Notes on Locality Transformations
CS4961
September 15, 2011

Today's lecture is about two things. We complete the discussion of loop transformations for locality optimization. We also formalize how to use dependence information to determine safety of reordering transformations.

Definitions:

Data Reuse: An intrinsic feature of a computation, the same data is used multiple times.

Data Locality: Same or adjacent data is reused and accessed from some nearby storage at low latency.

I. Dependence Analysis in loop nest computations

Last time we talked about how we use the idea that dependences are preserved in a computation to determine the safety of reordering computations. The process of dependence analysis of a loop nest computation is as follows:

1. Compare all pairs of array and scalar references to determine which might be involved in a dependence.
2. Where a dependence is possible, create a distance and/or direction vector that describes the temporal relationship of the source and sink of the dependence. A *distance vector*, which is what we will focus on in this class, represents the distance in the multi-dimensional iteration space of a loop between the dependence's source and sink.
3. For each dependence, we can compute which loop "carries" the dependence, which constrains the reordering transformations. The outermost non-zero distance carries the dependence.
4. For each transformation we are interested in applying, we can perform a safety test based on dependences to determine whether the transformation will preserve the dependences.

Note that a loop nest may have numerous dependences, and the safety tests for a transformation must hold for all dependences.

II. A Few Locality Transformations

For each transformation, we will have a definition, an example, the safety test and when it is profitable.

Loop permutation: permute the order of loops in a loop nest computation. For locality purposes, the new loop order results in a different order of traversing memory, potentially leading to better reuse in cache.

Example (matrix-matrix multiplication):

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

Permute to i,k,j order.

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

Safety: Permutation is safe if it does not reverse direction of any dependence distance vectors (see lecture notes for details).

Profitability: Permutation is used for a variety of purposes, to adjust parallelism granularity and to set up for other optimizations. In this example, we have reordered accesses so that now there is more spatial reuse for C and B. Moving the k loop inward is going to help us with other transformations later.

Loop tiling: Partition the iteration space of loops in a nest so that they iterate over a smaller chunk or block of data that can fit some capacity-constrained portion of the memory hierarchy (cache, registers, scratchpad), with the goal of ensuring the data remains in that storage until the data is reused.

Example (matrix-matrix multiplication): Tile i, k and j loops from previous version

```
for (ii=0; ii<N; ii+=TI)
   for (kk=0; kk<N; kk+=TK)
      for (jj=0; jj<N; jj+=TJ)
         for (i=ii; i<min(N,ii+TI); i++)
            for (k=kk; k<min(N,kk+TK); k++)
               for (j=jj; j<min(N,jj+TJ); j++)
                  C[i][j] += A[i][k] * B[k][j];
```

Safety: Tiling can be decomposed into strip-mine and permute transformations, so it is safe if permutation is safe (see lecture notes for details).

Profitability: When tiling allows data to be reused from low-latency nearby storage, it can greatly reduce latency to memory and achieve much higher performance. In this example, each of the loops controls some dimension of either an input or output of the computation. Because there is significant data reuse in the computation, tiling

can make a significant improvement in performance if problem size N is relatively large.

Unrolling, unroll-and-jam: A loop can be unrolled by replicating its loop body the number of times corresponding to an unroll factor, and adjusting the loop control accordingly, so that the unrolled statements execute if and only if they would have in the original code. Since unrolling is straightforward, we'll focus on unroll-and-jam in the example. Unroll-and-jam involves unrolling an outer loop in a nest and fusing the resulting inner loop bodies.

Example (matrix-matrix multiplication): unroll I and j loop from previous version, and also permute k tile outward

```
for (ii=0; ii<N; ii+=TI)
  for (kk=0; kk<N; kk+=TK)
    for (jj=0; jj<N; jj+=TJ)
      for (k=kk; k<min(N,kk+TK); k++)
        for (i=ii; i<min(N,ii+TI); i+=2)
          for (j=jj; j<min(N,jj+TJ); j+=2) {
            C[i][j] += A[i][k] * B[k][j];
            C[i+1][j] += A[i+1][k] * B[k][j];
            C[i][j+1] += A[i][k] * B[k][j+1];
            C[i+1][j+1] += A[i+1][k] * B[k][j+1];
          }
```

Safety: Unroll-and-jam can be decomposed into tile and unroll transformations, so it is safe if permutation is safe (see lecture notes for details).

Profitability: Unroll-and-jam now exposes a number of things that can be placed in registers. Each element of C can remain in a register for all of the I and j iterations. Also, there are two A elements and two B elements that are used in the inner loop and can be placed in registers.

Optimizing Compilers for Modern Architectures: A Dependence-Based Approach, Allen and Kennedy, 2002, Ch. 2.