8 个乘积保存到向量寄存器%ymm1。我们看到，一条指令能够产生对多个数据值的计算，因此称为“SIMD”。

GCC 支持对 C 语言的扩展，能够让程序员在程序中使用向量操作，这些操作能够被编译成 AVX 的向量指令(以及基于早前的 SSE 指令的代码)。这种代码风格比直接用汇编语言写代码要好，因为 GCC 还可以为其他处理器上的向量指令产生代码。

使用 GCC 指令、循环展开和多个累积变量的组合，我们的合并函数能够达到下面的性能：

方法	整数				浮点数			
	int		long		long		int	
	+	*	+	*	+	*	+	*
标量 10×10	0.54	1.01	0.55	1.00	1.01	0.51	1.01	0.52
标量吞吐量界限	0.50	1.00	0.50	1.00	1.00	0.50	1.00	0.50
向量 8×8	0.05	0.24	0.13	1.51	0.12	0.08	0.25	0.16
向量吞吐量界限	0.06	0.12	0.12	—	0.12	0.06	0.25	0.12

上表中，第一组数字对应的是按照 combine6 的风格编写的传统标量代码，循环展开因子为 10，并维护 10 个累积变量。第二组数字对应的代码编写形式可以被 GCC 编译成 AVX 向量代码。除了使用向量操作外，这个版本也进行了循环展开，展开因子为 8，并维护 8 个不同的向量累积变量。我们给出了 32 位和 64 位数字的结果，因为向量指令在第一种情况中达到 8 路并行，而在第二种情况中只能达到 4 路并行。

可以看到，向量代码在 32 位的 4 种情况下几乎都获得了 8 倍的提升，对于 64 位来说，在其中的 3 种情况下获得了 4 倍的提升。只有长整数乘法代码在我们尝试将其表示为向量代码时性能不佳。AVX 指令集不包括 64 位整数的并行乘法指令，因此 GCC 无法为此种情况生成向量代码。使用向量指令对合并操作产生了新的吞吐量界限。与标量界限相比，32 位操作的新界限小了 8 倍，64 位操作的新界限小了 4 倍。我们的代码在几种数据类型和操作的组合上接近了这些界限。

5.10　优化合并代码的结果小结

我们极大化对向量元素加或者乘的函数性能的努力获得了成功。下表总结了对于标量代码所获得的结果，没有使用 AVX 向量指令提供的向量并行性：

函数	方法	整数		浮点数	
		+	*	+	*
combine1	抽象-O1	10.12	10.12	10.17	11.14
combine6	2×2 循环展开	0.81	1.51	1.51	2.51
	10×10 循环展开	0.55	1.00	1.01	0.52
延迟界限		1.00	3.00	3.00	5.00
吞吐量界限		0.50	1.00	1.00	0.50

使用多项优化技术，我们获得的 CPE 已经接近于 0.50 和 1.00 的吞吐量界限，只受限于功能单元的容量。与原始代码相比提升了 10～20 倍，且使用普通的 C 代码和标准编译器就获得了所有这些改进。重写代码利用较新的 SIMD 指令得到了将近 4 倍或 8 倍的性能提升。比如单精度乘法，CPE 从初值 11.14 降到了 0.06，整体性能提升超过 180 倍。这个例子说明现代处理器具有相当的计算能力，但是我们可能需要按非常程式化的方式来编写程序以便将这些能力诱发出来。