```
4       addq    $2, %rdx                Increment i by 2
5       cmpq    %rdx, %rbp              Compare to limit:i
6       jg      .L35                    If >, goto loop
```

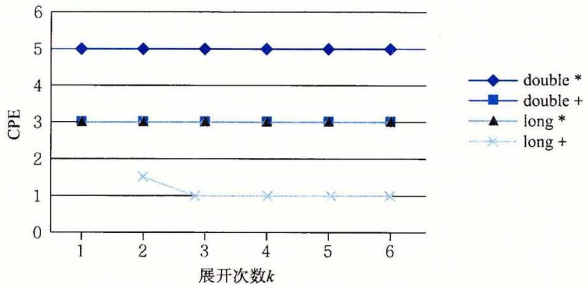

图 5-17　不同程度 $k\times1$ 循环展开的 CPE 性能。这种变换只改进了整数加法的性能

我们可以看到，相比 combine4 生成的基于指针的代码，GCC 使用了 C 代码中数组引用的更加直接的转换⊖。循环索引 i 在寄存器%rdx 中，data 的地址在寄存器%rax 中，和前面一样，累积值 acc 在向量寄存器%xmm0 中。循环展开会导致两条 vmulsd 指令——一条将 data[i]加到 acc 上，第二条将 data[i+ 1]加到 acc 上。图 5-18 给出了这段代码的图形化表示。每条 vmulsd 指令被翻译成两个操作：一个操作是从内存中加载一个数组元素，另一个是把这个值乘以已有的累积值。这里我们看到，循环的每次执行中，对寄存器%xmm0 读和写两次。可以重新排列、简化和抽象这张图，按照图 5-19a 所示的过程得到图 5-19b 所示的模板。然后，把这个模板复制 $n/2$ 次，给出一个长度为 n 的向量的计算，得到如图 5-20 所示的数据流表示。在此我们看到，这张图中关键路径还是 n 个 mul 操作——迭代次数减半了，但是每次迭代中还是有两个顺序的乘法操作。这个关键路径是循环没有展开代码的性能制约因素，而它仍然是 $k\times1$ 循环展开代码的性能制约因素。

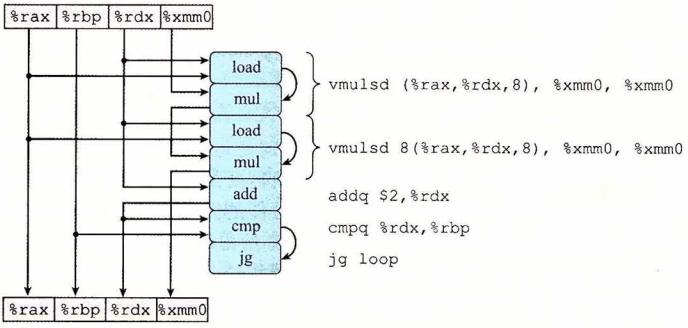

图 5-18　combine5 内循环代码的图形化表示。每次迭代有两条 vmulsd 指令，每条指令被翻译成一个 load 和一个 mul 操作

⊖ GCC 优化器产生一个函数的多个版本，并从中选择它预测会获得最佳性能和最小代码量的那一个。其结果就是，源代码中微小的变化就会生成各种不同形式的机器码。我们已经发现对基于指针和基于数组的代码的选择不会影响在参考机上运行的程序的性能。