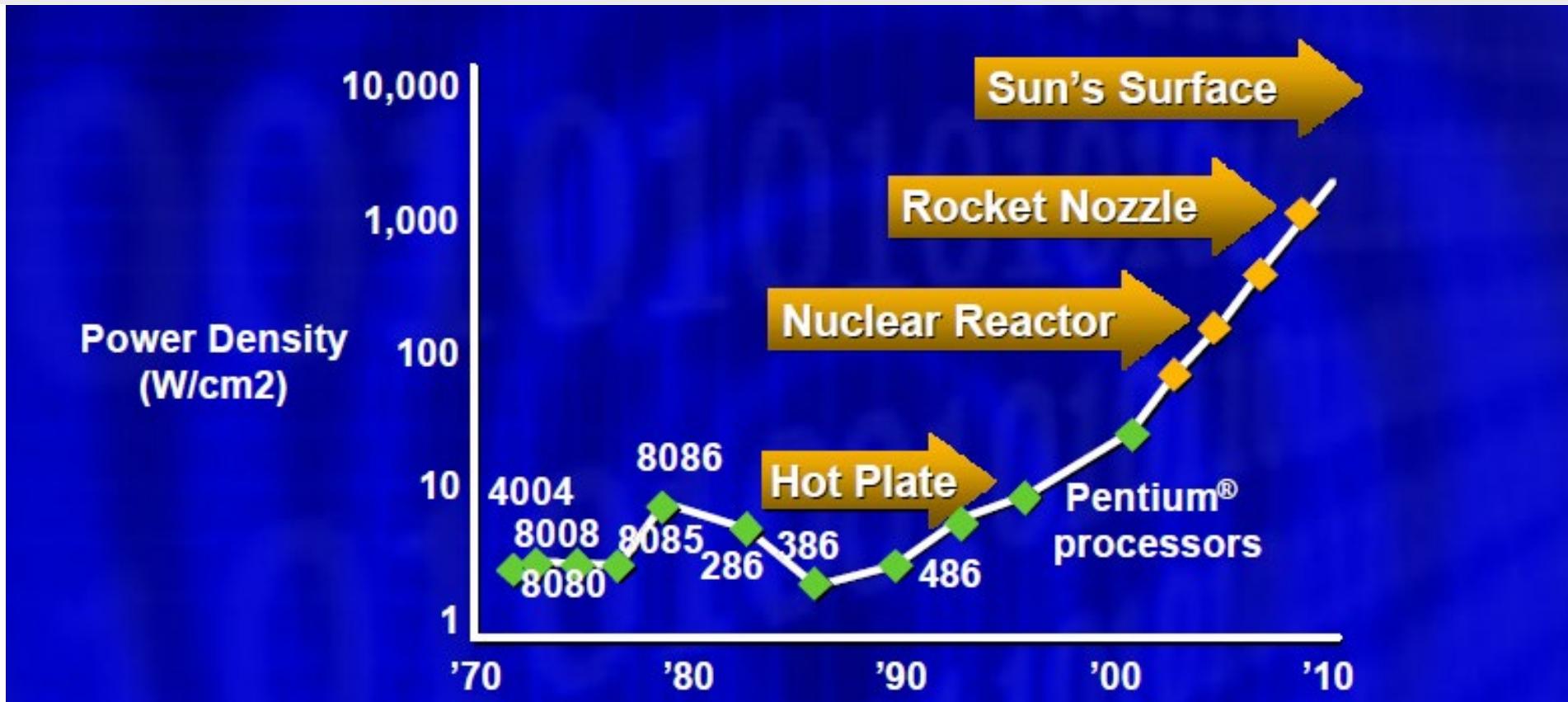


# Technology Scaling After 2004



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

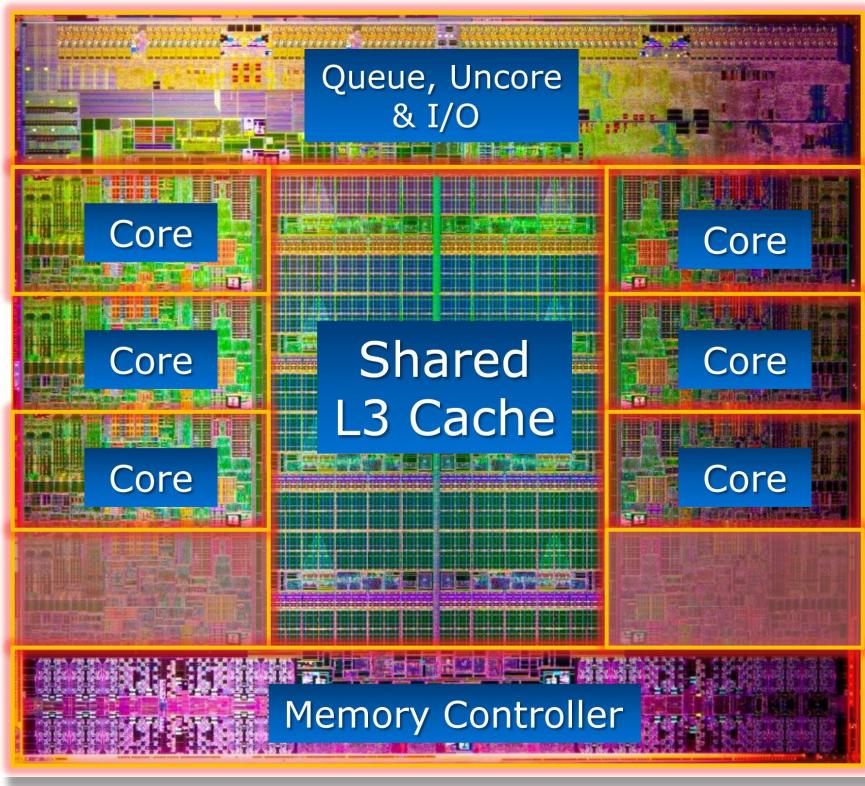
# Power Density



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

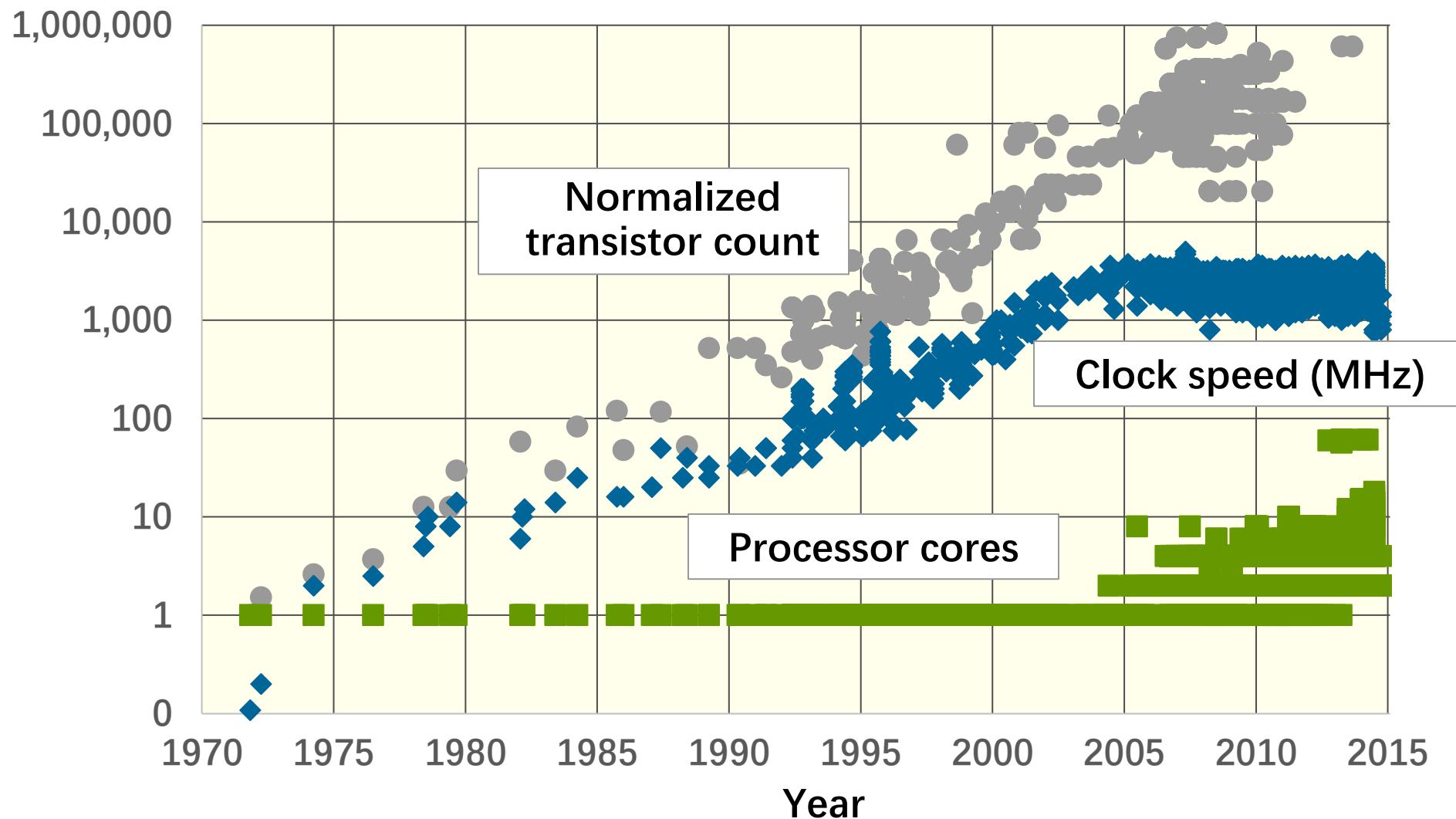


Intel Core i7 3960X  
(Sandy Bridge E), 2011

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

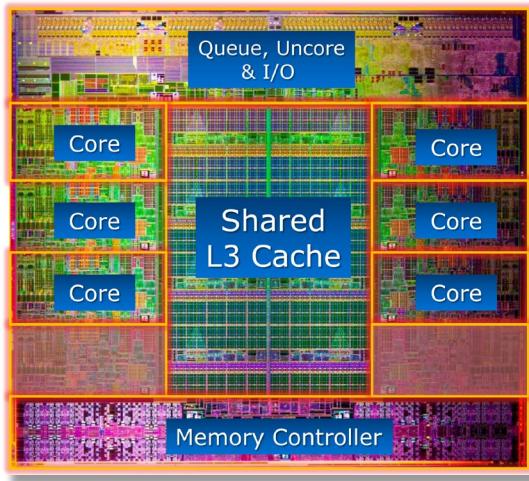
- To scale performance, processor manufacturers put many processing cores on the microprocessor chip.
- Each generation of Moore's Law potentially doubles the number of cores.

# Technology Scaling



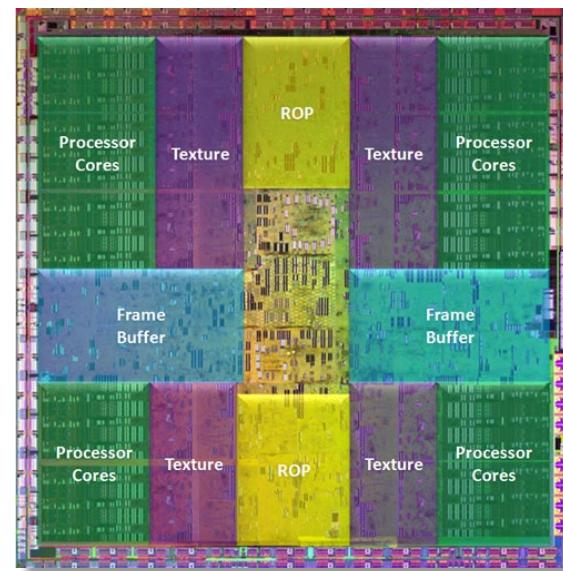
Processor data from Stanford's CPU DB [DKM12].

# Performance Is No Longer Free



2011 Intel  
Skylake  
processor

2008  
NVIDIA  
GT200  
GPU



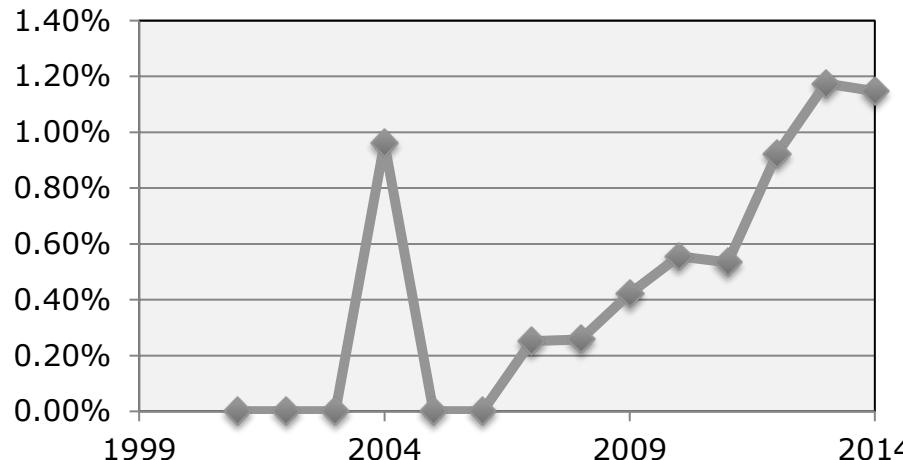
Moore's Law continued to increase computer performance.

But now that performance was available in the form of multicore processors with complex cache hierarchies, wide vector units, GPU's, FPGA's, etc.

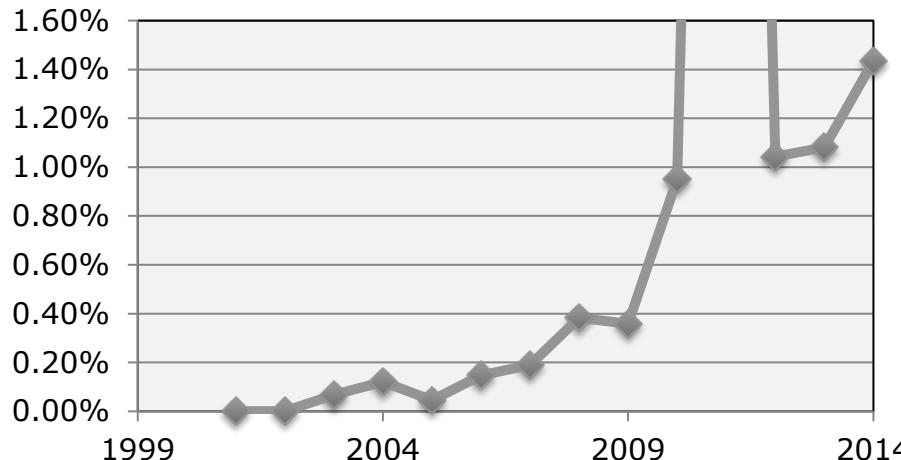
Generally, software must be **adapted** to utilize this hardware efficiently!

# Software Bugs Mentioning “Performance”

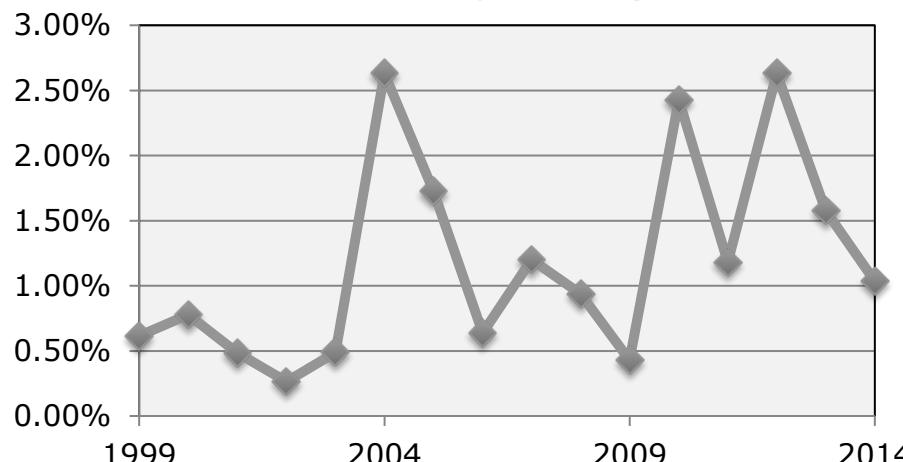
Bug reports for Mozilla “Core”



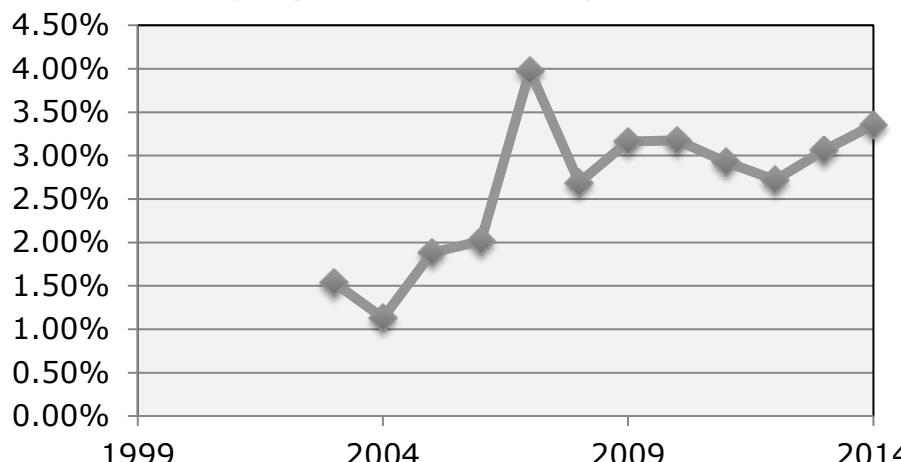
Commit messages for MySQL



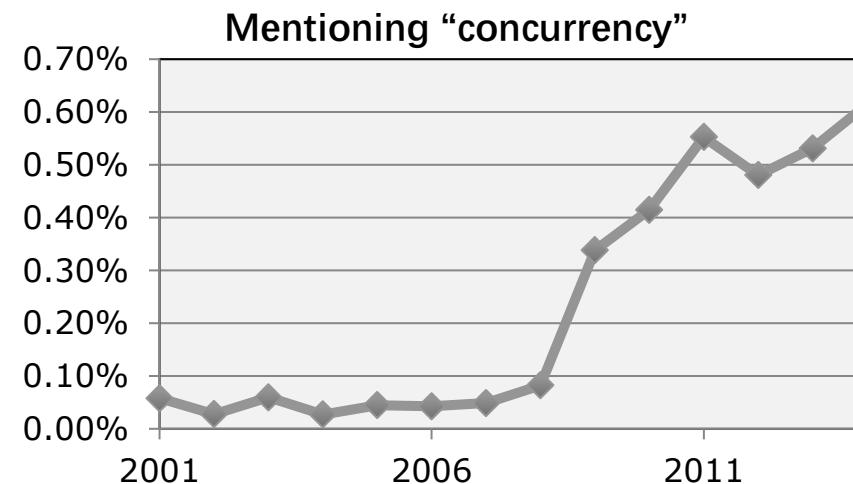
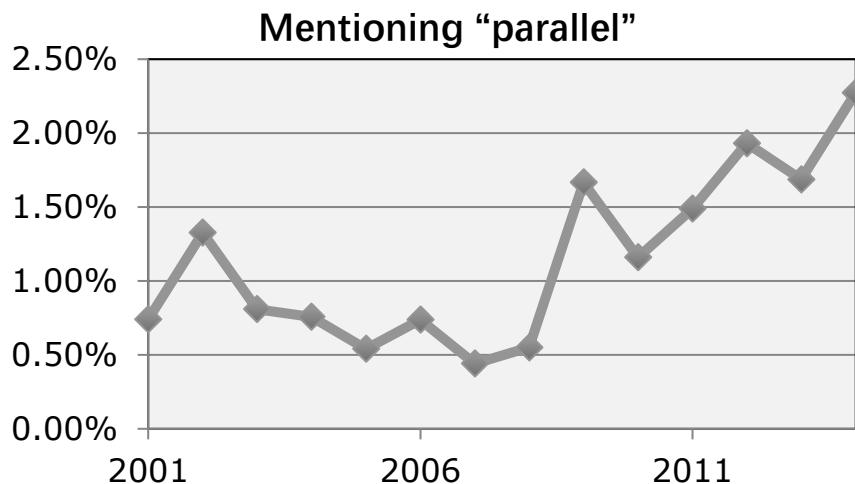
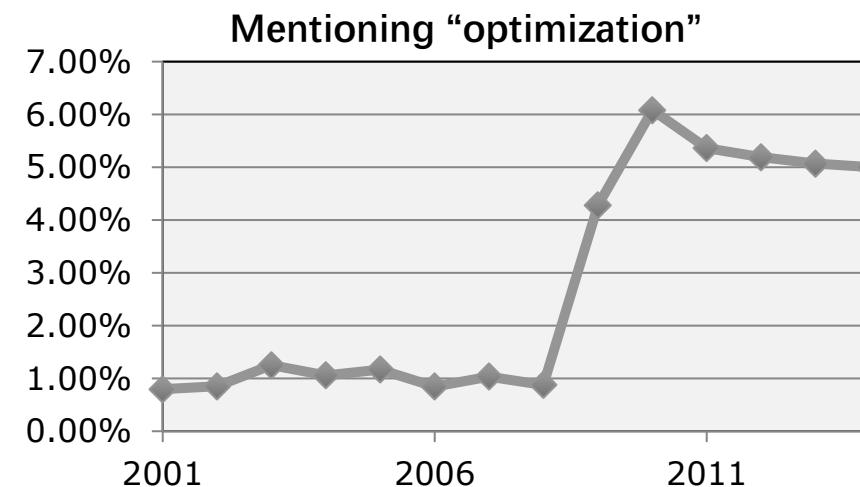
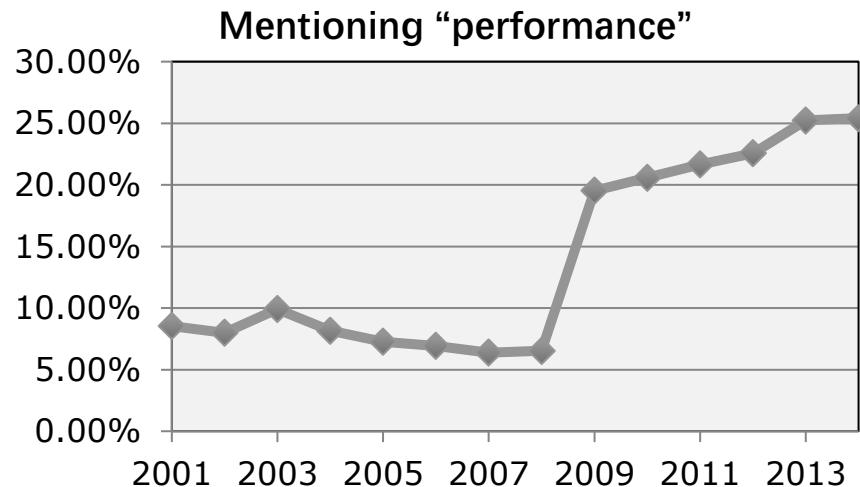
Commit messages for OpenSSL



Bug reports for the Eclipse IDE



# Software Developer Jobs



Source: Monster.com

# And Now, Moore's Law Is Over!



# Where Are We Now?

- Intel achieved 14 nanometers in 2014.
- **Doubling every two years**, according to Moore's Law, means that Intel should have achieved
  - 10 nanometers in 2016,
  - 7 nanometers in 2018,
  - 5 nanometers in 2020.
- But Intel did not release 10 nanometers until 2019!
- It took 5 years for what historically had taken only 2 years.

Semiconductor technology will no longer give applications free performance.

# Why Must the Party End?



# Darn That Physics!

- It's implausible that semiconductor technologists can make **wires thinner than atoms**, which are at most a few angstroms across.
- The silicon lattice constant is 0.543 nanometers = 5.43 angstroms.

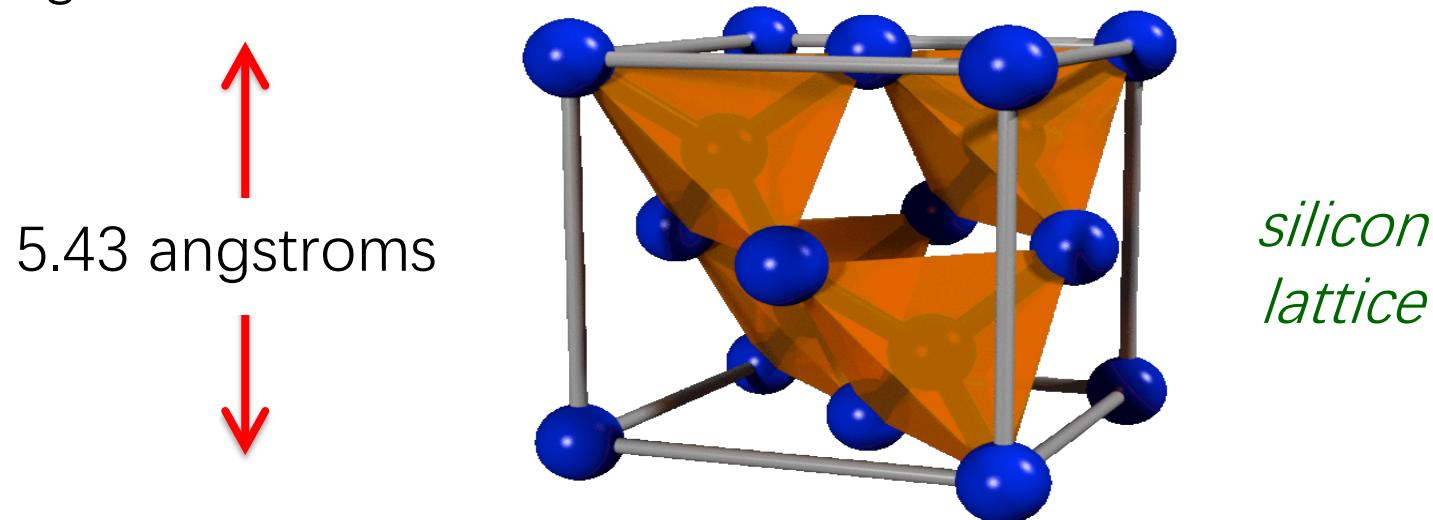


Image by Pieter Kuiper, Wikipedia Commons.

- **Technology roadmaps** see an end to transistor scaling around 5 nanometers. We're almost there!

# The Printing Press Is Grinding to a Halt

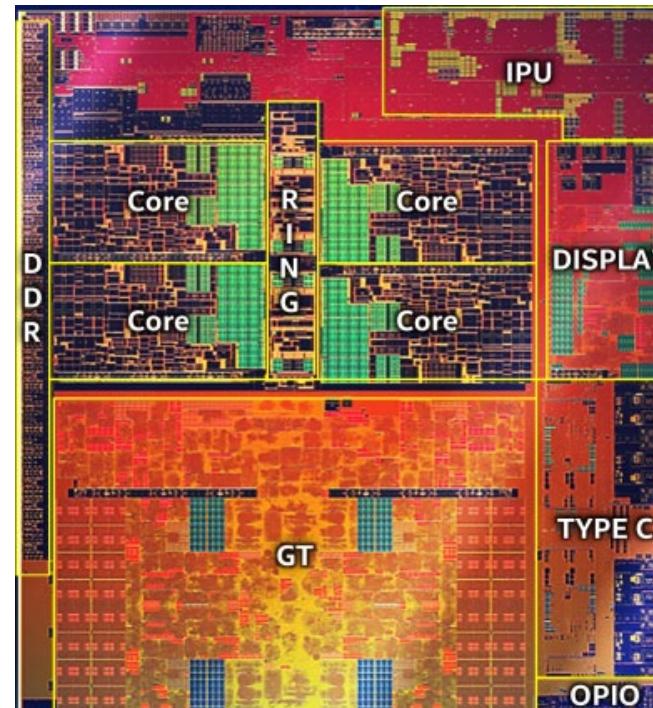


# Performance Engineering Redux

A modern multicore desktop processor contains

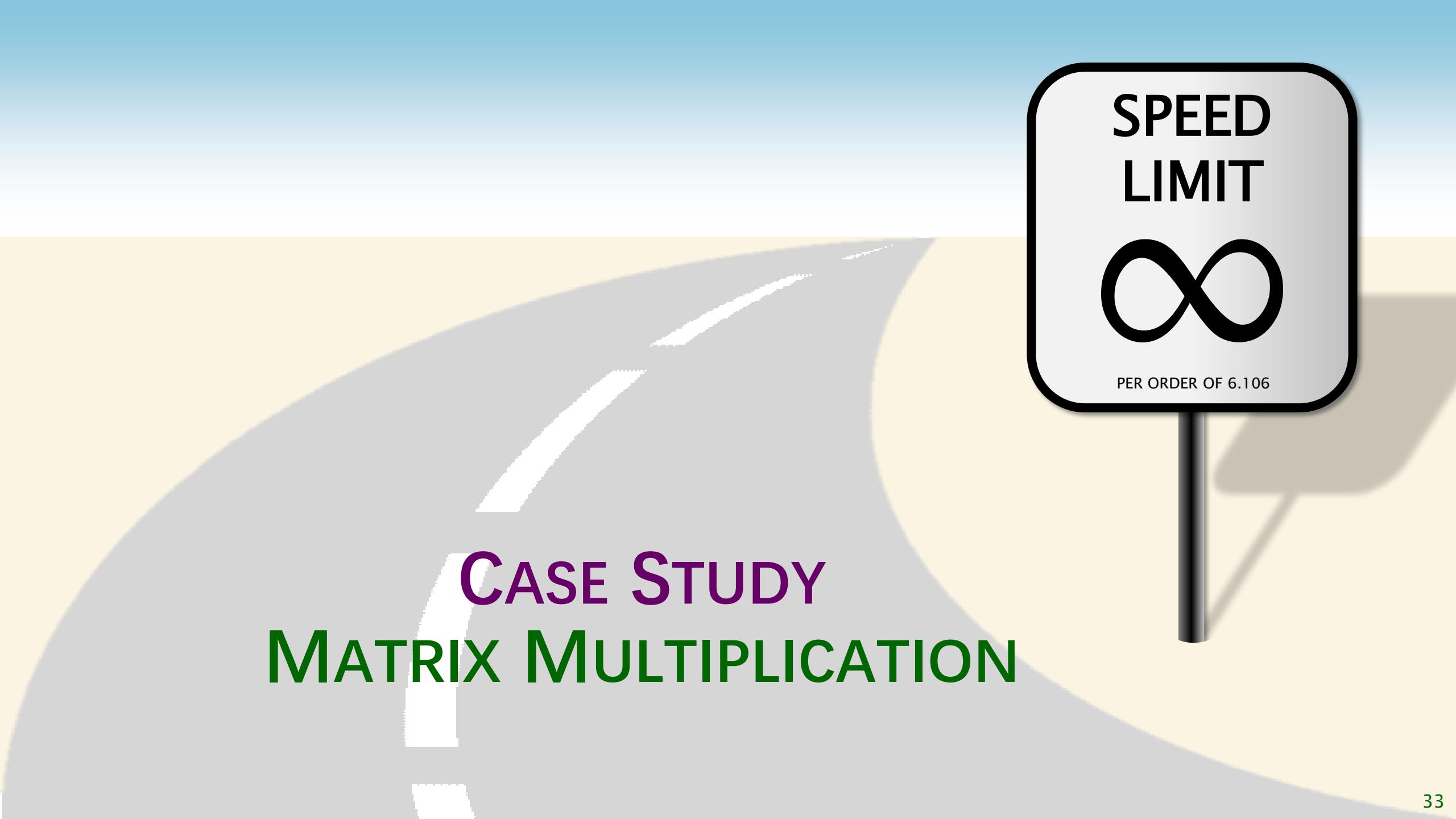
- parallel-processing cores
- vector units
- caches
- instruction prefetchers
- GPU's
- hyperthreading
- dynamic frequency scaling
- etc

These features can be challenging to exploit.

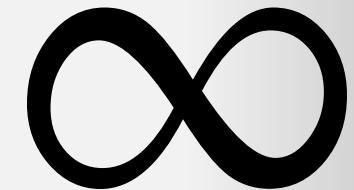


*2019 Intel 10nm processor*

**In this class you will learn the principles  
and practice of writing fast code.**



SPEED  
LIMIT



PER ORDER OF 6.106

# 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)
```

# 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:

- ≈ 6 microseconds?
- ≈ 6 milliseconds?
- ≈ 6 seconds?
- ≈ 6 hours?
- ≈ 6 days?

# 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

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

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

# Where We Stand So Far

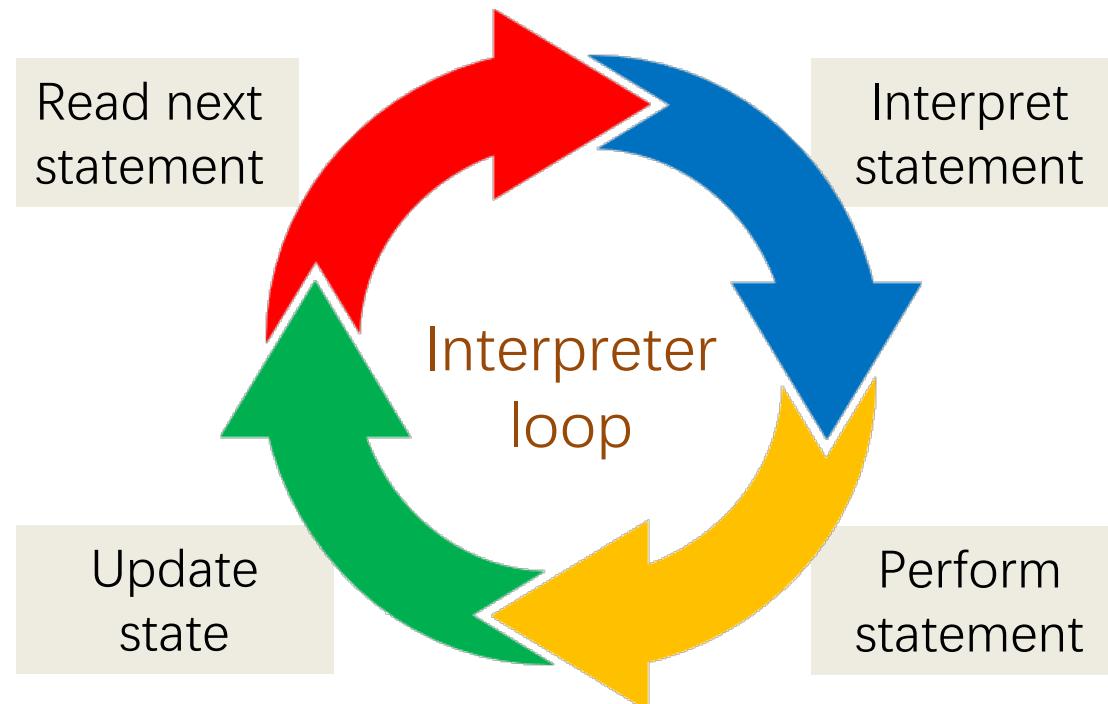
Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.007	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.119	0.014

## 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.
- **Future executions** of that code use the more-efficient **compiled** version.

# 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

# 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.

```
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

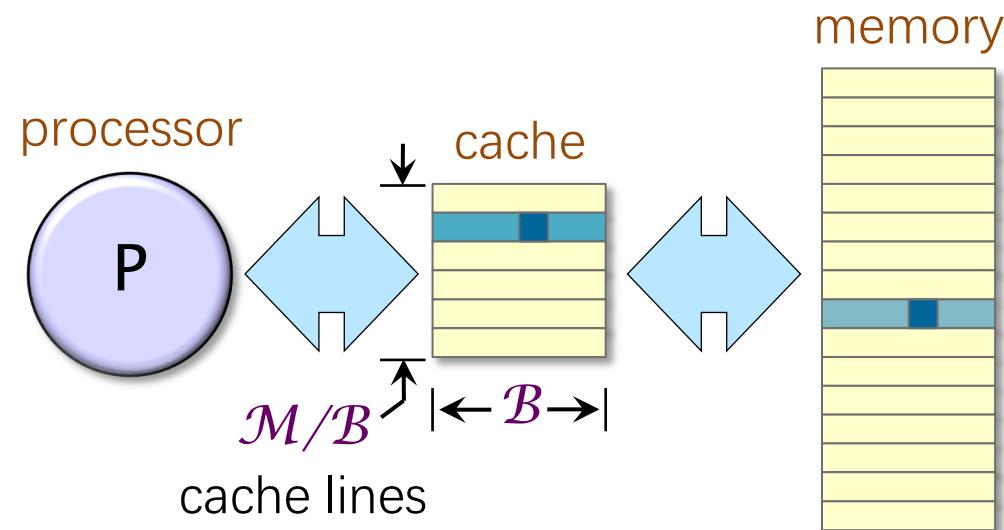
Loop order affects running time by a factor of 18!

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.



# 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

# 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.

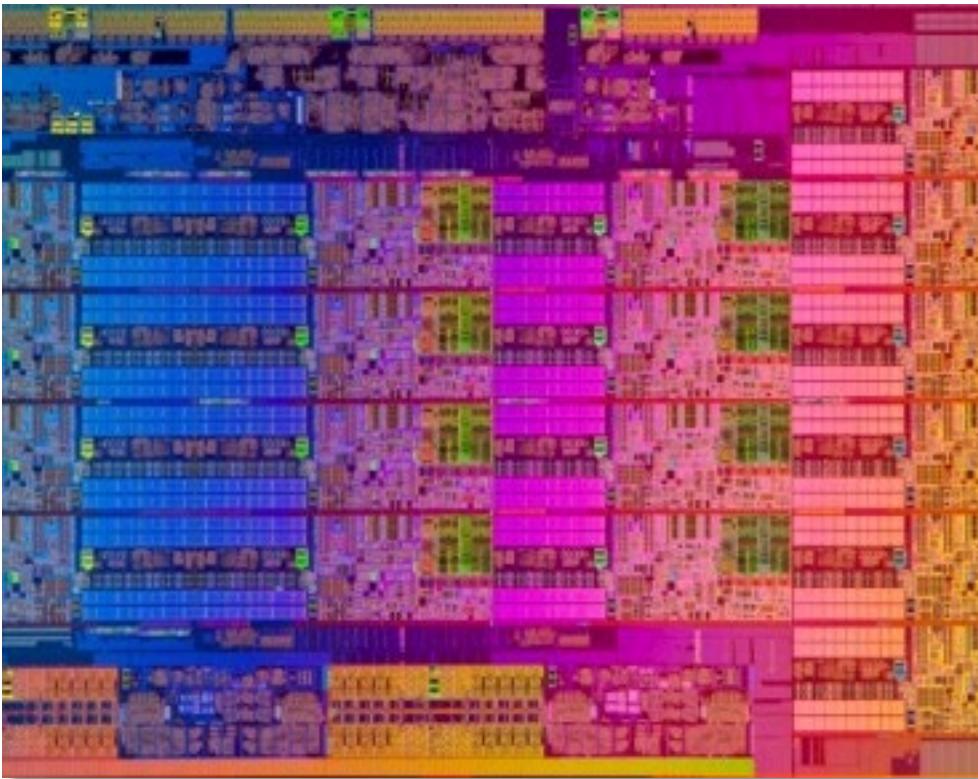
# Version 5: Optimization Flags

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.

*Where can we get more 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!*

# Parallel Loops

A `cilk_for` loop enables 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];
```

Both of these loops  
can be parallelized.

*Which parallel version works best?*

- parallelize just the `i` loop,
- parallelize just the `j` loop, or
- parallelize both the `i` and `j` loops.

# 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

Parallel **i** and **j** loops

```
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

Parallelizing the **i** loop yields a speedup of almost  $18\times$  on 18 cores!

- **Disclaimer:** It's rarely this easy to parallelize code effectively. Most code requires far more creativity to achieve a good speedup.

# 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

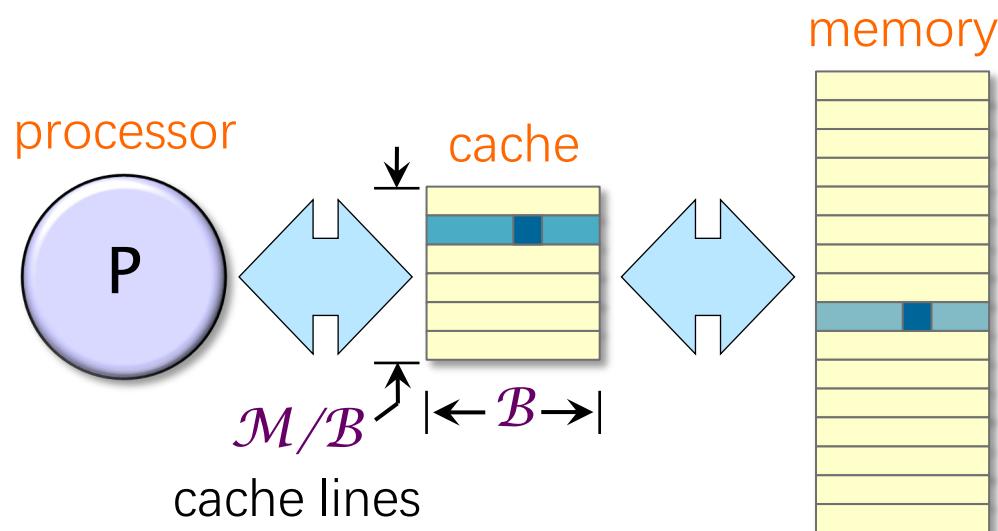
Using parallel loops gets us almost  $18\times$  speedup on  $18$  cores!  
(Disclaimer: Not all code is so easy to parallelize effectively.)

*Why are we still getting less than 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.



# D&C Matrix Multiplication

For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

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

**IDEA:** Divide the matrices into  $(n/2) \times (n/2)$  submatrices.

# D&C Matrix Multiplication

**IDEA:** For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$
$$= \left[ \begin{array}{cc} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{array} \right] + \left[ \begin{array}{cc} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{array} \right]$$

1. Compute  $C_{00} += A_{00}B_{00}$ ;  $C_{01} += A_{00}B_{01}$ ;  $C_{10} += A_{10}B_{00}$ ; and  $C_{11} += A_{10}B_{01}$  recursively in parallel.
2. Compute  $C_{00} += A_{01}B_{10}$ ;  $C_{01} += A_{01}B_{11}$ ;  $C_{10} += A_{11}B_{10}$ ; and  $C_{11} += A_{11}B_{11}$  recursively in parallel.

# 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;
    }
}
```

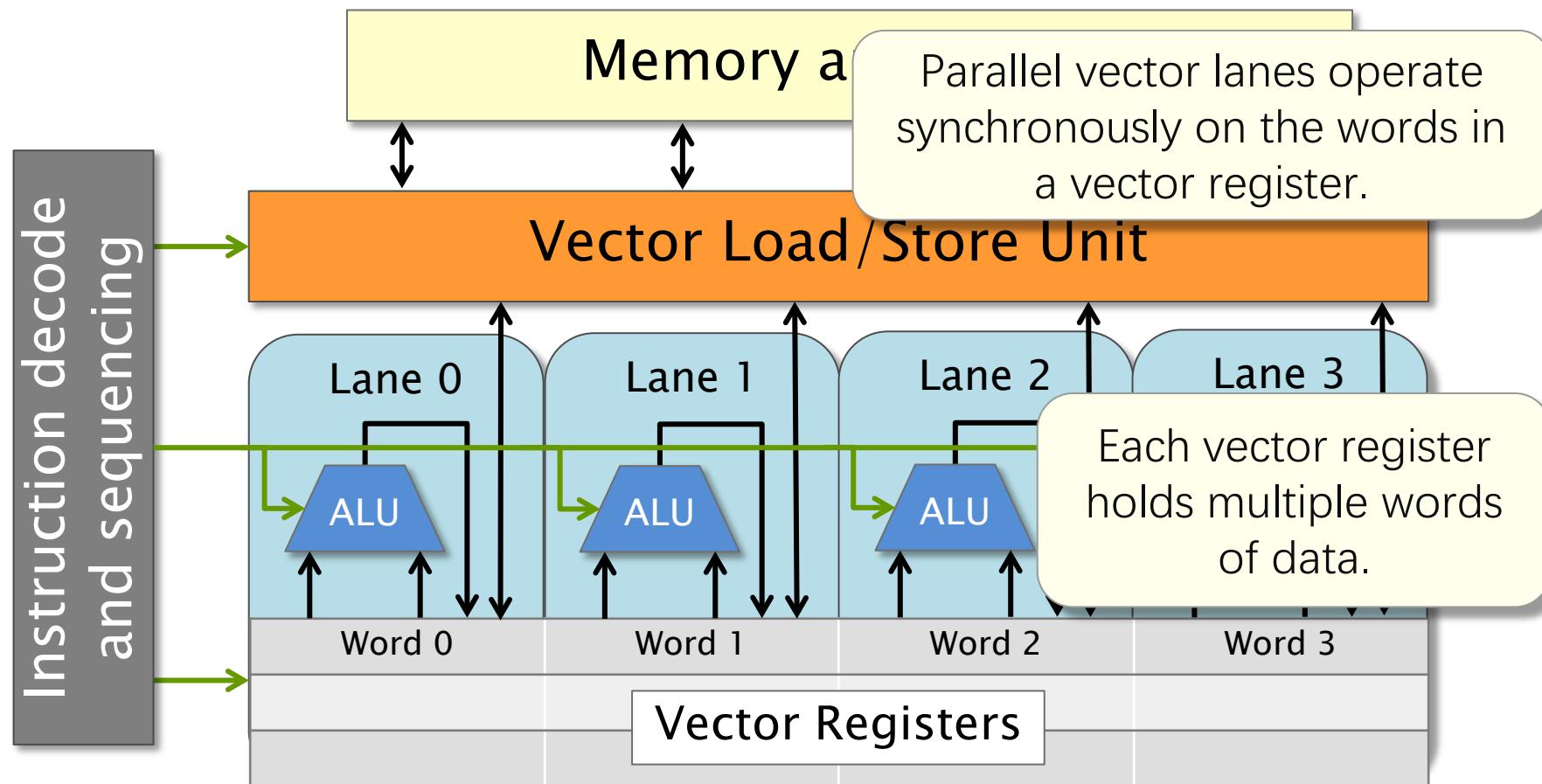
# Version 7: Parallel Divide-and-Conquer

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	Parallel divide-and-conquer	1.30	2.35	16,197	105.722	12.646

Implementation	Cache references × 10 <sup>6</sup>	Cache references × 10 <sup>6</sup>	L1-d cache misses × 10 <sup>6</sup>
Parallel loops	104,090	17,220	8,600
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.

Also, using AVX instructions slows down the microprocessor clock speed by about 20%!

# Version 8: 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	Parallel divide-and-conquer	1.30	2.35	16,197	105.722	12.646
8	+ 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/>

 **Intrinsics Guide**

**Technologies**

- MMX
- SSE
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX-512
- KNC
- SVML
- Other

**Categories**

- Application-Targeted
- Arithmetic
- Bit Manipulation
- Cast
- Compare
- Convert
- Cryptography
- Elementary Math Functions
- General Support
- Load
- Logical

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.

\_mm\_search ?

<code>__m256i _mm256_abs_epi16 (__m256i a)</code>	vpabsw
<code>__m256i _mm256_abs_epi32 (__m256i a)</code>	vpabsd
<code>__m256i _mm256_abs_epi8 (__m256i a)</code>	vpabsb
<code>__m256i _mm256_add_epi16 (__m256i a, __m256i b)</code>	vpaddw
<code>__m256i _mm256_add_epi32 (__m256i a, __m256i b)</code>	vpaddd
<code>__m256i _mm256_add_epi64 (__m256i a, __m256i b)</code>	vpaddq
<code>__m256i _mm256_add_epi8 (__m256i a, __m256i b)</code>	vpaddb
<code>__m256d _mm256_add_pd (__m256d a, __m256d b)</code>	vaddpd
<code>__m256 _mm256_add_ps (__m256 a, __m256 b)</code>	vaddps
<code>__m256i _mm256_adds_epi16 (__m256i a, __m256i b)</code>	vpaddsw
<code>__m256i _mm256_adds_epi8 (__m256i a, __m256i b)</code>	vpaddsb
<code>__m256i _mm256_adds_epu16 (__m256i a, __m256i b)</code>	vpaddusw
<code>__m256i _mm256_adds_epu8 (__m256i a, __m256i b)</code>	vpaddusb
<code>__m256d _mm256_addsub_pd (__m256d a, __m256d b)</code>	vaddsubpd
<code>__m256 _mm256_addsub_ps (__m256 a, __m256 b)</code>	vaddsubps
<code>__m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int count)</code>	vpalignr
<code>__m256d _mm256_and_pd (__m256d a, __m256d b)</code>	vandpd
<code>__m256 _mm256_and_ps (__m256 a, __m256 b)</code>	vandps
<code>__m256i _mm256_and_si256 (__m256i a, __m256i b)</code>	vpand
<code>__m256d _mm256_andnot_pd (__m256d a, __m256d b)</code>	vandnpd
<code>__m256 _mm256_andnot_ps (__m256 a, __m256 b)</code>	vandnps

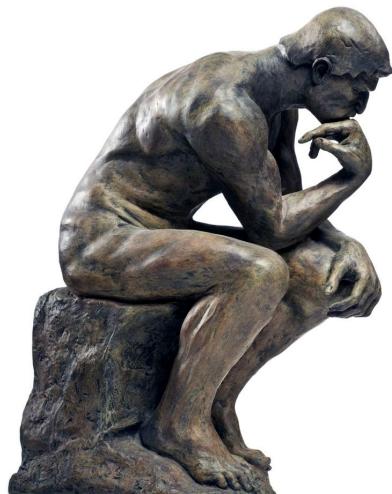
# Plus More Optimizations

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

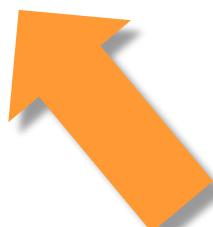
- Preprocessing
- Matrix transposition
- Data alignment
- Memory-management optimizations
- A clever algorithm for the base case that uses AVX intrinsic instructions explicitly

# Plus Performance Engineering

Think,



code,



run, run, run...



...to test and measure many  
different implementations

# Version 9: 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	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
8	+ compiler vectorization	0.70	2.35	30,272	196.341	23.486
9	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677

# Version 10: 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	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
8	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
9	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677
10	Intel MKL	0.41	0.97	51,497	335.217	40.098

Our Version 9 is competitive with Intel's professionally engineered Math Kernel Library (MKL)!

# Performance Engineering

- You won't generally see the magnitude of performance improvement we obtained for matrix multiplication.

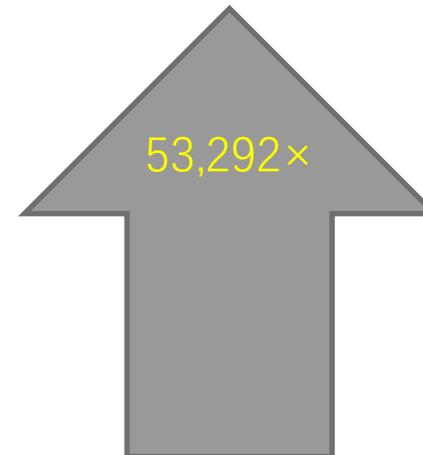
Galopagos  
Tortoise  
0.5 k/h



# Performance Engineering

- You won't generally see the magnitude of performance improvement we obtained for matrix multiplication.

Escape  
Velocity  
11 k/s



Galapagos  
Tortoise  
0.5 k/h

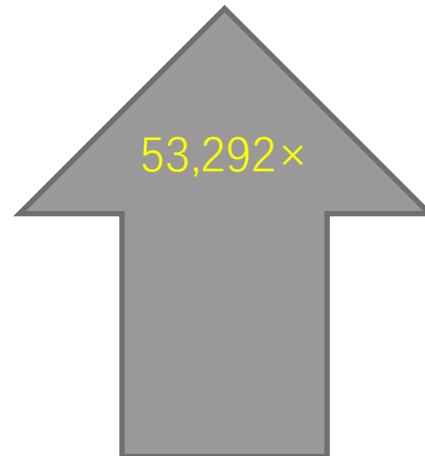


# Performance Engineering

- You won't generally see the magnitude of performance improvement we obtained for matrix multiplication.
- But this class will teach you how to **print the currency** of performance all by yourself.



Escape  
Velocity  
11 k/s



Galapagos  
Tortoise  
0.5 k/h