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

By using multiple optimizations, we have been able to achieve CPEs close to the throughput bounds of 0.50 and 1.00, limited only by the capacities of the functional units. These represent 10–20× improvements on the original code. This has all been done using ordinary C code and a standard compiler. Rewriting the code to take advantage of the newer SIMD instructions yields additional performance gains of nearly 4× or 8×. For example, for single-precision multiplication, the CPE drops from the original value of 11.14 down to 0.06, an overall performance gain of over 180×. This example demonstrates that modern processors have considerable amounts of computing power, but we may need to coax this power out of them by writing our programs in very stylized ways.

5.11 Some Limiting Factors

We have seen that the critical path in a data-flow graph representation of a program indicates a fundamental lower bound on the time required to execute a program. That is, if there is some chain of data dependencies in a program where the sum of all of the latencies along that chain equals T, then the program will require at least T cycles to execute.

We have also seen that the throughput bounds of the functional units also impose a lower bound on the execution time for a program. That is, assume that a program requires a total of N computations of some operation, that the microprocessor has C functional units capable of performing that operation, and that these units have an issue time of I. Then the program will require at least $N \cdot I/C$ cycles to execute.

In this section, we will consider some other factors that limit the performance of programs on actual machines.

5.11.1 Register Spilling

The benefits of loop parallelism are limited by the ability to express the computation in assembly code. If a program has a degree of parallelism P that exceeds the number of available registers, then the compiler will resort to *spilling*, storing some of the temporary values in memory, typically by allocating space on the run-time stack. As an example, the following measurements compare the result of extending the multiple accumulator scheme of `combine6` to the cases of $k = 10$ and $k = 20$:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine6`	573					
		10 × 10 unrolling	0.55	1.00	1.01	0.52
		20 × 20 unrolling	0.83	1.03	1.02	0.68
Throughput bound			0.50	1.00	1.00	0.50