

第五章知识点

1.一般有用的优化方法

(1) 代码移动和预先计算：例如将循环中每次都要计算的相同变量移动到循环外面，减少计算次数。GCC 优化等级 -O1 时，会执行此优化。

(2) 复杂运算简化：用简单的方法替换昂贵的操作，例如用加法、移位替代乘法和除法。

(3) 公用子表达式共享：当有些表达式中有公共的计算式时，可以将这个计算式先计算出来，不用每个表达式都计算一遍。

(4) 去掉不必要的过程调用（函数调用）：例如 PPT 中举的向量累加的例子，通过每次循环修改指向下一个元素的指针取代 `get_vec_element()` 函数的调用。

2.阻碍优化的因素（由于有这些因素，编译器只会做一些比较保守的优化）

(1) 为什么函数调用会阻碍优化：

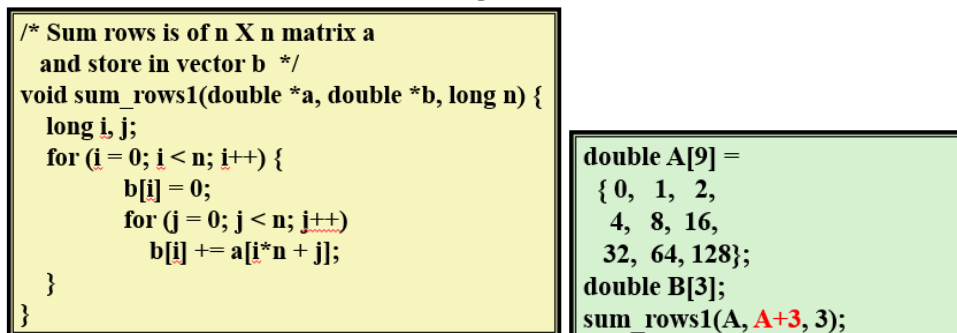
- 函数可能有副作用
 - 例如：每次被调用都改变全局变量/状态
- 对于给定的参数，函数可能返回不同的值
 - 依赖于全局状态/变量的其他部分
 - 函数 `lower` 可能与 `strlen` 相互作用

图 2-1 函数调用阻碍优化的原因

第一点是因为可能该函数每次调用会改变全局变量，编译器无法将处于循环中的函数调用移到函数外。

第二点很好理解，因为就算每次输入参数是相同的，可能返回值因为函数调用了全局变量或者与其他函数相互作用而不同。

(2) 内存别名的使用（即有两个或多个指针指向同一内存空间）：



```
/* 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[3];
sum_rows1(A, A+3, 3);
```

图 2-2 内存别名使用阻碍优化的例子

对于 PPT 中的例子，传入的两个指针 `a` (`A`)，`b` (`A+3`) 在遍历过程会访问同一段内存区域，编译器无法判断是否这是程序员的本意，因此无法用局部变量代替全局变量进行累加操作。

3.循环展开的知识总结

<pre>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; } }</pre>	<pre>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; //局部变量累计结果 for (i = 0; i < length; i++) t = t OP d[i]; //消除不必要的内存引用 *dest = t; }</pre>
--	--

图 2-3 combine1 和 combine4 函数

```
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;
}
```

图 2-4 循环展开 2x1

```
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;
}
```

图 2-5 循环展开 2x1a

```

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;
}

```

图 2-6 循环展开 2x2

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

图 2-7 循环展开各版本函数 CPE 值比较

(1) Combine1 -O1->Combine4，体现了编译器优化的局限性。

(2) 为什么 Combine4->Unroll2x1 只有整数+有提升，而其他三种操作没有提升？

因为对于整数+来说，每次循环累加两个元素，这样循环次数相比 Combine4 减少了，相应的减少了一些开销，例如减少了跳转的开销。而其他三种操作已经到达了延迟界限，减少的这些开销可能是 CPE 值只减少了 0.00 几，即有更为重要的因素影响了 CPE 值。

(3) 为什么 Unroll2x1-Unroll2x1a 其他操作都有提升，而只有整数+没有提升？（书上没解释，仅供参考）

因为加法运算只需要一个时钟周期，而 load 操作需要四个时钟周期（但是有两个加载单元，因此每四个时钟周期可以完成两个 load 指令，那为什么整数+的 CPE 是 1.01 而不是 2 呢，因为运算单元中的流水线使得每过一个周期可以发射一个 load 指令，但是该流水线并不是完全流水线，所以发射效率无法达到 0.5，随着循环展开因子的增加流水线会逐渐成为完全流水线）。因此，影响整数+的是 load 操作，而不是加法运算，所以关键路径是包含了 load 操作的路径。而从 Unroll2x1 到 Unroll2x1a 关键路径上的 load 操作数其实是不变的，因此 CPE 没变化。

(4) 从 Unroll2x1-Unroll2x2, 可以看到, 整数乘法和浮点数加法、乘法速度都提升了一倍, 加法也有所提升 (没到一倍), 因为消除了两条乘法语句之间的数据相关, 提升了流水线的效率 (之前由于已存在数据相关, 流水线无法启动)。加法没有提升一倍, 是因为还有其他因素限制 (循环开销不能忽视)。

(5) 理解延迟界限和吞吐量

运算	整数			浮点数		
	延迟	发射	容量	延迟	发射	容量
加法	1	1	4	3	1	1
乘法	3	1	1	5	1	2
除法	3 ~ 30	3 ~ 30	1	3 ~ 15	3 ~ 15	1

图 2-8 Haswell 架构 CPU 中整数、浮点数的加、乘、除操作的一些参数

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

图 2-9 Haswell 架构 CPU 中整数、浮点数的加、乘、除操作的延迟界限和吞吐量界限

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

图 2-10 Haswell 架构 CPU 中各种算术单元

在 PPT 中是以 Haswell 架构 CPU 为例子的, 要了解延迟界限和吞吐量, 首先要知道上图中的延迟、发射、容量。延迟: 执行该运算所需的时钟周期数, 发射: 两个相同运算之间间隔的时钟周期数, 容量: 每次能同时发射多少个这样的操作, 可以看到整数乘法延迟为 3, 可是却可以每过一个时钟周期就可以完成一条整数乘法 (发射为 1), 是因为利用了流水线 (pipeline), 达到了完全流水化。

延迟界限的单位是 CPE (每增加一个元素所增加的时钟周期数), 定义为严格按照顺序完成合并运算的函数所需的最小 CPE (CPE 是越小越好的), 其实在数值上就是等于该运算的延迟的。再来看吞吐量界限, 定义为根据功能单元产生结果的最大速率, 就是当所有参与运算的功能单元都完全利用起来后达到的最小 CPE 值, 因此吞吐量界限和发射成正比, 和容量成反比, 即 $\text{吞吐量界限} = \text{发射} / \text{容量}$ (由于整数加法只有两个加载单元, 因此整数加法的吞吐量界限 $= 1/2 = 0.5$, 而不是 $1/4$)。

(6) 关于 PPT 中的 AVX 指令集可以去下面的网址了解一下 (考试不作要求):

<https://post.smzdm.com/p/ax028v09/>