当平滑，不会出什么问题。即使有不连续的时候，它们通常也不会导致前面描述的条件那样的周期性模式。按照严格顺序对元素求积的准确性不太可能从根本上比“分成两组独立求积，然后再将这两个积相乘”更好。对大多数应用程序来说，使性能翻倍要比冒对奇怪的数据模式产生不同的结果的风险更重要。但是，程序开发人员应该与潜在的用户协商，看看是否有特殊的条件，可能会导致修改后的算法不能接受。大多数编译器并不会尝试对浮点数代码进行这种变换，因为它们没有办法判断引入这种会改变程序行为的转换所带来的风险，不论这种改变是多么小。

5.9.2　重新结合变换

现在来探讨另一种打破顺序相关从而使性能提高到延迟界限之外的方法。我们看到过做 $k\times1$ 循环展开的 combine5 没有改变合并向量元素形成和或者乘积中执行的操作。不过，对代码做很小的改动，我们可以从根本上改变合并执行的方式，也极大地提高程序的性能。

图 5-26 给出了一个函数 combine7，它与 combine5 的展开代码(图 5-16)的唯一区别在于内循环中元素合并的方式。在 combine5 中，合并是以下面这条语句来实现的

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

而在 combine7 中，合并是以这条语句来实现的

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

差别仅在于两个括号是如何放置的。我们称之为*重新结合变换*(reassociation transformation)，因为括号改变了向量元素与累积值 acc 的合并顺序，产生了我们称为“$2\times1a$”的循环展开形式。

```
1   /* 2 x 1a loop unrolling */
2   void combine7(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6       long limit = length-1;
7       data_t *data = get_vec_start(v);
8       data_t acc = IDENT;
9
10      /* Combine 2 elements at a time */
11      for (i = 0; i < limit; i+=2) {
12          acc = acc OP (data[i] OP data[i+1]);
13      }
14
15      /* Finish any remaining elements */
16      for (; i < length; i++) {
17          acc = acc OP data[i];
18      }
19      *dest = acc;
20  }
```

图 5-26　运用 $2\times1a$ 循环展开，重新结合合并操作。这种方法增加了可以并行执行的操作数量

对于未经训练的人来说，这两个语句可能看上去本质上是一样的，但是当我们测量