清華大學
Tsinghua University

高性能计算导论
第9讲：单机性能优化

翟季冬

计算机系

目录

- 性能优化概述
- 典型优化方式
- 循环优化
- 编译优化
- 性能优化示例
- 性能模型

性能优化概述

为什么要性能优化？

There's more to performance than asymptotic complexity !

- 常数因子同样很重要
  - 代码的不同实现方式可能导致10倍以上的性能差距
  - 程序员需要从多个层次进行性能优化
    - 数据表示、循环、函数等
- 需要充分了解体系结构才能优化性能
  - 程序是如何编译运行的
  - 现代处理器和存储系统是如何工作的
  - 如何测量程序的性能并定位瓶颈
  - 如何在不破坏代码模块化和通用性的情况下优化性能

must have a strong understanding of computer achitecture!
1. how do modern processors and storage sysems works?
2. how to measure the performance of program and locate bottleneck
3. how to optimize performance without breaking code modularity and versatility

典型优化方式

代码移动 Code Motion

- 在任何处理器架构上，程序员/编译器都应该进行的优化

- 代码移动
  - 减少计算的频率
    - 如果该优化不会影响结果的正确性
    - 将代码移动到循环之外

```
void set_row(double* a, double* b,
long i, long n) {
    long j;
    for (j = 0; j < n; j++)
        a[n * i + j] = b[j];
}
```

```
long j;
int ni = n * i;
for (j = 0; j < n; j++)
    a[ni + j] = b[j];
```

强度拆减 Strength Reduction

- 用简单的计算代替复杂的计算
- 用位移 (shift)、加法来代替乘法或除法

  16 * x ➔ x << 4
  - 依赖硬件平台
  - 取决于乘法或除法指令的开销
    - 在Intel Nehalem上，整数乘法需要 3 个 CPU 周期

```
for (i = 0; i < n; i++) {
    int ni = n * i;
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
}
```

```
int ni = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
    ni += n;
}
```

公共子表达式消除 Common Subexpressions Elimination

- 重用部分表达式的结果
- GCC 开 O1 优化

```
/* Sum neighbors of i,j */
up = val[(i - 1) * n + j];
down = val[(i + 1) * n + j];
left = val[i * n + j - 1];
right = val[i * n + j + 1];
sum = up + down + left + right;
```

```
long inj = i * n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3个乘法： i * n, (i – 1) * n, (i + 1) * n

1个乘法： i * n

```
leaq 1(%rsi), %rax # i+1
leaq -1(%rsi), %r8 # i-1
imulq %rcx, %rsi # i*n
imulq %rcx, %rax # (i+1)*n
imulq %rcx, %r8 # (i-1)*n
addq %rdx, %rsi # i*n+j
addq %rdx, %rax # (i+1)*n+j
addq %rdx, %r8 # (i-1)*n+j
```

```
imulq %rcx, %rsi # i*n
addq %rdx, %rsi # i*n+j
movq %rsi, %rax # i*n+j
subq %rcx, %rax # i*n+j-n
leaq (%rsi,%rcx), %rcx # i*n+j+n
```

常数折叠 Constant Folding

- 对常量计算在编译时完成
- 例子：sum 变量是两个常数之和，编译器可以预先计算出结果
  - 避免运行时的计算

```
add = 100;
aug = 200;
sum = add + aug;
```

```
sum = 300;
```

死代码消除 Dead Code Removal

- 编译器消除不会被执行的代码

```
var = 5;
printf("%d", var);
exit(-1);
printf("%d", var * 2);
```

```
var = 5;
printf("%d", var);
exit(-1);
```

复制传播 Copy Propagation

- 消除语句之间的依赖关系
- 提高指令级并行性

```
x = y;
z = 1 + x;
```

存在数据依赖

```
x = y;
z = 1 + y;
```

消除数据依赖

变量重命名 Variable Renaming

- 消除输出变量之间的依赖关系

```
x = y * z;
q = r + x * 2;
x = a + b;
```

```
x0 = y * z;
q = r + x0 * 2;
x = a + b;
```

- 前者存在输出变量之间的依赖，而后者不存在
- 但是变量 x 的最后结果相同

循环优化

循环优化

循环优化是程序优化一个重要的环节：

- 提取循环不变量
- 消除判断
- 迭代剥离
- 循环集分割
- 循环交换
- 循环展开
- 循环融合
- 循环分裂

iterative stripping?

提取循环不变量 Hoisting Loop Invariant Code

- 循环内部不会改变的变量为循环不变量
- 不需要每次循环都计算一遍

```
for (i = 0; i < n; i++) {
    a[i] = b[i] + c * d;
    e = g(n);
}
```

```
temp = c * d;
for (i = 0; i < n; i++) {
    a[i] = b[i] + temp;
}
e = g(n);
```

循环判断外提

```
for (i = 0; i < n; i++) {
    for (j = 1; j < n; j++) {
        if (t[i] > 0)
            a[i][j] = a[i][j] * t[i] + b[j];
        else
            a[i][j] = 0.0;
    }
}
```

```
for (i = 0; i < n; i++) {
    if (t[i] > 0) {
        for (j = 1; j < n; j++) {
            a[i][j] = a[i][j] * t[i] + b[j];
        }
    } else {
        for (j = 1; j < n; j++) {
            a[i][j] = 0.0;
        }
    }
}
```

该判断与循环变量 j 无关，所以将该判断挪到循环之外

迭代剥离

```
for (i = 0; i < n; i++) {
    if ((i == 0) || (i == (n - 1))) {
        x[i] = y[i];
    } else {
        x[i] = y[i + 1] + y[i - 1];
    }
}
```

- 将不一样的循环迭代剥离到循环之外

```
x[0] = y[0];
for (i = 1; i < n - 1; i++) {
    x[i] = y[i + 1] + y[i - 1];
}
x[n - 1] = y[n - 1];
```

下标集分割 Index Set Splitting

```
for (i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
    if (i >= 10) {
        d[i] = a[i] + b[i - 10];
    }
}
```

```
for (i = 0; i < 10; i++) {
    a[i] = b[i] + c[i];
}
for (i = 10; i < n; i++) {
    a[i] = b[i] + c[i];
    d[i] = a[i] + b[i - 10];
}
```

- 这是迭代剥离更一般的形式

循环交换 Loop Interchange

```
for (j = 0; j < nj; j++) {
    for (i = 0; i < ni; i++) {
        a[i][j] = b[i][j];
    }
}
```

```
for (i = 0; i < ni; i++) {
    for (j = 0; j < nj; j++) {
        a[i][j] = b[i][j];
    }
}
```

- 在二维数组中，a[i][j] 与 a[i][j + 1] 在内存中连续，而 a[i][j] 和 a[i + 1][j] 相距比较远，所以用 j 作为内层循环能更好发挥局部性

循环展开 Unrolling

- 循环展开：对于计算较多，访存较少的循环有更好的效果
- 提高单个循环内的计算
  - 利用 SIMD 加速
  - 增加指令级并行
- 可以减少循环变量的比较次数和分支跳转次数

```
for (i = 0; i < n; i++) {
    a[i] = a[i] + b[i];
}
```

```
for (i = 0; i < n; i += 4) {
    a[i] = a[i] + b[i];
    a[i + 1] = a[i + 1] + b[i + 1];
    a[i + 2] = a[i + 2] + b[i + 2];
    a[i + 3] = a[i + 3] + b[i + 3];
}
```

循环融合 Loop Fusion

```
for (i = 0; i < n; i++) {
    a[i] = b[i] + 1;
}
for (i = 0; i < n; i++) {
    c[i] = a[i] / 2;
}
for (i = 0; i < n; i++) {
    d[i] = 1 / c[i];
}
```

```
for (i = 0; i < n; i++) {
    a[i] = b[i] + 1;
    c[i] = a[i] / 2;
    d[i] = 1 / c[i];
}
```

- 更少的分支跳转
- 更少的内存访问
  - Cache

循环分裂 Loop Fission

```
for (i = 0; i < n; i++) {
    a[i] = b[i] + 1;
    c[i] = a[i] / 2;
    d[i] = 1 / c[i];
}
```

```
for (i = 0; i < n; i++) {
    a[i] = b[i] + 1;
}
for (i = 0; i < n; i++) {
    c[i] = a[i] / 2;
}
for (i = 0; i < n; i++) {
    d[i] = 1 / c[i];
}
```

- 循环融合或循环分裂需要根据具体的程序和硬件决定
- 在某些情况下：循环内的访存比较复杂，分裂可以减少程序访存范围

内联 Inlining

```
float func(x) {
    return x * 3;
}

for (i = 0; i < n; i++) {
    a[i] = func(i + 1);
}
```

```
for (i = 0; i < n; i++) {
    a[i] = (i + 1) * 3;
}
```

- 内联函数会被直接展开到调用者的位置，消除了函数调用的开销

编译优化

编译优化

- 编译优化：在编译器中实现的优化
- 编译器：代码翻译、将程序翻译成高效机器码
  - 寄存器分配
  - 代码重排
  - 死代码消除
  - 循环优化
- 不改变程序计算复杂度
  - 程序员决定选择最好的算法
  - 但常数因子也同样重要

编译优化：参数选择

- 为什么需要参数选择？
  - 理论上，编译器“明白”之前提到的所有优化，但实际上不是这样
  - 编译器不知道针对特定的处理器而言，哪种优化算法更合适

- 编译器可能需要你的帮助：
  - 选择不同的编译参数
    - 如下是影响单机优化的编译参数：
    - 超标量、流水、向量化、循环优化、内联、过程间优化等
  - 重构代码：使代码结构更简单清楚
  - 使用特殊的函数 (intrinsics) 或者内嵌汇编

```
compiler doesn't always know the best
optimization algorithms, requires to choose among
 different compilation parameters

factors that affect signle machine
1. superscaler, pipeline, vectorization, loop
optimization, inline,
```

Intel 编译器

- 多体系结构支持
  - 多体系结构支持：IA-32、IA-64、 Intel ® 64（包括 AMD 处理器)
  - 跨平台兼容性
  - 多语言支持：C、C++、Fortran
  - 多操作系统支持：Windows、macOS、Linux

Host \ Target	IA-32 Architecture	Intel 64 Architecture	IA-64 Architecture
IA-32 Architecture	Yes	Yes	Yes
Intel 64 Architecture	Yes	Yes	Yes
IA-64 Architecture	No	No	Yes

Intel 编译器：典型参数

macOS/Linux	Windows	描述
-O1	/O1	面向可执行文件大小的优化
-O2	/O2	面向执行速度的优化（默认）
-O3	/O3	O2 + 进一步优化：面向以浮点运算为主的循环和大规模数据处理的程序
-fast	/fast	集成各种优化方法
-g	/Zi	加入调试信息

✓ 阅读不同编译器的参数说明文档、选取合适的编译参数

O2 + further optimization: for loops and large-scale data processing programs dominated by floating-point arithmetic

编译优化的限制

limitations
1. can't change the behavior of a program
2. most code is static analysis (can't predict input)

1. most analysis is limited to functions, overal overhead is too large, new gcc can' perform interprocedural analysis within a signle file, still room for exploration in code optimization between different files

- 编译优化的限制
  - 不能改变程序的行为
    - 除非程序使用了不标准的语言特性（“未定义行为”）
  - 大多数代码分析都是静态分析
    - 编译器难以预测运行时的输入
  - 虽然某些条件只能在特殊情况出现甚至不出现，但仍会阻碍编译器的优化
- 大部分的代码分析都局限在函数内部
  - 全局的代码分析开销太大
  - 新版本的 GCC 能够进行单个文件内部的过程间分析
    - 但是，不同文件间的代码优化仍有探索空间

阻碍编译优化
—函数副作用

hinder compilation optimization

阻碍编译优化-1: 函数副作用

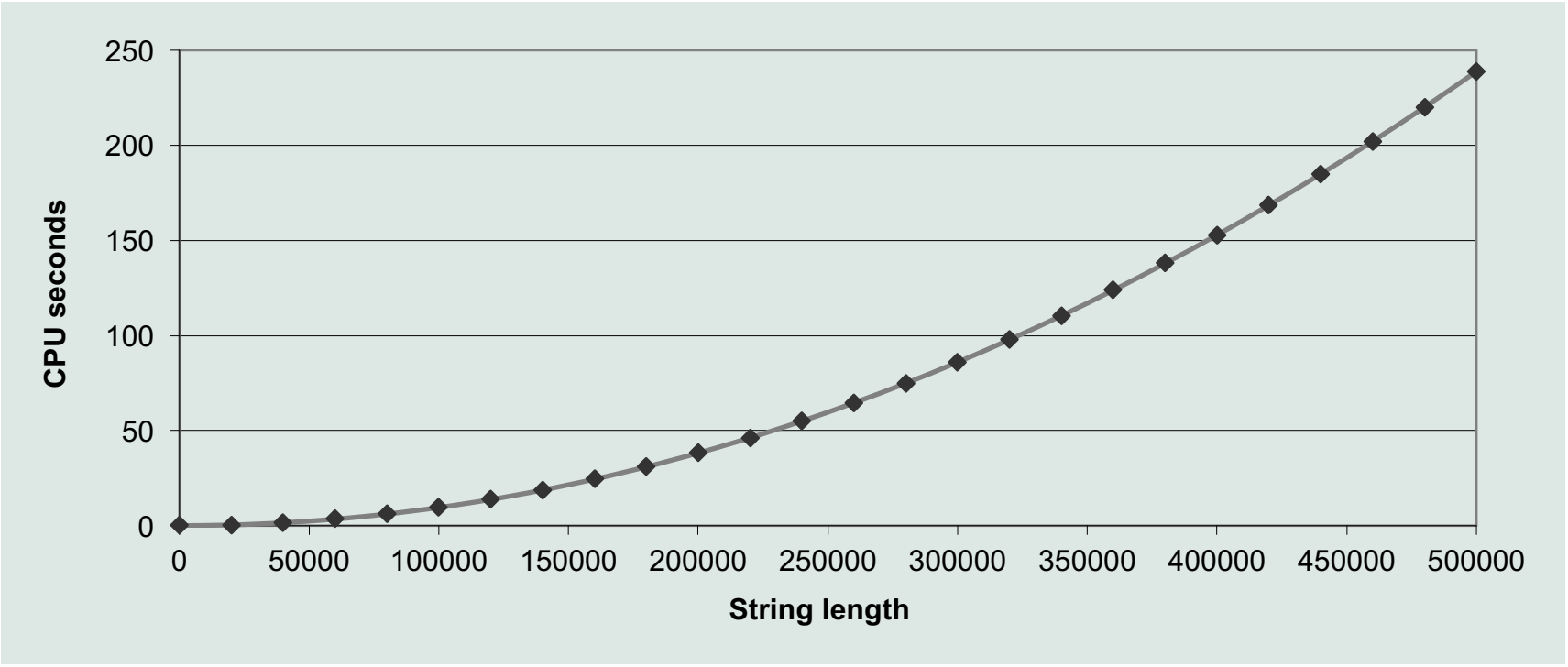
- 阻碍编译优化的因素　function side effects
  - 函数副作用
  - 内存别名

```
void lower1(char* s) {
    size_t i;
    for (i = 0; i < strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
    }
}
```

将 string 转换成小写的函数

性能分析

- 当字符串长度增加一倍时，时间增加了四倍
- 平方量级的时间复杂度



原因分析

```c
/* My version of strlen */
size_t strlen(const char* s) {
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- Strlen 的性能
  - 确定字符串长度的方式是扫描整个字符串，直到找到 null 字符
- 时间复杂度，假设字符串的长度为 N
  - N 次调用 strlen
  - $O(N^2)$

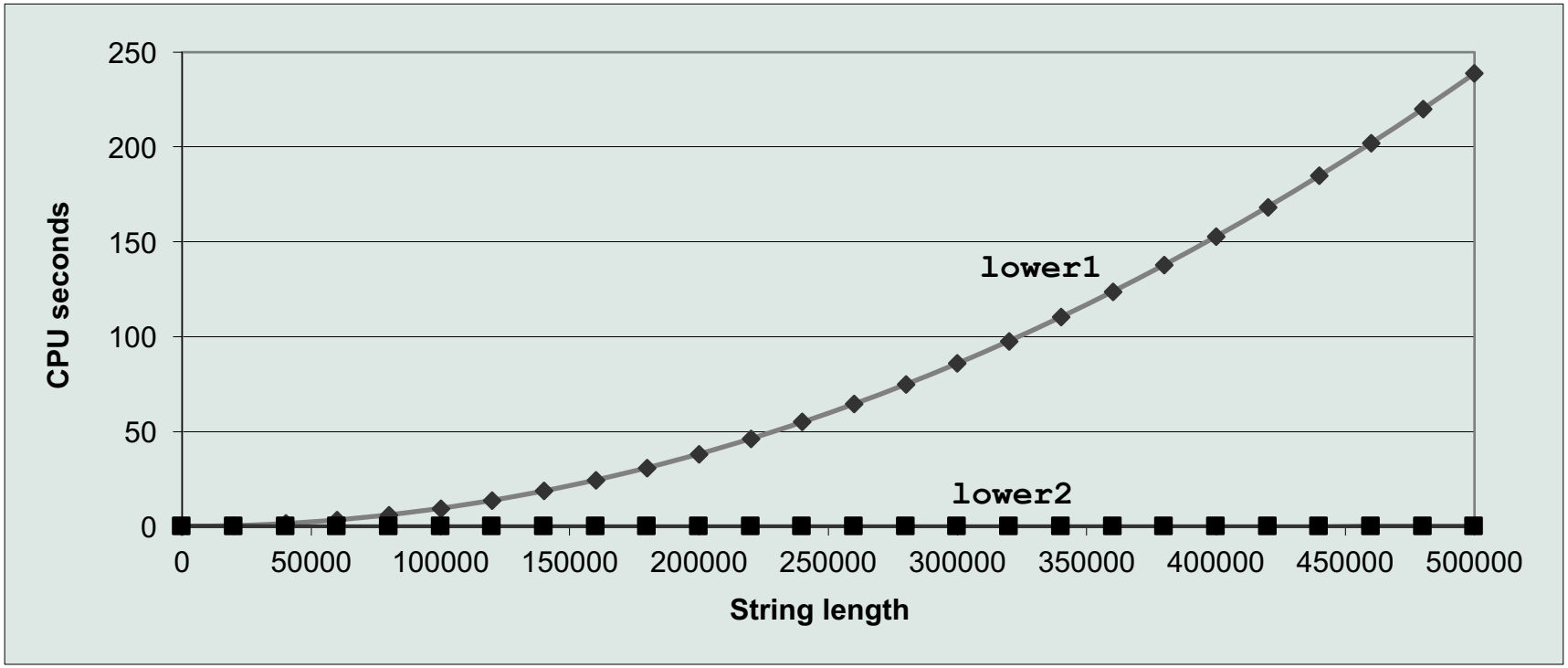
性能优化

- 因为 strlen 的结果不会随着循环而改变
- 将 strlen 函数的调用移出循环
- 代码移动的一种方式

```
void lower2(char* s) {
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++) {
        if (s[i] >= 'A' && s[i] <= 'Z`)
            s[i] -= ('A' - 'a`);
    }
}
```

改进后的性能

- 当字符串长度加倍时，时间也加倍
- 线性时间复杂度



阻碍编译优化因素1: 函数副作用

- 为什么编译器不能将 strlen 移动到循环外面？
  - 函数可能存在副作用
    - 每次调用时都会改变全局状态
  - 即使传入参数相同，函数可能返回不同的结果
    - 依赖于全局的状态（全局变量）
- 注意
  - 编译器对待函数就像黑盒
  - 编译器对函数调用的优化很弱
- 补救方式
  - 使用内联函数
  - 自己做代码移动等优化
  - C++ constexpr 函数

```
size_t lencnt = 0;
size_t strlen(const char* s) {
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    lencnt += length;
    return length;
}
```

有副作用的函数调用

阻碍编译优化

—内存别名

阻碍编译优化因素2: 内存别名 Memory Alias

- 内存别名
  - 两个不同的指针都指向了同一个内存地址
  - 在 C 中很容易出现
    - C 允许地址的算术运算
    - 直接访问存储结构
  - 建议使用局部变量
    - 在循环内进行累加
    - 显士告诉编译器不检查内存别名

内存访问

```
/* Sum rows is of n X n matrix a and store in vector b */
void sum_rows1(double* a, double* b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i * n + j];
    }
}
```

```
# sum_rows1 inner loop
.L4:
    movsd (%rsi,%rax,8), %xmm0 # FP load b[i]
    addsd (%rdi), %xmm0 # FP add a[i*n+j]
    movsd %xmm0, (%rsi,%rax,8) # FP store b[i]
    addq $8, %rdi
    cmpq %rcx, %rdi
    jne .L4
```

- 每次迭代都会更新内存中 b[i] 的值
- 编译器不能进行优化

内存别名示例

```
/* Sum rows is of n X n matrix a and store in vector b */
void sum_rows1(double* a, double* b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i * n + j];
    }
}
```

```
double A[9] =
    { 0, 1, 2,
      4, 8, 16},
     32, 64, 128};

double *B = A + 3;
sum_rows1(A, B, 3);
```

B 的值：

```
init:  [4, 8, 16]
i = 0: [3, 8, 16]
i = 1:[3, 22, 16]
i = 2:[3, 22,224]
```

- 每一轮迭代都会更新 b[i]，而 b[1] 和 a[4] 指向了相同的内存地址
- 优化时必须考虑这些情况，否则会改变程序行为

消除内存别名

```
/* Sum rows is of n X n matrix a
and store in vector b */
void sum_rows2(double* a, double* b, long
n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i * n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L10:
    addsd (%rdi), %xmm0 # FP load + add
    addq $8, %rdi
    cmpq %rax, %rdi
    jne .L10
```

- 可以将中间结果保存在寄存器里

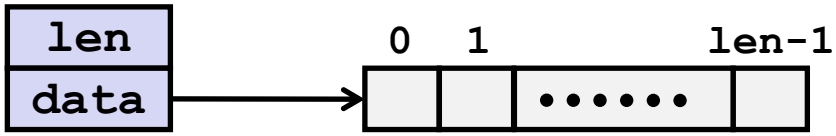
```
/* Sum rows is of n X n matrix a and
store in vector b */
void sum_rows1(double* a, double* b,
long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i * n + j];
    }
}
```

```
# sum_rows1 inner loop
.L4:
    movsd (%rsi,%rax,8), %xmm0 # FP
load b[i]
    addsd (%rdi), %xmm0
    movsd %xmm0, (%rsi,%rax,8) # FP
store b[i]
    addq $8, %rdi
    cmpq %rcx, %rdi
    jne .L4
```

优化示例

待优化程序 - 数据类型

```
/** data structure for vectors **/
typedef struct{
    size_t len;
    data_t *data;
} vec;
```



- 数据类型 **data_t**
  - **int**
  - **long**
  - **float**
  - **double**

```
/** retrieve vector element
   and store at val **/
int get_vec_element
  (*vec v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

待优化程序

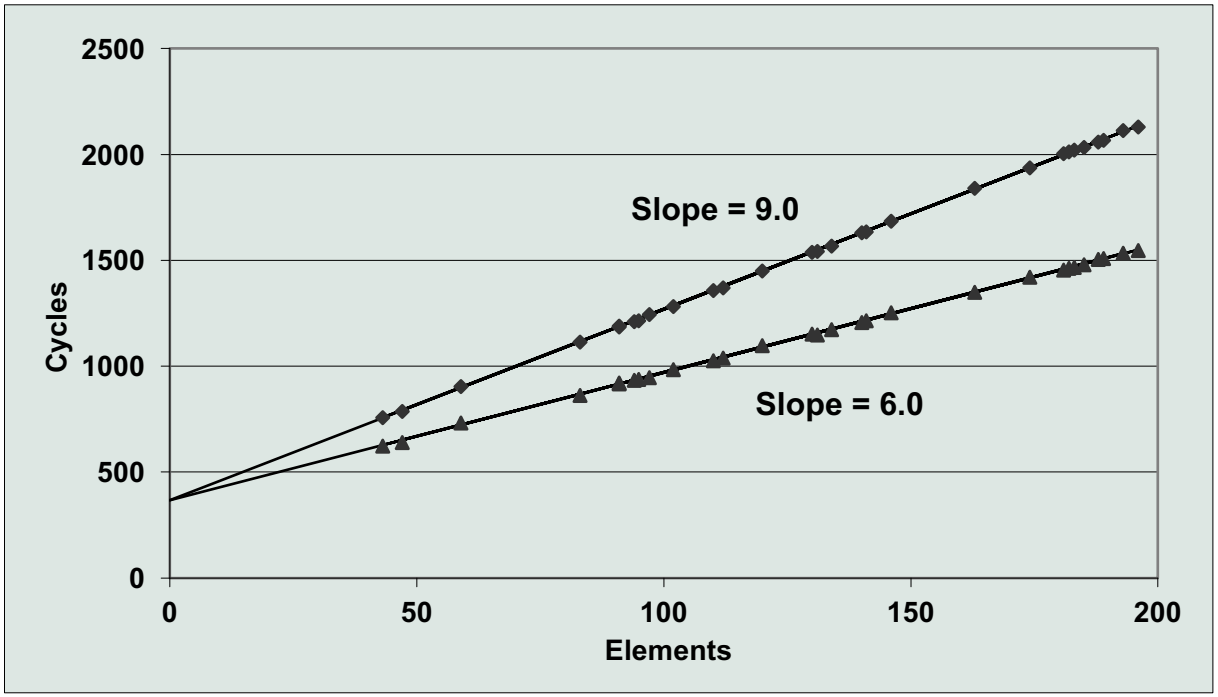
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

计算向量元素的和或积

- 数据类型 **data_t**
  - **int**
  - **long**
  - **float**
  - **double**
- 操作类型
  - 使用不同定义 **OP** 和 **IDENT**
  - + / 0
  - * / 1

性能指标

- 性能指标：方便描述测试程序的性能
- N 个元素
- CPE：Cycle Per Element
- T = CPE * N + Overhead



程序初始性能

- 优化目标：提升 400 倍性能

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18

编译优化

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

通用优化方法

- 函数外移
- 累加：临时变量

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

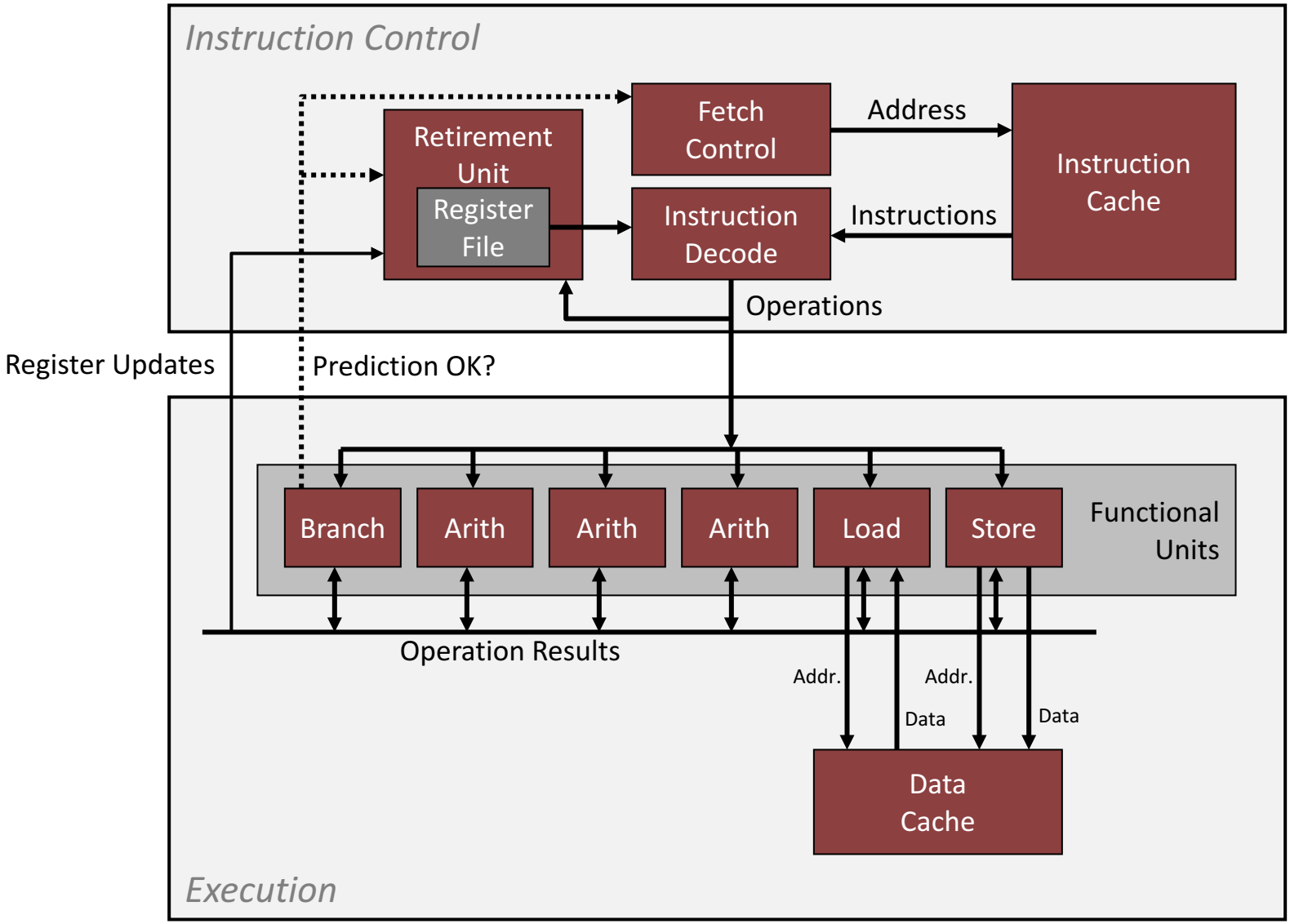
```
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

通用优化性能

```
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

现代处理器设计回顾

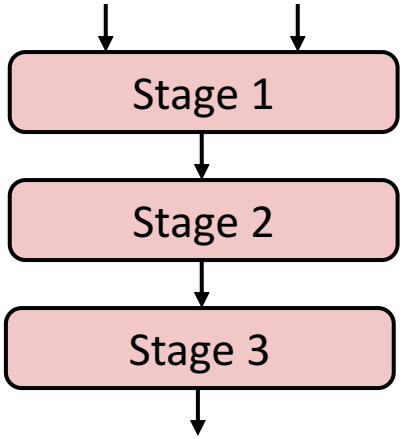

Instruction Control
Retirement Unit
Register File
Fetch Control
Address
Instruction Cache
Instruction Decode
Instructions
Operations
Register Updates
Prediction OK?
Branch
Arith
Arith
Arith
Load
Store
Functional Units
Operation Results
Addr.
Addr.
Data
Data
Data Cache
Execution

超标量处理器

- 超标量：
  - 超标量处理器能在一个周期内发射和执行多条指令
  - 处理器同时加载多条指令并动态地调度执行
- 优点：
  - 超标量处理器能够自动地利用程序本身的指令级并行
- 目前大多数处理器都是超标量的
- Intel：从 Pentium (1993) 开始

流水线加速执行

```
long mult_eg(long a, long b, long c) {
    long p1 = a * b;
    long p2 = a * c;
    long p3 = p1 * p2;
    return p3;
}
```



	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

- 将计算分成 3 个阶段
- 当stage i 将结果传给 stage i + 1 后，stage i 可以开始新的计算
- 例如：完成 3 个乘法运算需要 7 个周期，即使每个乘法需要 3 个周期
- 工作条件：没有数据依赖关系

Haswell 架构

- 共有 8 个计算单元、多个指令可以同时执行
  - 4 integer
  - 2 FP multiply
  - 1 FP add
  - 1 FP divide
  - 2 个包含地址计算的 load
  - 1 个包含地址计算的 store
- 有些指令执行完需要 1 个周期以上，但是可以流水执行

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

指令级并行优化

- 需要了解现代处理器的设计
  - 硬件可以并行执行多条指令
- 数据依赖会影响性能
- 简单的转化可以带来很大的性能提升
  - 编译器一般无法进行这些转化
  - 浮点数计算不满足结合律和分配律（由于精度原因）

Combine 函数

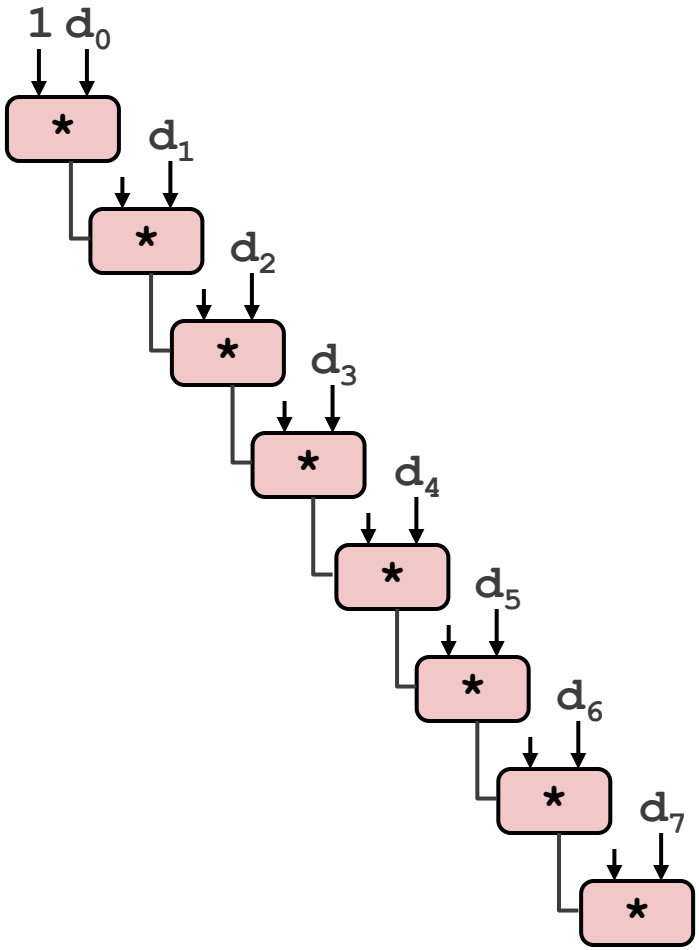
- 主要的循环部分

```
for (int i = 0; i < length; ++i) {
    t = t * d[i];
}
```

```
L519: # Loop:
    imull (%rax,%rdx,4), %ecx # t = t * d[i]
    addq $1, %rdx # i++
    cmpq %rdx, %rbp # Compare length:i
    jl .L519 # If <, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine	1.27	3.01	3.01	5.01
延迟限制	1.00	3.00	3.00	5.00

Combine 计算流程（以乘法为例）



- 核心计算（假设length = 8)

```
(((((((1 * d[0]) * d[1]) *
d[2]) * d[3]) * d[4]) * d[5])
* d[6]) * d[7])
```

- 顺序依赖
  - 性能：取决于每个计算的延迟

如何增加指令级并行

```
void combine4(vec_ptr v, data_t* dest) {
    long i;
    long length = vec_length(v);
    data_t* d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

循环展开
Loop Unrolling

循环展开 - 2x1方式

```
void unroll2a_combine(vec_ptr v, data_t*
dest) {
    long length = vec_length(v);
    long limit = length - 1;
    data_t* d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2) {
        x = (x OP d[i]) OP d[i + 1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- 每个迭代执行两倍的计算

循环展开的结果

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

- 整数加法
  - 帮助整数达到延迟上限
- 其他的计算都没有优化
  - 仍然存在顺序依赖

```
x = (x OP d[i]) OP d[i+1];
```

integer addition
1. help integer reach upper limit of delay

other calculations are not optimized, order dependency

循环展开并重新结合 - 2x1a 方式

```
void unroll2aa_combine(vec_ptr v, data_t*
dest) {
    long length = vec_length(v);
    long limit = length - 1;
    data_t* d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2) {
        x = x OP (d[i] OP d[i + 1]);
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

和之前进行对比

```
x = (x OP d[i]) OP d[i+1];
```

- 这样会改善运算结果吗？

展开后的性能

latency vs throughput

latency is the time required to perform some actin or to produce some result, latency is measured in units of time -- hours, minutes, seconds, nanoseconds, or clock periods,

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

throughoput is the number of such actions executed or results produced per unit of time

2 load
1 store
4 integer add
2 FP multiply
1 FP add
1 FP divide

- 在 int *, FP +, FP * 上取得 2 倍的加速比
  - 加速原因: 打破了顺序依赖关系

```
x = x OP (d[i] OP d[i+1]);
```

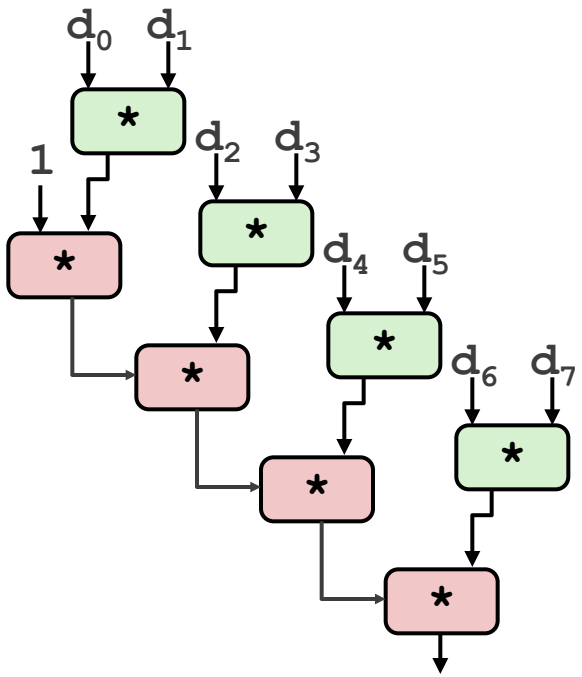
  - 为什么?

4 func. units for int +
2 func. units for load

2 func. units for FP *
2 func. units for load

重新结合律之后的计算流程

```
x = x OP (d[i] OP d[i+1]);
```



- 改变了什么？
  - 下一个循环迭代可以提前开始计算
  - 没有数据依赖
- 性能分析：
  - N 个元素, OP 的延迟为 D 个周期
  - 总共需要：(N/2+1)*D 个周期
  - **CPE = D/2**

循环展开并使用单独累加器 - 2x2 方式

```
void unroll2a_combine(vec_ptr v, data_t* dest) {
    long length = vec_length(v);
    long limit = length - 1;
    data_t* d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i + 1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- 与之前重新结合的方式不相同 `x = x OP (d[i] OP d[i+1]);`

分开累加的结果

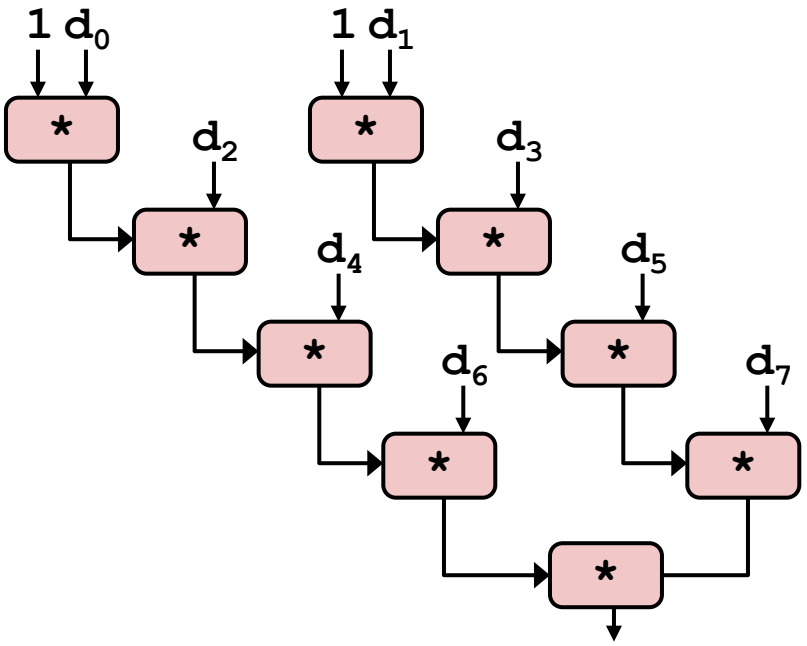
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- Int + 充分使用了两个 load 单元

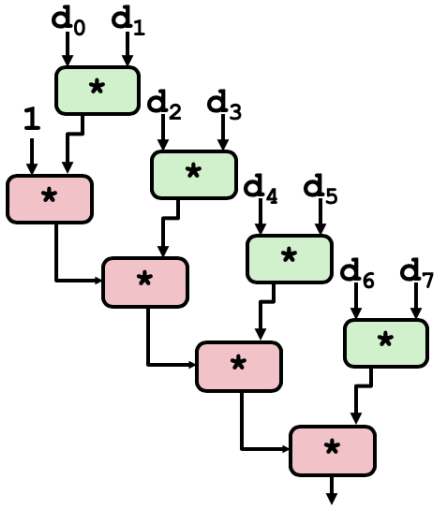
```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

分开累加的计算图

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- 改变了什么？
  - 两个独立的计算流
- 整体的性能
  - N 个元素, OP 的延迟为D个周期
  - 总共需要 (N/2+1)*D 个周期：
    CPE = D/2
  - 结果符合预期



循环展开 & 累加器

- 方法扩展：
  - 可以展开任意 L 层循环
  - 可以并行累加 K 个结果
  - **L 一定要是 K 的倍数**

- 性能限制
  - 收益递减
    - 不可能超越硬件计算单元的性能上限
  - 当元素个数很少时，开销比较大
    - 完成剩下迭代需要按顺序

循环展开 & 累加器: Double *

- 示例：
  - Intel Haswell
  - Double FP 乘法
  - 延迟限制：5，吞吐限制：0.5

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Accumulators

循环展开 & 累加器: Int +

- 例子
  - Intel Haswell
  - Integer 加法
  - 延迟限制：5，吞吐限制：0.5

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Accumulators

达到的性能

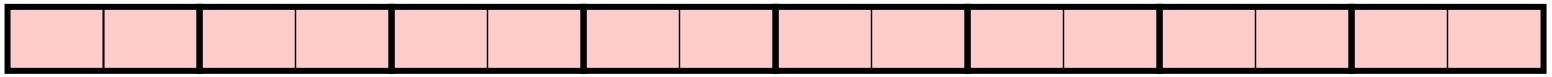
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- 基本达到硬件限制的上限
- 与未优化的代码相比，最多能有 42X 的加速比

使用 SIMD 加速

- AVX2：YMM寄存器 256 位
  - 16 个 YMM 寄存器，每个寄存器有 32 个字节
  - 32 个单字节的整型

  - 16个16位的整型



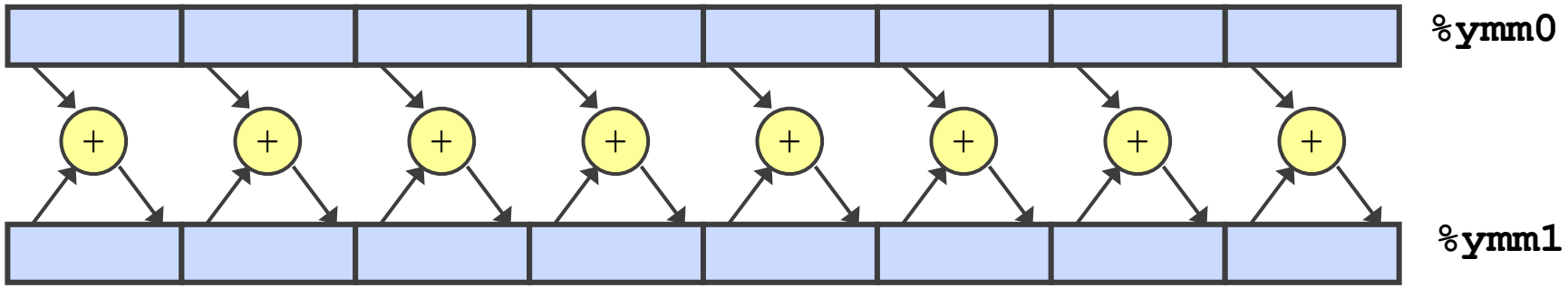
  - 8个32位的整型或单精度浮点数

  - 4个双精度浮点数

使用 SIMD 加速

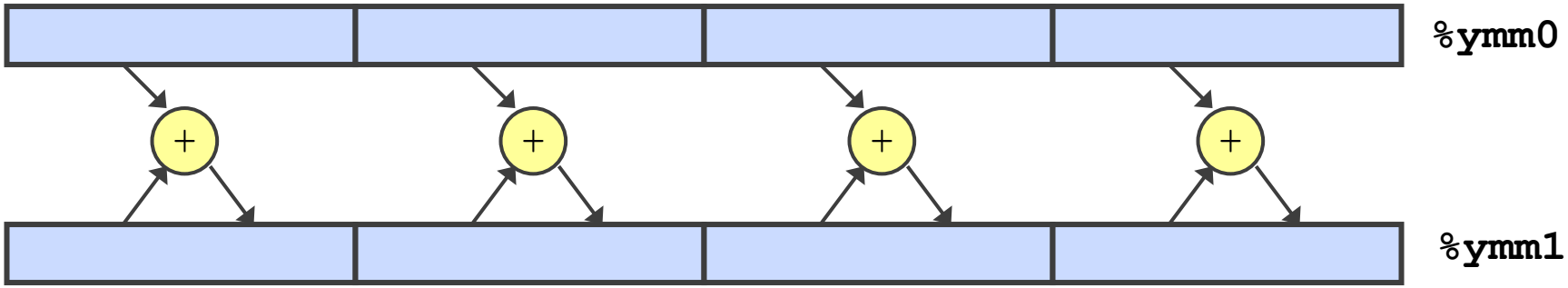
- 加法：单精度

```
vaddsd %ymm0, %ymm1, %ymm1
```



- 加法：双精度

```
vaddpd %ymm0, %ymm1, %ymm1
```



向量指令的优化效果

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	0.50	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50
Vec Throughput Bound	0.06	0.12	0.25	0.12

- 使用向量指令
  - 并行的计算多个元素
- 与未优化的代码相比，最多有 378X 的加速比

性能模型

为什么性能模型

- 理解性能
  - 不同硬件平台、编程模型、实现方式导致的性能差异
- 预测性能
  - 预测在未来系统上的性能
  - 对采购机器有合理的性能期望
  - 有助于软硬件协同设计，设计更符合当前应用的硬件架构
- 定位性能瓶颈
- 确定性能优化是否已经达到极限

性能模型：计算复杂度 Computational Complexity

- 认为程序运行时间与运算的次数相关
  - 比如浮点计算次数
- 运行时间可以表示为用户程序参数为自变量的函数

```
for(i=0;i<N;i++){
  Z[i] = alpha*X[i] + Y[i];
}
```

DAXPY: O(N) complexity where N is the number of elements

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

DGEMM: $O(N^3)$ complexity where N is the number of rows (equations)

FFTs: O(NlogN) in the number of elements

CG: $O(N^{1.33})$ in the number of elements (equations)

MG: O(N) in the number of elements (equations)

N-body: $O(N^2)$ in the number of particles (per time step)

性能模型：数据移动复杂度 Data Movement Complexity

- 认为程序运行时间与数据移动的次数有关
- 统计数据访问的总次数
- 困难在于需要对 Cache 的机制有深入了解
  - Cache 的大小限制
  - Cache 的替换策略

Operation	Flops	Data
DAXPY	$O(N)$	$O(N)$
DGEMV	$O(N^2)$	$O(N^2)$
DGEMM	$O(N^3)$	$O(N^2)$
FFTs	$O(NlogN)$	$O(N)$

性能影响因素

- 很多因素会对程序时间产生影响
- 有些与软件相关，有些与硬件相关，而有些与两者都相关
- 例如下面这些指标：

程序特征	硬件指标
#FP operations	FLOP/s
Cache data movement	Cache GB/s
DRAM data movement	DRAM GB/s
PCIe data movement	PCIe Bandwidth
Loop Depth	OMP Overhead
MPI Message Size	Network Bandwidth
#MPI Send	Network Latency

性能模型：Roofline 模型

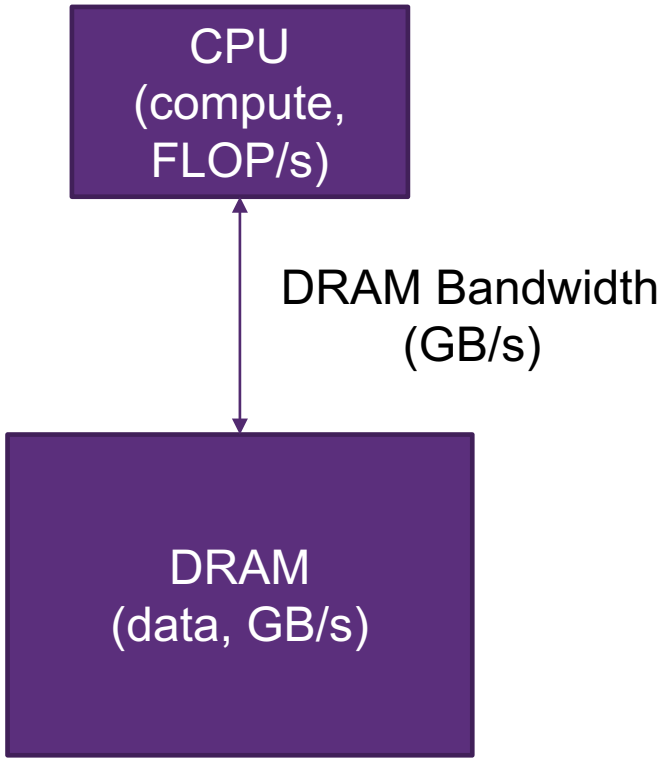
throughput performance model, visual performance model used to provide performance estimates. naive method gives upperbound to performance

- Roofline 是一个面向吞吐量的性能模型
- 适合分析应用的瓶颈和上限
- 与具体的硬件架构无关（适用于CPU，GPU，TPU等）
- 选用了部分性能指标来构建模型

程序特征	硬件指标
#FP operations	FLOP/s
Cache data movement	Cache GB/s
DRAM data movement	DRAM GB/s
PCIe data movement	PCIe Bandwidth
Loop Depth	OMP Overhead
MPI Message Size	Network Bandwidth
#MPI Send	Network Latency

性能模型： DRAM Roofline 模型

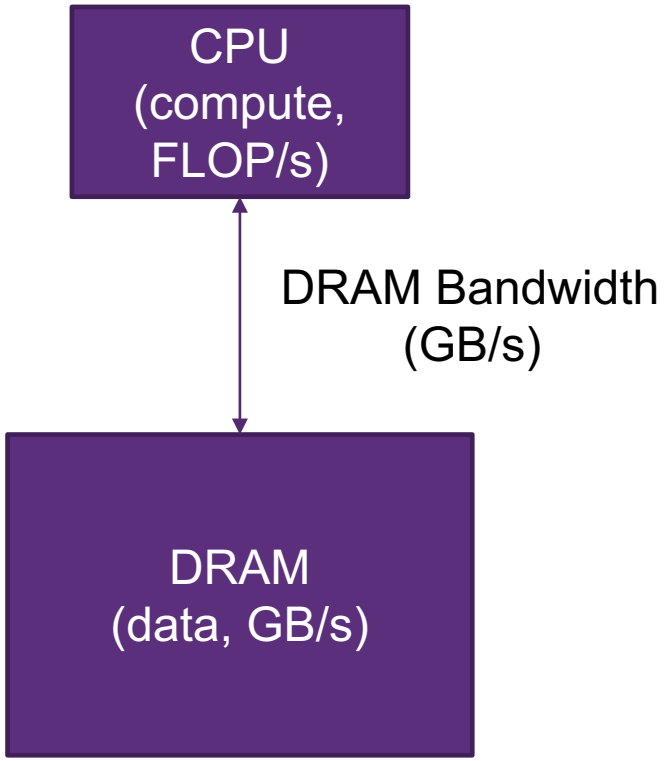
- 我们总希望能够达到最高的性能
- 然而，由于数据局部性和带宽的限制，影响了程序的性能上限
- 假设：
  - 理想情况下的CPU
  - 初始时数据都在DRAM上



$$Time = max \begin{cases} \dfrac{FP\ ops}{Peak\ GFLOP/s} \\ \dfrac{Data\ \text{move}\ size}{Peak\ GB/s} \end{cases}$$

性能模型： DRAM Roofline 模型

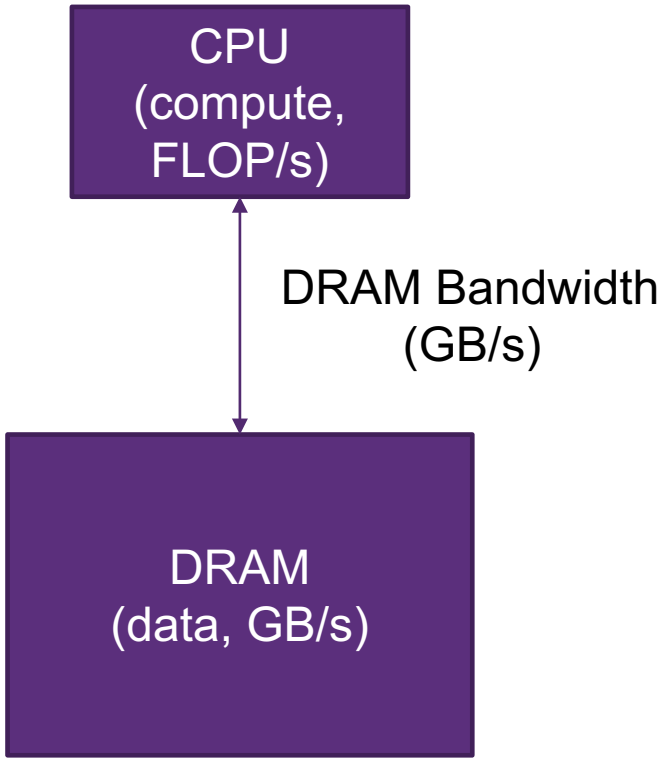
- 我们总希望能够达到最高的性能
- 然而，由于数据局部性和带宽的限制，影响了程序的性能上限
- 假设：
  - 理想情况下的CPU
  - 初始时数据都在DRAM上



$$\frac{Time}{FP\ ops} = max \begin{cases} \frac{1}{Peak\ GFLOP/s} \\ \frac{Data\ \text{move}\ size}{Peak\ GB/s\ *\ FP\ ops} \end{cases}$$

性能模型： DRAM Roofline 模型

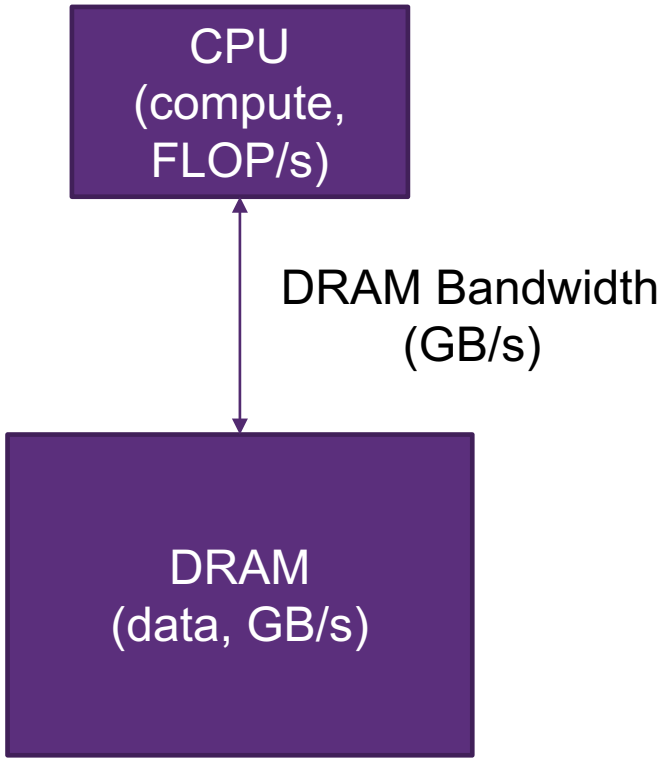
- 我们总希望能够达到最高的性能
- 然而，由于数据局部性和带宽的限制，影响了程序的性能上限
- 假设：
  - 理想情况下的CPU
  - 初始时数据都在 DRAM 上



$$\frac{FP\ ops}{Time} = min \begin{cases} Peak\ GFLOP/s \\ \dfrac{Peak\ GB/s\ *\ FP\ ops}{Data\ move\ size} \end{cases}$$

性能模型： DRAM Roofline 模型

- 我们总希望能够达到最高的性能
- 然而，由于数据局部性和带宽的限制，影响了程序的性能上限
- 假设：
  - 理想情况下的CPU
  - 初始时数据都在 DRAM 上

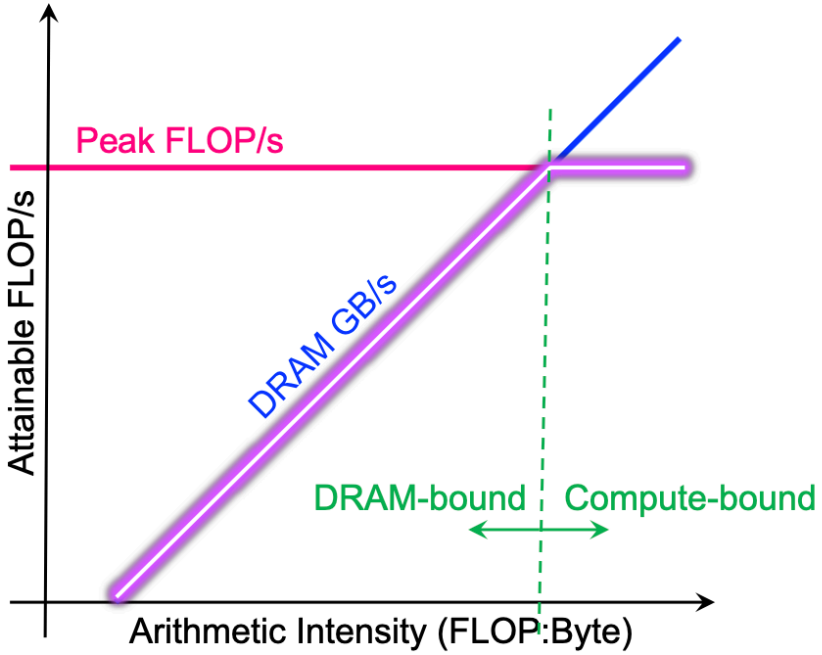


$$\frac{FP\ ops}{Time} = min \begin{cases} Peak\ GFLOP/s \\ Peak\ GB/s\ *\ \ AI \end{cases}$$

这里 AI (arithmetic intensity) 即计算密度，单位是 Flops / Bytes

性能模型：Roofline 模型

- Roofline 模型中最重要的概念就是计算密度
- 表示总共的计算次数和总共的数据移动规模的比例
- 以计算密度（AI）为 x 轴，画 Roofline 模型图
- log-log scale更适合作图，且能够外推到摩尔定律
- 当计算密度（AI）较小时，程序往往是 DRAM-bound 的



总结

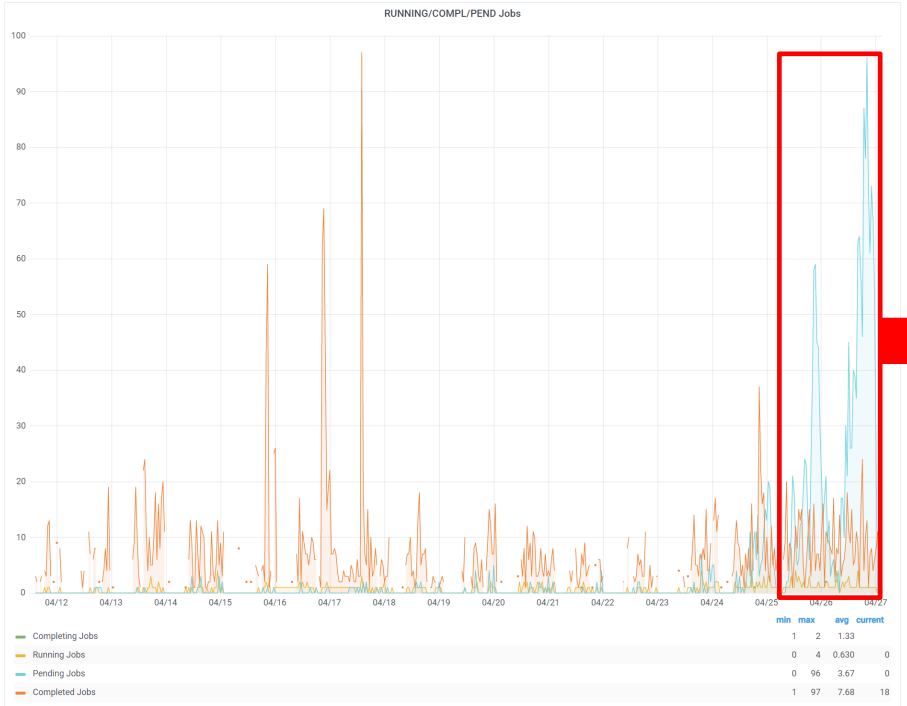
- 性能优化
- 循环优化
- 编译优化
- 性能模型
- 理解并使用体系结构相关性能优化

实验三：GPU 全源最短路

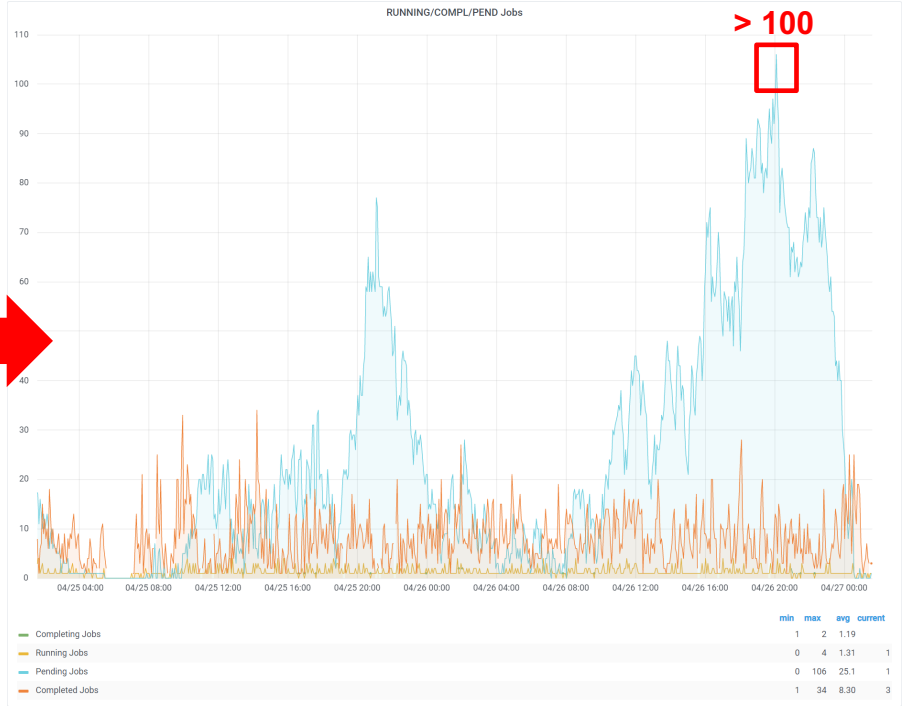
- 任务：GPU 加速的 Floyd-Warshall 算法
- 目标：
  - 熟悉 GPU（CUDA）编程
  - 熟悉针对 GPU 体系结构的优化和“分块”优化方法
- 详见实验指导书
  - https://lab.cs.tsinghua.edu.cn/hpc/doc/exp/3.apsp/
- 时间安排：
  - 五周时间（截止日期 5/31）

提醒：尽早开始作业！

- 集群计算能力有限，晚做**必然**导致作业累积
  - PA2 deadline 前，平均等待作业 > 50（最大 106），等待时间 > 20 分钟
  - 调度策略调整/资源限制均治标不治本
- 部分同学需要及时杀死卡死的进程，避免浪费机时



PA1 deadline 后



最后两天