Loop Optimizations - Operation Counting*

Charles Severance
Kevin Dowd

Before you begin to rewrite a loop body or reorganize the order of the loops, you must have some idea of what the body of the loop does for each iteration. *Operation counting* is the process of surveying a loop to understand the operation mix. You need to count the number of loads, stores, floating-point, integer, and library calls per iteration of the loop. From the count, you can see how well the operation mix of a given loop matches the capabilities of the processor. Of course, operation counting doesn't guarantee that the compiler will generate an efficient representation of a loop.[1] But it generally provides enough insight to the loop to direct tuning efforts.

Bear in mind that an instruction mix that is balanced for one machine may be imbalanced for another. Processors on the market today can generally issue some combination of one to four operations per clock cycle. Address arithmetic is often embedded in the instructions that reference memory. Because the compiler can replace complicated loop address calculations with simple expressions (provided the pattern of addresses is predictable), you can often ignore address arithmetic when counting operations.[2]

Let's look at a few loops and see what we can learn about the instruction mix:

```
DO I=1,N
  A(I,J,K) = A(I,J,K) + B(J,I,K)
ENDDO
```

This loop contains one floating-point addition and three memory references (two loads and a store). There are some complicated array index expressions, but these will probably be simplified by the compiler and executed in the same cycle as the memory and floating-point operations. For each iteration of the loop, we must increment the index variable and test to determine if the loop has completed.

A 3:1 ratio of memory references to floating-point operations suggests that we can hope for no more than 1/3 peak floating-point performance from the loop unless we have more than one path to memory. That's

*Version 1.3: Aug 25, 2010 11:02 am -0500

†http://creativecommons.org/licenses/by/3.0/

[1] Take a look at the assembly language output to be sure, which may be going a bit overboard. To get an assembly language listing on most machines, compile with the **–S** flag. On an RS/6000, use the **–qlist** flag.

[2] The compiler reduces the complexity of loop index expressions with a technique called *induction variable simplification*. See (<http://cnx.org/content/m33690/latest/>).

bad news, but good information. The ratio tells us that we ought to consider memory reference optimizations first.

The loop below contains one floating-point addition and two memory operations — a load and a store. Operand `B(J)` is loop-invariant, so its value only needs to be loaded once, upon entry to the loop:

```
DO I=1,N
  A(I) = A(I) + B(J)
ENDDO
```

Again, our floating-point throughput is limited, though not as severely as in the previous loop. The ratio of memory references to floating-point operations is 2:1.

The next example shows a loop with better prospects. It performs element-wise multiplication of two vectors of complex numbers and assigns the results back to the first. There are six memory operations (four loads and two stores) and six floating-point operations (two additions and four multiplications):

```
for (i=0; i<n; i++) {
    xr[i] = xr[i] * yr[i] - xi[i] * yi[i];
    xi[i] = xr[i] * yi[i] + xi[i] * yr[i];
}
```

It appears that this loop is roughly balanced for a processor that can perform the same number of memory operations and floating-point operations per cycle. However, it might not be. Many processors perform a floating-point multiply and add in a single instruction. If the compiler is good enough to recognize that the multiply-add is appropriate, this loop may also be limited by memory references; each iteration would be compiled into two multiplications and two multiply-adds.

Again, operation counting is a simple way to estimate how well the requirements of a loop will map onto the capabilities of the machine. For many loops, you often find the performance of the loops dominated by memory references, as we have seen in the last three examples. This suggests that memory reference tuning is very important.