Loop Optimizations - Memory Access Patterns[*]

Charles Severance
Kevin Dowd

The best pattern is the most straightforward: increasing and unit sequential. For an array with a single dimension, stepping through one element at a time will accomplish this. For multiply-dimensioned arrays, access is fastest if you iterate on the array subscript offering the smallest *stride* or step size. In FORTRAN programs, this is the leftmost subscript; in C, it is the rightmost. The FORTRAN loop below has unit stride, and therefore will run quickly:

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

In contrast, the next loop is slower because its stride is N (which, we assume, is greater than 1). As N increases from one to the length of the cache line (adjusting for the length of each element), the performance worsens. Once N is longer than the length of the cache line (again adjusted for element size), the performance won't decrease:

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

Here's a unit-stride loop like the previous one, but written in C:

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

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

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i][j] = a[i][j] + c[i][j] * d;
```

Unit stride gives you the best performance because it conserves cache entries. Recall how a data cache works.[1] Your program makes a memory reference; if the data is in the cache, it gets returned immediately. If not, your program suffers a cache miss while a new cache line is fetched from main memory, replacing an old one. The line holds the values taken from a handful of neighboring memory locations, including the one that caused the cache miss. If you loaded a cache line, took one piece of data from it, and threw the rest away, you would be wasting a lot of time and memory bandwidth. However, if you brought a line into the cache and consumed everything in it, you would benefit from a large number of memory references for a small number of cache misses. This is exactly what you get when your program makes unit-stride memory references.

The worst-case patterns are those that jump through memory, especially a large amount of memory, and particularly those that do so without apparent rhyme or reason (viewed from the outside). On jobs that operate on very large data structures, you pay a penalty not only for cache misses, but for TLB misses too.[2] It would be nice to be able to rein these jobs in so that they make better use of memory. Of course, you can't eliminate memory references; programs have to get to their data one way or another. The question is, then: how can we restructure memory access patterns for the best performance?

In the next few sections, we are going to look at some tricks for restructuring loops with strided, albeit predictable, access patterns. The tricks will be familiar; they are mostly loop optimizations from here[3], used here for different reasons. The underlying goal is to minimize cache and TLB misses as much as possible. You will see that we can do quite a lot, although some of this is going to be ugly.

1 Loop Interchange to Ease Memory Access Patterns

Loop interchange is a good technique for lessening the impact of strided memory references. Let's revisit our FORTRAN loop with non-unit stride. The good news is that we can easily interchange the loops; each iteration is independent of every other:

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

After interchange, A, B, and C are referenced with the leftmost subscript varying most quickly. This modification can make an important difference in performance. We traded three N-strided memory references for unit strides:

[1] See (<http://cnx.org/content/m32733/latest/>).

[2] The Translation Lookaside Buffer (TLB) is a cache of translations from virtual memory addresses to physical memory addresses. For more information, refer back to (<http://cnx.org/content/m32733/latest/>).

[3] "Eliminating Clutter - Introduction" <http://cnx.org/content/m33720/latest/>

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

2 Matrix Multiplication

Matrix multiplication is a common operation we can use to explore the options that are available in optimizing a loop nest. A programmer who has just finished reading a linear algebra textbook would probably write matrix multiply as it appears in the example below:

```
DO I=1,N
  DO J=1,N
    SUM = 0
    DO K=1,N
      SUM = SUM + A(I,K) * B(K,J)
    ENDDO
    C(I,J) = SUM
  ENDDO
ENDDO
```

The problem with this loop is that the `A(I,K)` will be non-unit stride. Each iteration in the inner loop consists of two loads (one non-unit stride), a multiplication, and an addition.

Given the nature of the matrix multiplication, it might appear that you can't eliminate the non-unit stride. However, with a simple rewrite of the loops all the memory accesses can be made unit stride:

```
DO J=1,N
  DO I=1,N
    C(I,J) = 0.0
  ENDDO
ENDDO

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

Now, the inner loop accesses memory using unit stride. Each iteration performs two loads, one store, a multiplication, and an addition. When comparing this to the previous loop, the non-unit stride loads have been eliminated, but there is an additional store operation. Assuming that we are operating on a cache-based system, and the matrix is larger than the cache, this extra store won't add much to the execution time. The store is to the location in `C(I,J)` that was used in the load. In most cases, the store is to a line that is already in the in the cache. The `B(K,J)` becomes a constant scaling factor within the inner loop.