《软件系统优化》第1课
引言 & 矩阵乘法案例

郭健美

2021年秋

SOLE 系统优化实验室
华东师范大学

内容

- 引言
  - 为何需要软件系统优化?
  - 课程教学大纲
- 矩阵乘法案例

引言

为何需要软件系统优化?

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?

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.

There's Plenty of Room at the Bottom

An Invitation to Enter a New Field of Physics

by Richard P. Feynman

Miniaturizing the computer

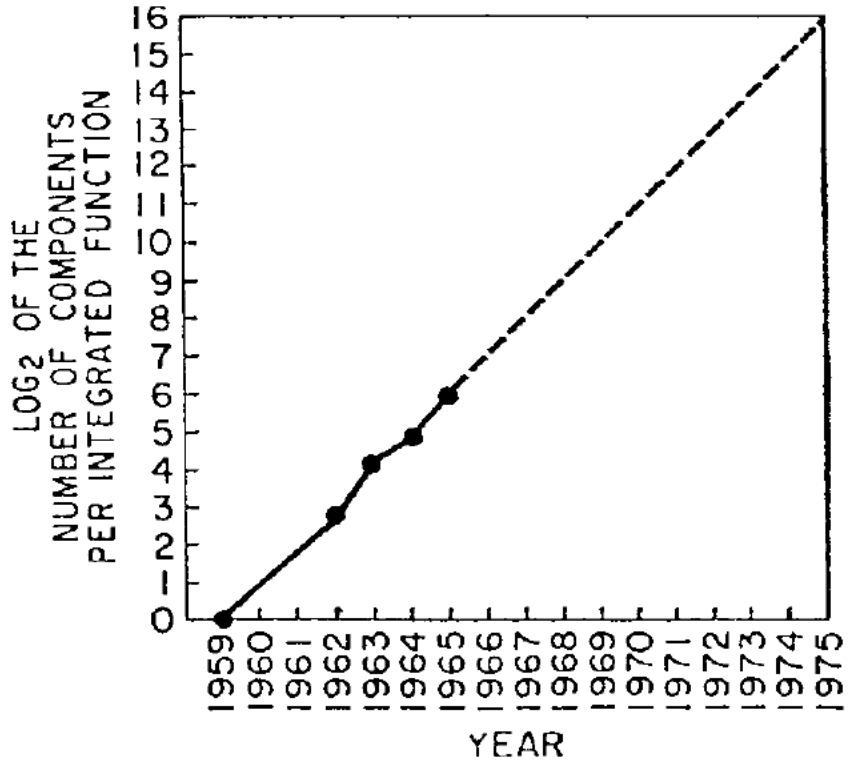
I don't know how to do this on a small scale in a practical way, but I do know that computing machines are very large; they fill rooms. Why can't we make them very small, make them of little wires, little elements---and by little, I mean *little*. For instance, the wires should be 10 or 100 atoms in diameter, and the circuits should be a few thousand angstroms across. Everybody who has analyzed the logical theory of computers has come

disadvantages. First, it requires too much material; there may not be enough germanium in the world for all the transistors which would have to be put into this enormous thing. There is also the problem of heat generation and power consumption; TVA would be needed to run the computer. But an even more practical difficulty is that the computer would be limited to a certain speed. Because of its large size, there is finite time required to get the information from one place to another. The information cannot go any faster than the speed of light---so, ultimately, when our computers get faster and faster and more and more elaborate, we will have to make them smaller and smaller.

But there is plenty of room to make them smaller. There is nothing that I can see in the physical laws that says the computer elements cannot be made enormously smaller than they are now. In fact, there may be certain advantages.

[R. P. Feynman, **There's plenty of room at the bottom**. Eng. Sci. 23, 22–36 (1960).]

Moore's Law 摩尔定律盛行半个世纪



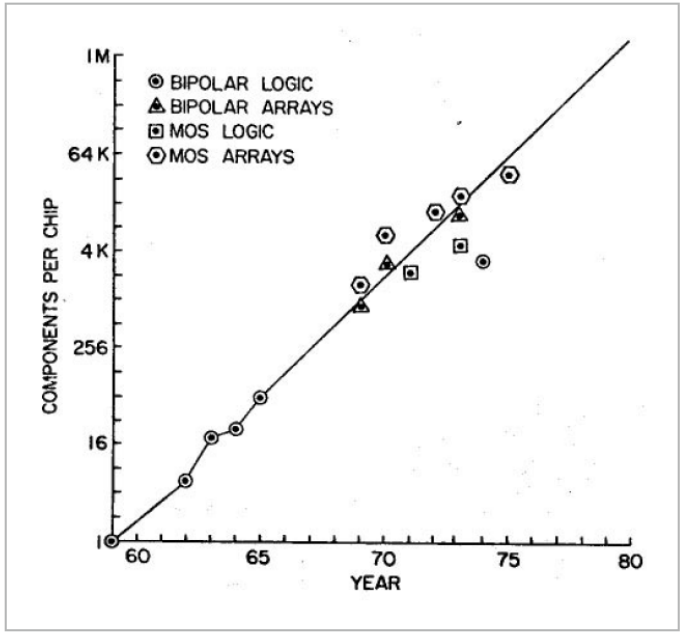
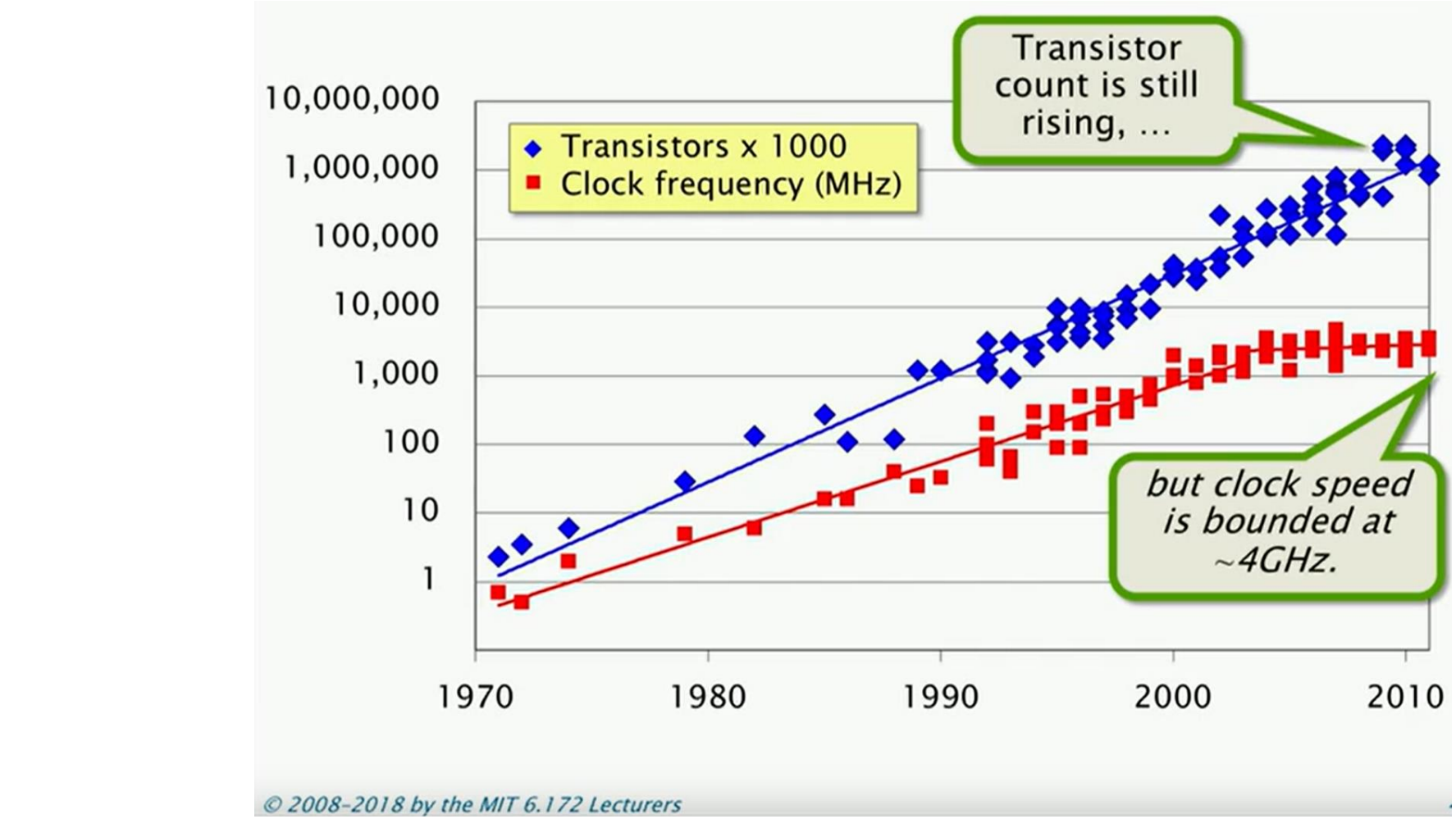


Figure 1 Approximate component count for complex integrated circuits vs. year of Introduction.

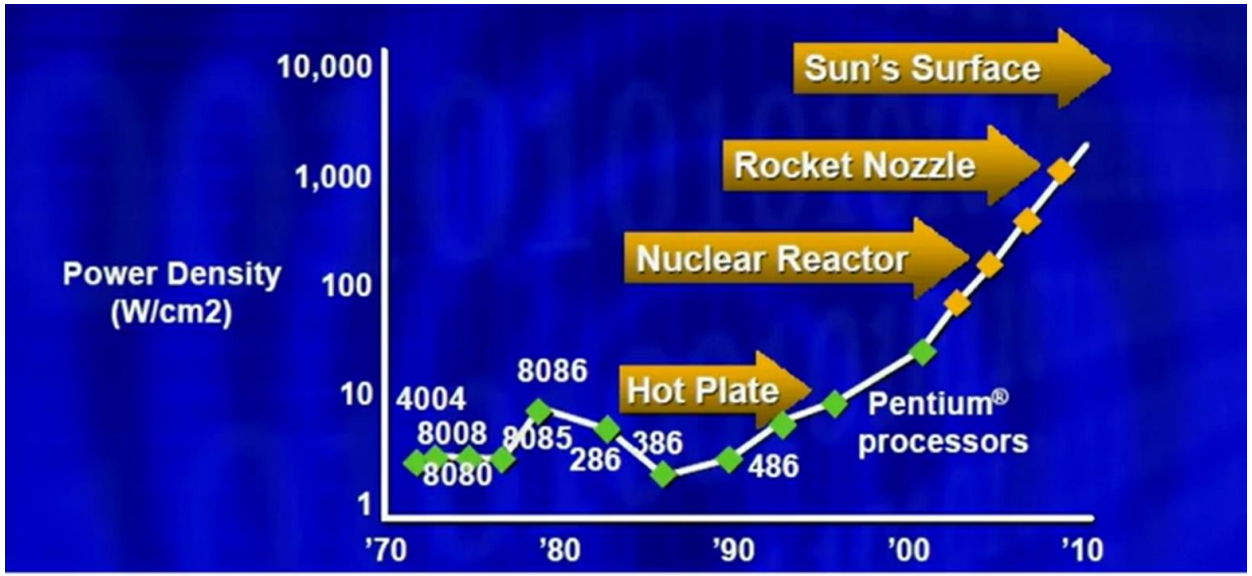
The number of transistors per computer chip would double every 2 years

[G. E. Moore, **Cramming more components onto integrated circuits**. Electronics 38, 1–4 (1965).]
[G. E. Moore, **Progress in digital integrated electronics**, in International Electron Devices Meeting Technical Digest (IEEE, 1975), pp. 11–13.]

摩尔定律逐渐失效

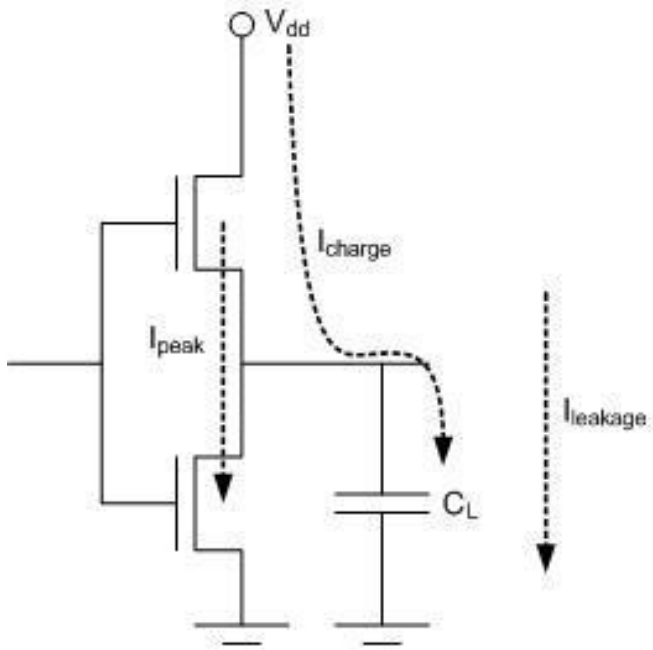

Transistor count is still rising, ...
Transistors x 1000
Clock frequency (MHz)
but clock speed is bounded at ~4GHz.
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
1970
1980
1990
2000
2010
© 2008–2018 by the MIT 6.172 Lecturers

Dennard Scaling 登纳德缩放定律逐渐失效



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

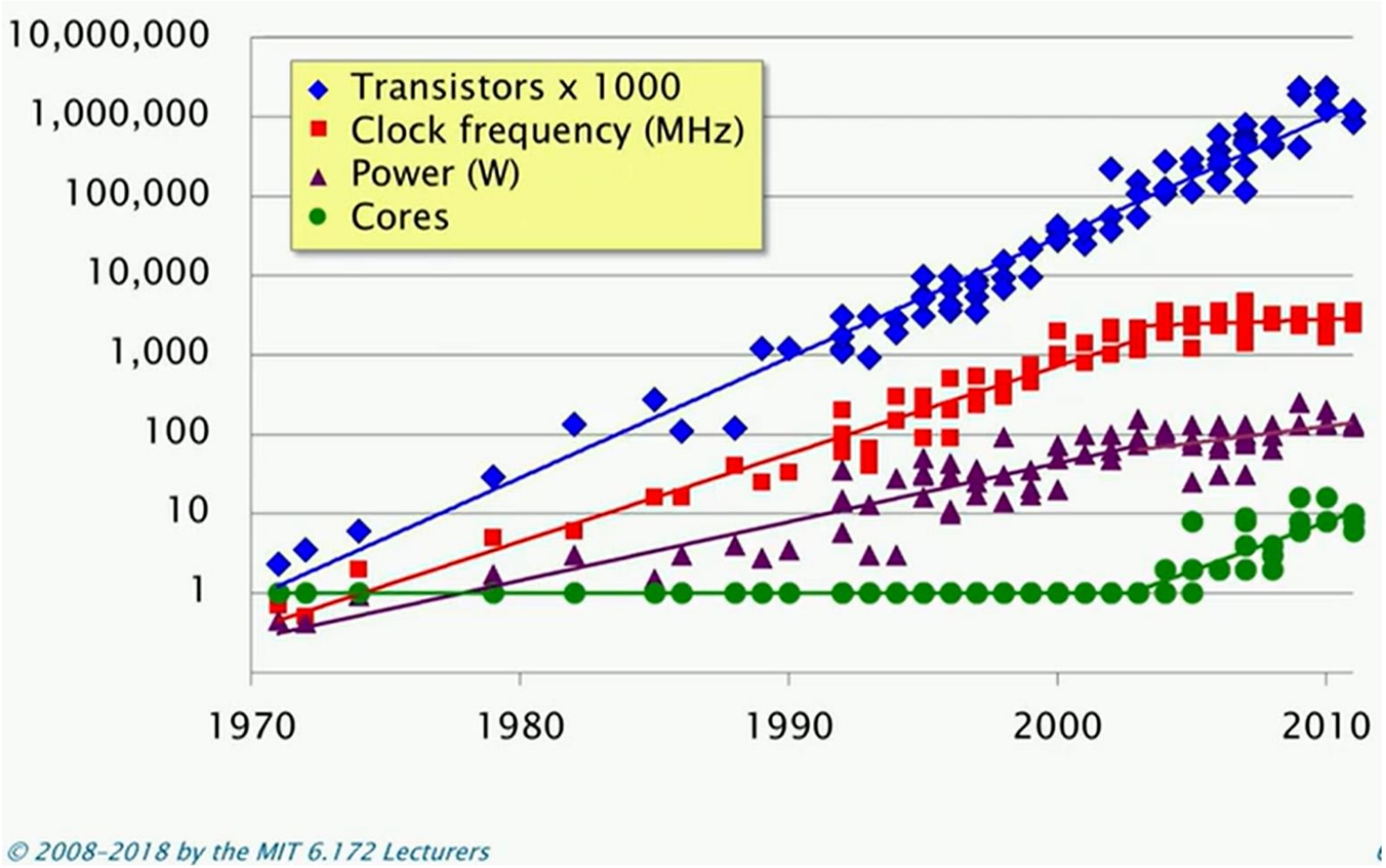
Projected power density, if clock frequency had continued its trend of scaling 25%–30% per year.



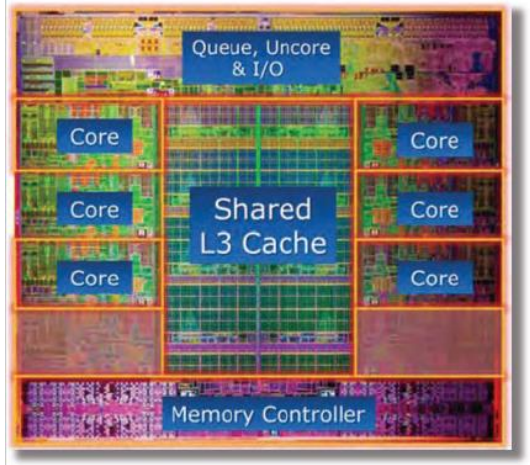
As transistors shrink, they become faster, consume less power, and are cheaper to manufacture.

[R. H. Dennard et al., **Design of ion-implanted MOSFET's with very small physical dimensions**. JSSC 9, 256–268 (1974).]
[Intel. **Why P scales as C*V^2*f is so obvious**. https://software.intel.com/content/www/us/en/develop/blogs/why-p-scales-as-cv2f-is-so-obvious.html]

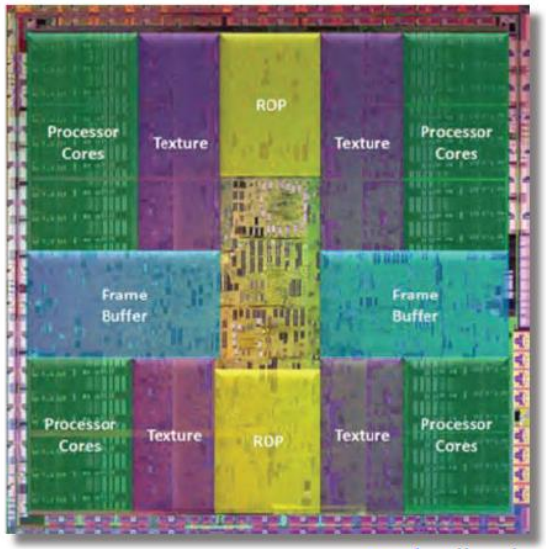
多核/众核时代来临


10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
Transistors x 1000
Clock frequency (MHz)
Power (W)
Cores
1970
1980
1990
2000
2010
© 2008–2018 by the MIT 6.172 Lecturers

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.

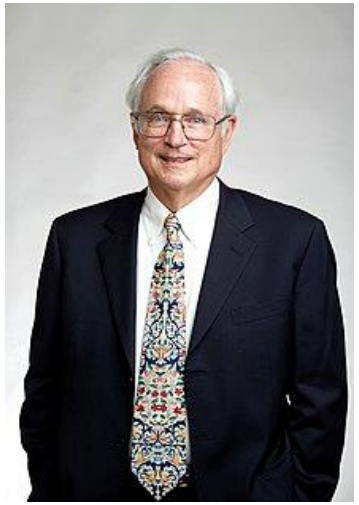
Generally, software must be adapted to utilize this hardware efficiently!

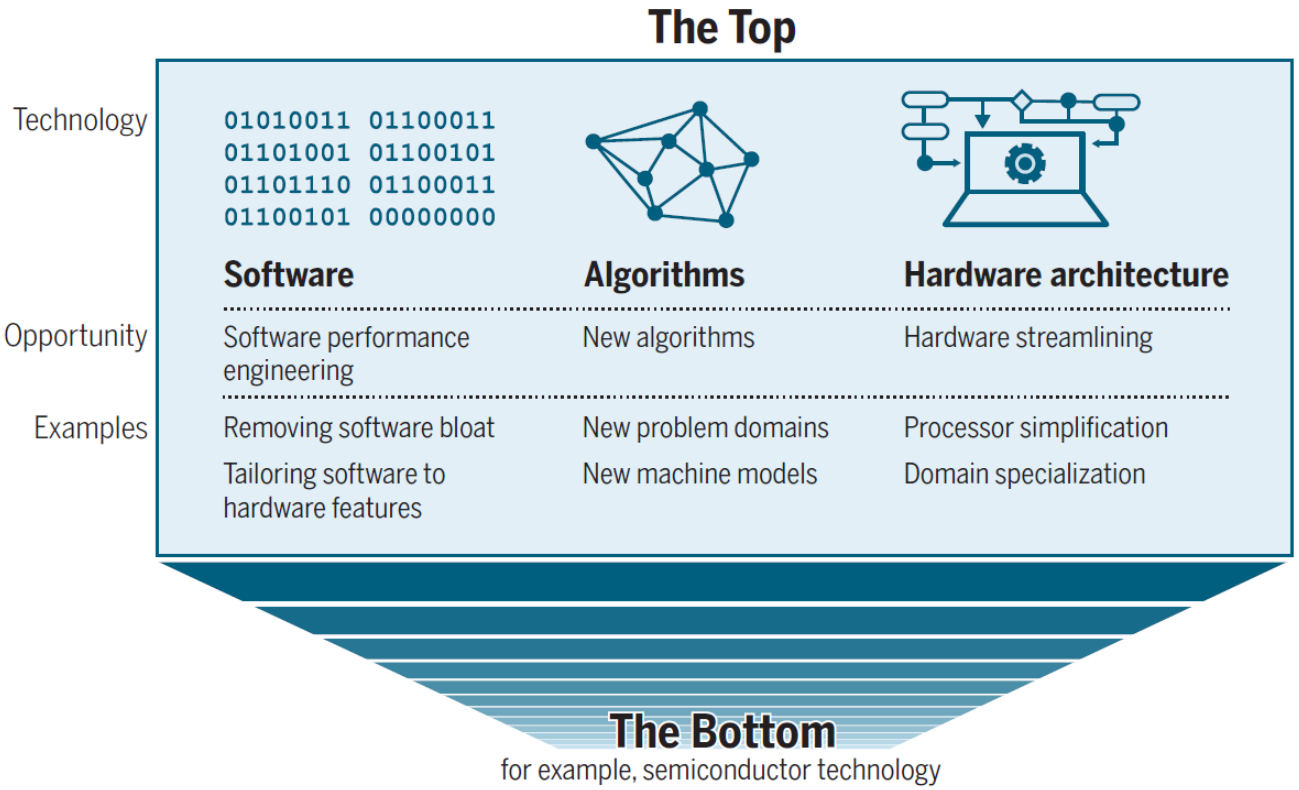
How?

There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson[1], Neil C. Thompson[1,2]*, Joel S. Emer[1,3], Bradley C. Kuszmaul[1]†, Butler W. Lampson[1,4], Daniel Sanchez[1], Tao B. Schardl[1]

The miniaturization of semiconductor transistors has driven the growth in computer performance for more than 50 years. As miniaturization approaches its limits, bringing an end to Moore's law, performance gains will need to come from software, algorithms, and hardware. We refer to these technologies as the "Top" of the computing stack to distinguish them from the traditional technologies at the "Bottom": semiconductor physics and silicon-fabrication technology. In the post-Moore era, the Top will provide substantial performance gains, but these gains will be opportunistic, uneven, and sporadic, and they will suffer from the law of diminishing returns. Big system components offer a promising context for tackling the challenges of working at the Top.





Performance gains after Moore's law ends. In the post-Moore era, improvements in computing power will increasingly come from technologies at the "Top" of the computing stack, not from those at the "Bottom", reversing the historical trend.

[C. E. Leiserson et al., **There's plenty of room at the Top: What will drive computer performance after Moore's law?** Science 368, eaam9744 (2020)]

Software Performance Engineering
软件性能工程

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

[MIT 6.172, **Performance Engineering of Software Systems**, https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/]
[C. E. Leiserson et al., **There's plenty of room at the Top: What will drive computer performance after Moore's law?** Science 368, eaam9744 (2020)]

New Algorithms 新型算法

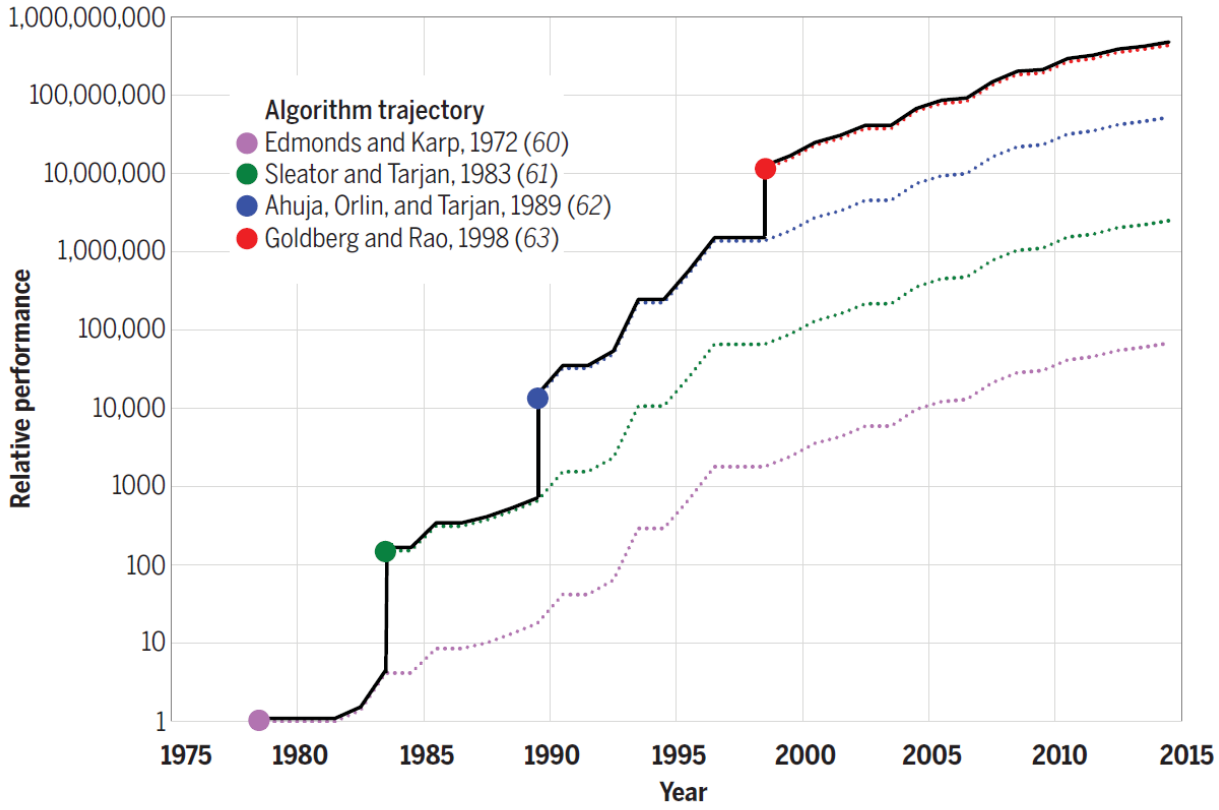


Fig. 1. Major algorithmic advances in solving the maximum-flow problem on a graph with $n = 10^{12}$ vertices and $m = n^{1.1}$ edges. The vertical axis shows how many problems (normalized to the year 1978) could theoretically be solved in a fixed time on the best microprocessor system available in that year. Each major algorithm is shown as a circle in the year of its invention, except the first, which was the best algorithm in 1978. The dotted trajectory shows how faster computers [as measured by SPECint scores in Stanford's CPU database (56)] make each algorithm faster over time. The solid black line shows the best algorithm using the best computer at any point in time. Each algorithm's performance is measured by its asymptotic complexity, as described in the Methods.

Hardware Streamlining
硬件精简

Implementing hardware functions using fewer transistors and less silicon area

- **Processor simplification :** replaces a complex processing core with a simpler core that requires fewer transistors.
- **Domain specialization** : may be even more important than simplification. Hardware that is customized for an application domain can be much more streamlined and use many fewer transistors, enabling applications to run tens to hundreds of times faster. ***Hennessy and Patterson*** foresee a move from general-purpose toward domain-specific architectures that run small compute-intensive kernels of larger systems for tasks such as object recognition or speech understanding.
  - GPU, TPU

AWARDS & RECOGNITION

John Hennessy and David Patterson Receive 2017 ACM A.M. Turing Award

ACM has named John L. Hennessy, former President of Stanford University, and David A. Patterson, retired Professor of the University of California, Berkeley, recipients of the 2017 ACM A.M. Turing Award for pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry.

[C. E. Leiserson et al., **There's plenty of room at the Top: What will drive computer performance after Moore's law?** Science 368, eaam9744 (2020)]
[J. L. Hennessy, D. A. Patterson, **A new golden age for computer architecture**. Commun. ACM 62, 48–60 (2019)]

系统优化的未来之路

There's plenty of room at the Top: What will drive computer performance after Moore's law?

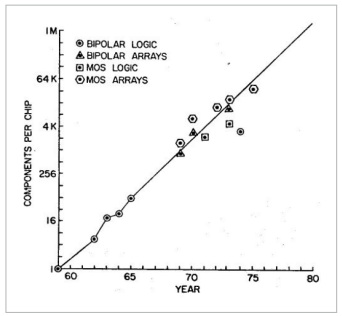


Figure 1 Approximate component count for complex integrated circuits vs. year of Introduction.

2070?

2020

2014

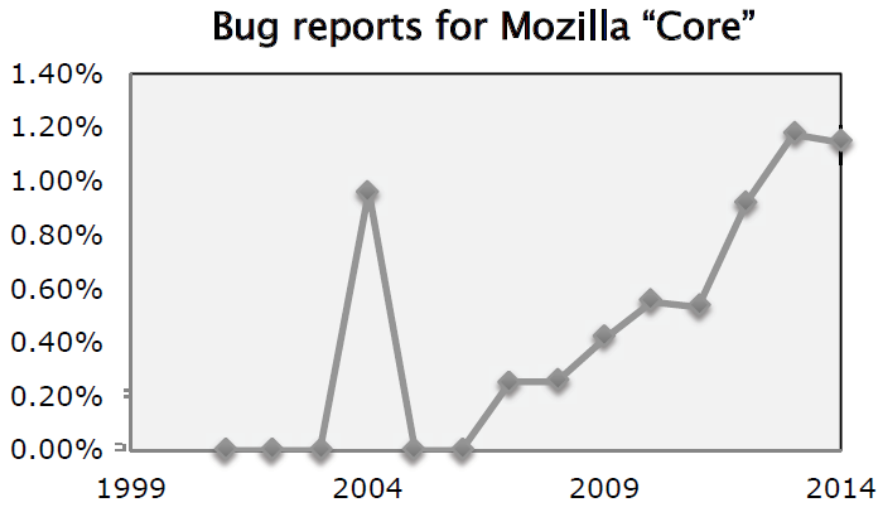
1960

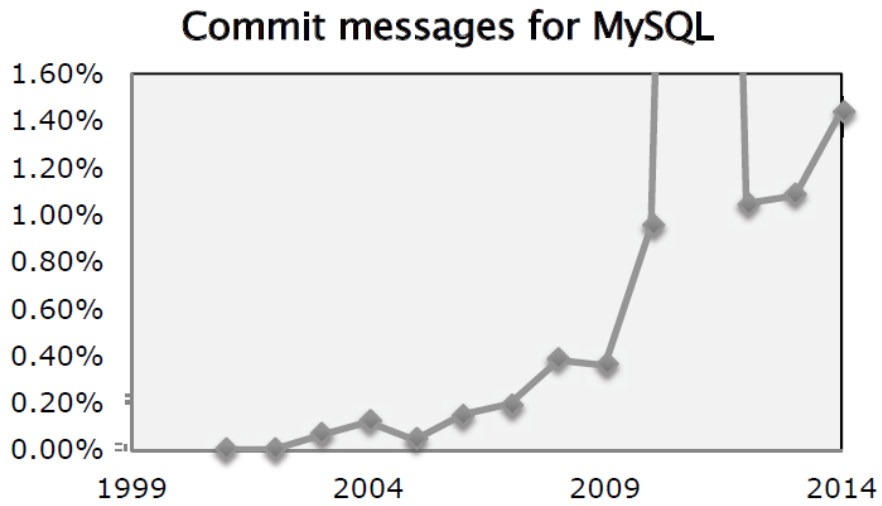
There's Plenty of Room at the Bottom

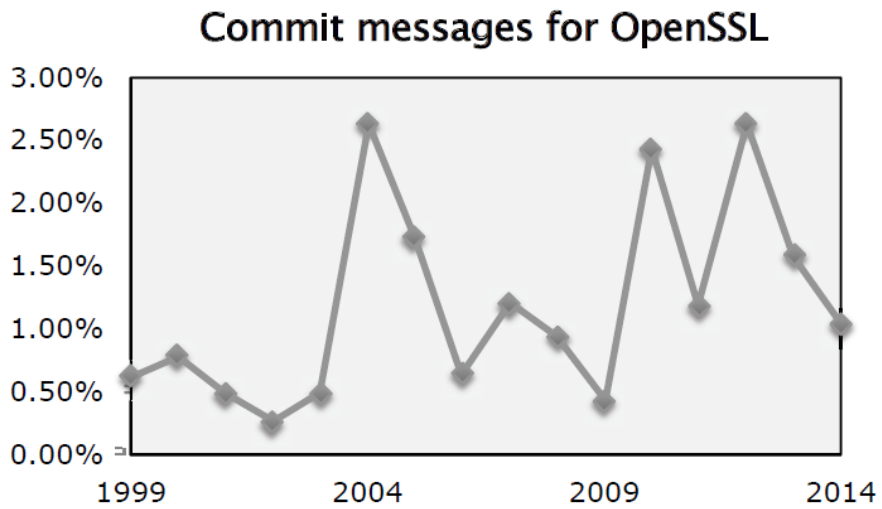
An Invitation to Enter a New Field of Physics

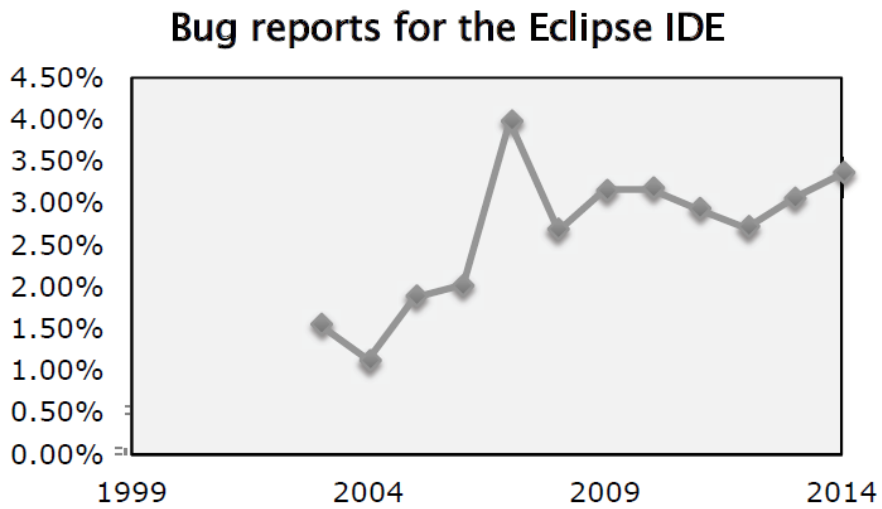
by Richard P. Feynman

Software Bugs Mentioning “Performance”

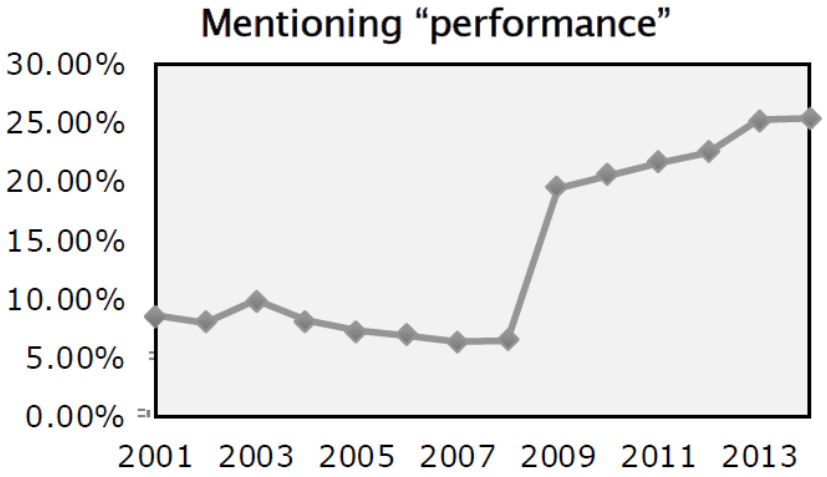


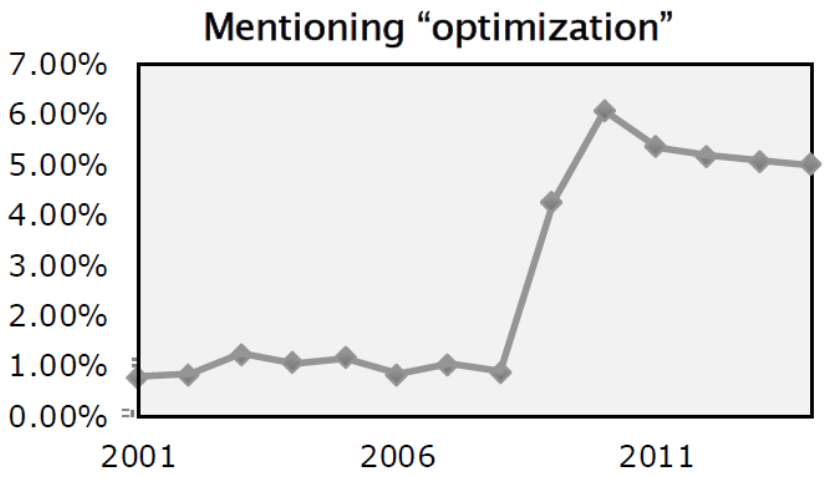


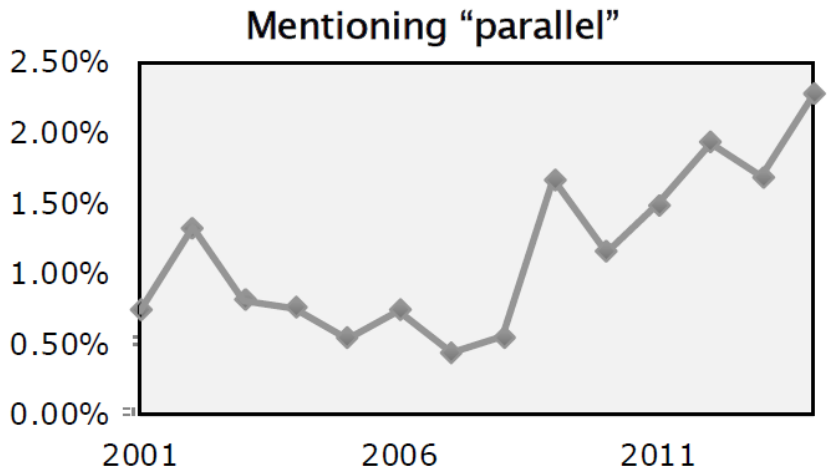


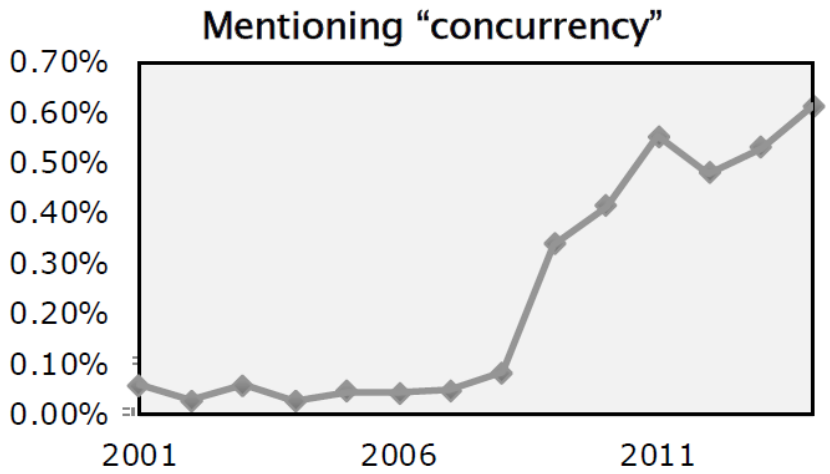


Software Developer Jobs









Source: Monster.com

引言

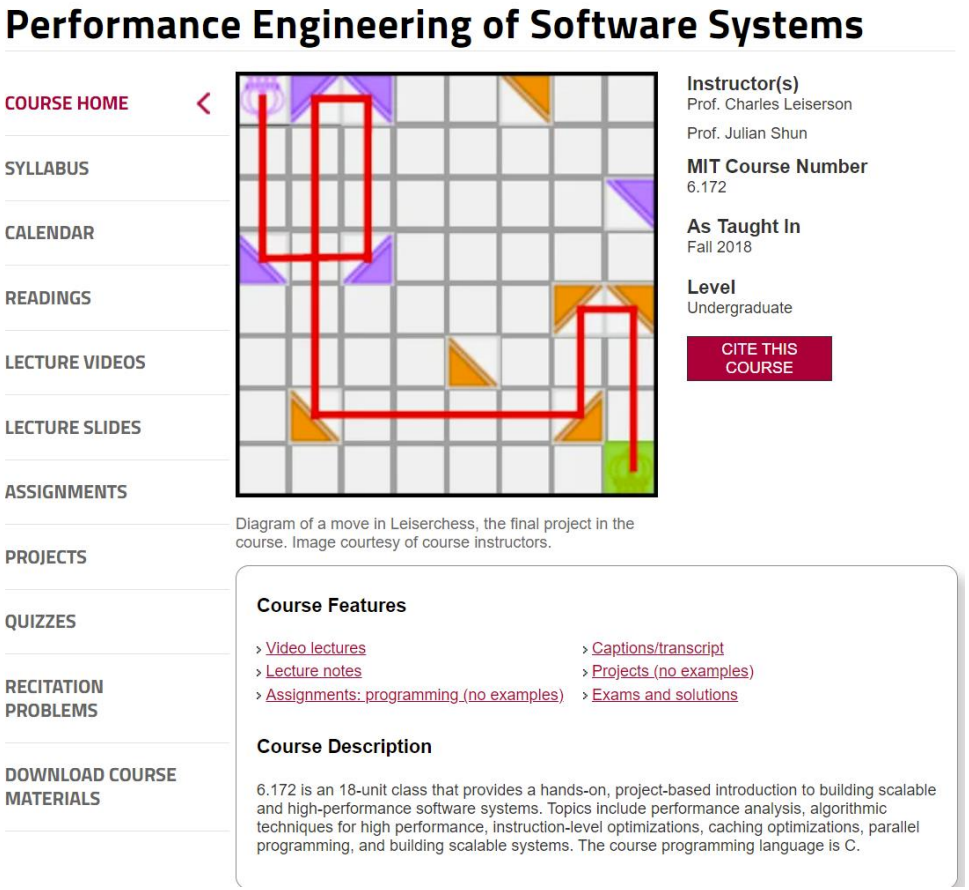
课程教学大纲

课程简介

- **如何在给定的硬件系统资源配置下提升软件系统的性能**是数字化系统的设计和实现必须思考和解决的问题，同时也是最大化利用硬件平台资源的有效手段。
- 软件系统优化的原理、技术和实践是一位**卓越的系统架构师或数据科学家**必备的素养，发起软件系统优化方面的课程设置及教学同时也是解决国家计算机系统方面“卡脖子”问题人才培养的有效措施，在培养学生解决实际问题的过程中可以充分培养学生的逻辑思维、批判性思维和创造性思维。

主要参考课程：MIT 6.172 Performance Engineering of Software Systems

- Course Website：https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/
- Video： https://www.bilibili.com/video/BV1wA411h7N7/
- Using the course materials was confirmed by Prof. Charles E. Leiserson via emails



先修课程

- 程序设计（C语言、汇编语言；Python、Java、Julia）
- 计算机系统
- 算法设计与分析

理论课目标和安排

- 掌握软件系统性能优化的基本概念和原理，培养系统观。
- 掌握程序性能优化的基本原则与方法，学会如何编写高性能代码。
- 熟练运用系统性能优化的工具和方法，了解相关开源技术社区。
- 掌握编译器、计算机体系结构、并行编程、性能工程的基本概念和原理，了解如何通过相关分析和优化方法来定位和解决软件系统的性能瓶颈。
- 了解系统性能优化的演进过程和研究动向，为后续相关课程学习和研究打下良好基础，为后续职业发展奠定基石。

序号	模块	主题	内容	课时
L1	导论	Introduction & Matrix Multiplication	为什么要进行软件性能优化？用矩阵乘法示例各类优化方法和效果	2
L2		Bentley Rules for Optimizing Work	枚举介绍常见的各类优化手段（New Bentley Rules）	2
L3	编译器和编译优化	编译器概论	编译器简介、静态/动态编译	2
L4		Assembly Language	x86-64 汇编简介、浮点和向量化指令	2
L5		C to Assembly	C 函数与 LLVM IR 及 x86-64 汇编的对应，控制流图及函数调用约定	2
L6		What Compilers Can and Cannot Do	常见编译优化方法（向量、结构、函数调用、循环）	2
L7	计算机体系结构和并行编程	计算机体系结构概论	CPU 微体系结构、流水线、超标量、乱序、分支预测等	2
L8		Multicore Programming	共享内存硬件架构及多核编程	2
L9		Caching Optimization	高速缓存及其优化	2
L10		Races and Parallelism	数据竞争及并行性、Amdahl 定律	2
L11	性能工程	Measurement and Timing	软件性能测量方法、测量工具及性能建模	2
L12		Benchmarking and Autotuning	基准测试和实验设计	2
L13		Metrics and Errors	性能指标和统计错误	2
L14		Analysis Methods	性能分析方法	2
L15	前沿研究和应用	Binary Translation	静态/动态二进制翻译	2
L16		Data Centers and Cloud Optimization	数据中心和云上优化	2
L17		Project Studies	实际项目分析	2
L18	总结	Course Review	课程总结和复习	2

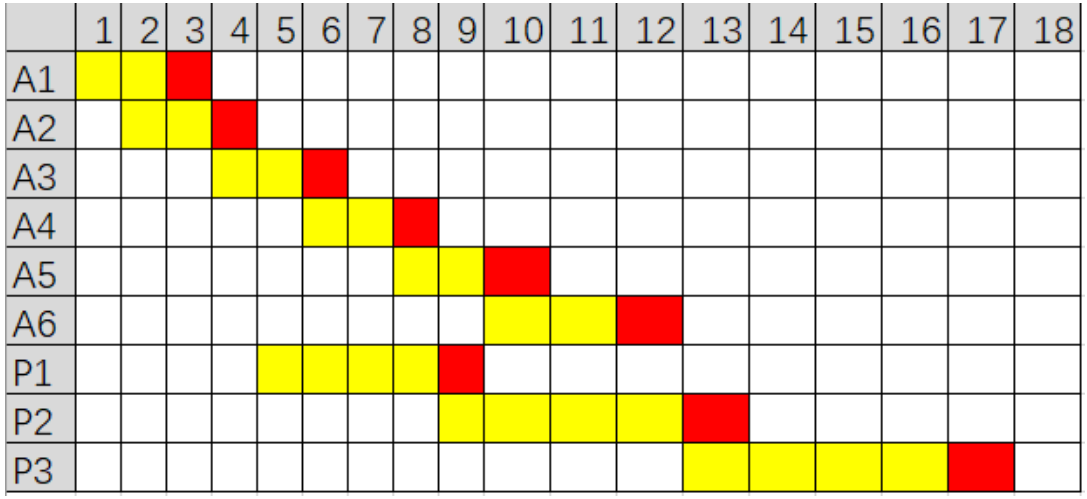
实践课目标和安排

- 掌握编译基本过程和工具框架。
- 掌握性能数据采集、处理和分析的基本工具和技能。
- 掌握基准测试和系统性能优化的基本方法。
- 掌握通过实测和数据分析不断定位性能问题并调优的基本技巧。

序号	主题模块	内容	课时	布置周	提交周
A1	导论	初试环境和工具	2	1	3
A2		OpenTuner Tutorial	2	2	4
A3	编译器和编译优化	GCC 与 Clang/LLVM 优化比较	2	4	6
A4		Vectorization	2	6	8
P1		交叉编译与跨平台应用仿真	8	5	9
A5	计算机体系结构和并行编程	OpenCilk Tutorial	2	8	10
A6	性能工程	SPECjvm2008 基准测试	2	10	12
P2		Matrix Multiplication Autotuner	8	9	13
P3		Profiling Serial Merge Sort	8	13	17

作业要求：

- 实践项目和上机作业都需要独立完成，并提交电子版报告。作业要求会在“布置周”实践课上发放，作业必须在“提交周”实践课结束前完成提交，过期将影响成绩。
- 实践项目报告需要包括完整的项目介绍、优化过程描述、算法描述、代码实现和具体的执行过程和结果。上机作业报告主要包括上机执行过程和结果，如有代码实现或其他要求，也要附上。



参考书目或其他学习材料

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Pearson Education, 2007.
- John L. Hennessy, David A. Patterson. Computer Architecture: A Quantitative Approach (Sixth Edition). Morgan Kaufmann, 2019.
- David J. Lilja. Measuring Computer Performance: A Practitioner's Guide. Cambridge University Press, 2004.
- Denis Bakhvalov. Performance Analysis and Tuning on Modern CPUs. 2020.
- Brendan Gregg. System Performance: Enterprise and the Cloud (Second Edition). Pearson Education, 2021.

课堂文明 Class Civility

- 原则：**不要影响其他人**（老师、同学）
- 不迟到、不早退
- 不交头接耳、不大声喧哗
- 手机关机或静音

考核办法和评分规则

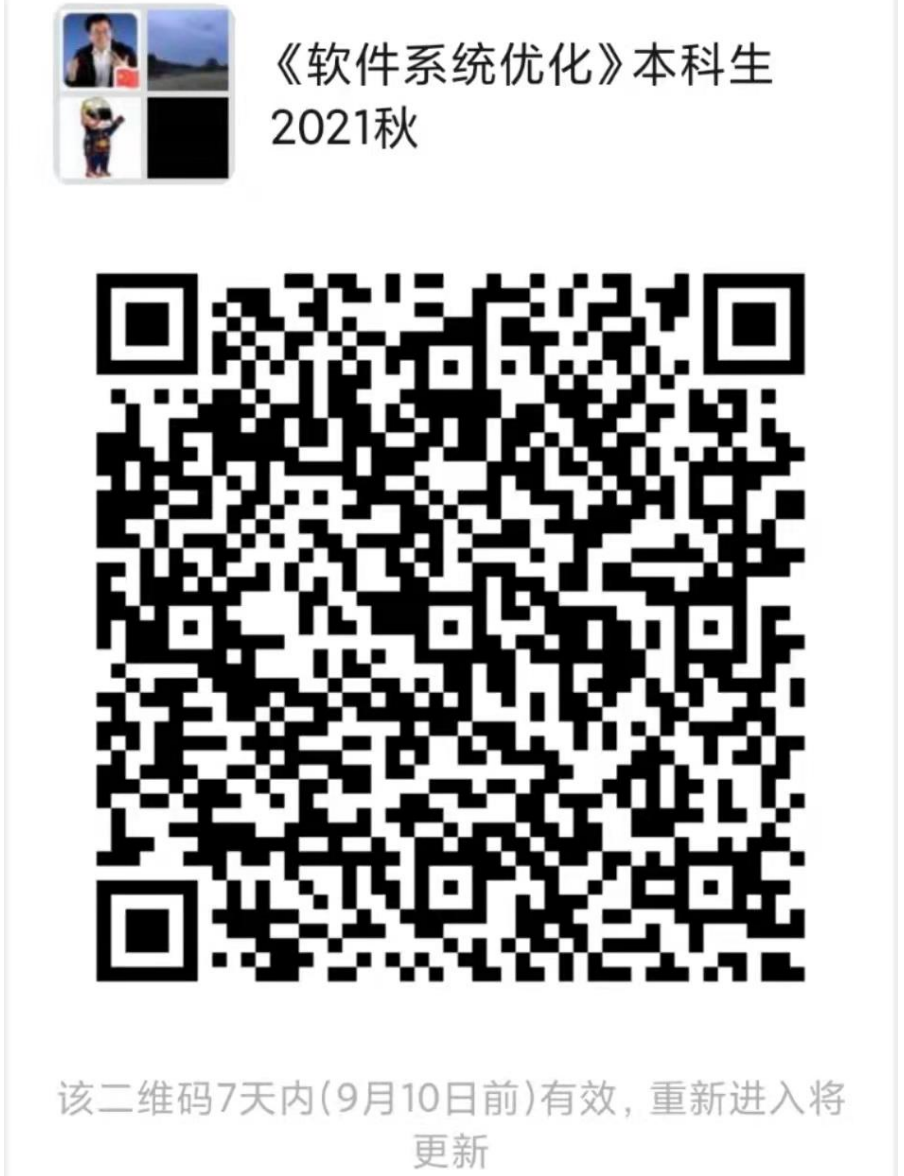
- 考核：采用考查方式（项目报告+上机作业+平时成绩）和百分制
- 评分：
  - 实践项目报告3份，占60%，每份占20%；
  - 上机作业报告6份，占30%，每份占5%；
  - 平时成绩（考勤+课堂表现）占10%。

- 禁忌：
  - 发现项目报告、上机作业互相抄袭1次即作零分处理
  - 3次缺勤即作零分处理

课程联络

- 助教
  - 张义磊 52215903013@stu.ecnu.edu.cn
  - 梁文辉 51215903095@stu.ecnu.edu.cn

- 微信群



矩阵乘法案例

Square-Matrix Multiplication

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

$C \qquad A \qquad B$

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

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
≈ 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 =

star
for

end

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

Running time
= 21042 seconds
≈ 6 hours

Is this fast?

Back-of-the-envelope calculation

$2n^3 = 2(2^{12})^3 = 2^{37}$ floating-point operations
Running time = 21042 seconds
∴ Python gets $2^{37}/21042 \approx 6.25$ MFLOPS
Peak ≈ 836 GFLOPS
Python gets ≈ 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?

Where We Stand So Far

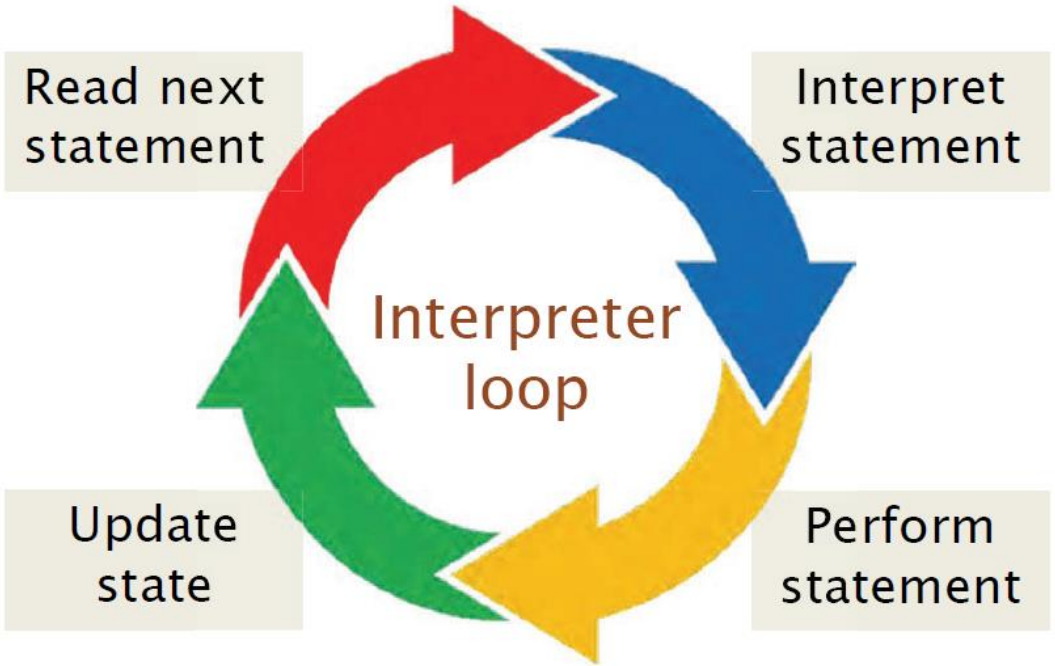
Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.007	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.119	0.014

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.

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

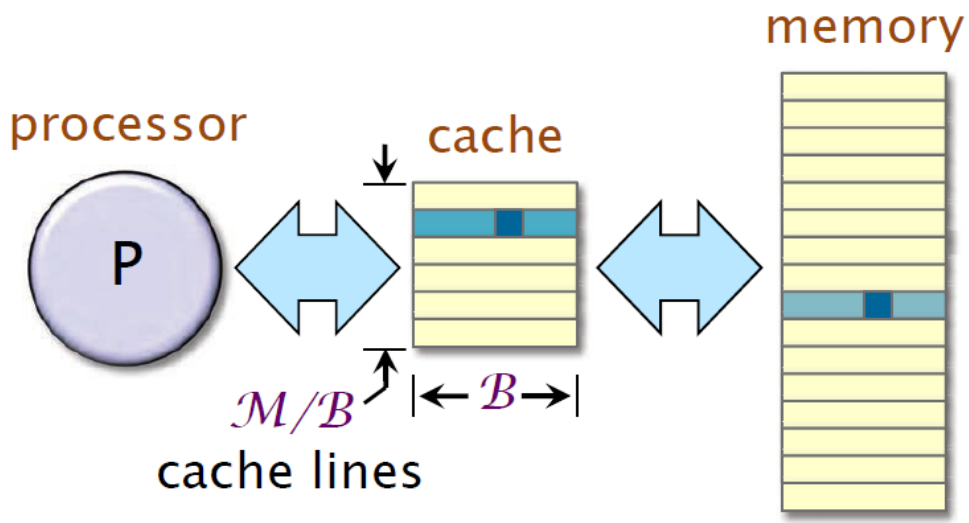
Loop order affects running time by a factor of 18!

What's going on!?

Hardware Caches

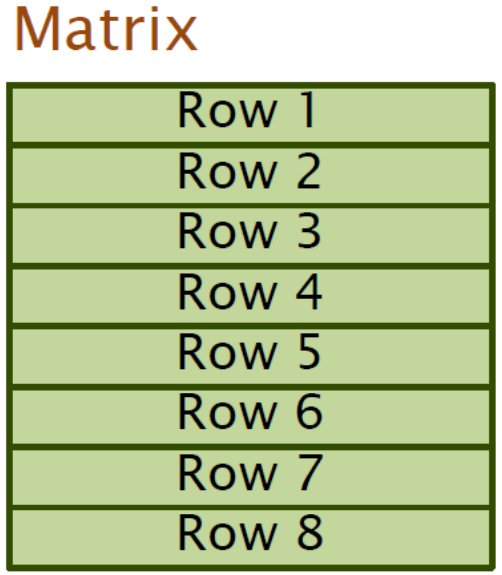
Each processor reads and writes main memory in contiguous blocks, called *cache lines*.

- 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.

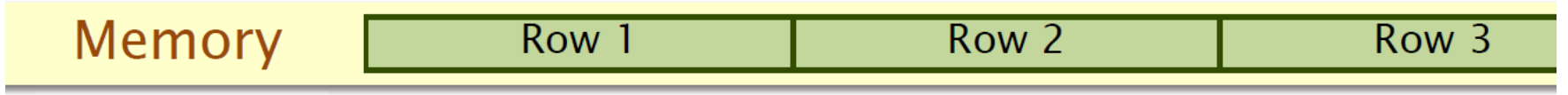


Memory Layout of Matrices

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



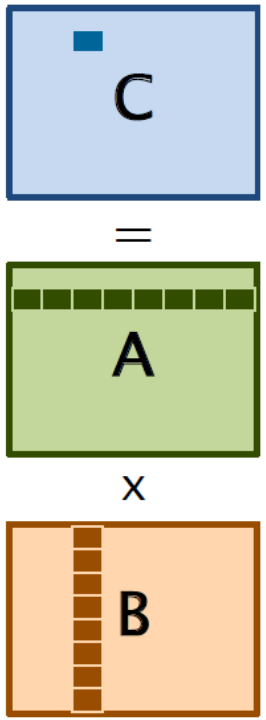
What does this layout imply about the performance of different loop orders?



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

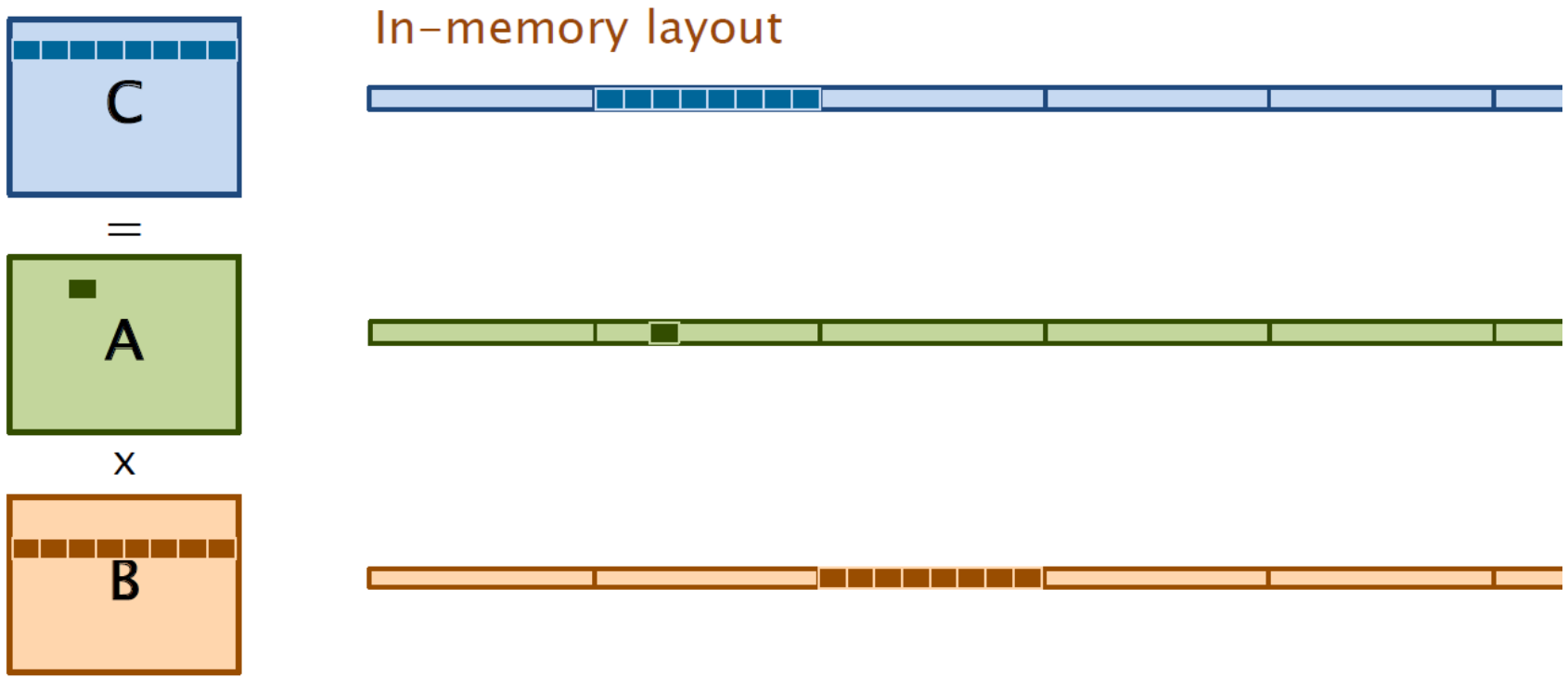
4096 elements apart

33

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];
```

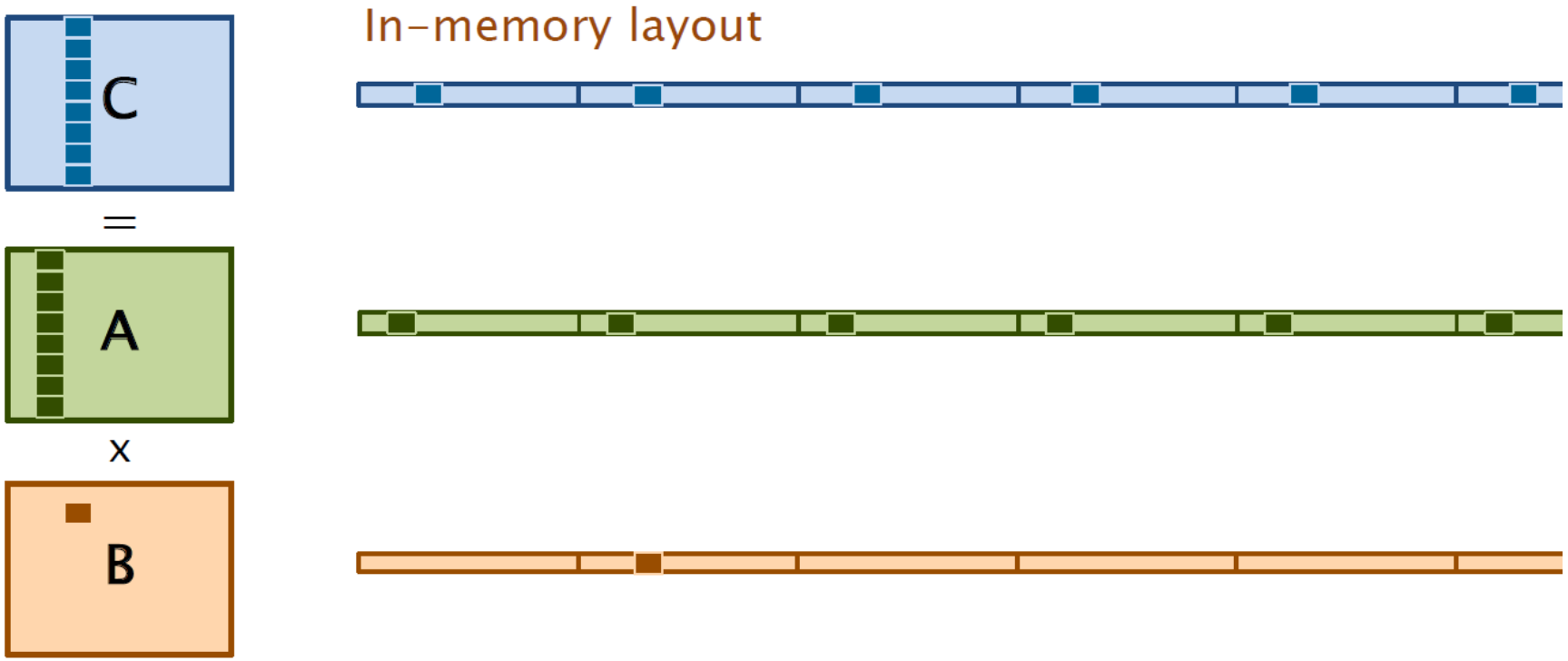
Running time:
177.68s



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



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)
-O0	Do not optimize	177.54
-O1	Optimize	66.24
-O2	Optimize even more	54.63
-O3	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.

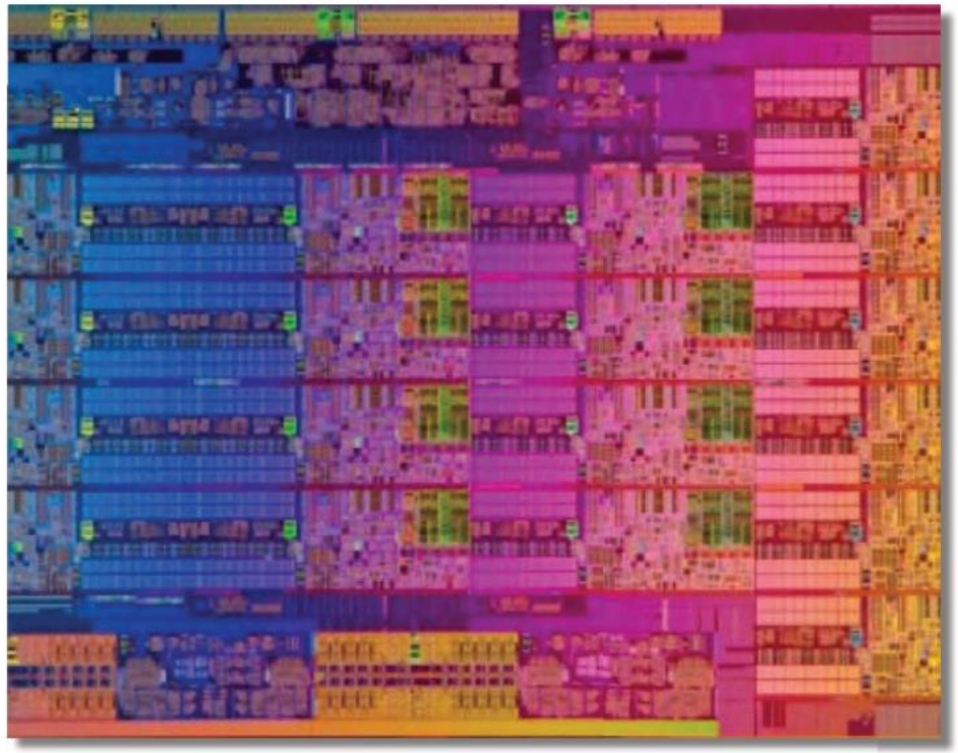
Version 5: Optimization Flags

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!

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
    cilk_for (int
      C[i][j] += A
```

Running time: 531.71s

Rule of Thumb
Parallelize outer loops rather than inner loops.

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

scheduling overhead

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

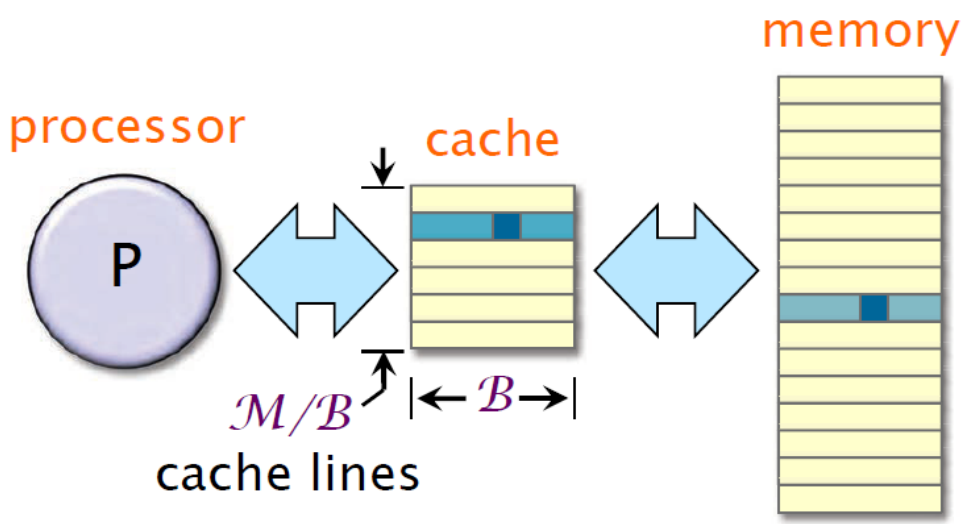
Using parallel loops gets us almost 18× speedup on 18 cores! (Disclaimer: Not all code is so easy to parallelize effectively.)

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.

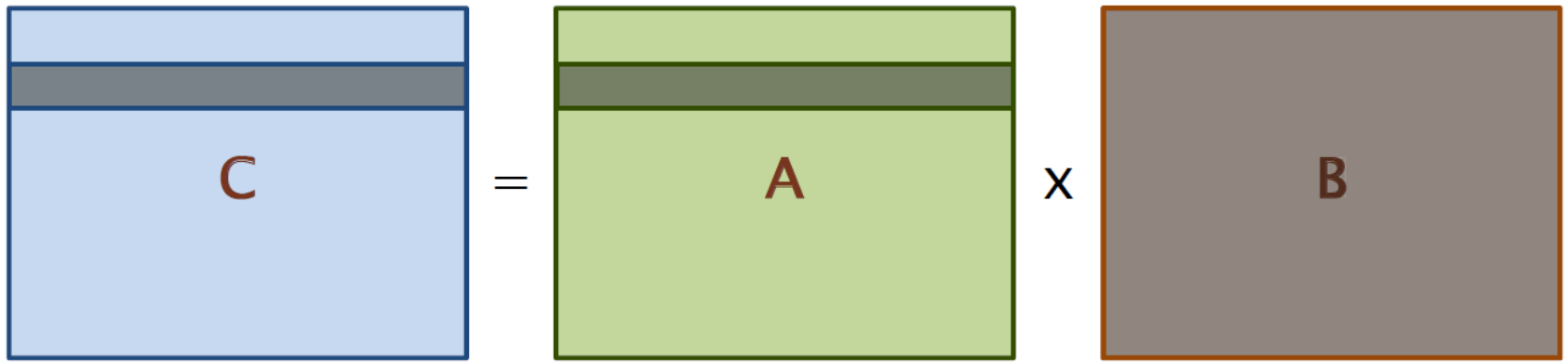
- Cache misses are slow, and cache hits are fast.
- Try to make the most of the cache by reusing the data that's already there.



Data Reuse: Loops

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

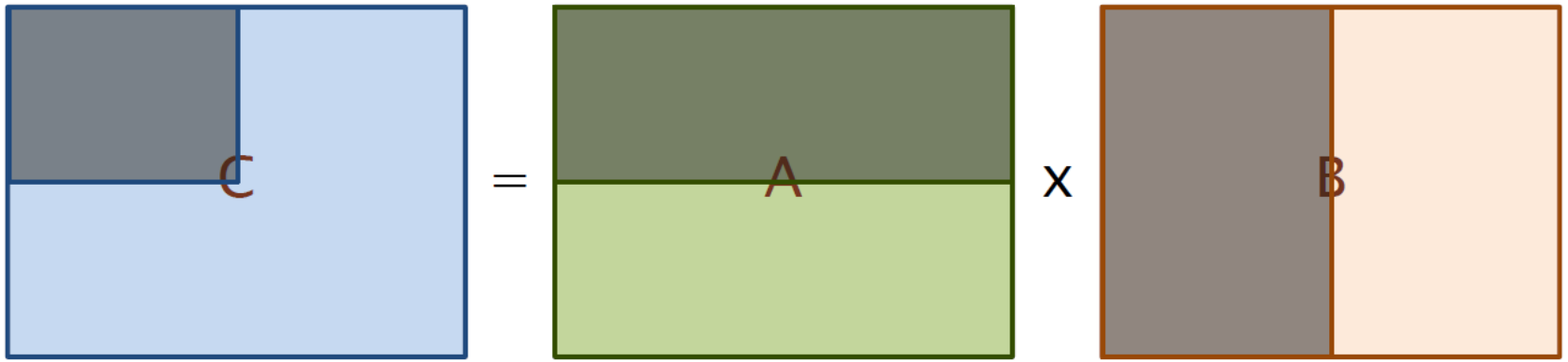
- 4096 * 1 = 4096 writes to C,
- 4096 * 1 = 4096 reads from A, and
- 4096 * 4096 = 16,777,216 reads from B, which is
- 16,785,408 memory accesses total.



Data Reuse: Blocks

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

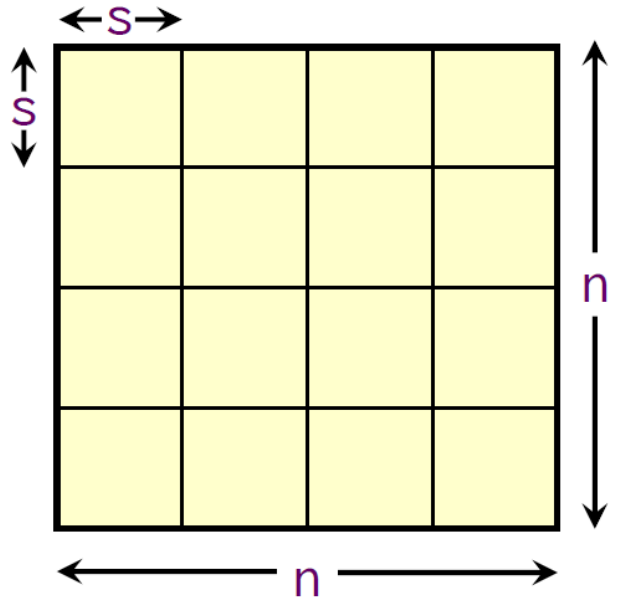
- $64 \cdot 64 = 4096$ writes to C,
- $64 \cdot 4096 = 262{,}144$ reads from A, and
- $4096 \cdot 64 = 262{,}144$ reads from B, or
- 528,384 memory accesses total.



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];
```

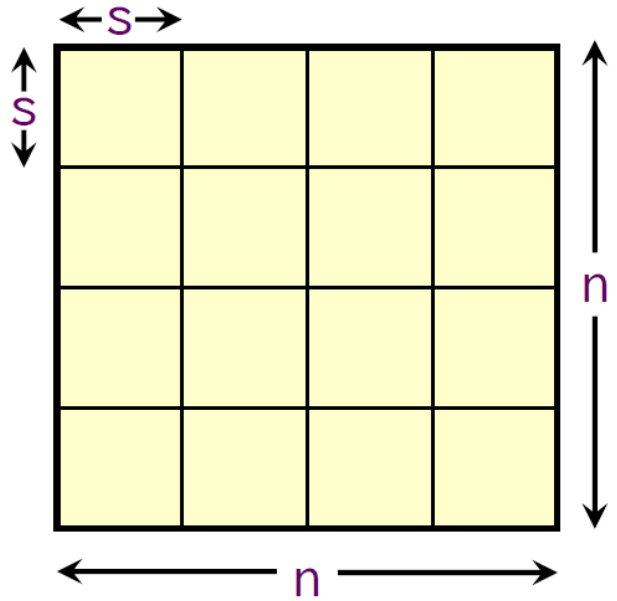
Tuning parameter
How do we find the right value of s?



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

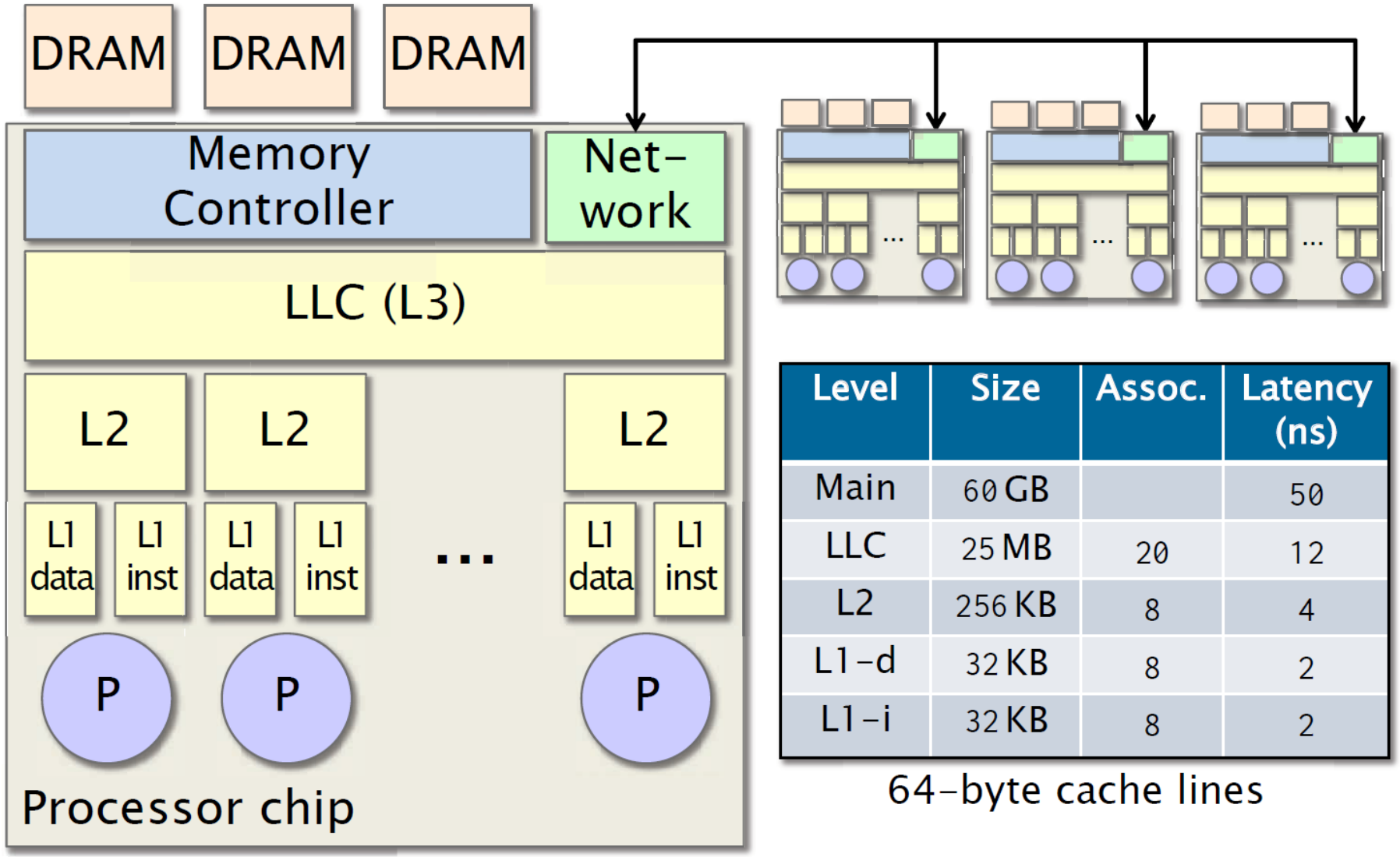
Version 7: Tiling

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

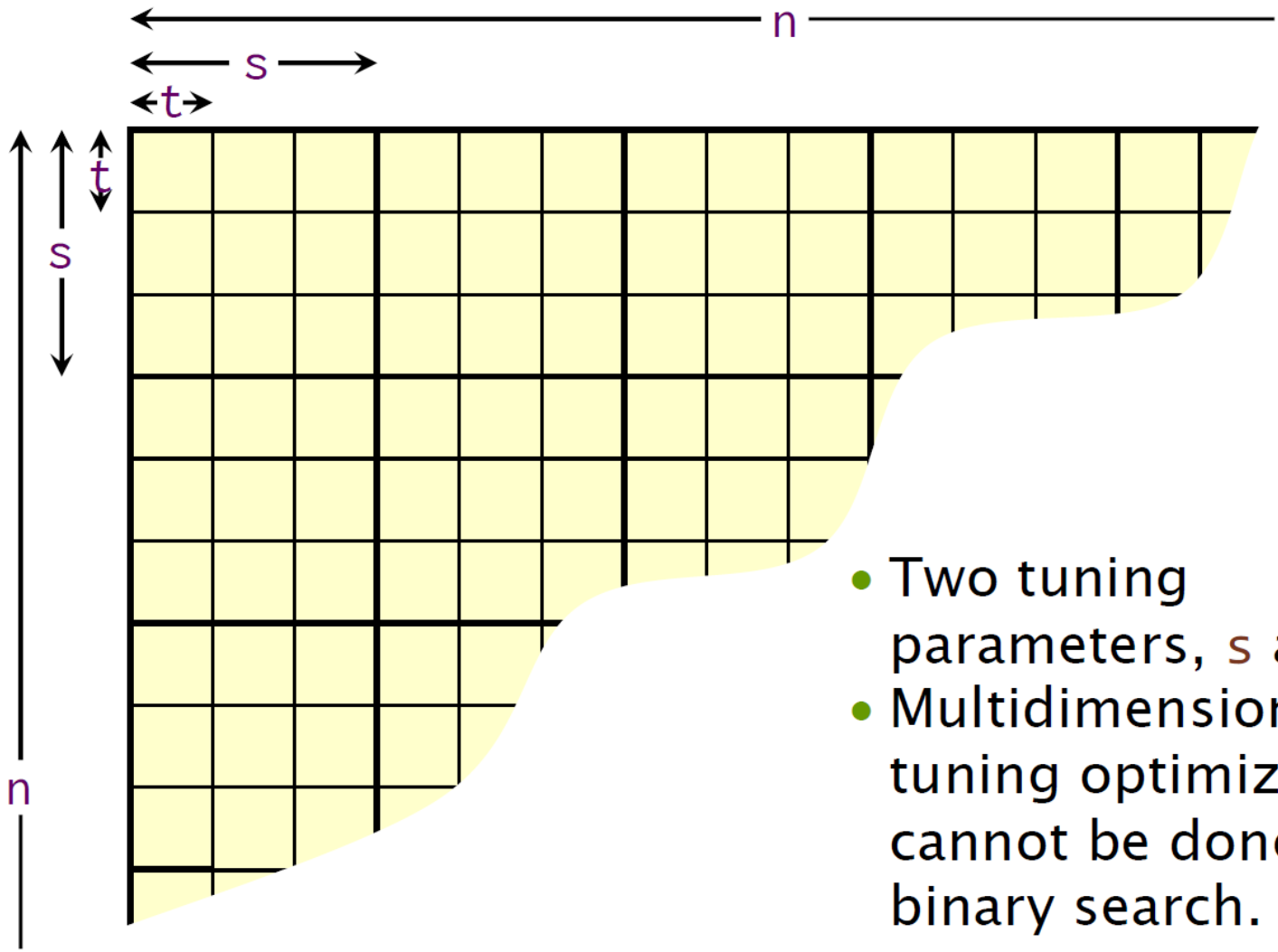


Level	Size	Assoc.	Latency (ns)
Main	60 GB		50
LLC	25 MB	20	12
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

64-byte cache lines

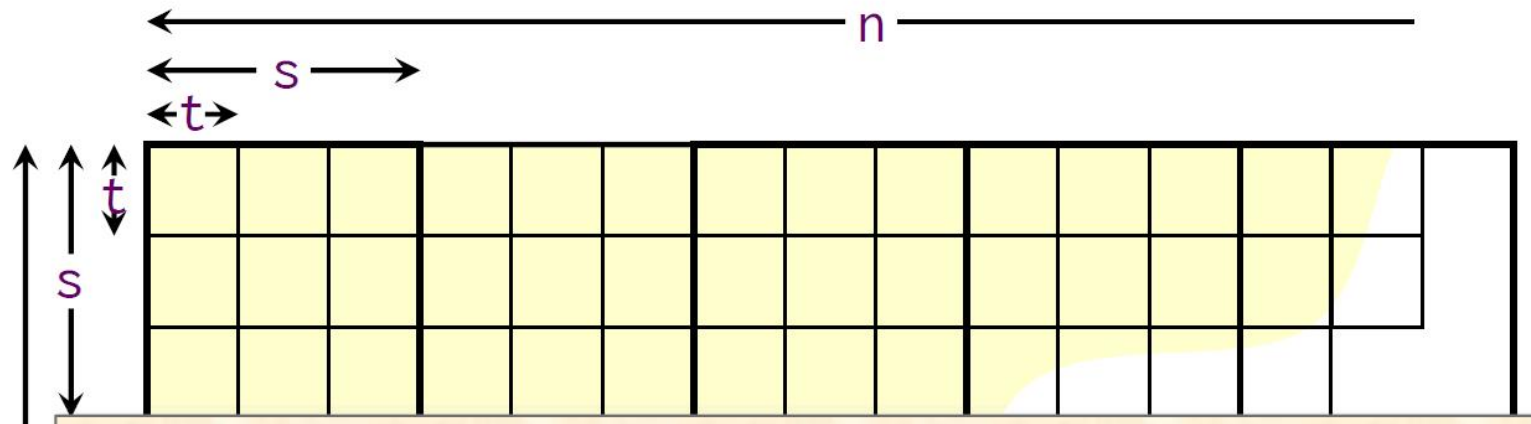
49

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];
```

51

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(
               mm_dac(X(C,1,1), n_C, X
    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_
            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
    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
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,
               mm_dac(X(C,1,1), n_C, X(A,1,
    cilk_sync;
  }
}
```

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

Running time: 93.93s

… about 50× 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_
               mm_
    cilk_sync;
    cilk_spawn mm_
    cilk_spawn mm_
    cilk_spawn mm_
               mm_
    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

```
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,
    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,
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A,
    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;
  }
}
```

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

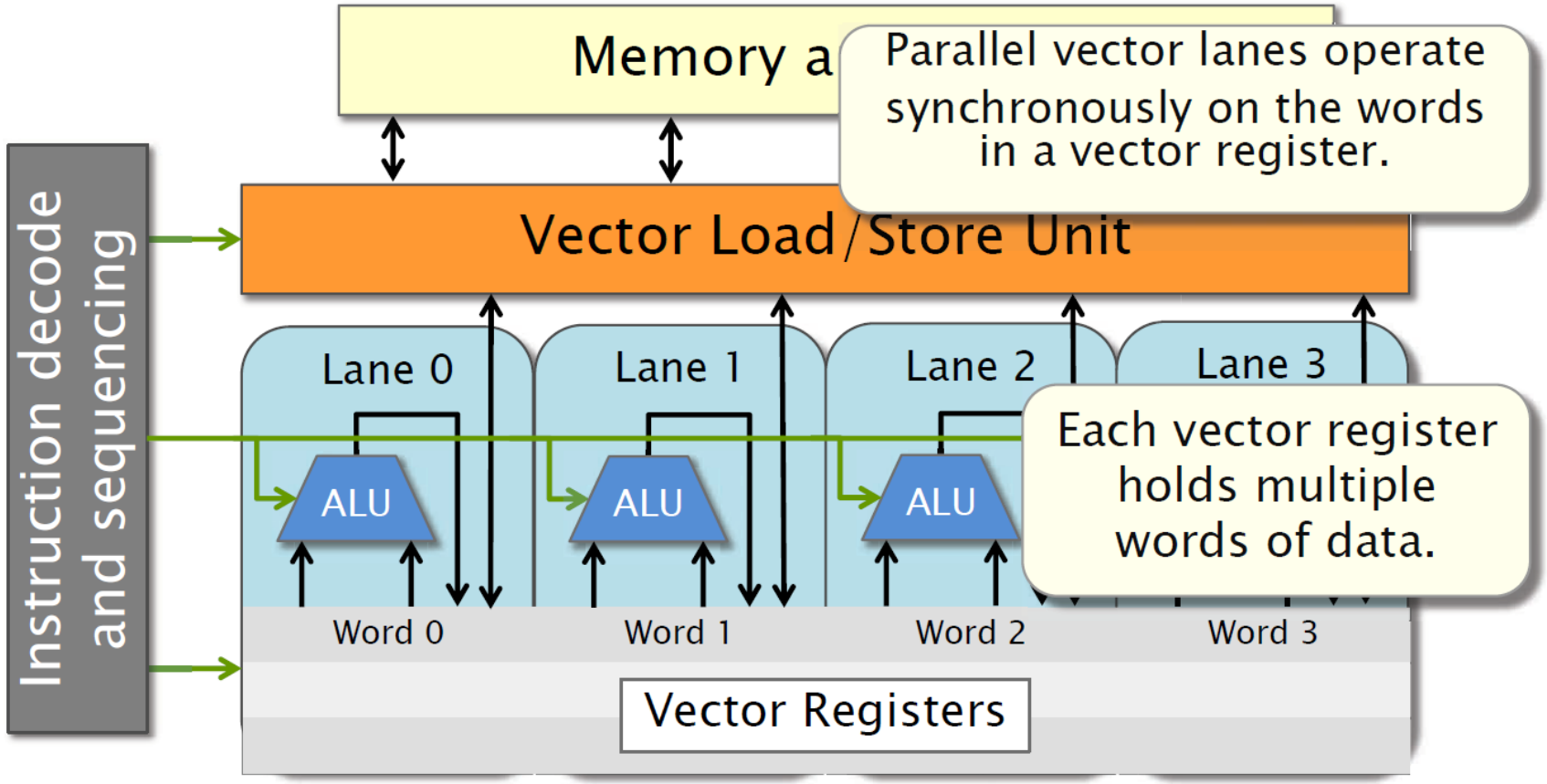
8. Divide-and-Conquer

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.



59

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

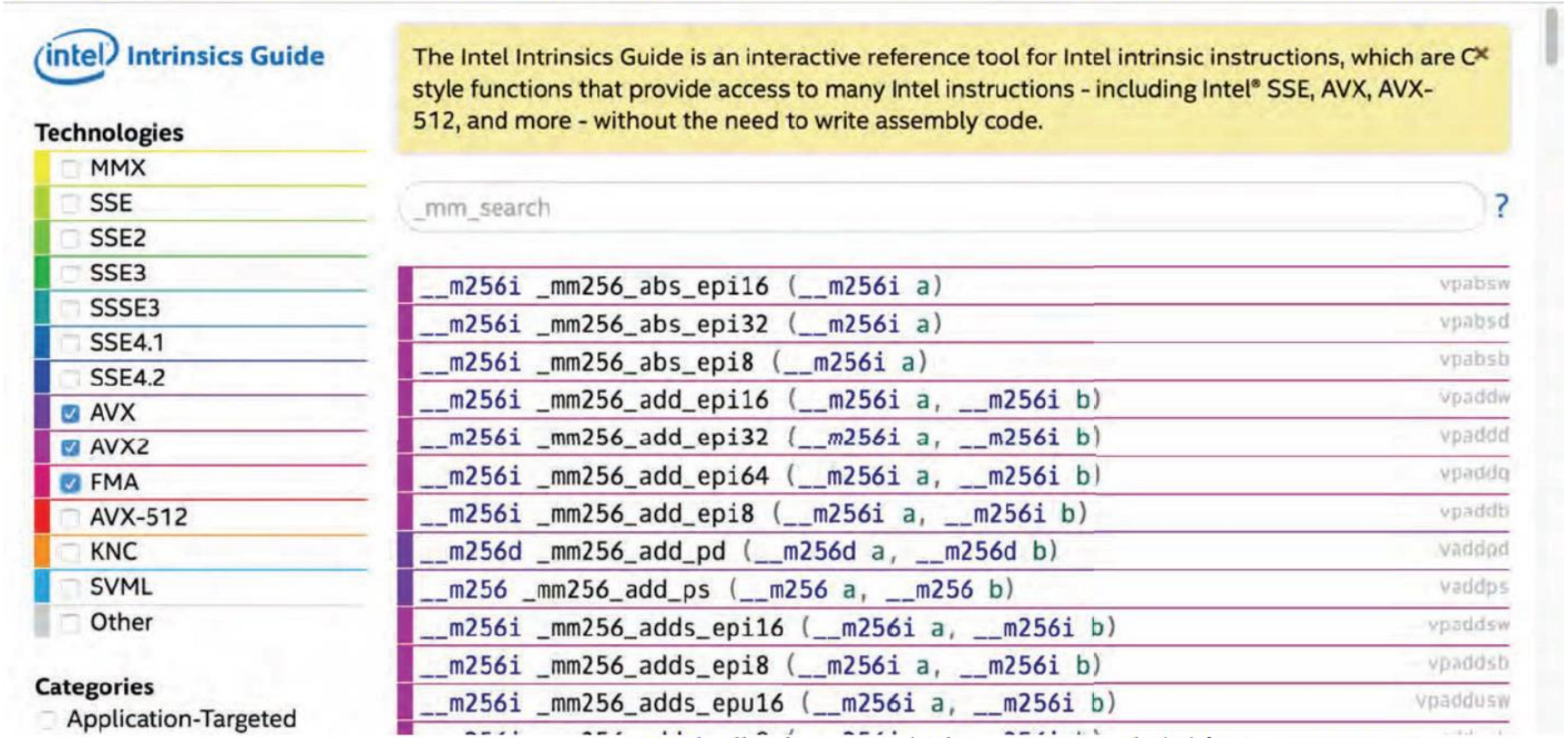
Using the flags `-march=native -ffast-math` nearly doubles the program's performance!

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/



 63

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

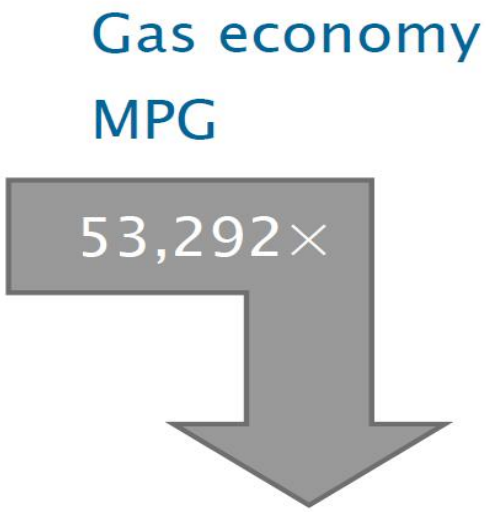
Version 11: Final Reckoning

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.

Courtesy of pngimg.
Used under CC-BY-NC.

Note: GPU, file systems, network performance are out of scope of this course, but they are hugely important in practice.

《软件系统优化》第1课
引言 & 矩阵乘法案例

郭健美

2021年秋

SOLE 系统优化实验室
华东师范大学