5.9 Enhancing Parallelism

At this point, our functions have hit the bounds imposed by the latencies of the arithmetic units. As we have noted, however, the functional units performing addition and multiplication are all fully pipelined, meaning that they can start new operations every clock cycle, and some of the operations can be performed by multiple functional units. The hardware has the potential to perform multiplications and additions at a much higher rate, but our code cannot take advantage of this capability, even with loop unrolling, since we are accumulating the value as a single variable `acc`. We cannot compute a new value for `acc` until the preceding computation has completed. Even though the functional unit computing a new value for `acc` can start a new operation every clock cycle, it will only start one every L cycles, where L is the latency of the combining operation. We will now investigate ways to break this sequential dependency and get performance better than the latency bound.

5.9.1 Multiple Accumulators

For a combining operation that is associative and commutative, such as integer addition or multiplication, we can improve performance by splitting the set of combining operations into two or more parts and combining the results at the end. For example, let P_n denote the product of elements $a_0, a_1, \ldots, a_{n-1}$:

$$P_n = \prod_{i=0}^{n-1} a_i$$

Assuming n is even, we can also write this as $P_n = PE_n \times PO_n$, where PE_n is the product of the elements with even indices, and PO_n is the product of the elements with odd indices:

$$PE_n = \prod_{i=0}^{n/2-1} a_{2i}$$

$$PO_n = \prod_{i=0}^{n/2-1} a_{2i+1}$$

Figure 5.21 shows code that uses this method. It uses both two-way loop unrolling, to combine more elements per iteration, and two-way parallelism, accumulating elements with even indices in variable `acc0` and elements with odd indices in variable `acc1`. We therefore refer to this as "2 × 2 loop unrolling." As before, we include a second loop to accumulate any remaining array elements for the case where the vector length is not a multiple of 2. We then apply the combining operation to `acc0` and `acc1` to compute the final result.

Comparing loop unrolling alone to loop unrolling with two-way parallelism, we obtain the following performance:

```
1   /* 2 x 2 loop unrolling */
2   void combine6(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 acc0 = IDENT;
9       data_t acc1 = IDENT;
10
11      /* Combine 2 elements at a time */
12      for (i = 0; i < limit; i+=2) {
13          acc0 = acc0 OP data[i];
14          acc1 = acc1 OP data[i+1];
15      }
16
17      /* Finish any remaining elements */
18      for (; i < length; i++) {
19          acc0 = acc0 OP data[i];
20      }
21      *dest = acc0 OP acc1;
22  }
```

Figure 5.21 Applying 2×2 loop unrolling. By maintaining multiple accumulators, this approach can make better use of the multiple functional units and their pipelining capabilities.

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine4`	551	Accumulate in temporary	1.27	3.01	3.01	5.01
`combine5`	568	2×1 unrolling	1.01	3.01	3.01	5.01
`combine6`	573	2×2 unrolling	0.81	1.51	1.51	2.51
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

We see that we have improved the performance for all cases, with integer product, floating-point addition, and floating-point multiplication improving by a factor of around 2, and integer addition improving somewhat as well. Most significantly, we have broken through the barrier imposed by the latency bound. The processor no longer needs to delay the start of one sum or product operation until the previous one has completed.

To understand the performance of `combine6`, we start with the code and operation sequence shown in Figure 5.22. We can derive a template showing the

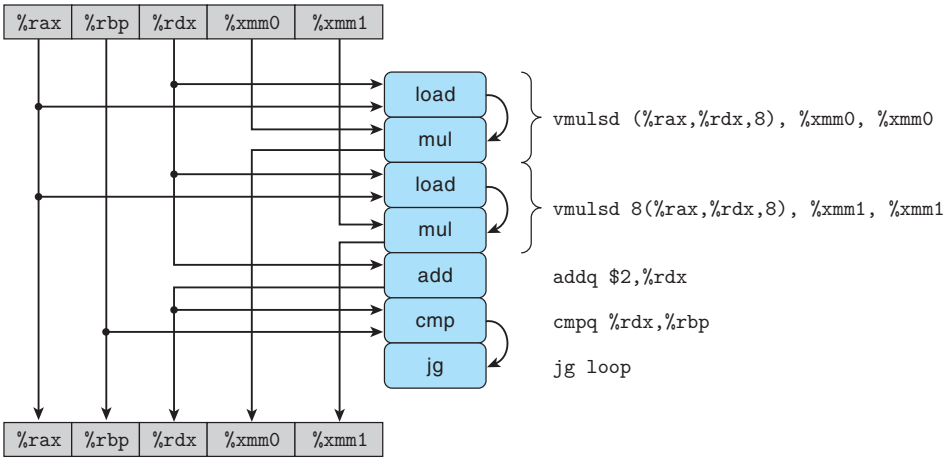

Figure 5.22 **Graphical representation of inner-loop code for** `combine6`. Each iteration has two `vmulsd` instructions, each of which is translated into a load and a mul operation.

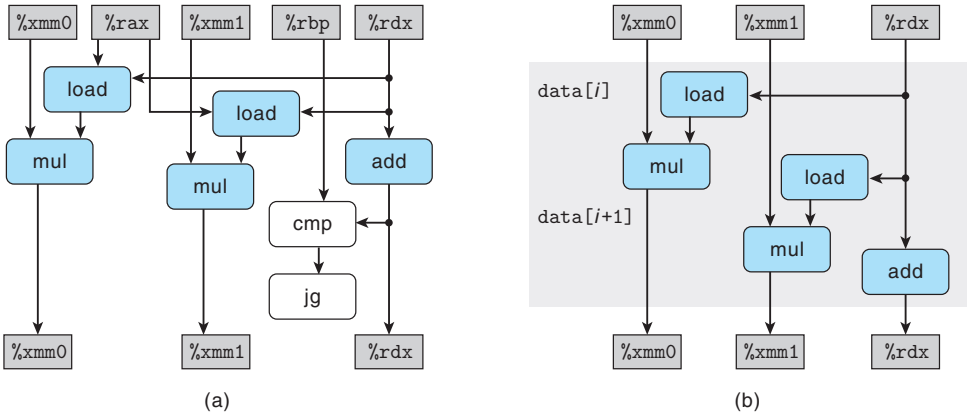

Figure 5.23 **Abstracting** `combine6` **operations as a data-flow graph.** We rearrange, simplify, and abstract the representation of Figure 5.22 to show the data dependencies between successive iterations (a). We see that there is no dependency between the two mul operations (b).

data dependencies between iterations through the process shown in Figure 5.23. As with `combine5`, the inner loop contains two `vmulsd` operations, but these instructions translate into mul operations that read and write separate registers, with no data dependency between them (Figure 5.23(b)). We then replicate this template $n/2$ times (Figure 5.24), modeling the execution of the function on a vector of length n. We see that we now have two critical paths, one corresponding to computing the product of even-numbered elements (program value `acc0`) and

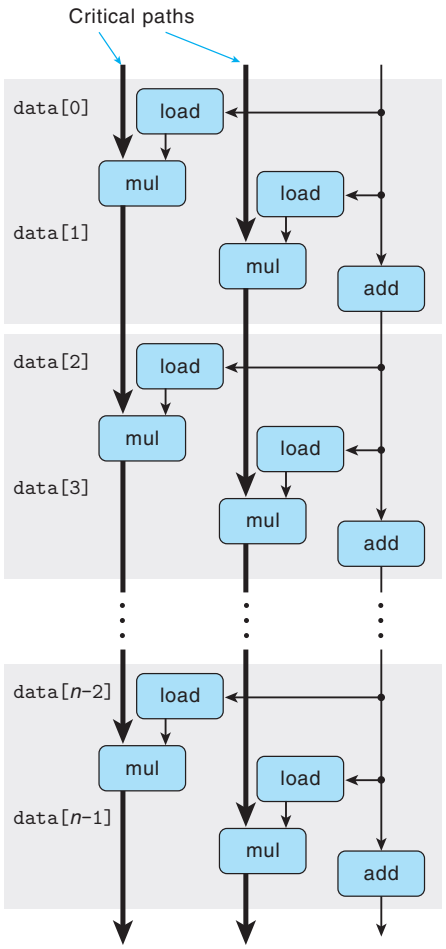

Figure 5.24
Data-flow representation of `combine6` operating on a vector of length n. We now have two critical paths, each containing $n/2$ operations.

one for the odd-numbered elements (program value `acc1`). Each of these critical paths contains only $n/2$ operations, thus leading to a CPE of around $5.00/2 = 2.50$. A similar analysis explains our observed CPE of around $L/2$ for operations with latency L for the different combinations of data type and combining operation. Operationally, the programs are exploiting the capabilities of the functional units to increase their utilization by a factor of 2. The only exception is for integer addition. We have reduced the CPE to below 1.0, but there is still too much loop overhead to achieve the theoretical limit of 0.50.

We can generalize the multiple accumulator transformation to unroll the loop by a factor of k and accumulate k values in parallel, yielding $k \times k$ *loop unrolling*. Figure 5.25 demonstrates the effect of applying this transformation for values up to $k = 10$. We can see that, for sufficiently large values of k, the program can

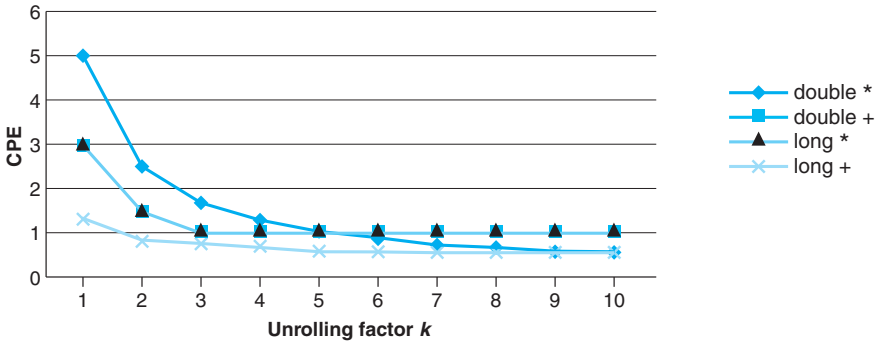

Figure 5.25 CPE performance of $k \times k$ loop unrolling. All of the CPEs improve with this transformation, achieving near or at their throughput bounds.

achieve nearly the throughput bounds for all cases. Integer addition achieves a CPE of 0.54 with $k = 7$, close to the throughput bound of 0.50 caused by the two load units. Integer multiplication and floating-point addition achieve CPEs of 1.01 when $k \geq 3$, approaching the throughput bound of 1.00 set by their functional units. Floating-point multiplication achieves a CPE of 0.51 for $k \geq 10$, approaching the throughput bound of 0.50 set by the two floating-point multipliers and the two load units. It is worth noting that our code is able to achieve nearly twice the throughput with floating-point multiplication as it can with floating-point addition, even though multiplication is a more complex operation.

In general, a program can achieve the throughput bound for an operation only when it can keep the pipelines filled for all of the functional units capable of performing that operation. For an operation with latency L and capacity C, this requires an unrolling factor $k \geq C \cdot L$. For example, floating-point multiplication has $C = 2$ and $L = 5$, necessitating an unrolling factor of $k \geq 10$. Floating-point addition has $C = 1$ and $L = 3$, achieving maximum throughput with $k \geq 3$.

In performing the $k \times k$ unrolling transformation, we must consider whether it preserves the functionality of the original function. We have seen in Chapter 2 that two's-complement arithmetic is commutative and associative, even when overflow occurs. Hence, for an integer data type, the result computed by `combine6` will be identical to that computed by `combine5` under all possible conditions. Thus, an optimizing compiler could potentially convert the code shown in `combine4` first to a two-way unrolled variant of `combine5` by loop unrolling, and then to that of `combine6` by introducing parallelism. Some compilers do either this or similar transformations to improve performance for integer data.

On the other hand, floating-point multiplication and addition are not associative. Thus, `combine5` and `combine6` could produce different results due to rounding or overflow. Imagine, for example, a product computation in which all of the elements with even indices are numbers with very large absolute values, while those with odd indices are very close to 0.0. In such a case, product PE_n might overflow, or PO_n might underflow, even though computing product P_n pro-

ceeds normally. In most real-life applications, however, such patterns are unlikely. Since most physical phenomena are continuous, numerical data tend to be reasonably smooth and well behaved. Even when there are discontinuities, they do not generally cause periodic patterns that lead to a condition such as that sketched earlier. It is unlikely that multiplying the elements in strict order gives fundamentally better accuracy than does multiplying two groups independently and then multiplying those products together. For most applications, achieving a performance gain of 2× outweighs the risk of generating different results for strange data patterns. Nevertheless, a program developer should check with potential users to see if there are particular conditions that may cause the revised algorithm to be unacceptable. Most compilers do not attempt such transformations with floating-point code, since they have no way to judge the risks of introducing transformations that can change the program behavior, no matter how small.

5.9.2 Reassociation Transformation

We now explore another way to break the sequential dependencies and thereby improve performance beyond the latency bound. We saw that the $k \times 1$ loop unrolling of `combine5` did not change the set of operations performed in combining the vector elements to form their sum or product. By a very small change in the code, however, we can fundamentally change the way the combining is performed, and also greatly increase the program performance.

Figure 5.26 shows a function `combine7` that differs from the unrolled code of `combine5` (Figure 5.16) only in the way the elements are combined in the inner loop. In `combine5`, the combining is performed by the statement

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

while in `combine7` it is performed by the statement

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

differing only in how two parentheses are placed. We call this a *reassociation transformation*, because the parentheses shift the order in which the vector elements are combined with the accumulated value `acc`, yielding a form of loop unrolling we refer to as "$2 \times 1a$."

To an untrained eye, the two statements may seem essentially the same, but when we measure the CPE, we get a surprising result:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine4`	551	Accumulate in temporary	1.27	3.01	3.01	5.01
`combine5`	568	2×1 unrolling	1.01	3.01	3.01	5.01
`combine6`	573	2×2 unrolling	0.81	1.51	1.51	2.51
`combine7`	578	$2 \times 1a$ unrolling	1.01	1.51	1.51	2.51
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

```
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  }
```

Figure 5.26 Applying $2 \times 1a$ unrolling. By reassociating the arithmetic, this approach increases the number of operations that can be performed in parallel.

The integer addition case matches the performance of $k \times 1$ unrolling (`combine5`), while the other three cases match the performance of the versions with parallel accumulators (`combine6`), doubling the performance relative to $k \times 1$ unrolling. These cases have broken through the barrier imposed by the latency bound.

Figure 5.27 illustrates how the code for the inner loop of `combine7` (for the case of multiplication as the combining operation and `double` as data type) gets decoded into operations and the resulting data dependencies. We see that the load operations resulting from the `vmovsd` and the first `vmulsd` instructions load vector elements i and $i + 1$ from memory, and the first mul operation multiplies them together. The second mul operation then multiples this result by the accumulated value `acc`. Figure 5.28(a) shows how we rearrange, refine, and abstract the operations of Figure 5.27 to get a template representing the data dependencies for one iteration (Figure 5.28(b)). As with the templates for `combine5` and `combine7`, we have two load and two mul operations, but only one of the mul operations forms a data-dependency chain between loop registers. When we then replicate this template $n/2$ times to show the computations performed in multiplying n vector elements (Figure 5.29), we see that we only have $n/2$ operations along the critical path. The first multiplication within each iteration can be performed without waiting for the accumulated value from the previous iteration. Thus, we reduce the minimum possible CPE by a factor of around 2.

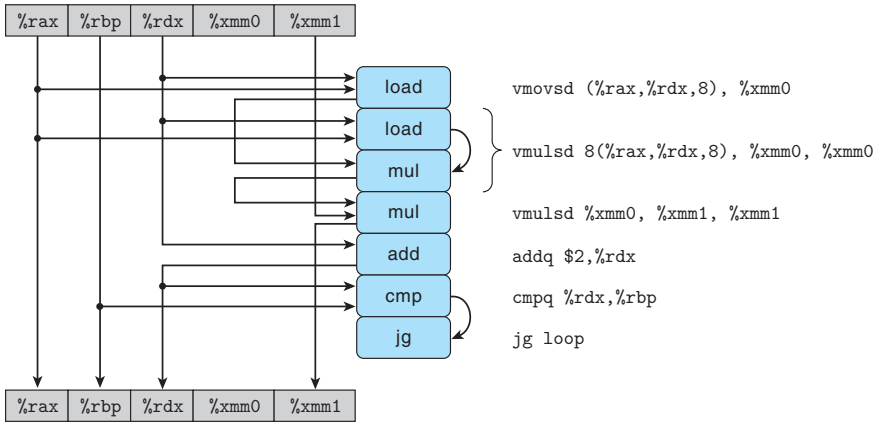

Figure 5.27 Graphical representation of inner-loop code for `combine7`. Each iteration gets decoded into similar operations as for `combine5` or `combine6`, but with different data dependencies.

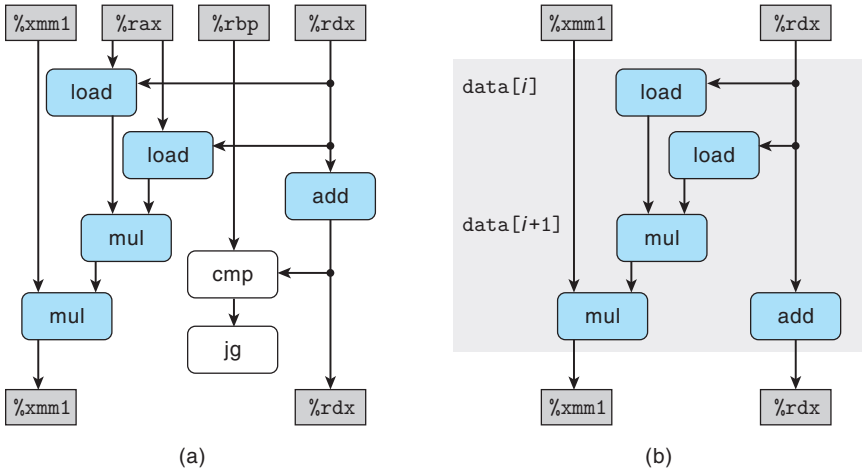

Figure 5.28 Abstracting `combine7` operations as a data-flow graph. We rearrange, simplify, and abstract the representation of Figure 5.27 to show the data dependencies between successive iterations. The upper mul operation multiplies two 2-vector elements with each other, while the lower one multiplies the result by loop variable `acc`.

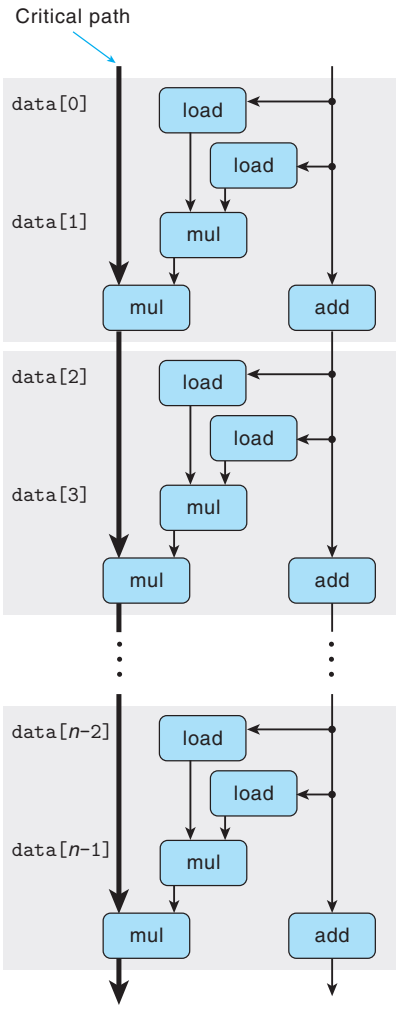

Figure 5.29
Data-flow representation of `combine7` operating on a vector of length n. We have a single critical path, but it contains only $n/2$ operations.

Figure 5.30 demonstrates the effect of applying the reassociation transformation to achieve what we refer to as $k \times 1a$ *loop unrolling* for values up to $k = 10$. We can see that this transformation yields performance results similar to what is achieved by maintaining k separate accumulators with $k \times k$ unrolling. In all cases, we come close to the throughput bounds imposed by the functional units.

In performing the reassociation transformation, we once again change the order in which the vector elements will be combined together. For integer addition and multiplication, the fact that these operations are associative implies that this reordering will have no effect on the result. For the floating-point cases, we must once again assess whether this reassociation is likely to significantly affect

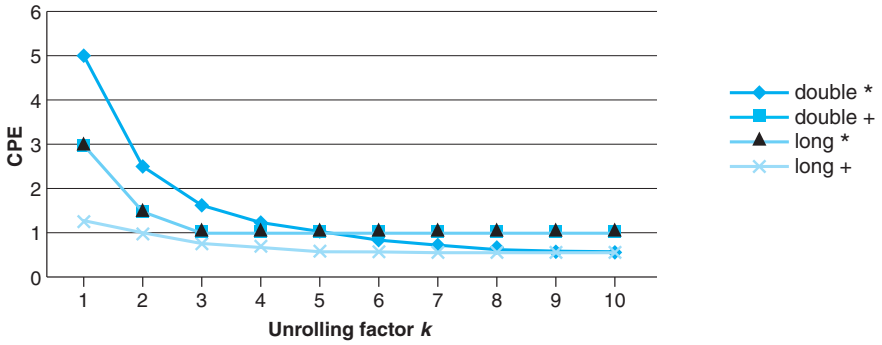

Figure 5.30 CPE performance for $k \times 1a$ loop unrolling. All of the CPEs improve with this transformation, nearly approaching their throughput bounds.

the outcome. We would argue that the difference would be immaterial for most applications.

In summary, a reassociation transformation can reduce the number of operations along the critical path in a computation, resulting in better performance by better utilizing the multiple functional units and their pipelining capabilities. Most compilers will not attempt any reassociations of floating-point operations, since these operations are not guaranteed to be associative. Current versions of GCC do perform reassociations of integer operations, but not always with good effects. In general, we have found that unrolling a loop and accumulating multiple values in parallel is a more reliable way to achieve improved program performance.

Practice Problem 5.8 (solution page 612)

Consider the following function for computing the product of an array of n double-precision numbers. We have unrolled the loop by a factor of 3.

```
double aprod(double a[], long n)
{
    long i;
    double x, y, z;
    double r = 1;
    for (i = 0; i < n-2; i+= 3) {
        x = a[i]; y = a[i+1]; z = a[i+2];
        r = r * x * y * z; /* Product computation */
    }
    for (; i < n; i++)
        r *= a[i];
    return r;
}
```

For the line labeled "`Product computation`," we can use parentheses to create five different associations of the computation, as follows:

```
r = ((r * x) * y) * z; /* A1 */
r = (r * (x * y)) * z; /* A2 */
r = r * ((x * y) * z); /* A3 */
r = r * (x * (y * z)); /* A4 */
r = (r * x) * (y * z); /* A5 */
```

Assume we run these functions on a machine where floating-point multiplication has a latency of 5 clock cycles. Determine the lower bound on the CPE set by the data dependencies of the multiplication. (*Hint:* It helps to draw a data-flow representation of how `r` is computed on every iteration.)

Web Aside OPT:SIMD Achieving greater parallelism with vector instructions

As described in Section 3.1, Intel introduced the SSE instructions in 1999, where SSE is the acronym for "streaming SIMD extensions" and, in turn, SIMD (pronounced "sim-dee") is the acronym for "single instruction, multiple data." The SSE capability has gone through multiple generations, with more recent versions being named *advanced vector extensions*, or AVX. The SIMD execution model involves operating on entire vectors of data within single instructions. These vectors are held in a special set of *vector registers*, named `%ymm0`–`%ymm15`. Current AVX vector registers are 32 bytes long, and therefore each can hold eight 32-bit numbers or four 64-bit numbers, where the numbers can be either integer or floating-point values. AVX instructions can then perform vector operations on these registers, such as adding or multiplying eight or four sets of values in parallel. For example, if YMM register `%ymm0` contains eight single-precision floating-point numbers, which we denote $a_0, \ldots, a_7$, and `%rcx` contains the memory address of a sequence of eight single-precision floating-point numbers, which we denote $b_0, \ldots, b_7$, then the instruction

```
vmulps  (%rcs), %ymm0, %ymm1
```

will read the eight values from memory and perform eight multiplications in parallel, computing $a_i \leftarrow a_i \cdot b_i$, for $0 \leq i \leq 7$ and storing the resulting eight products in vector register `%ymm1`. We see that a single instruction is able to generate a computation over multiple data values, hence the term "SIMD."

GCC supports extensions to the C language that let programmers express a program in terms of vector operations that can be compiled into the vector instructions of AVX (as well as code based on the earlier SSE instructions). This coding style is preferable to writing code directly in assembly language, since GCC can also generate code for the vector instructions found on other processors.

Using a combination of GCC instructions, loop unrolling, and multiple accumulators, we are able to achieve the following performance for our combining functions:

Web Aside OPT:SIMD Achieving greater parallelism with vector instructions *(continued)*

	Integer				Floating point			
	`int`		`long`		`int`		`long`	
Method	+	*	+	*	+	*	+	*
Scalar 10 × 10	0.54	1.01	0.55	1.00	1.01	0.51	1.01	0.52
Scalar throughput bound	0.50	0.50	1.00	1.00	1.00	1.00	0.50	0.50
Vector 8 × 8	0.05	0.24	0.13	1.51	0.12	0.08	0.25	0.16
Vector throughput bound	0.06	0.12	0.12	—	0.12	0.06	0.25	0.12

In this chart, the first set of numbers is for conventional, *scalar* code written in the style of `combine6`, unrolling by a factor of 10 and maintaining 10 accumulators. The second set of numbers is for code written in a form that GCC can compile into AVX vector code. In addition to using vector operations, this version unrolls the main loop by a factor of 8 and maintains eight separate vector accumulators. We show results for both 32-bit and 64-bit numbers, since the vector instructions achieve 8-way parallelism in the first case, but only 4-way parallelism in the second.

We can see that the vector code achieves almost an eightfold improvement on the four 32-bit cases, and a fourfold improvement on three of the four 64-bit cases. Only the long integer multiplication code does not perform well when we attempt to express it in vector code. The AVX instruction set does not include one to do parallel multiplication of 64-bit integers, and so GCC cannot generate vector code for this case. Using vector instructions creates a new throughput bound for the combining operations. These are eight times lower for 32-bit operations and four times lower for 64-bit operations than the scalar limits. Our code comes close to achieving these bounds for several combinations of data type and operation.

5.10 Summary of Results for Optimizing Combining Code

Our efforts at maximizing the performance of a routine that adds or multiplies the elements of a vector have clearly paid off. The following summarizes the results we obtain with *scalar* code, not making use of the vector parallelism provided by AVX vector instructions:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine1`	543	Abstract `-O1`	10.12	10.12	10.17	11.14
`combine6`	573	2 × 2 unrolling	0.81	1.51	1.51	2.51
		10 × 10 unrolling	0.55	1.00	1.01	0.52
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50