

第五章 优化程序性能

教 师： 郑贵滨

听觉智能研究中心

哈尔滨工业大学，计算机科学与技术学院

要点

- 综述
- 普遍有用的优化方法
 - 代码移动/预先计算
 - 复杂运算简化Strength reduction
 - 共享公用子表达式
 - 去掉不必要的过程调用
- 妨碍优化的因素Optimization Blockers（优化障碍）
 - 过程调用
 - 存储器别名使用Memory aliasing（不同名字指向相同内存）
- 运用指令级并行
- 处理条件分支

性能的现实

- 性能比时间复杂度（asymptotic complexity，渐进时间复杂度/渐进复杂性）更重要
- 常数因子也很重要！
 - 代码编写不同，性能会差10倍！
 - 要在多个层次进行优化：
 - 算法、数据表示、过程、循环
- 优化性能一定要理解“系统”
 - 程序是怎样被编译和执行的
 - 现代处理器+存储系统是怎么运作的
 - 怎样测量程序性能、确定“瓶颈”
 - 如何在不破坏代码模块性和通用性的前提下提高性能

编译器优化

- 提供从程序到机器的有效映射
 - 寄存器分配
 - 代码选择与排序（调度）
 - 消除死代码
 - 消除轻微的低效率问题
- (通常)不提高渐进效率asymptotic efficiency
 - 由程序员来选择最佳的总体算法
 - 大O常常比常数因子更重要，但常数因子也需要考虑
- 难以克服的“优化障碍”
 - 潜在的内存别名使用memory aliasing(两个指针可能指向同一内存位置)
 - 潜在的函数副作用

编译器优化的局限性

- 在基本约束条件下运行
 - 不能引起程序行为的任何改变
 - 例外：可能由于程序使用了非标准的语言特性
 - 通常不去优化那些畸形情况下(pathological conditions)的程序行为
- 对程序员来说很明显的行为，可能会因语言和编码风格而变得模糊、混乱
 - 如：变量值的范围 \ll 变量类型对应的范围

编译器优化的局限性

- 大多数分析只在过程范围内进行
 - 在大多数情况下，全程序分析的代价会很高
 - 教新版本的GCC在单个文件中进行过程间的优化分析
 - 但是, 不做文件间的代码优化分析
- 大多数分析都是基于静态信息的
 - 编译器很难预测运行时的输入
- 当有疑问时，编译器必须是保守的

通常有用的优化

程序员或编译器应该做（不依赖处理器）的优化

■ 代码移动

- 减少计算执行的频率
 - 如果它总是产生相同的结果
 - 将代码从循环中移出

```
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];
```

编译器生成的代码移动 (-O1)

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

<pre>set_row: testq %rcx, %rcx jle .L1 imulq %rcx, %rdx leaq (%rdi,%rdx,8), %rdx movl \$0, %eax .L3: movsd (%rsi,%rax,8), %xmm0 movsd %xmm0, (%rdx,%rax,8) addq \$1, %rax cmpq %rcx, %rax jne .L3 .L1: rep ; ret</pre>	<pre># Test n # If 0, goto done # ni = n*i # rowp = a + ni*8 # j = 0 # loop: # t = b[j] # M[a+ni*8 + j*8] = t # j++ # j:n # if !=, goto loop # done:</pre>
--	--

复杂运算简化Reduction in Strength

- 用更简单的方法替换昂贵的操作

- 移位、加，替代乘法/除法

$16 * x \implies x \ll 4$

- 实际效果依赖于机器

- 取决于乘法或除法指令的成本

- Intel Nehalem CPU整数乘需要3个CPU周期

- 识别乘积的数值序列

```
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;  
}
```

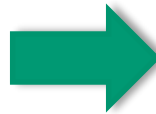
共享公用子表达式

- 重用表达式的一部分
- 使用 -O1时, GCC会做这个优化

```
/* 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;
```

3 乘法: $i*n$, $(i-1)*n$, $(i+1)*n$

```
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
```



```
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;
```

1 乘法: $i*n$

```
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
```

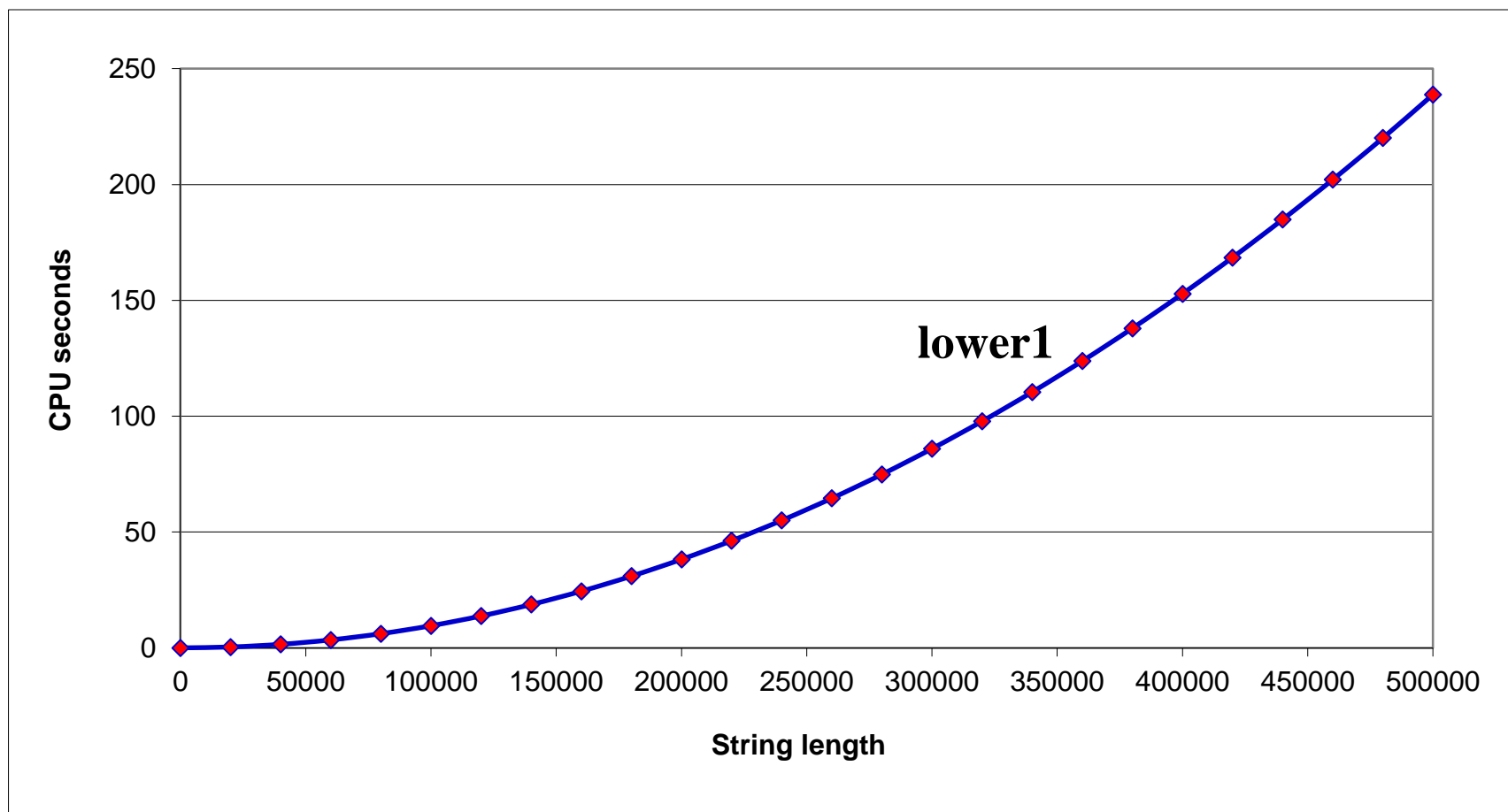
妨碍优化的因素/优化障碍#1: 函数调用

■ 将字符串转小写的函数

```
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');
}
```

小写转换性能

- 当字符串长度双倍时，时间增加了四倍
- 二次方（平方）的性能Quadratic performance



把循环变成 Goto形式

```
void lower(char *s){
    size_t i = 0;
    if (i >= strlen(s))
        goto done;
loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
done:
}
```

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

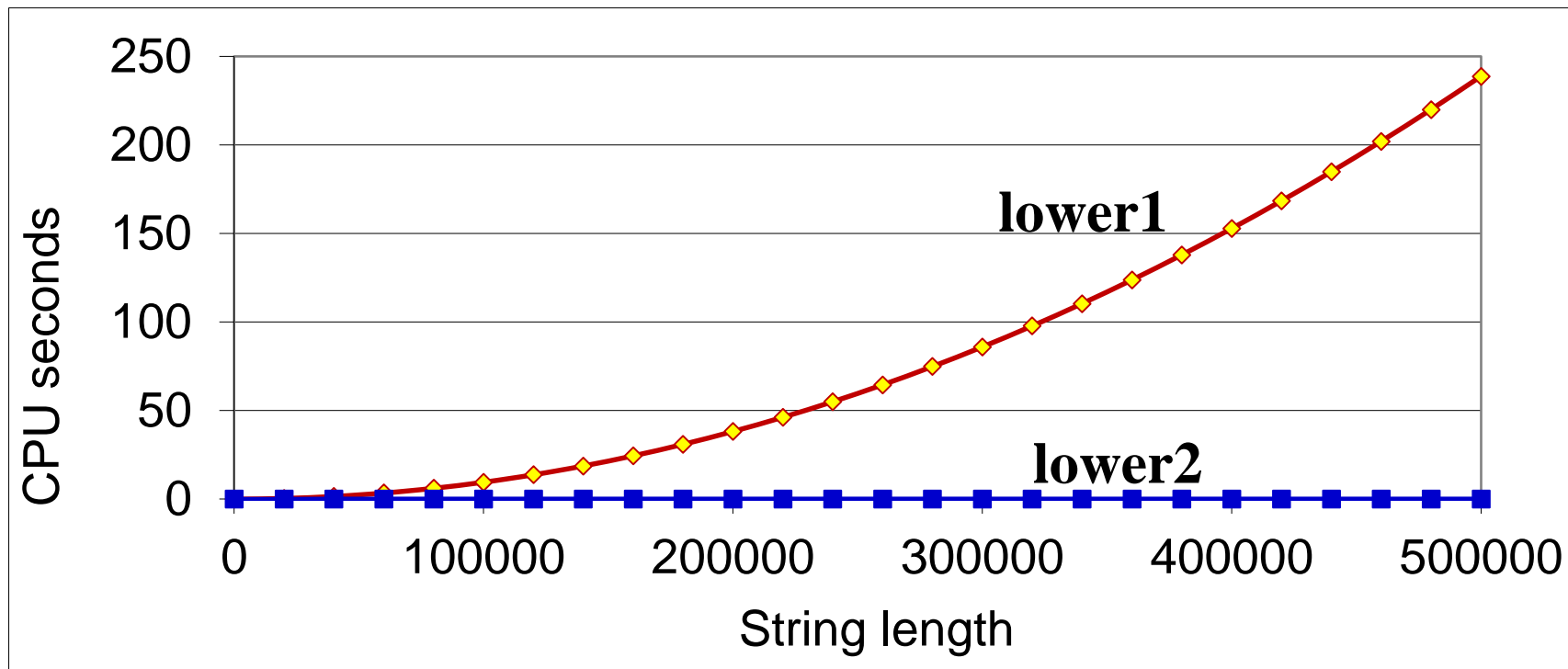
- **strlen**每次循环都要重复执行
- **strlen** 性能
 - 确定字符串长度的唯一方法是扫描它的整个长度，查找null字符
- **整体性能，长度为N的字符串**
 - N 次调用 strlen
 - 整体 $O(N^2)$ 性能

提高性能

```
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');
}
```

- 代码移动：把 `strlen` 调用移到循环外
- 根据：从一次迭代到另一次迭代时结果不会变化

Lower 小写转换的效率



- 字符串长度2倍时，时间也2倍
- `lower2` 的线性效率

妨碍优化的因素: 函数调用

■ 为什么编译器不能将`strlen`从内层循环中移出呢?

- 函数可能有副作用
 - 例如: 每次被调用都改变全局变量/状态
- 对于给定的参数, 函数可能返回不同的值
 - 依赖于全局状态/变量的其他部分
 - 函数`lower`可能与 `strlen` 相互作用

■ Warning:

- 编译器将函数调用视为黑盒
- 在函数附近进行弱优化

■ 补救措施:

- 使用 `inline` 内联函数
 - 用 `-O1` 时GCC这样做, 但局限于单个文件之内
- 自己做代码移动

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


妨碍优化的因素: 内存别名使用

■ 别名使用

- 两个不同的内存引用指向相同的位置
- C很容易发生
 - 因为允许做地址运算
 - 直接访问存储结构
- 养成使用局部变量的习惯
 - 在循环内部累积
 - 是一种告诉编译器不用检查内存别名使用的方法

内存的麻烦

```

/* 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
addsd  (%rdi), %xmm0        # FP add
movsd  %xmm0, (%rsi,%rax,8) # FP store
addq    $8, %rdi
cmpq    %rcx, %rdi
jne     .L4

```

- 代码每次循环都更新 **b[i]**
- 为什么编译器不能优化这个？

内存别名的(Memory Aliasing)使用

```
/* 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];
    }
}
```

- 代码每次循环都更新 **b[i]**
- 必须考虑：更新可能会影响程序的行为

```
double A[9] =
{ 0, 1, 2,
  4, 8, 16,
  32, 64, 128};
double B[3];
sum_rows1(A, A+3, 3);
```

Value of **b**:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, **22**, 16]

i = 2: [3, 22, 224]

移除 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;
    }
}

```

- 不需要存储中间结果

sum_rows2 inner loop

.L10:

```

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

```

表示程序性能

- CPE: 每个元素的周期数(Cycles Per Element)
- 表示向量或列表操作的程序性能的方便方式
- $\text{Length} = n$
- In our case: **CPE = cycles per OP**
- $T = \text{CPE} * n + \text{经常开销(Overhead)}$
 - CPE 是线的斜率slope

CPE

```
1 void psum1(float a[], float p[], long int n)
2 {
3     long int i;
4
5     p[0] = a[0] ;
6     for (i = 1; i < n; i++)
7         p[i] = p[i-1] + a[i];
8 }
9
```

CPE

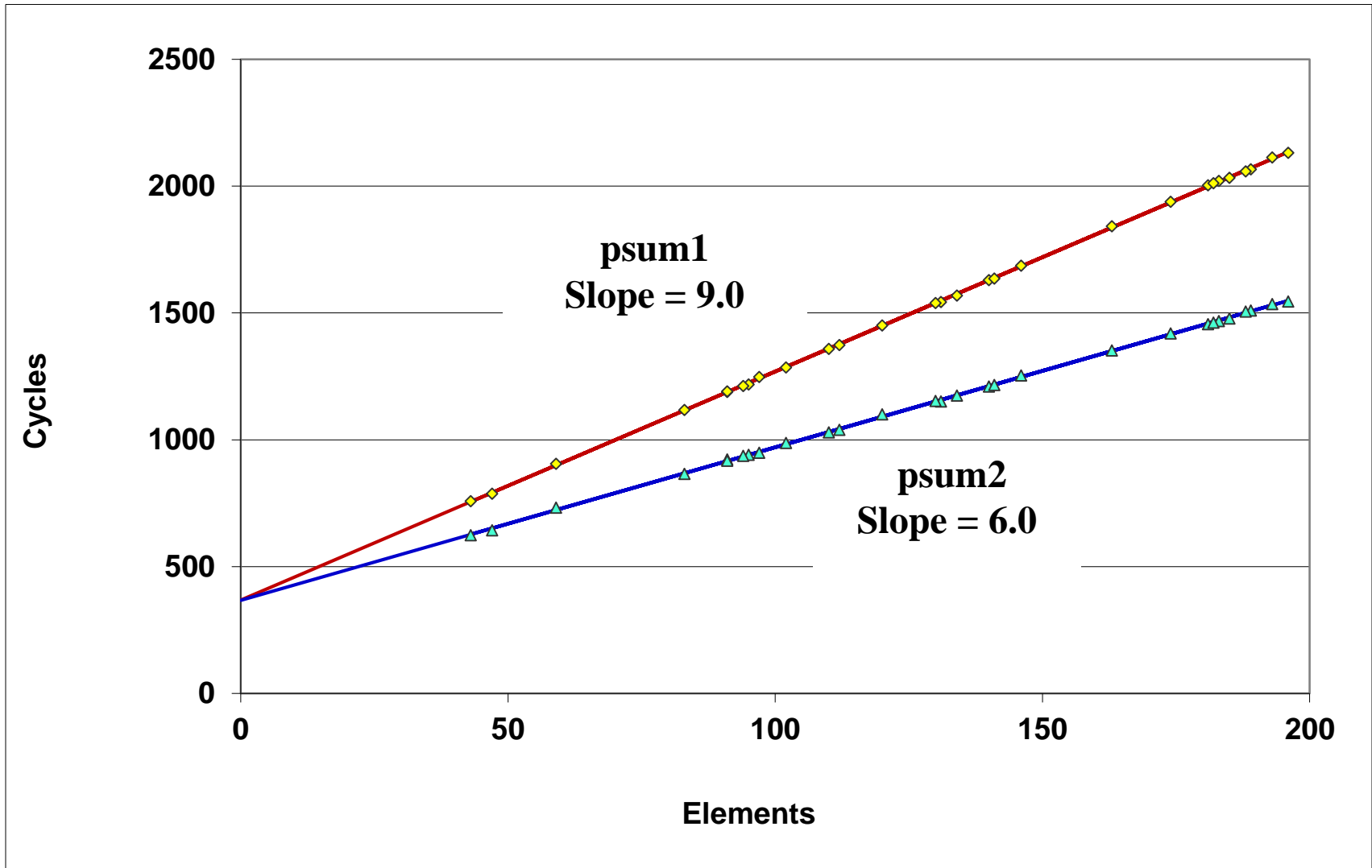
■ 降低循环开销

```

10 void psum2(float a[], float p[]; long int n)
11 {
12     long int i;
13     p[0] = a[0] ;
14     for (i = 1; i < n-1; i+=2) { //减少循环边界检查的次数
15         float mid_val = p[i-1] + a[i] ;
16         p[i] = mid_val ;
17         p[i+1] = mid_val + a[i+1];
18     }
19     /* For odd n, finish remaining element */
20     if ( i < n )
21         p[i] = p[i-1] + a[i] ;
22 }

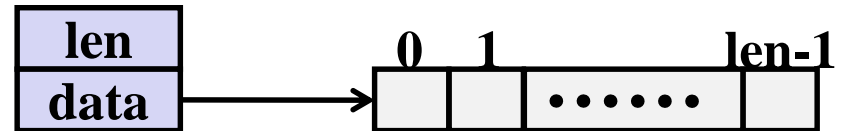
```

CPE



程序示例: 抽象数据类型——向量

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



■ 数据类型 **data_t** 使用的不同声明:

- **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;
}
```

基准程序的计算

```
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** 使用的不同声明：

- int
- long
- float
- double

■ 操作类型： **OP**

- 不同 OP, IDENT 使用不同的定义
- 0 for +
- 1 for *

基准程序的性能

计算向量元素的和/积

```
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;
    }
}
```

不同方法的CPE比较

方 法 \ 操作OP	Integer		Double FP	
	+	*	+	*
Combine1 未优化	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14

基本优化

- 减少函数调用：

把函数`vec_length`移到循环外——代码移动

- 用局部变量累积结果

- 避免每个循环的边界检查

尽量减少循环边界的检查

基本优化的效果

```
void combine4(vec_ptr v, data_t *dest)
{//增加一个局部累积量acc
  long i;
  long length = vec_length(v); //代码移动
  data_t *data = get_vec_start(v); //减少过程调用
  data_t acc = IDENT; //局部变量累积结果
  for (i = 0; i < length; i++)
    acc = acc OP data[i]; //消除不必要的内存引用
  *dest = acc;
}
```

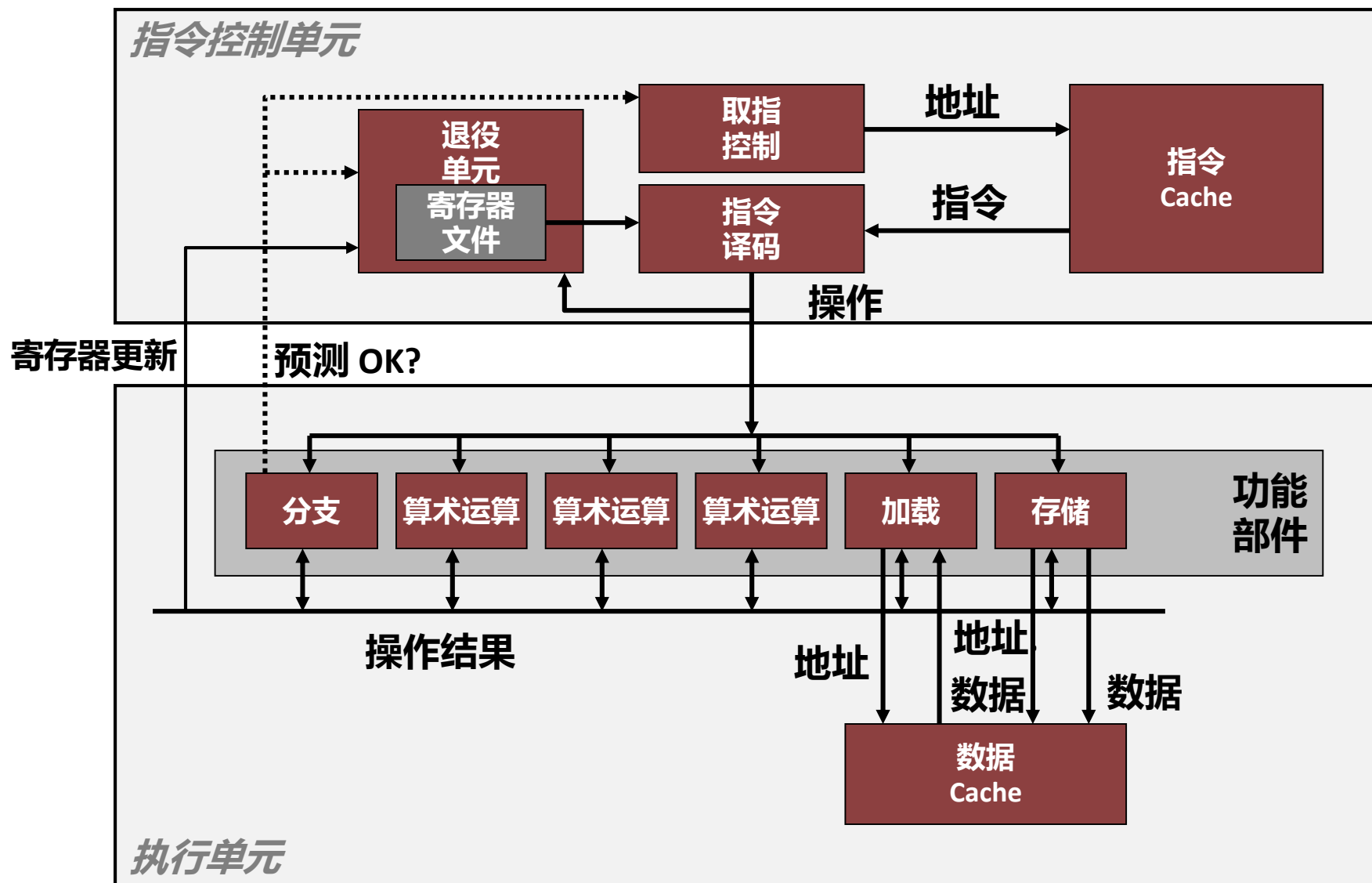
速度比较 (CPE)

方 法 \ 操作OP	Integer		Double FP	
	+	*	+	*
Combine1 -O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

高级优化：利用指令级并行

- 理解现代处理器的设计——流水线
 - 硬件可以并行执行多个指令
- 流水线性能障碍：
 - 数据依赖的限制
 - 控制冒险
- 简单的转换可以带来显著的性能改进
 - 编译器通常无法进行这些转换
 - 浮点运算缺乏结合性和可分配性

现代CPU设计

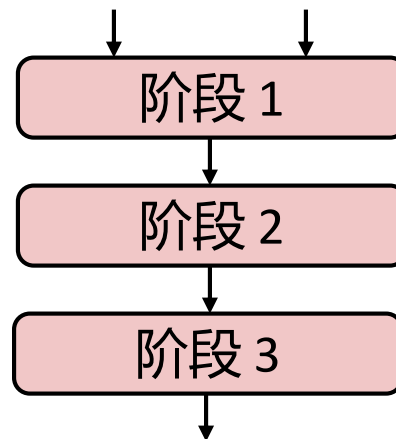


超标量Superscalar处理器

- **定义:** 一个时钟周期可发射、执行多条指令。这些指令是从一个连续的指令流获取、动态调度。
- **好处:** 不需要编程的努力, 超标量处理器 (CPU中有一条以上的流水线) 可以利用大多数程序都具有的**指令级并行性**。
- 大多数现代的cpu都是超标量
- Intel: 从Pentium (1993)起

流水线功能单元

```
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
阶段 1	a*b	a*c			p1*p2		
阶段 2		a*b	a*c			p1*p2	
阶段 3			a*b	a*c			p1*p2

- 把计算分解为多个阶段
- 在阶段间传递部分计算
- 一旦值传送给阶段 $i+1$, 阶段 i 就能开始新的计算,
- 例如, 即使每个乘法需要3个周期, 但在7个周期里可完成3个乘法

Haswell 架构的Intel CPU

- 8 个功能单元—P359
- 可并行执行多条指令

2 个加载，带地址计算

1 个存储，带地址计算

4 个整数运算

2 个浮点乘法运算

1 个浮点加法

1 个浮点除法

- 某些指令 > 1 周期，但能流水执行

延迟(latency): 表示完成运算所需要的总时间

发射时间(issue time): 连续两个同类型运算之间需要的最小间隔(时钟周期数)

容量(capacity): 该运算的功能单元数

指令

延迟Latency (周期)

发射(周期)

Load / Store

4

1

Integer 乘

3

1

Integer/Long 除

3-30

3-30

Single/Double FP 乘

5

1

Single/Double FP 加

3

1

Single/Double FP 除

3-15

3-15

Combine4的x86-64 编译结果

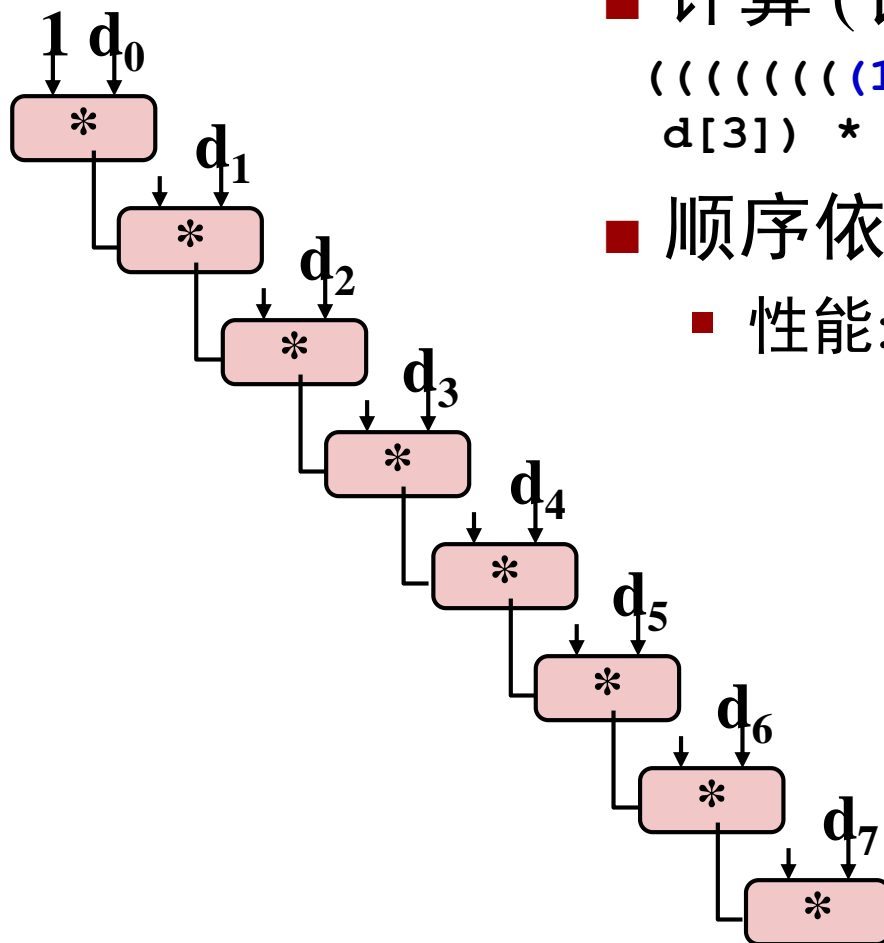
■ 内循环(做整数乘法)

.L519:	# Loop:
imull (%rax,%rdx,4), %ecx	# acc = acc OP data[i]; OP = *
addq \$1, %rdx	# i++
cmpq %rdx, %rbp	# Compare length:i
jg .L519	# If >, goto Loop

速度比较 (CPE)

方 法 \ 操作	Integer		Double FP	
	+	*	+	*
Combine4	1.27	3.01	3.01	5.01
延迟界限	1.00	3.00	3.00	5.00

Combine4 = 串行计算(操作OP = *)



■ 计算 (长度=8)

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

■ 顺序依赖性 Sequential dependence

- 性能: 由OP的延迟决定

教材P364的图5-15: 展示关键路径的简化版

循环展开

```

void combine5(vec_ptr v, data_t *dest)
{
  // unroll 2x1, (1个累积量), combine5
  long length = vec_length(v);
  long limit = length-1;
  data_t *d = get_vec_start(v);
  data_t acc = IDENT;
  long i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    acc = (acc OP d[i]) OP d[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    acc = acc OP d[i];
  }
  *dest = acc;
}

```

■ 每个循环完成2倍的有效工作

循环展开的汇编分析

■ combine5: unroll2x1 的内层循环

.L35:

```
vmulsd (%rax, %rdx, 8), %xmm0, %xmm0#acc*data[i]
vmulsd 8(%rax, %rdx, 8), %xmm0, %xmm0#acc*data[i+1]
addq $2, %rdx
cmpq %rdx, %rbp
jg .L35
```

combine4:

.L519:

```
imull (%rax,%rdx,4), %ecx
addq $1, %rdx
cmpq %rdx, %rbp
jg .L519
```

速度比较(CPE)

方法	Integer		Double FP	
操作	+	*	+	*
Combine4	1.27	3.01	3.01	5.01
Combine5 2x1循环展开 (1个累积量)	1.01	3.01	3.01	5.01
延迟界限	1.00	3.00	3.00	5.00

$$x = (x \text{ OP } d[i]) \text{ OP } d[i+1];$$

- 对整数 + 有帮助
 - 达到延迟界限
- 其他没有改善, *Why?*
 - 仍然是顺序依赖、已达到了延迟界限

循环展开+重组

```
void combine7 (vec_ptr v, data_t *dest)
{
    //unroll2x1a, 1个累积量
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t acc = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc = acc OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc = acc OP d[i];
    }
    *dest = acc;
}
```

combine5:

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

■ 这能改变运算速度？

■ 是的，尤其对 FP(浮点数), *Why?*

Combine程序中OP的极限——吞吐量界限

吞吐量界限(CPE)=发射/容量

	Integer		Double FP	
操作OP	+	*	+	*
延迟界限	1	3	3	5
发 射	1	1	1	1
容量/功能单元数	4	1	1	2
吞吐量界限(单位:CPE)	0.5	1	1	0.5
整个CPU有2个加载单元（读内存数据的单元）				

尽管有4个整数加法功能单元
但只有2个加载功能单元，运算
涉及读内存，所以容量按2算

2个浮点乘法功能单元
2个浮点加载功能单元

速度比较 (CPE)

方 法	Integer		Double FP	
操作OP	+	*	+	*
Combine4	1.27	3.01	3.01	5.01
Combine5 循环展开 2x1	1.01	3.01	3.01	5.01
Combine7 循环展开 2x1a	1.01	1.51	1.51	2.51
延迟界限	1.00	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50

■ 接近 2 倍的速度提升: **Int ***, **FP +**, **FP ***

■ 原因: 打破了顺序依赖

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

■ 为何是这样?

循环展开+重组

■ combine7的内层循环

loop:

`vmovsd (%rax, %rdx, 8), %xmm0`**#读内存**

`vmulsd 8(%rax, %rdx, 8), %xmm0, %xmm0`**#读内存、乘**

`vmulsd %xmm0, %xmm1, %xmm1`**#乘, 更新xmm1**

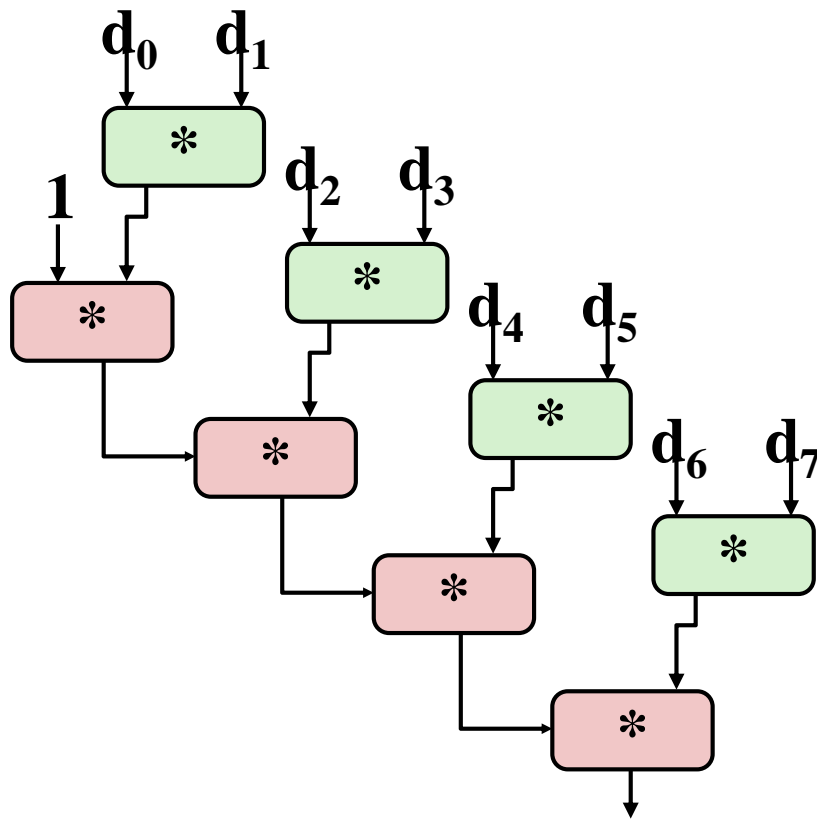
`addq $2, %rdx`

`cmpq %rdx, %rbp`

`jg loop`

重组的计算

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



- 什么改变了？
 - 下一个循环的部分操作可以早一些开始 (没有依赖的部分，第一个 mul 和相应加载)

- 整体性能
 - N 个元素, 每个操作 D 个周期延迟
 - $(N/2+1)*D$ cycles:
 $CPE = D/2$

循环展开：使用分离的累积量

```
void combine6 (vec_ptr v, data_t *dest)
```

```
{ //unroll2x2, 2个累积量
```

```
    long length = vec_length(v);
```

```
    long limit = length-1;
```

```
    data_t *d = get_vec_start(v);
```

```
    data_t acc0 = IDENT;
```

```
    data_t acc1 = IDENT;
```

```
    long i;
```

```
    /* Combine 2 elements at a time */
```

```
    for (i = 0; i < limit; i+=2) {
```

```
        acc0 = acc0 OP d[i];
```

```
        acc1 = acc1 OP d[i+1];
```

```
    }
```

```
    /* Finish any remaining elements */
```

```
    for (; i < length; i++) { acc0 = acc0 OP d[i]; }
```

```
    *dest = acc0 OP acc1;
```

```
}
```

■ 重组的不同形式

速度比较 (CPE)

方法	Integer		Double FP	
操作	+	*	+	*
Combine4	1.27	3.01	3.01	5.01
combine5: unroll 2x1	1.01	3.01	3.01	5.01
combine7: unroll 2x1a	1.01	1.51	1.51	2.51
combine6: unroll 2x2	0.81	1.51	1.51	2.51
延迟界限	1.00	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50

- 整数 加 + 使用 了两个加载单元

```
acc0 = acc0 OP d[i];
acc1 = acc1 OP d[i+1];
```

- 相比Unroll 2x1, 2倍速度提升: Int *, FP +、FP *

循环展开：使用分离的累积量

■ combine6: unroll2x2 的内层循环

.L35:

```
vmulsd (%rax, %rdx, 8), %xmm0, %xmm0
```

```
vmulsd 8(%rax, %rdx, 8), %xmm1, %xmm1
```

```
addq $2, %rdx
```

```
cmpq %rdx, %rbp
```

```
jg .L35
```

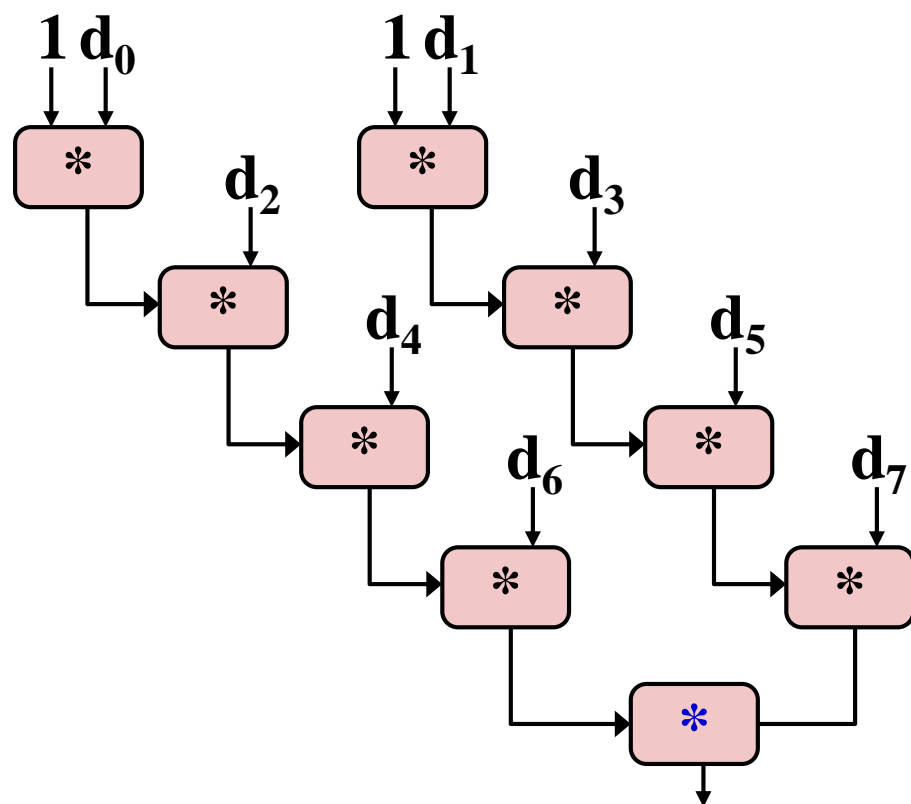
combine5:

```
vmulsd (%rax, %rdx, 8), %xmm0, %xmm0
```

```
vmulsd 8(%rax, %rdx, 8), %xmm0, %xmm0
```

分离的累积量

```
acc0 = acc0 OP d[i];
acc1 = acc1 OP d[i+1];
```



- 什么改变了:
 - 两个独立的操作的“流水”
- 整体性能
 - N 个元素, 每个操作 D 个周期延迟
 - 应为 $(N/2+1)*D$ cycles:
CPE = D/2
 - CPE 与预测匹配!

循环展开 & 累积量

■ 想法

- 循环展开到某一程度 K
- 并行累积 A 个
- K 是 A 的倍数
 - 灌满流水线

■ 限制

- 回报递减 Diminishing returns
 - 不能超出执行单元的吞吐量限制
- 长度小开销大 Large overhead for short lengths
 - 顺序地完成循环

循环展开 & 累积量: Double * 的最好性能

■ 案例 Intel Haswell

- 延迟界限: **5**, 容量: **2**, 吞吐量界限: 0.50
- **Load / Store** 延迟界限: 4
- 灌满流水线, 实现最高性能: $A \geq \text{延迟} * \text{容量}$ (10) 才好

累积量	FP *	循环展开因子 K							
	A	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

循环展开 & 累积量: Int + 的最好性能

■ 案例 Intel Haswell

- Integer 加法: 4 个功能单元 (容量4)、2 个加载单元
- Int+延迟界限: 1(应该和寻址方式有关), 吞吐量界限: 0.5
- Load / Store 延迟界限: 4

累积量	Int +	循环展开因子 K							
	A	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

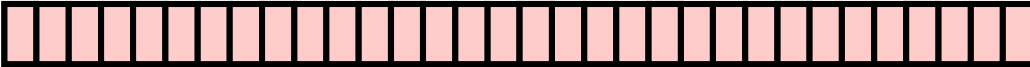






最好性能

- 受功能单元的吞吐量限制
- 比原始未优化的代码提高了42倍

<div>操作</div> <div>方法</div>	Integer		Double FP	
	+	*	+	*
Combine1 未优化	22.68	20.02	19.98	20.18
最好性能	0.54	1.01	1.01	0.52
延迟界限	1.00	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50

用 AVX2 编程

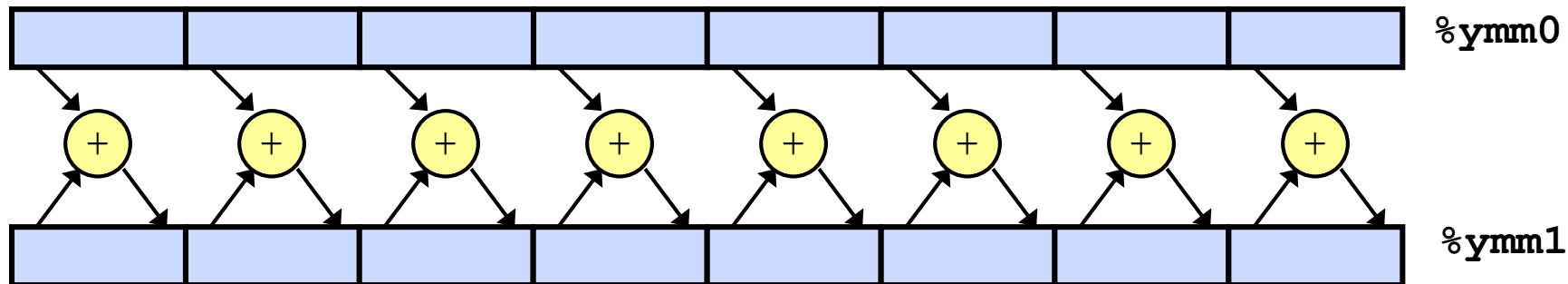
YMM 寄存器：16 个，每个32字节

- 32个单字节整数 
- 16个 16位整数 
- 8 个 32位整数 
- 8 个单精度浮点数 
- 4 个双精度浮点数 
- 1个单精度浮点数 
- 1 个双精度浮点数 

SIMD 操作

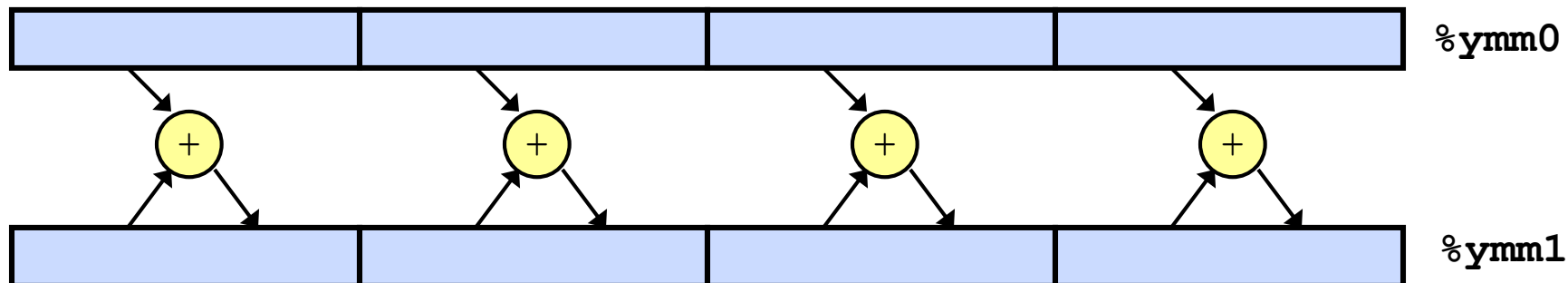
■ SIMD 操作: 单精度

`vaddsd %ymm0, %ymm1, %ymm1`



■ SIMD 操作: 双精度

`vaddpd %ymm0, %ymm1, %ymm1`



使用向量指令

方法 \ 操作	Integer		Double FP	
	+	*	+	*
标量 Best	0.54	1.01	1.01	0.52
向量 Best	0.06	0.24	0.25	0.16
延迟界限	0.50	3.00	3.00	5.00
吞吐量界限	0.50	1.00	1.00	0.50
向量 吞吐量界限	0.06	0.12	0.25	0.12

■ 使用AVX 指令

- 多数据元素的并行操作
- 看网络旁注 OPT:SIMD on CS:APP web 页面

分支怎么处理?

■ 挑战

- **指令控制单元**必须提前为**执行单元**生成足够的操作：使EU保持繁忙

```
404663: mov    $0x0,%eax
```

```
404668: cmp    (%rdi),%rsi
```

```
40466b: jge    404685 ←
```

```
40466d: mov    0x8(%rdi),%rax
```

```
...
```

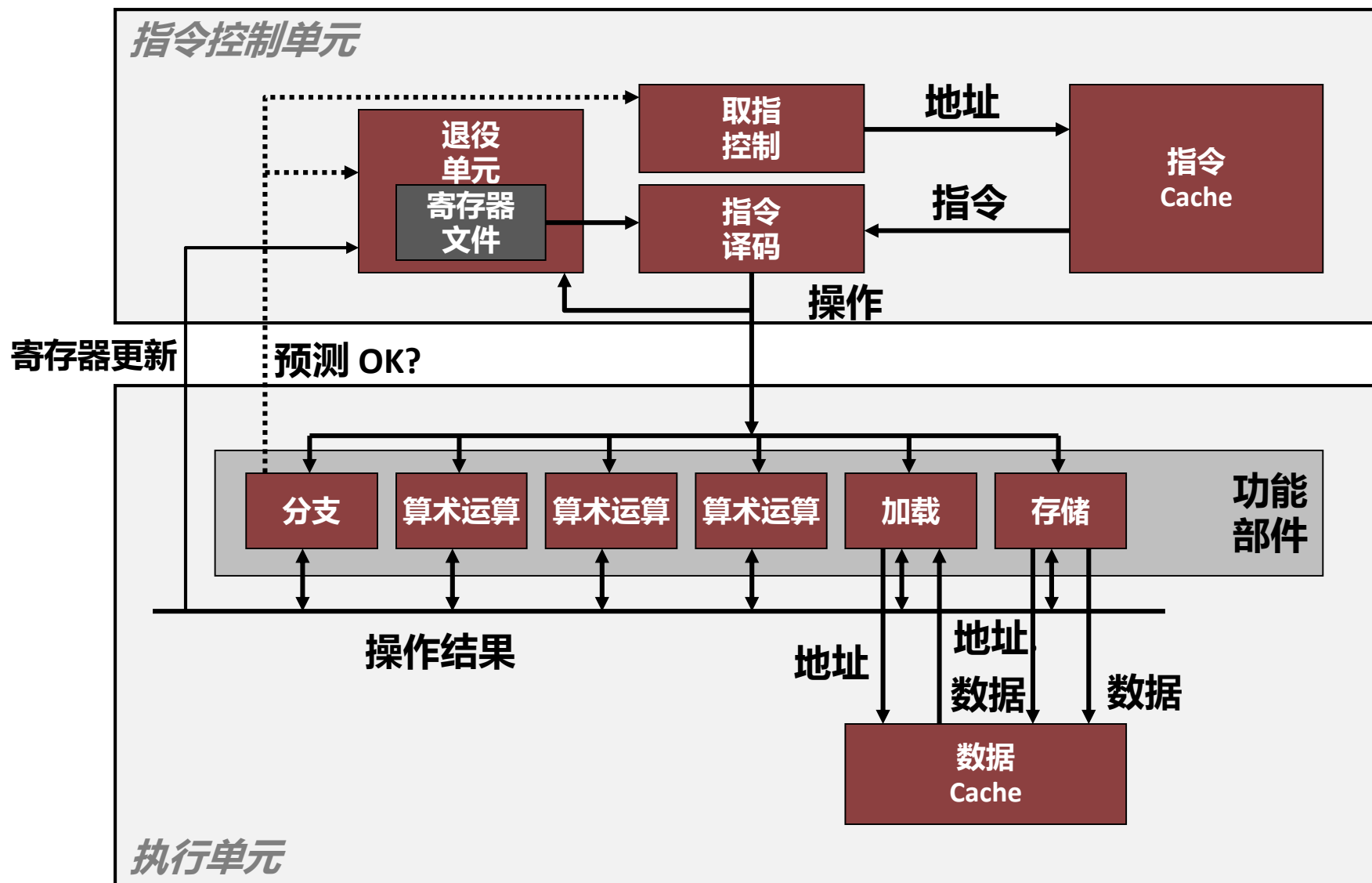
```
404685: repz retq
```

} Executing

How to continue?

- 遇到条件分支时，无法可靠地确定继续取指的位置

现代CPU设计



分支的结果

- 当遇到条件分支时，无法确定继续取指的位置
 - 选择分支:将控制转移到分支目标
 - 不选择分支:继续下一个指令
- 直到分支/整数单元(Y86的E步)结果出来后才能确定

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax
. . .
404685:  repz   retq
```

不选择分支

选择分支

分支预测

■ 设想

- 猜测会走哪个分支
- 在预测的位置开始执行指令
 - 但不要真修改寄存器或内存数据

```
404663: mov    $0x0,%eax
404668: cmp    (%rdi),%rsi
40466b: jge     404685
40466d: mov    0x8(%rdi),%rax

...

404685: repz retq
```

预测选择分支

} 开始执行

穿过循环的分支预测

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

*i = 98***假定****向量长度 = 100****预测选择分支(OK)**

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

*i = 99***预测选择分支(Oops)**

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

*i = 100***读无效位置****执行****取指**

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 101

分支预测错误的失效/撤销

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 98

假定

向量长度 = 100

预测选择分支(OK)

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 99

预测选择分支(Oops)

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 100

无效

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 101

分支错误预测的恢复

```
401029: vmulsd (%rdx),%xmm0,%xmm0
40102d: add  $0x8,%rdx
401031: cmp  %rax,%rdx
401034: jne  401029
401036: jmp  401040
...
401040: vmovsd %xmm0,(%r12)
```

i = 99

确定不会选择分支

重新加载流水线

■ 性能开销/代价

- 现代处理器上的多个时钟周期
- 可能是一个主要的性能限制因素

获得高性能

- 用好的编译器、合适的编译选项
- 别做傻事
 - 当心隐藏的算法效率低下
 - 编写对编译器友好的代码
 - 小心妨碍优化的因素:
函数调用 & 内存引用
 - 重点、仔细观察最内层循环 (多数工作在那儿完成)
- 为机器优化代码
 - 利用指令级并行
 - 尽量避免不可预测的分支
 - 使代码能较好地缓存 (在后续的课程介绍)