

LOOP OPTIMIZATIONS - NESTED LOOPS*

Charles Severance

Kevin Dowd

This work is produced by OpenStax-CN X and licensed under the Creative Commons Attribution License 3.0[†]

When you embed loops within other loops, you create a *loop nest*. The loop or loops in the center are called the *inner* loops. The surrounding loops are called *outer* loops. Depending on the construction of the loop nest, we may have some flexibility in the ordering of the loops. At times, we can swap the outer and inner loops with great benefit. In the next sections we look at some common loop nestings and the optimizations that can be performed on these loop nests.

Often when we are working with nests of loops, we are working with multidimensional arrays. Computing in multidimensional arrays can lead to non-unit-stride memory access. Many of the optimizations we perform on loop nests are meant to improve the memory access patterns.

First, we examine the computation-related optimizations followed by the memory optimizations.

1 Outer Loop Unrolling

If you are faced with a loop nest, one simple approach is to unroll the inner loop. Unrolling the innermost loop in a nest isn't any different from what we saw above. You just pretend the rest of the loop nest doesn't exist and approach it in the normal way. However, there are times when you want to apply loop unrolling not just to the inner loop, but to outer loops as well — or perhaps only to the outer loops. Here's a typical loop nest:

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

To unroll an outer loop, you pick one of the outer loop index variables and replicate the innermost loop body so that several iterations are performed at the same time, just like we saw in the here¹. The difference is in the index variable for which you unroll. In the code below, we have unrolled the middle (j) loop twice:

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

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

¹"Loop Optimizations - Qualifying Candidates for Loop Unrolling" <<http://cnx.org/content/m33733/latest/>>

```

for (i=0; i<n; i++)
  for (j=0; j<n; j+=2)
    for (k=0; k<n; k++) {
      a[i][j][k] = a[i][j][k] + b[i][k][j] * c;
      a[i][j+1][k] = a[i][j+1][k] + b[i][k][j+1] * c;
    }

```

We left the *k* loop untouched; however, we could unroll that one, too. That would give us outer *and* inner loop unrolling at the same time:

```

for (i=0; i<n; i++)
  for (j=0; j<n; j+=2)
    for (k=0; k<n; k+=2) {
      a[i][j][k]      = a[i][j][k]      + b[i][k][j] * c;
      a[i][j+1][k]    = a[i][j+1][k]    + b[i][k][j+1] * c;
      a[i][j][k+1]    = a[i][j][k+1]    + b[i][k+1][j] * c;
      a[i][j+1][k+1]  = a[i][j+1][k+1]  + b[i][k+1][j+1] * c;
    }

```

We could even unroll the *i* loop too, leaving eight copies of the loop innards. (Notice that we completely ignored preconditioning; in a real application, of course, we couldn't.)

2 Outer Loop Unrolling to Expose Computations

Say that you have a doubly nested loop and that the inner loop trip count is low — perhaps 4 or 5 on average. Inner loop unrolling doesn't make sense in this case because there won't be enough iterations to justify the cost of the preconditioning loop. However, you may be able to unroll an outer loop. Consider this loop, assuming that *M* is small and *N* is large:

```

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

```

Unrolling the *I* loop gives you lots of floating-point operations that can be overlapped:

```

II = IMOD (N,4)
DO I=1,II
  DO J=1,M
    A(J,I) = B(J,I) + C(J,I) * D

```

```

        ENDDO
    ENDDO

DO I=1,N,4
    DO J=1,M
        A(J,I) = B(J,I) + C(J,I) * D
        A(J,I+1) = B(J,I+1) + C(J,I+1) * D
        A(J,I+2) = B(J,I+2) + C(J,I+2) * D
        A(J,I+3) = B(J,I+3) + C(J,I+3) * D
    ENDDO
ENDDO

```

In this particular case, there is bad news to go with the good news: unrolling the outer loop causes strided memory references on A, B, and C. However, it probably won't be too much of a problem because the inner loop trip count is small, so it naturally groups references to conserve cache entries.

Outer loop unrolling can also be helpful when you have a nest with recursion in the inner loop, but not in the outer loops. In this next example, there is a first- order linear recursion in the inner loop:

```

DO J=1,M
    DO I=2,N
        A(I,J) = A(I,J) + A(I-1,J) * B
    ENDDO
ENDDO

```

Because of the recursion, we can't unroll the inner loop, but we can work on several copies of the outer loop at the same time. When unrolled, it looks like this:

```

JJ = IMOD (M,4)
DO J=1,JJ
    DO I=2,N
        A(I,J) = A(I,J) + A(I-1,J) * B
    ENDDO
ENDDO

DO J=1+JJ,M,4
    DO I=2,N
        A(I,J) = A(I,J) + A(I-1,J) * B
        A(I,J+1) = A(I,J+1) + A(I-1,J+1) * B
        A(I,J+2) = A(I,J+2) + A(I-1,J+2) * B
        A(I,J+3) = A(I,J+3) + A(I-1,J+3) * B
    ENDDO
ENDDO

```

You can see the recursion still exists in the I loop, but we have succeeded in finding lots of work to do anyway.

Sometimes the reason for unrolling the outer loop is to get a hold of much larger chunks of things that can be done in parallel. If the outer loop iterations are independent, and the inner loop trip count is high, then each outer loop iteration represents a significant, parallel chunk of work. On a single CPU that doesn't matter much, but on a tightly coupled multiprocessor, it can translate into a tremendous increase in speeds.