


正如前面讲到过的，combine3 将它的结果累积在目标位置中，在本例中，目标位置就是向量的最后一个元素。因此，这个值首先被设置为 1，然后设为 $2 \cdot 1 = 2$ ，然后设为 $3 \cdot 2 = 6$ 。最后一次迭代中，这个值会乘以它自己，得到最后结果 36。对于 combine4 的情况来说，直到最后向量都保持不变，结束之前，最后一个元素会被设置为计算出来的值 $1 \cdot 2 \cdot 3 \cdot 5 = 30$ 。

当然，我们说明 combine3 和 combine4 之间差别的例子是人为设计的。有人会说 combine4 的行为更加符合函数描述的意图。不幸的是，编译器不能判断函数会在什么情况下被调用，以及程序员的本意可能是什么。取而代之，在编译 combine3 时，保守的方法是不断地读和写内存，即使这样做效率不太高。

 **练习题 5.4** 当用带命令行选项“-O2”的 GCC 来编译 combine3 时，得到的代码 CPE 性能远好于使用-O1 时的：

函数	方法	整数		浮点数	
		+	*	+	*
combine3	用-O1 编译	7.17	9.02	9.02	11.03
combine3	用-O2 编译	1.60	3.01	3.01	5.01
combine4	累积在临时变量中	1.27	3.01	3.01	5.01

由此得到的性能与 combine4 相当，不过对于整数求和的情况除外，虽然性能已经得到了显著的提高，但还是低于 combine4。在检查编译器产生的汇编代码时，我们发现对内循环的一个有趣的变化：

Inner loop of combine3. data_t = double, OP = *. Compiled -O2
dest in %rbx, data+i in %rdx, data+length in %rax
Accumulated product in %xmm0

```

1  .L22:                                loop:
2      vmulsd (%rdx), %xmm0, %xmm0      Multiply product by data[i]
3      addq   $8, %rdx                  Increment data+i
4      cmpq   %rax, %rdx                Compare to data+length
5      vmovsd %xmm0, (%rbx)             Store product at dest
6      jne    .L22                      If !=, goto loop

```

把上面的代码与用优化等级 1 产生的代码进行比较：

Inner loop of combine3. data_t = double, OP = *. Compiled -O1
dest in %rbx, data+i in %rdx, data+length in %rax

```

1  .L17:                                loop:
2      vmovsd (%rbx), %xmm0             Read product from dest
3      vmulsd (%rdx), %xmm0, %xmm0      Multiply product by data[i]
4      vmovsd %xmm0, (%rbx)             Store product at dest

5      addq   $8, %rdx                  Increment data+i
6      cmpq   %rax, %rdx                Compare to data+length
7      jne    .L17                      If !=, goto loop

```

我们看到，除了指令顺序有些不同，唯一的区别就是使用更优化的版本不含有 vmovsd 指令，它实现的是从 dest 指定的位置读取数据（第 2 行）。

- 寄存器 %xmm0 的角色在两个循环中有什么不同？
- 这个更优化的版本忠实地实现了 combine3 的 C 语言代码吗（包括在 dest 和向量数据之间使用内存别名的时候）？