Assignment 3: Locality Optimization

15-745: Optimizing Compilers

Checkpoint: 10:30am, Thursday, February 28
Due: 10:30am, Thursday, March 6

In this assignment you will use the results of the previous assignment to implement two optimizations, loop interchange and blocking, that improve the locality of memory references. You should implement both optimizations in the file `LocalityOptimizer.cpp`. Note that the Allen and Kennedy book is probably the best reference for this assignment and is on reserve in the E&S library.

1 Loop Interchange

The loop interchange transformation attempts to reorder the loops of a loop nest to maximize spacial locality. For example, the following loop nest:

```
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    A[j][i] += 1;
```

would be rearranged so that the `i` loop is on the inside to exploit the spatial locality of A.

To implement this optimization you will need to determine when an interchange is legal and when it is desirable. You can use the results from your dependence analyzer to determine if loops can be safely interchanged. You will have to modify `LoopDependenceAnalysis.cpp` and create `LoopDependence-Analysis.h` so that your optimizer can access this information. Two loops can not be interchanged if the interchange would result in an illegal direction matrix. In addition, you must avoid invalidating any loop carried scalar dependencies (requiring the LCSSA pass should make this easier).

You may use a simple greedy algorithm to determine which loops should be innermost. You do not need to consider all possible loop orderings. Instead, you can heuristically evaluate the impact on spatial locality of each loop and simply sort them by this heuristic value.

Your fully functional loop interchange optimization is due at the checkpoint date, February 28.

2 Blocking

We can sometimes further improve the locality of memory references by blocking a loop. We apply *strip-mine-and-interchange* to the innermost loop. This transformation decomposes the inner loop into two loops. One loop iterates a fixed number of times, the block size, and the other loop iterates over these *strips*. The *by-strip* loop is then interchanged to be the outermost loop. For example, consider classical matrix multiply:

```
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    for(k = 0; k < n; k++)
    {
      C[i][j] += A[i][k] * B[k][j];
    }
```

If we strip-mine-and-interchange we get:

```
for(k = 0; k < n; k += BS)
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      for(kk = k; kk < min(n,k+BS); kk++)
      {
        C[i][j] += A[i][kk] * B[kk][j];
      }
```

If the block size, BS, is small enough then `A[i][kk]` should remain in the cache for each iteration of the `j` loop and `B[kk][j]` will benefit from spatial locality as the `j` loop iterates. In general, blocking should be performed on the innermost loop anytime there is a reuse in an outer loop. We consider there to be a reuse in a loop if there is a depenendence of any type, including input (read after read) or the loop index appears in the continuous dimension of a multidimensional array. For example, in the above case there is a reuse on `A[i][k]` in the `j` loop and `B[k][j]` is being indexed by `j` in the contiguous dimension.

After performing the transformation, we must re-evaluate the loop nest to see if additional blocking is desirable. In the transformed loop the innermost loop is now the `j` loop (we ignore the already blocked loops). In order to determine if it is profitable to block the `j` loop we look for reuse in the `i` loop. Since `B[kk][j]` is a reuse, we proceed to block the `j` loop:

```
for(k = 0; k < n; k += BS)
  for(j = 0; j < n; j += BS)
    for(i = 0; i < n; i++)
      for(jj = j; jj < min(n,j+BS); jj++)
        for(kk = k; kk < min(n,k+BS); kk++)
        {
          C[i][jj] += A[i][kk] * B[kk][jj];
        }
```

The result is the classical form of blocked matrix multiply. In your implementation the block size should be a simple user provided constant set with the `-block-size=`BS command line option.

3 Testing

Your optimizer should correctly compile all the provided benchmarks. We provide a script, `optbench`, that compiles, runs, and verifies the output of all the benchmarks. You may find the `-march=c` option to `llc` useful in debugging your code. You should also be able to successfully optimize the provided suboptimal version of matrix multiplication (`mm.c`).

Your are responsible for creating a test case in `test.c` that computes something nontrivial and that your optimizer can successfully improve.

4 Writeup and Handin

Optimize and run `mm.c` and `test.c` for all power of two block sizes between 2^1 and 2^{10}. Plot the performance as you vary the block size as well as the original, unoptimized, performance. In your writeup include these plots, a description of the caches on the machine you are using, and an explanation of the results. In addition, include a description of the heuristic you use when performing loop interchange.

You should handin your writeup (in pdf form), your version of `LocalityOptimizer.cpp`, `LoopDependenceAnalysis.cpp`, `LoopDependenceAnalysis.h`, and your test case, `test.c` into your course handin directory.[1]

[1]If you have trouble, just email your submission to `dkoes`.