Imperfectly Nested Loop Transforms

by

Induprakas Kodukula

→ Motivation

- Framework
- Expressiveness
- An Application - Tiling
- Future Work and Conclusion

Compiling for High performance hardware

Apply code transformations to

- Optimize for data locality for uni- and multi- processors.
- Obtain for parallelization for multiprocessors.

Why Imperfectly nested loops?

- Most numerical solvers are imperfectly nested.
- Existing technology inadequate for imperfectly nested loops.
- Unified framework for loop transformations desirable for imperfectly nested loops.

- Motivation

$\rightarrow$ Framework

- Expressiveness
- An Application - Tiling
- Future Work and Conclusion

Limitations of existing technology

Cannot handle imperfectly nested loop transformations in a systematic manner.

Gaussian Elimination

```
for (i = 0; i < 100; i++) {
  for (j = i+1; j < 100; j++)
  pivot = A[j][i] / A[i][i];
  for (k = i; k < 100; k++)
    A[j][k] -= A[i][k] * pivot;
}
```

Review of perfectly nested loops

- Formulation of an integer space
- Program order $\rightleftharpoons$ Lexicographic order in integer space
- Dependece $\rightleftharpoons$ Lexicographically positive dependence vectors
- Loop Transformations $\rightleftharpoons$ Matrices
- Transformations legal iff dependences are preserved
- Automatic Code generation from transformation matrix

Example - Loop reversal

```
for (i = 1; i < 100; i++)
  S1: A[i] = A[i-1];
```

Iteration space = $1 \ldots 10$

Initial Dependence vector = [1]

Transformation = [-1]

Final Dependence Vector = [-1]

So, Loop reversal is illegal

Nice things about previous framework

- Transformations as matrices
- Transformations are composable
- Easy test for legality
- "Good" transformations can be chosen easily
- Automatic code generation from transformation

What leads to such nice properties?

- Separate basis vectors for each different object being transformed
- Program order corresponds to lexicographic order in integer space

Intuition

Perfectly Nested Loops	Imperfectly Nested Loops
□ Iteration space formulation	□ Iteration space formulation
□ Program Order	□ Program Order
□ Dependence	□ Dependence
□ Transformations as matrices	□ Transformations as matrices
□ Legality	□ Legality

Iteration Space formulation - Dictionary Order

```
for (i = 0; i < 100; i++) {
  S1:
  S2:
  S3:
}
```

Necessary:

- Capture program order between loop iterations
- Capture program order between multiple statements

S1: $\begin{bmatrix} i \\ 1 \end{bmatrix}$, S2: $\begin{bmatrix} i \\ 2 \end{bmatrix}$, S3: $\begin{bmatrix} i \\ 3 \end{bmatrix}$, S4: $\begin{bmatrix} i \\ 4 \end{bmatrix}$

```
for (i=0;i<100;i++) {                for (i=0;i<100;i++) {
  S1:                                  S2:
  S2:                        =>        S1:
  S3:                                  S3:
}                                    }
```

$$\begin{bmatrix} i \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} i \\ 2 \end{bmatrix}, \begin{bmatrix} i \\ 2 \end{bmatrix} \rightarrow \begin{bmatrix} i \\ 1 \end{bmatrix}, \begin{bmatrix} i \\ 3 \end{bmatrix} \rightarrow \begin{bmatrix} i \\ 3 \end{bmatrix},$$

So, one dimension per statement:

$$\text{S1:} \begin{bmatrix} i \\ 0 \\ 0 \\ 1 \end{bmatrix}, \text{S2:} \begin{bmatrix} i \\ 0 \\ 1 \\ 0 \end{bmatrix}, \text{S3:} \begin{bmatrix} i \\ 1 \\ 0 \\ 0 \end{bmatrix},$$

Transformation:

$$\text{S1: } \begin{bmatrix} i \\ 0 \\ 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} i \\ 0 \\ 1 \\ 0 \end{bmatrix}, \text{ S2: } \begin{bmatrix} i \\ 0 \\ 1 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} i \\ 0 \\ 0 \\ 1 \end{bmatrix}, \text{ S3: } \begin{bmatrix} i \\ 1 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} i \\ 1 \\ 0 \\ 0 \end{bmatrix},$$

$$\text{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

```
for (i = 0; i < 100; i++) {
  S1: A[i][i] = 100;
  for (j = i+1; j < 100; j++) {
    S2: A[i][j] /= A[i][i];
  }
}
```

S1: $\begin{bmatrix} i \\ 0 \\ 1 \end{bmatrix}$ j loop: $\begin{bmatrix} i \\ 1 \\ 0 \end{bmatrix}$ so, S2: $\begin{bmatrix} i \\ 1 \\ 0 \\ j \end{bmatrix}$ Pad S1 to $\begin{bmatrix} i \\ 0 \\ 1 \\ 0 \end{bmatrix}$

Syntax tree based computation easy

Program Order

```
for (i = 0; i < 100; i++) {
  A[i][i] = 100;
  for (j = 0; j < i; j++)
    A[i][j] /= A[i][i];
}
```

Simple inductive argument

Dependences

Dependences are lexicographically positive

```
for (i = 0; i < 100; i++) {
  A[i][i] = 100;
  for (j = 0; j < i; j++)
    A[i][j] /= A[i][i];
}
```

$$\text{Dependence Vector} = \begin{bmatrix} i \\ 1 \\ 0 \\ j \end{bmatrix} - \begin{bmatrix} i \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ -1 \\ j \end{bmatrix}$$

Legality

Transformation T is legal iff

For all dependence vectors d in the initial code, T $* \vec{d} > \vec{0}$

- Motivation
- Framework

$\longrightarrow$ Expressiveness

- An Application - Tiling
- Future Work and Conclusion

Fission

```
for (i = 0; i < 100; i++)
  S1: A[i] = 101;
  S2: B[i] = A[i];
```

$\downarrow$

```
for (i = 0; i < 100; i++)
  S1: A[i] = 101;
for (i = 0; i < 100; i++)
  S2: B[i] = A[i];
```

$$\begin{pmatrix} i \\ 0 \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1 \\ i \end{pmatrix}, \begin{pmatrix} i \\ 1 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 0 \\ i \end{pmatrix} \mathbf{T} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Legality of fission

Dependence in the initial code is

$$\begin{pmatrix} i \\ 1 \\ 0 \end{pmatrix} - \begin{pmatrix} i \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}$$

Transformed dependence vector is

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}$$

Since the transformed dependence vector is $> \vec{0}$, the transformation is legal.

fusion

```
for (i = 0; i < 100; i++)
  S1: A[i] = 101;
for (i = 0; i < 100; i++)
  S2: B[i] = A[i];
```

$\downarrow$

```
for (i = 0; i < 100; i++)
  S1: A[i] = 101;
  S2: B[i] = A[i];
```

$$\begin{pmatrix} 0 \\ 1 \\ i \end{pmatrix} \rightarrow \begin{pmatrix} i \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ i \end{pmatrix} \rightarrow \begin{pmatrix} i \\ 1 \\ 0 \end{pmatrix} \mathbf{T} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Legality of fusion

Dependence in the initial code is

$$\begin{pmatrix} 1 \\ 0 \\ i \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ i \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}$$

Transformed dependence vector is

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}$$

Since the transformed dependence vector is $> \vec{0}$, the transformation is legal.

Imperfect Loop Interchange

```
for (i = k+1; i < N; i++) {
  S1: A[i][i] = A[i-1][k+1] * A[k][k];
  for (j = k+1; j < N; j++) {
    S2: A[i][j] = A[i][j] + A[i][k] * A[k][j];
  }
}
```

$\downarrow\uparrow$

```
for (j = k+1; j < N; j++) {
  S1: A[j][j] = A[j-1][k+1] * A[k][k];
  for (i = k+1; i < N; i++) {
    S2: A[i][j] = A[i][j] + A[i][k] * A[k][j];
  }
}
```

Motivation for loop interchange

- Unit stride access in inner loops
- Necessary to tile imprefectly nested loops

General Transformations

```
for (k = 0; k < N; k++) {
  A[k][k] = sqrt(A[k][k]);
  for (i = k+1; i < N; i++)                    Right Looking Choleky Solver
    A[i][k] /= A[k][k];
  for (j = k+1; j < N; j++)
    for (l = k+1; l < j; l++)
      A[j][l] -= A[j][k] * A[l][k];
}
```

↓

```
for (k = 0; k < N; k++) {
  for (j = k; j < N; j++)
    for (l = 0; l < k; l++)                    Left Looking Cholesky Solver
      A[j][k] -= A[j][l] * A[k][l];
  A[k][k] = sqrt(A[k][k]);
  for (i = k+1; i < N; i++)
    A[i][k] /= A[k][k];
}
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 \\ -1 \\ 1 \\ 0 \\ 0 \\ * \\ * \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ -1 \\ * \\ 1 \\ * \\ 1 \end{pmatrix}$$

Dependence from last iteration of update loop of previous iteration to the square root step from the next iteration

Motivation for general transformations

Right Looking Cholesky	Left Looking Cholesky
80.9	20.1

Table 1: Run time (sec)

On an HP Snake 9000/712, cc +O4 (version - A.09.61), on input matrices 1000x1000 doubles

- Motivation
- Framework
- Expressiveness

→ An Application - Tiling

- Future Work and Conclusion

Tiling

Claim:

If transformation from code A to code B is invertible, but non linear, but transformation from code B to code A is linear and invertible and preserves dependences, then code A and code B have the same dependences.

```
for i = 0..100,1
  for j = 0..100,1
    A[i][j] = 0;
```

```
for i = 0..100, 10
  for j = 0..100, 10
    for itile = i..i+9,1
      for jtile = j..j+9,1
        A[itile][jtile] = 0;
```

Example

```
for (k = 0; k < 100; k++) {
  A[k][k] = sqrt(A[k][k]);
  for (i = k+1; i < 100; i++)
    A[i][k] /= A[k][k];
  for (j = k+1; j < 100; j++)
    for (l = k+1; l < j; l++)
      A[j][l] -= A[j][k] * A[l][k];
}
```

Notes on the previous slide

- The transformations involved are distribution and index set splitting. The reverse transformation is linear and invertible.
- Last part of the code is perfectly nested, and can be trivially tiled for j and l as well.

Performance

Program Name	Untiled(s)	Tiled(s)	Improvement(%)
Right Looking Cholesky	80.9	37.5	54
Left looking Cholesky	20.1	11.9	41
LAPACK cholesky	21.2	16.9	20

Summary of tiling

Issues:

- Is tiling of code same as blocking of algorithms in libraries?
- Choice of optimal tile size
- Heuristics for obtaining good index set splits (use a greedy strategy now)

- Motivation
- Framework
- Expressiveness
- An Application - Tiling

$\rightarrow$ Future Work and Conclusion

Future Work - Code Generation

Framework needs three parts:

- Legality
- Utility
- Code generation.

We have provided the first. We will develop techniques for others.

Issues:

- Dimensions of Transformations.
- Sizes of the transformation matrices.

Generation and completion procedures

- Generate partial transformations from desired goal
- Complete partially generated tramsformations

Determine techniques that will help us to get performance comparable to handwritten code.

Is imperfectly nested loop tiling it for LAPACK?

Conclusion

- Systematic framework for imperfectly nested loops
- Expressive framework
- Application of framework to find transformations necessary to approach performance of library codes written by hand
- Example: Imperfect-loop tiling lets us acheive performance comparable to (sometimes better than) handwritten library of solvers from LAPACK