Standard Loop Transformations

Louis-Noël Pouchet

CS & ECE
Colorado State University

February 23, 2020

PPoPP'20 Tutorial

Some Reference

Most of the material from this section is covered in:
Advanced Compilation for High Performance Computing *Randy Allen and Ken Kennedy*, Morgan Kaufmann

Some Key Properties

Definition (Program equivalence)

Two computations are equivalent if given the same input they produce identical values for the output variables at the time output statements are executed and the output statements are executed in the same order.

Definition (Reordering transformation)

A reordering transformation is any program transformation that changes the order of execution in the code without adding or deleting any execution of any statement.

Definition (Legality of reordering transformations)

A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence. Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

Example

Example (dgemm original)

```
   /* C := alpha*A*B + beta*C */
   for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
S1:    C[i][j] = 0;
   for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
      for (k = 0; k < nk; ++k)
S2:     C[i][j] += alpha * A[i][k] * B[k][j];
```

Example

Example (dgemm incorrect)

```
   /* C := alpha*A*B + beta*C */
   for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
      for (k = 0; k < nk; ++k)
S2:      C[i][j] += alpha * A[i][k] * B[k][j];
   for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
S1:    C[i][j] = 0;
```

Question: Why is this code incorrect?

Data Dependence

Definition

Two statements R and S, with R coming before S in the instruction stream can be reordered freely if:

- $use(S) \cap def(R) = \emptyset$ (flow dependence otherwise)
- $use(R) \cap def(S) = \emptyset$ (anti dependence otherwise)
- $def(R) \cap def(S) = \emptyset$ (output dependence otherwise)

Example

```
A = B * C;
D = A * E + F; //(flow on A)
D = B * C; //(output on D)
B = F * C; //(anti on B)
```

Control Dependence

Definition

A statement S is in control dependence with a statement R if S is guarded by R

Example

```
R: if (x == 2);
S:    D = A * E + F;
T: F = G * H;
```

Data Dependence

Definition (Bernstein conditions)

Two operations are in dependence if they access the same memory location, and one of these access is a write.

Classification:

- flow dependence (Read-after-Write – RAW)
- anti dependence (Write-after-Read – WAR)
- output dependence (Write-after-Write – RAW)

A First Parallelization Approach

- If two statements have no data/control dependence, then they can be reordered freely
- Parallelization is a reordering transformation
- Naive algorithm: detect independent statements, and parallelize consecutive sets of independent operations

Example

Example (input code)

```
A = B * C;
F = G * H;
U = F * C;
```

Example

Example (Valid transformation)

```
F = G * H;
U = F * C;
A = B * C;
```

Example

Example (Parallel code)

```
finish {
  async {
    F = G * H;
    U = F * C;
  }
  async { A = B * C; }
}
```

Returning to DGEMM

Example (dgemm original)

```
   /* C := alpha*A*B + beta*C */
   for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
S1:    C[i][j] = 0;
   for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
      for (k = 0; k < nk; ++k)
S2:     C[i][j] += alpha * A[i][k] * B[k][j];
```

According to our definition, this code is sequential!

Data dependences in loops

- Intuition: distinguish each memory cell accessed by an array
  - $C \rightarrow C(i,j)$
- Intuition: distinguish each dynamic instance of the statements
  - $S1 \rightarrow S1(i,j)$
- Intuition: apply Bernstein conditions between statement instances, looking at the particular memory address accessed each time.
  - $def_{S1}(i,j) \cap use_{S2}(i,j,k)$ for a flow dependence
  - only instances meeting this property are in dependence, others are not!

More on this later :-)

Catalogue of loop transformations

- loop permutation (a.k.a. interchange)
- loop distribution (a.k.a. fission)
- loop fusion (a.k.a. merging)
- loop peeling
- loop shifting
- loop unrolling
- loop strip-mining
- loop unroll-and-jam
- loop tiling (a.k.a. blocking)
- Index-set splitting
- ...
- loop parallelization
- loop vectorization
- ...

Loop Permutation

Example (original)

```
   for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
S1:     C[i][j] = 0;
```

Example (permute(i,j))

```
   for (j = 0; j < nj; j++)
    for (i = 0; i < ni; i++)
S1:     C[i][j] = 0;
```

This transformation may be illegal

Loop Distribution

Example (original)

```
   for (i = 0; i < ni; i++)
S1:     C[i] = 0;
S2:     D[i] = 0;
```

Example (distribute(S1,S2))

```
   for (i = 0; i < ni; i++)
S1:     C[i] = 0;
   for (i = 0; i < ni; i++)
S2:     D[i] = 0;
```

This transformation may be illegal

Loop Fusion

Example (original)

```
   for (i = 0; i < ni; i++)
S1:     C[i] = 0;
   for (i = 0; i < ni; i++)
S2:     D[i] = 0;
```

Example (fuse(S1,S2))

```
   for (i = 0; i < ni; i++)
S1:     C[i] = 0;
S2:     D[i] = 0;
```

This transformation may be illegal

Loop Shifting

Example (original)

```
   for (i = 0; i < ni; i++)
S1:    C[i] = 0;
S2:    D[i] = 0;
```

Example (shift(S2,1))

```
S1:  C[0] = 0;
     for (i = 1; i < ni; i++)
S1:    C[i] = 0;
S2:    D[i-1] = 0;
S2:  D[ni-1] = 0;
```

This transformation may be illegal

Loop Unrolling

Example (original)

```
   for (i = 0; i < ni; i++)
S1:    C[i] = 0;
```

Example (unroll(i, 2))

```
   for (i = 0; i < ni; i += 2) {
S1:    C[i] = 0;
S1:    C[i+1] = 0;
   }
```

This transformation is always legal

Loop Stripmining

Example (original)

```
    for (i = 0; i < ni; i++)
S1:     C[i] = 0;
```

Example (stripmine(i, 4))

```
    for (i = 0; i < ni; i += 4)
      for (ii = i; ii < i + 4; ii++)
S1:       C[ii] = 0;
```

This transformation is always legal

Loop Unroll-and-Jam

Example (original)

```
   for (i = 0; i < ni; i++)
     for (j = 0; j < nj; j++)
S1:    C[i][j] = 0;
```

Example (uaj(i, j, 2×2))

```
   for (i = 0; i < ni; i += 2)
     for (j = 0; j < nj; j += 2) {
S1:    C[i][j] = 0;
S1:    C[i][j+1] = 0;
S1:    C[i+1][j] = 0;
S1:    C[i+1][j+1] = 0;
     }
```

This transformation may be illegal

Loop Tiling

Example (original)

```
    for (i = 0; i < ni; i++)
      for (j = 0; j < nj; j++)
S1:     C[i][j] = 0;
```

Example (tile(i, j, 2×2))

```
    for (i = 0; i < ni; i += 2)
      for (j = 0; j < nj; j += 2)
        for (ii = i; ii < i + 2; ii++)
          for (jj = j; jj < j + 2; jj++)
S1:           C[ii][jj] = 0;
```

This transformation may be illegal

Remarks

- Be careful about matching loop bounds and divisibility by the stride
  - Ex: for tiling, the good loop bound for `ii` is `ii < min(ni, i+2)`
- Fusion/distribution on non-matching loop bounds is properly defined in the polyhedral model (using min/max for the loop bounds)
- Transformations can be composed in sequence
  - Example for dgemm: `fuse(S1,S2);tile(i,j,32,32)`

Loop Parallelization (OpenMP)

Example (original)

```
    for (i = 0; i < ni; i++)
      for (j = 0; j < nj; j++)
S1:     C[i][j] = 0;
```

Example (omp(i))

```
    #pragma omp parallel for private(j)
    for (i = 0; i < ni; i++)
      for (j = 0; j < nj; j++)
S1:     C[i][j] = 0;
```

This transformation may be illegal

Loop Vectorization

Example (original)

```
    for (i = 0; i < ni; i++)
      for (j = 0; j < nj; j++)
S1:     C[i][j] = 0;
```

Example (vectorize(j))

```
    for (i = 0; i < ni; i++)
      for (j = 0; j < nj; j += 4)
S1:     C[i][j:0-3] = [0:3];
```

Concluding Remarks

- Determining the legality of a loop transformation requires data dependence anlysis
- Some transformations are composition of other, basic ones
  - Need to effectively compose the transformations
  - Search space is infinite
- Applying loop transformations can be challenging
  - non-matching loop bounds
  - control dependences, gotos, ...
  - imperfectly nested loops
- Current compiler framework are limited (work on subset of programs)