《软件系统优化》
绪论

郭健美

2024年秋

SOLE 系统优化实验室
华东师范大学

内容

- **课程介绍**
- 矩阵乘法优化案例
- 方法论概述

教学团队

- 教师
  - 郭健美 教授 jmguo@dase.ecnu.edu.cn
  - 黄波 特聘教授 bhuang@dase.ecnu.edu.cn
  - 林晓东 兼职教授（Intel）
  - 赵鹏 兼职副教授（Intel）

- 助教
  - 徐静 51265903015@stu.ecnu.edu.cn
  - 施瑾睿 51265903112@stu.ecnu.edu.cn
  - 杨桐 51265903072@stu.ecnu.edu.cn

为何要上这门课?

- **性能**是衡量软件系统质量和竞争力的一个重要方面，是软件系统设计、开发和应用过程中必须关注的一个基本属性。如何在给定的硬件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，同时也是优化利用软硬件资源的有效途径。
- **软件系统优化**的原理、技术和实践是一位卓越的软件系统工程师、架构师或研究人员必备的素养。发起软件系统优化方面的课程设置和教学是解决我国计算机系统方面“卡脖子”问题人才培养的有效措施。我们力求在训练相关人员解决实际问题的过程中围绕“优化思维”培养“系统观”和工程能力，锻炼逻辑思维、批判性思维和创造性思维。

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

[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 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.

课程目标（对照教学大纲）

- 目标1：了解应用负载、操作系统、编译器、计算机体系结构的基本概念，掌握软件系统性能优化的基本方法和原理，培养系统观。（支撑毕业要求7、13、14）
- 目标2：掌握程序性能优化的基本原则与方法，掌握编译基本过程和工具框架，学会如何编写高性能代码。（支撑毕业要求7、9、12）
- 目标3：熟练运用系统性能优化的工具和套件，掌握性能数据采集、处理和分析的基本方法和技能，了解相关开源技术社区。（支撑毕业要求9、12）
- 目标4：掌握软件系统性能工程和基准测试的基本概念和方法，了解如何通过实测数据分析和优化方法来定位和解决软件系统的性能瓶颈。（支撑毕业要求9、12、13、14）
- 目标5：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。（支撑毕业要求7、13、14）

先修课程

- 计算机程序设计
- 数据结构
- 算法设计与分析
- 计算机系统
  - 编译原理
  - 计算机组成与系统结构

教学内容和安排

	理论课（周一3-4节）				实践课（周三3-4节）			
周	模块	主题	教师	日期	上机作业（2周/个）	实践项目（4周/个）	助教	日期
1	绪论	矩阵乘法优化案例、方法论概述	郭健美	9.9			徐静 施瑾睿 杨桐	9.11
2	性能工程基础	性能测量		9.16->9.14	A1 初试环境和工具			9.18
3		基准评测		9.23	A2 SPECjvm2008基准测试	P1 矩阵乘法自动调优器		9.25
4		配置优化		9.30	A1 提交			~~10.2~~
5		性能评价		10.7->10.12	1. A2 提交 2. A1 反馈和检查			10.9
6	计算机体系结构优化	处理器优化		10.14	A2 反馈和检查			10.16
7		存储器优化		10.21		P1 提交		~~10.23~~
8		微体系结构性能分析		10.28		1. P2 剖析合并排序 2. P1 反馈和检查		10.30
9		异构计算与编程	赵鹏	11.4	A3 oneAPI异构编程			11.6
10	编译优化	源程序级别的常见优化方法	黄波	11.11				11.13
11		编译器概述		11.18	1. A4 GCC与Clang/LLVM优化比较 2. A3 提交	P3 交叉编译与跨平台应用仿真		11.20
12		目标指令集架构及汇编语言		11.25	A3 反馈和检查	P2 提交		11.27
13		C程序的汇编代码生成		12.2	A4 提交	P2 反馈和检查		12.4
14		编译器的优化能力		12.9	1. A5 向量化 2. A4 反馈和检查			12.11
15		程序插桩及优化机会识别		12.16		P3提交		12.18
16	专题讨论	数据中心的性能优化	郭健美	12.23	A5 提交	P3 反馈和检查		12.25
17		深度学习框架的优化	林晓东	12.30	A5 反馈和检查			~~1.1~~

- 本科生
  - 3学分，72学时（理论 36学时，上机 36学时）
- 研究生
  - 2学分，36学时（理论）
  - 建议参加实践课（布置 - 提交 - 反馈和检查）
- 本科生与研究生的作业要求和考核办法一样，但在两套系统里分开评价和打分

作业要求

- 完成时间：上机作业 2周/个，实践项目 4周/个。
- 实践项目和上机作业都需要**独立完成**，并**按时提交**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”实践课当天完成提交，过期将影响成绩（每迟交1天扣30%分数）。
- 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等。
- 上机作业报告主要包括上机执行过程和结果等，如有代码实现或其他要求，也要附上。

考核办法和评分规则

- 考核
  - 考查方式：实践项目+上机作业+平时表现
  - 五级制+过程性评价：A / B+ / B- / C / F
- 评分
  - 3个实践项目（报告+检查），占60%，每个占20%；
  - 5个上机作业（报告+检查），占40%，每个占8%。
  - 检查包括项目/作业报告检查和平时上课表现。
- 禁忌
  - 发现项目/作业互相抄袭1次，抄袭者与被抄袭者都按零分处理。

课程资源

- 教材：郭健美,黄波,刘通宇,林晓东,赵鹏. 软件系统优化. 机械工业出版社. 即将出版
  https://solelab.tech/sso
- 示例代码：https://github.com/solecnugit/sso_code
- 课程网站（课件和作业发布）：水杉学堂
  https://www.shuishan.net.cn/mooc/course/softsysopt
- 作业提交系统：水杉码园 https://gitea.shuishan.net.cn/SoftSysOpt.jmguo.2024Fall.DASE
- 课程点播（校内选课同学） https://courses.ecnu.edu.cn/

课堂纪律和沟通

- 原则：**不要影响他人**（教师、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

群聊：软件系统优化2024秋

- 微信群（仅用于课程通知和作业反馈发布）

该二维码7天内(9月15日前)有效，重新进入将更新

- 请假：请在上课前发**电子邮件**，同时抄送**所有**教师和助教，说明具体事由，以备存档。

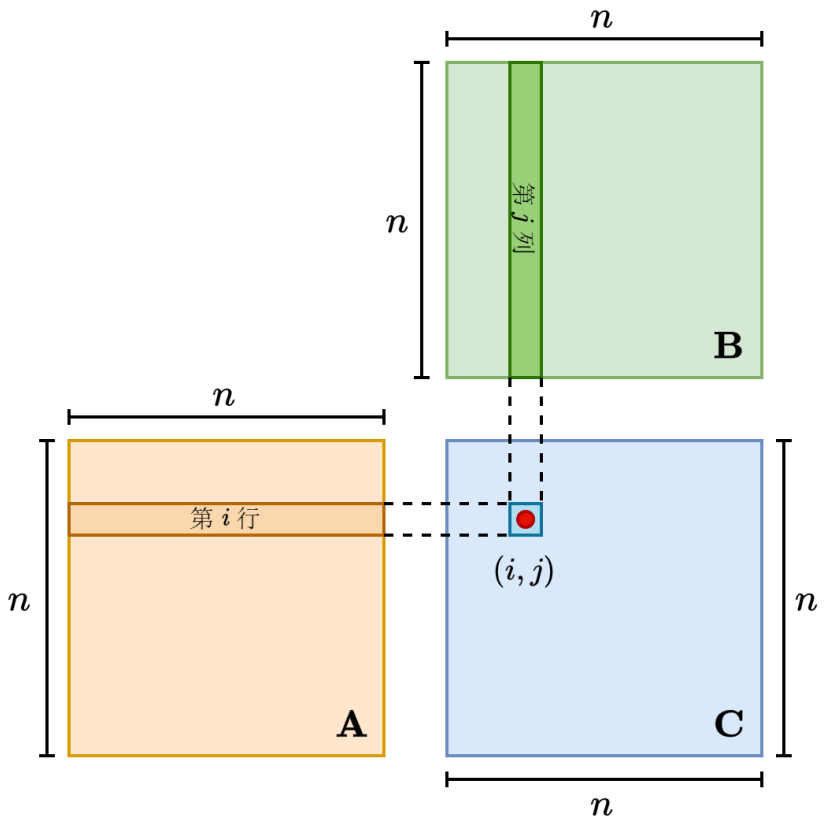
内容

- 课程介绍
- **矩阵乘法优化案例**
- 方法论概述

$n \times n$ 矩阵乘法

$$C = AB$$

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



实验环境

硬件配置		软件配置	
服务器	浪潮 NF5468M6 双路服务器	操作系统	Ubuntu 22.04 LTS / 5.15.0-91-generic
CPU 处理器	2 x Intel Xeon Gold 6326	编译器	Clang/LLVM 15.07
CPU 基频	2.90GHz		Intel oneAPI ICX/ICPX 2023.2.0.20230721
物理核数	16 cores per socket	Python	3.10.12
超线程数	2 threads per core	OpenJDK	17.0.9
L1 Cache	32 KiB I + 48KiB D per core	Linux Perf	5.15.131
L2 Cache	1.25 MiB per core	Intel oneAPI VTune	2023.2.0
L3 Cache	24 MiB per socket	Valgrind	3.18.1
内存	16 x 32G DDR4 3200MT/s	OpenMP	5.0

代码1.1：Python语言实现

```python
import random
from time import *

n = 4096
A = [[random.random() for row in range(n)] for col in range(n)]
B = [[random.random() for row in range(n)] for col in range(n)]
C = [[0 for row in range(n)] for col in range(n)]

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

print("%.2f" % (end - start))
```

运行时间:
18 023.02秒（约 5小时）

代码1.2 & 1.3：Java & C语言实现

```java
import java.util.Random;

public class gemm {
  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 end = System.nanoTime();

    double tdiff = (end - start) * 1e-9;
    System.out.printf("%.2f\n", tdiff);
  }
}
```

```c
#include <stdio.h>
#include <stdlib.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[]) {
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (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 (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            for (k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    gettimeofday(&end, NULL);

    printf("%.2f\n", tdiff(&start, &end));
    return 0;
}
```

不同语言实现的性能差异

编程语言	编译及运行指令	运行时间（单位：秒）
Python	`python ./gemm.py`	18 023.02
Java	`javac ./gemm.java && java gemm`	702.56
C	`icx -O0 gemm.c -o gemm && ./gemm`	781.12

不同语言实现的性能差异

编程语言	编译及运行指令	运行时间（单位：秒）
Python	`python ./gemm.py`	18 023.02
Java	`javac ./gemm.java && java gemm`	702.56
C	`icx -O0 gemm.c -o gemm && ./gemm`	781.12

为何会产生性能差异?

不同语言实现的性能差异

- Python 语言实现是通过Python 解释器（interpreter）进行动态解释运行的。在执行Python 程序时，需要不断地在解释器中对Python 语句进行解释并进行相关对象的构造与销毁，同时还需要对解释器的状态进行动态更新。
- Java 语言实现会先利用 javac 编译器将源程序编译为Java 字节码（bytecode），然后在Java程序实际运行时利用Java 虚拟机（Java Virtual Machine，JVM）对编译好的字节码文件进行解释或者即时（Just-In-Time，JIT）编译成机器码执行。
- C 语言实现则更加简单直接，会由编译器直接编译成机器码执行。
- 通常来说，编译型的C 语言实现要快于混合了解释和编译的Java 语言实现。然而，Java 虚拟机会对Java 程序做许多运行时优化，比如将热点代码即时编译成本地机器码，以减少解释执行的开销。所以，在C 语言编译器禁用优化时（如，编译时使用了-O0 选项），Java 语言实现可能比C语言实现还要快。

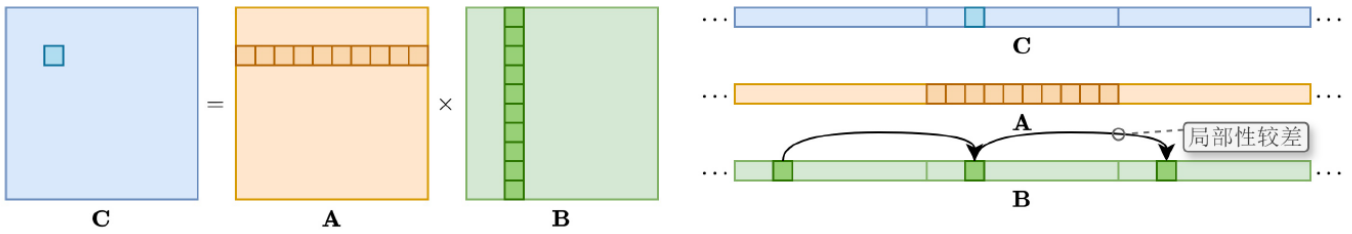
代码1.4：循环交换

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

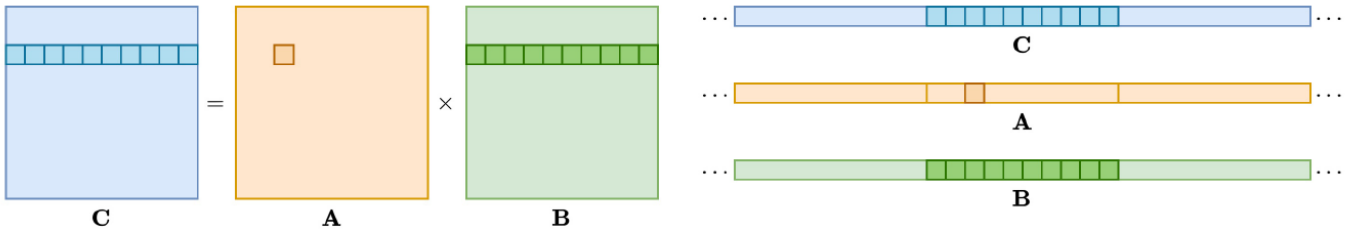
- 什么条件下可以采用循环交换?
- 为何循环交换会提升性能?

不同循环顺序对访存局部性的影响

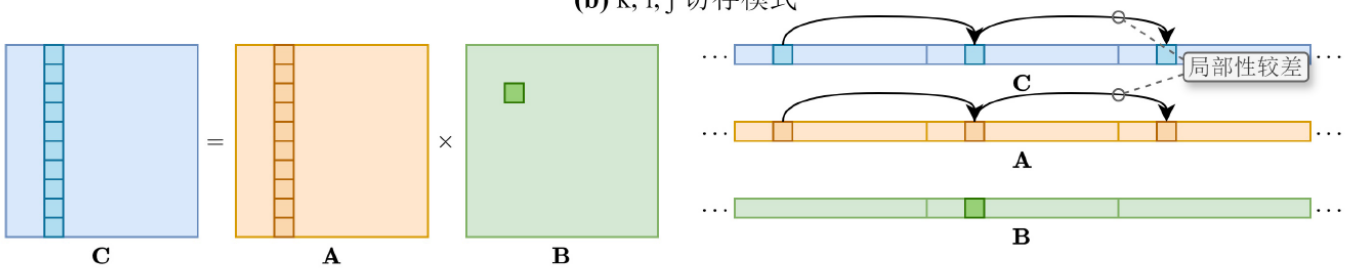
- 数组在内存中是以行主序（row-major order）存储的。
- 提高访存的空间局部性，可以提高缓存命中率，减少对内存的访问，最终提升程序性能。



(a) i, j, k 访存模式



(b) k, i, j 访存模式



(c) j, k, i 访存模式

循环交换的效果

循环顺序	运行时间（单位：秒）	LLC 未命中率
i, j, k	781.12	7.7%
i, k, j	157.53	1.0%
j, i, k	531.30	8.6%
j, k, i	1 080.15	15.4%
k, i, j	143.01	1.0%
k, j, i	988.40	15.4%

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

编译器的不同优化级别

编译优化级别	运行时间（单位：秒）
-O0	143.01
-O1	61.80
-O2	52.02
-O3	52.22

```
$ icx -O2 gemm.c -o gemm
```

多核并行优化

硬件配置		软件配置	
服务器	浪潮 NF5468M6 双路服务器	操作系统	Ubuntu 22.04 LTS / 5.15.0-91-generic
CPU 处理器	2 x Intel Xeon Gold 6326	编译器	Clang/LLVM 15.07
CPU 基频	2.90GHz		Intel oneAPI ICX/ICPX 2023.2.0.20230721
物理核数	16 cores per socket	Python	3.10.12
超线程数	2 threads per core	OpenJDK	17.0.9
L1 Cache	32 KiB I + 48KiB D per core	Linux Perf	5.15.131
L2 Cache	1.25 MiB per core	Intel oneAPI VTune	2023.2.0
L3 Cache	24 MiB per socket	Valgrind	3.18.1
内存	16 x 32G DDR4 3200MT/s	OpenMP	5.0

代码1.6：OpenMP并行化

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <omp.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[]) {
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (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);
    #pragma omp parallel for schedule(static) shared(A, B, C)
    for (i = 0; i < n; i++) {
        for (k = 0; k < n; k++) {
            for (j = 0; j < n; j++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    gettimeofday(&end, NULL);

    printf("%.2f\n", tdiff(&start, &end));
    return 0;
}
```

线程数	i,k,j 循环顺序 运行时间（单位：秒）	k,i,j 循环顺序 运行时间（单位：秒）
4	13.19	16.04
8	6.38	9.90
12	4.39	8.36
16	3.51	7.71

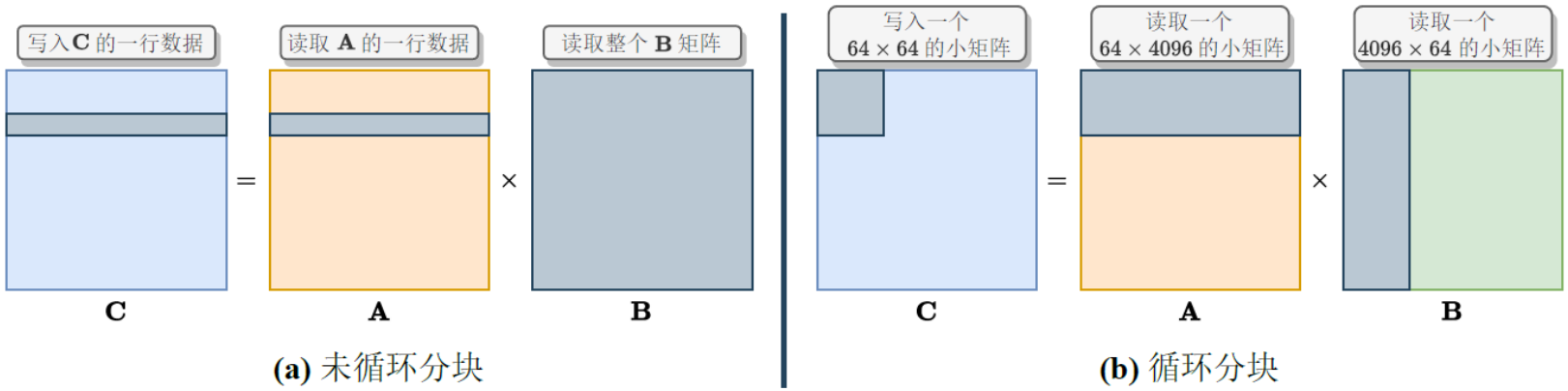
注意：在编写并行化 for 循环程序的时候，尽量对最外层的循环进行并行化，这样往往可以获得最佳的性能。

循环分块

硬件配置	
服务器	浪潮 NF5468M6 双路服务器
CPU 处理器	2 x Intel Xeon Gold 6326
CPU 基频	2.90GHz
物理核数	16 cores per socket
超线程数	2 threads per core
L1 Cache	32 KiB I + 48KiB D per core
L2 Cache	1.25 MiB per core
L3 Cache	24 MiB per socket
内存	16 x 32G DDR4 3200MT/s

循环分块

硬件配置	
服务器	浪潮 NF5468M6 双路服务器
CPU 处理器	2 x Intel Xeon Gold 6326
CPU 基频	2.90GHz
物理核数	16 cores per socket
超线程数	2 threads per core
L1 Cache	32 KiB I + 48KiB D per core
L2 Cache	1.25 MiB per core
L3 Cache	24 MiB per socket
内存	16 x 32G DDR4 3200MT/s



- 未循环分块时，写入C的一行数据（1*4096）
  - 共计 $2^{12} + 2^{12} + 2^{24}$ 次内存访问，约128MiB
- 循环分块后，写入C的一个小分块（64*64）
  - 共计 $2^{12} + 2^{18} + 2^{18}$ 次内存访问，约4MiB

代码1.7：循环分块

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <omp.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[]) {
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (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;
    int ih, il, jh, jl, kh, kl;
    gettimeofday(&start, NULL);

    // S 为具体的分块大小，编译时通过命令行传入
    #pragma omp parallel for shared(A, B, C) schedule(static) collapse(2)
    for (ih = 0; ih < n; ih += S)
        for (jh = 0; jh < n; jh += S)
            for (kh = 0; kh < n; kh += S)
                for (il = 0; il < S; il++)
                    for (kl = 0; kl < S; kl++)
                        for (jl = 0; jl < S; jl++)
                            C[ih + il][jh + jl] += A[ih + il][kh + kl] * B[kh + kl][jh + jl];
    gettimeofday(&end, NULL);

    printf("%.2f\n", tdiff(&start, &end));
    return 0;
}
```

如何确定分块的最佳配置？

代码1.7：循环分块

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <omp.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[]) {
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (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;
    int ih, il, jh, jl, kh, kl;
    gettimeofday(&start, NULL);

    // S 为具体的分块大小，编译时通过命令行传入
    #pragma omp parallel for shared(A, B, C) schedule(static) collapse(2)
    for (ih = 0; ih < n; ih += S)
        for (jh = 0; jh < n; jh += S)
            for (kh = 0; kh < n; kh += S)
                for (il = 0; il < S; il++)
                    for (kl = 0; kl < S; kl++)
                        for (jl = 0; jl < S; jl++)
                            C[ih + il][jh + jl] += A[ih + il][kh + kl] * B[kh + kl][jh + jl];
    gettimeofday(&end, NULL);

    printf("%.2f\n", tdiff(&start, &end));
    return 0;
}
```

如何确定分块的最佳配置？

实验设计 + 实际测量！

分块大小 S	运行时间（单位：秒）
4	4.91
8	3.74
16	1.41
32	1.60
64	1.38
128	1.20
256	1.79

代码1.8：二级循环分块

```
// S 和 T 为多层拆分后的子矩阵的维度，编译时通过命令行传入
#pragma omp parallel for shared(A, B, C) schedule(static) collapse(2)
for (ih = 0; ih < n; ih += S)
    for (jh = 0; jh < n; jh += S)
        for (kh = 0; kh < n; kh += S)
            for (im = 0; im < S; im += T)
                for (jm = 0; jm < S; jm += T)
                    for (km = 0; km < S; km += T)
                        for (il = 0; il < T; ++il)
                            for (kl = 0; kl < T; ++kl)
                                for (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];
```

S	T			
	4	8	16	32
32	1.40	2.89	1.20	\
64	1.79	2.28	1.02	1.34
128	1.67	2.02	1.00	1.27
256	1.88	3.15	1.03	1.30

代码1.9：Intel AVX内建函数（intrinsics）

```
struct timeval start, end;
int ih, il, im, jh, jl, jm, kh, kl, km;
__m256d packedA, packedB, packedC;
gettimeofday(&start, NULL);

#pragma omp parallel for shared(A, B, C) schedule(static) collapse(2)
for (ih = 0; ih < n; ih += 128)
    for (jh = 0; jh < n; jh += 128)
        for (kh = 0; kh < n; kh += 128)
            for (im = 0; im < 128; im += 16)
                for (jm = 0; jm < 128; jm += 16)
                    for (km = 0; km < 128; km += 16)
                        for (il = 0; il < 16; ++il) {
                            for (kl = 0; kl < 16; ++kl) {
                                // 将 AVX 寄存器设置为 4 个 A[ih + im + il][kh + km + kl]
                                packedA = _mm256_set1_pd(A[ih + im + il][kh + km + kl]);
                                for (jl = 0; jl < 16; jl += 4) {
                                    // 加载 C[ih+im+il][kh+km+kl] ~ C[ih+im+il][kh+km+kl+3]
                                    packedC = _mm256_load_pd(&C[ih + im + il][jh + jm + jl]);
                                    // 加载 B[kh+km+kl][jh+jm+jl] ~ B[kh+km+kl][jh+jm+jl+3]
                                    packedB = _mm256_load_pd(&B[kh + km + kl][jh + jm + jl]);
                                    // packedC += packedA * packedB
                                    packedC = _mm256_fmadd_pd(packedA, packedB, packedC);
                                    // 将 packedC 写入内存
                                    _mm256_store_pd(&C[ih + im + il][jh + jm + jl], packedC);
                                }
                            }
                        }
gettimeofday(&end, NULL);
```

```
$ icx ./gemm.c -qopenmp -O2 -march=native -o gemm
```

五万余倍的性能提升

实现版本	代码版本	运行时间（单位：秒）	相对加速比	绝对加速比	GFLOPS
Python 语言实现	代码 1.1	18 023.02	1.0	1	0.008
Java 语言实现	代码 1.2	702.56	25.7	26	0.197
C 语言实现	代码 1.3	781.12	0.9	23	0.176
循环交换	代码 1.4（O0 级别）	143.01	5.5	126	0.961
编译器的不同优化级别	代码 1.4（O2 级别）	52.02	2.8	346	2.642
多核并行优化	代码 1.6	3.51	14.8	5135	39.156
循环分块	代码 1.7	1.20	2.9	15 019	114.532
二级循环分块	代码 1.8	1.00	1.2	18 023	137.439
AVX 内建函数	代码 1.9	0.36	2.8	50 064	381.775

$$\text{相对加速比} = \frac{\text{前一版本的运行时间}}{\text{当前版本的运行时间}}$$

$$\text{绝对加速比} = \frac{\text{性能最差版本的运行时间}}{\text{其它版本的运行时间}}$$

对于 4096×4096 的矩阵乘法，总运算量约为 $2 \times (2^{12})^3$ 次浮点运算

内容

- 课程介绍
- 矩阵乘法优化案例
- **方法论概述**

There's Plenty of Room at the Bottom

An Invitation to Enter a New Field of Physics

by Richard P. Feynman

Miniaturizing the computer

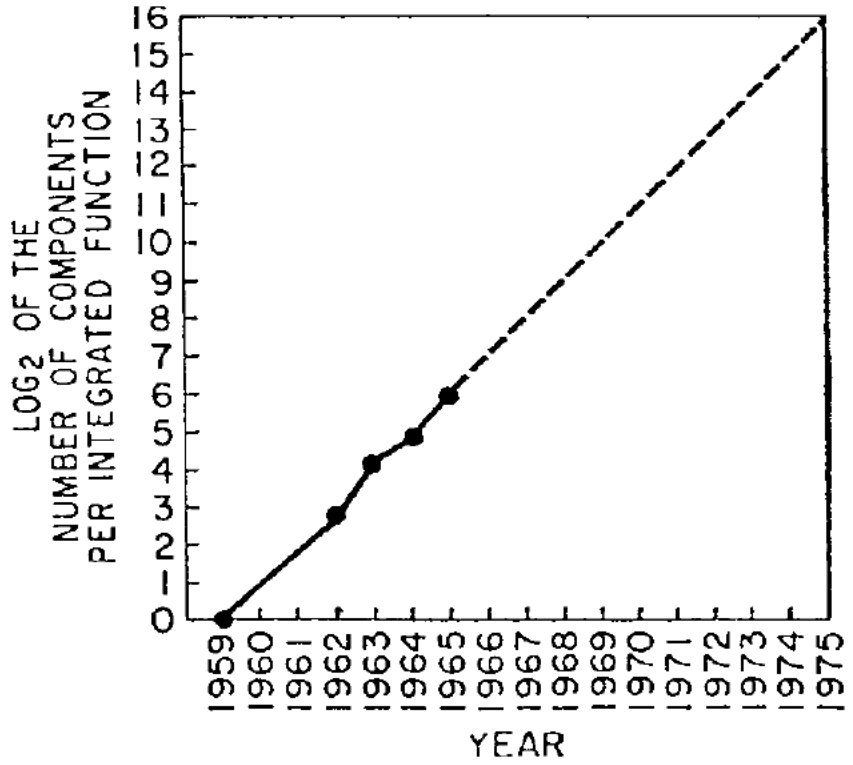
I don't know how to do this on a small scale in a practical way, but I do know that computing machines are very large; they fill rooms. Why can't we make them very small, make them of little wires, little elements---and by little, I mean *little*. For instance, the wires should be 10 or 100 atoms in 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



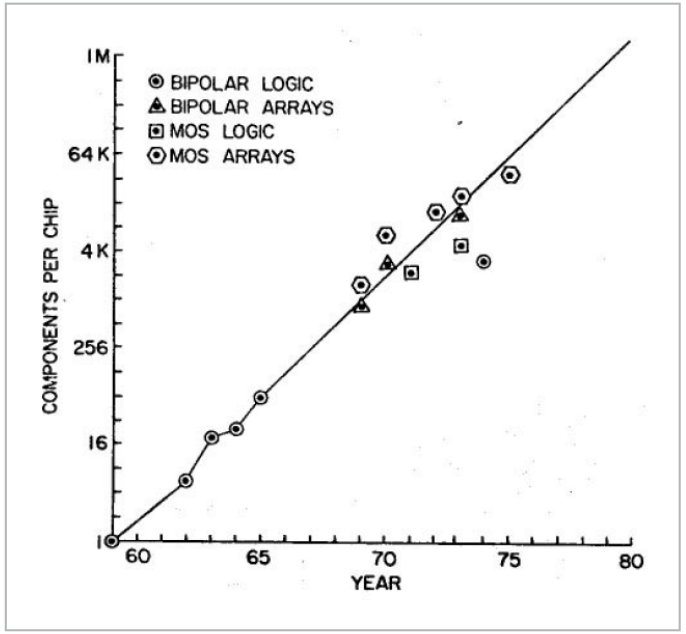
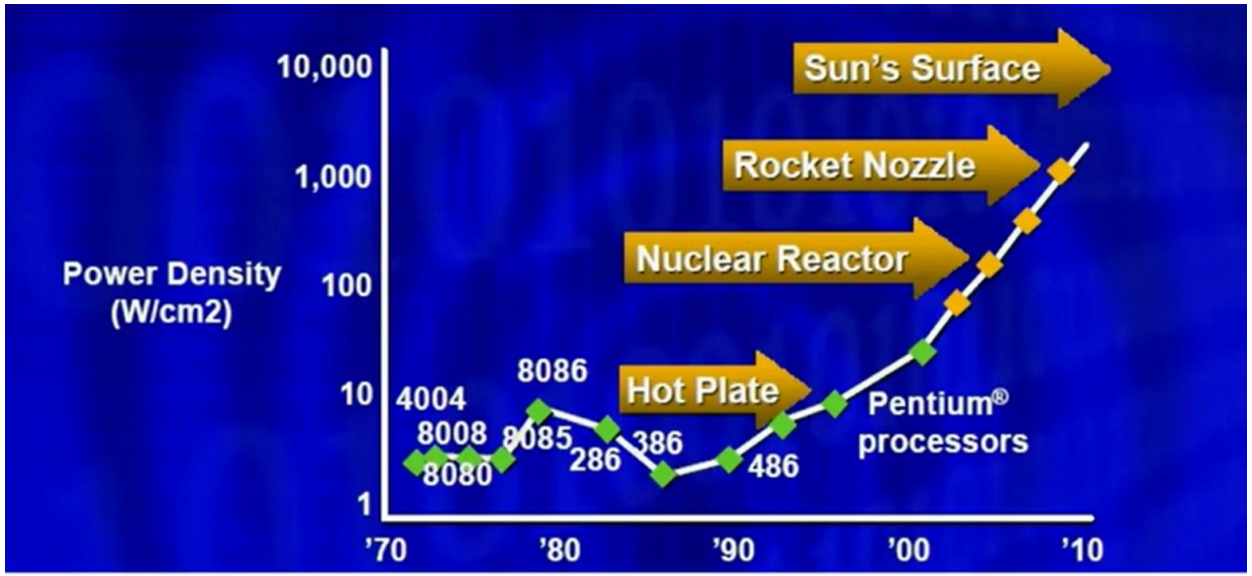


Figure 1 Approximate component count for complex integrated circuits vs. year of Introduction.

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

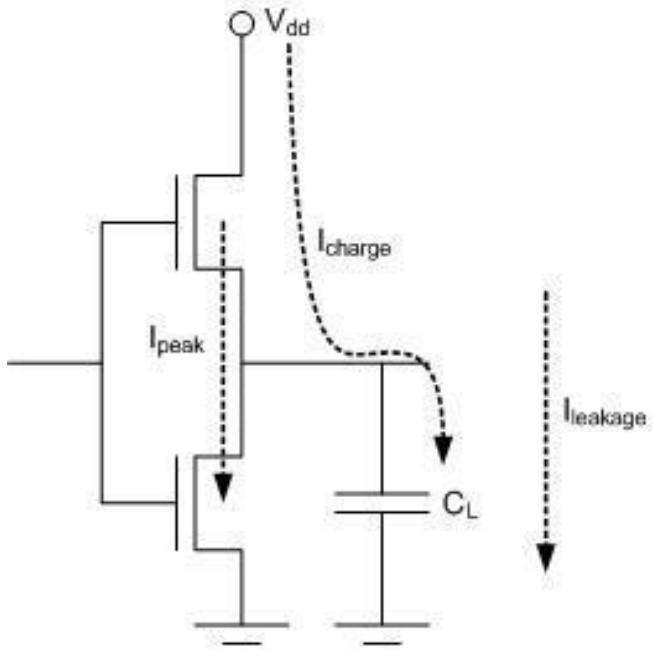
[G. E. Moore, **Cramming more components onto integrated circuits**. Electronics 38, 1–4 (1965).]
[G. E. Moore, **Progress in digital integrated electronics**, in International Electron Devices Meeting Technical Digest (IEEE, 1975), pp. 11–13.]

Dennard Scaling 登纳德缩放定律逐渐失效



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

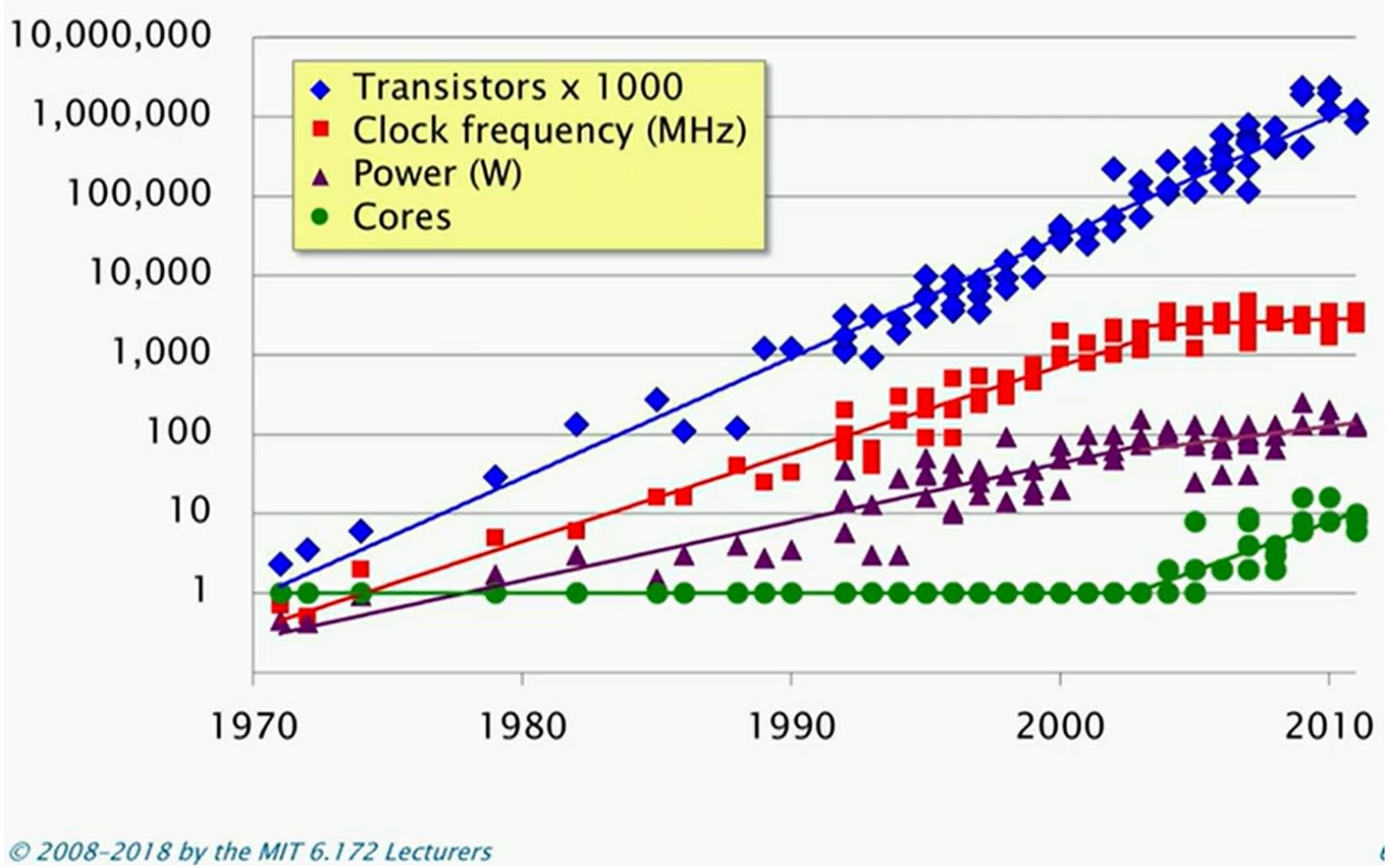
Projected power density, if clock frequency had continued its trend of scaling 25%–30% per year.



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]

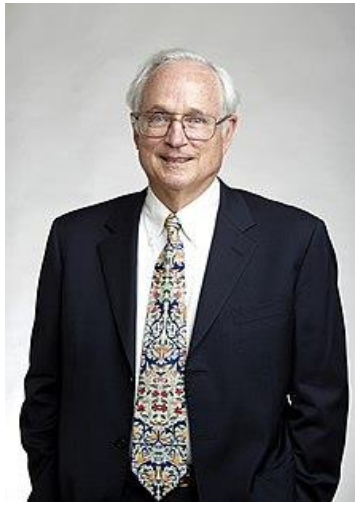
摩尔定律逐渐失效

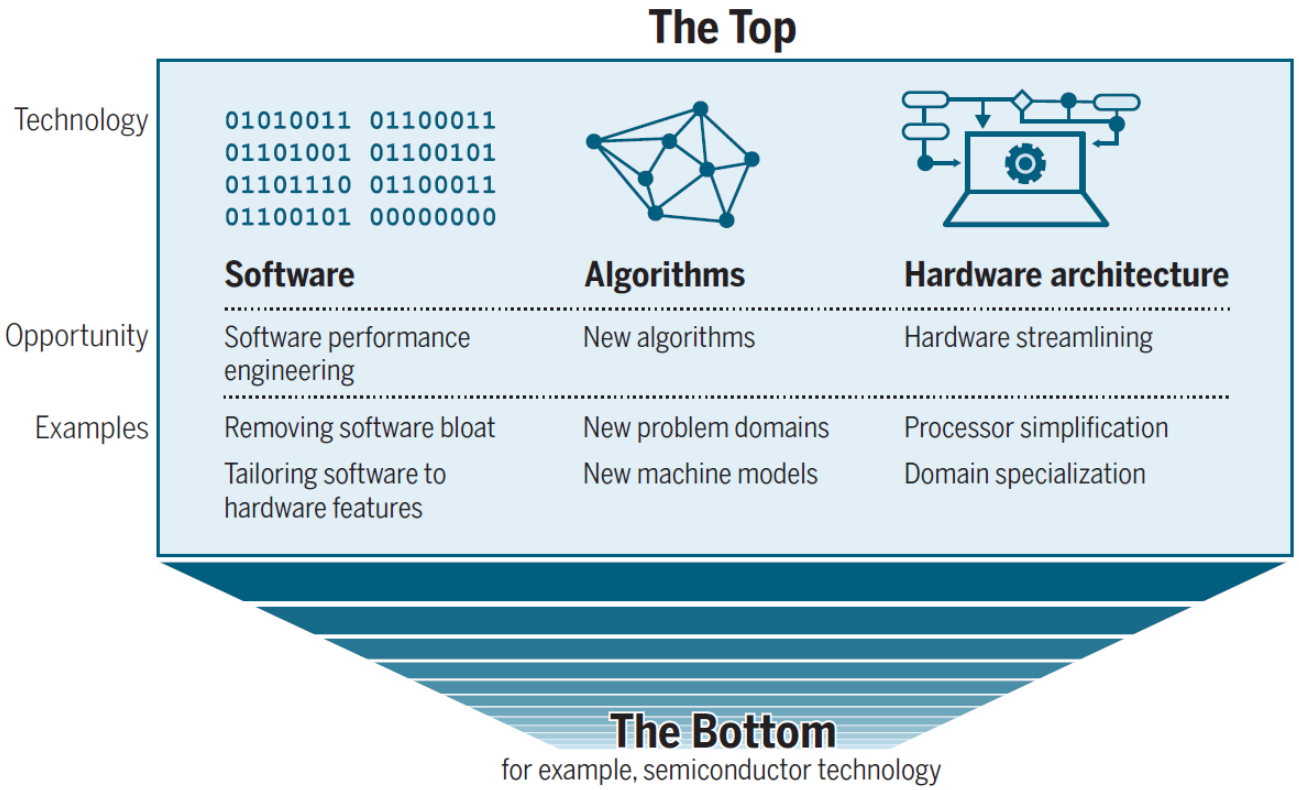

10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
Transistors x 1000
Clock frequency (MHz)
Power (W)
Cores
1970
1980
1990
2000
2010
© 2008–2018 by the MIT 6.172 Lecturers

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.



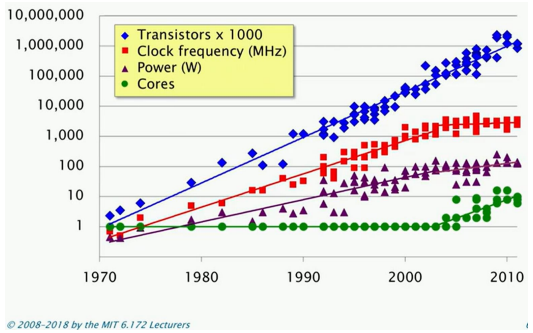


Performance gains after Moore's law ends. In the post-Moore era, improvements in computing power will increasingly come from technologies at the "Top" of the computing stack, not from those at the "Bottom", reversing the historical trend.

[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)]

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?



2070?

2020

2014

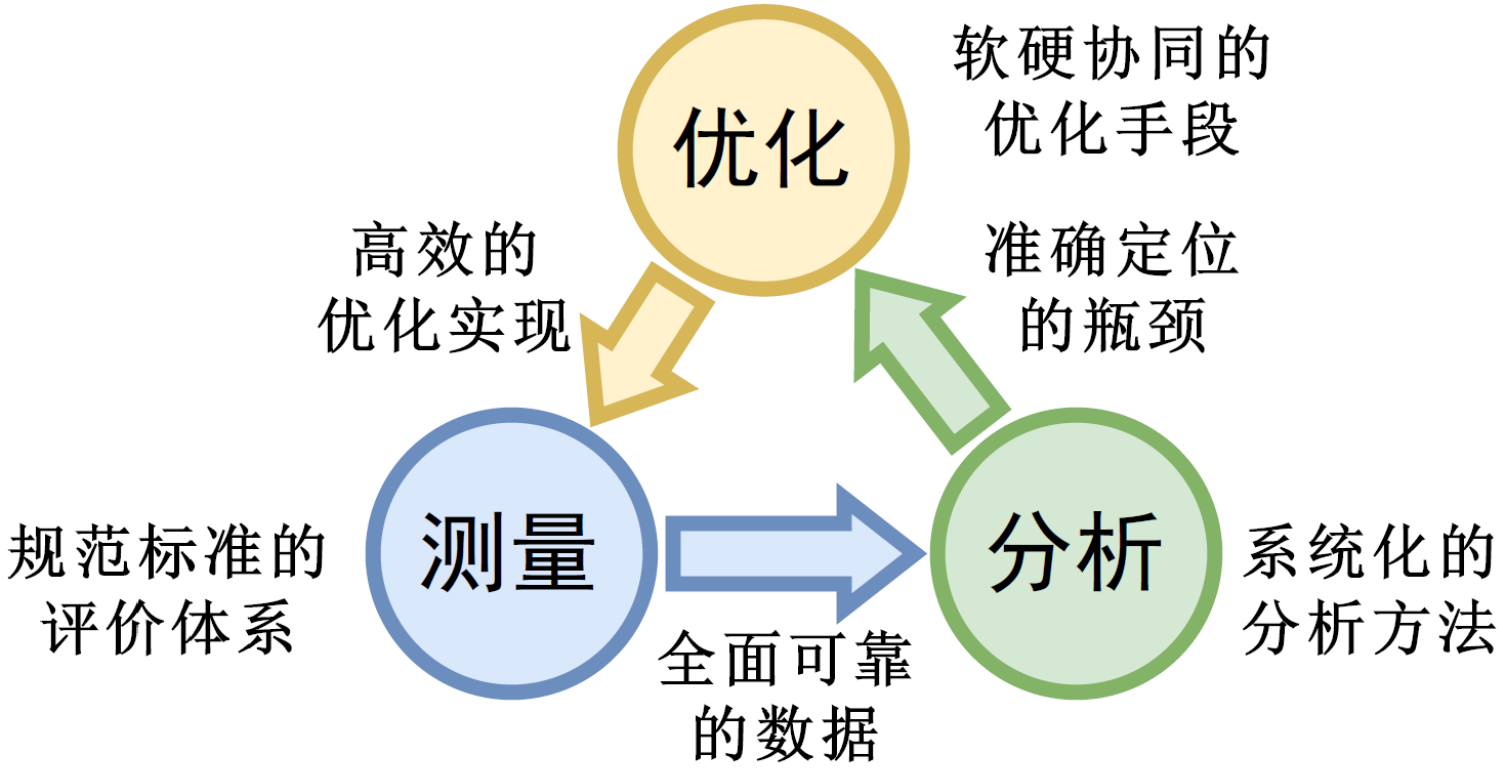
1960

There's Plenty of Room at the Bottom

An Invitation to Enter a New Field of Physics

by Richard P. Feynman

数据驱动的系统优化方法



从单点到全局的“系统观”