CHiLL

Authors:
Chun Chen, Jacqueline Chame and Mary Hall

Justin Szaday, CS598APK, October 5th, 2018

Introduction

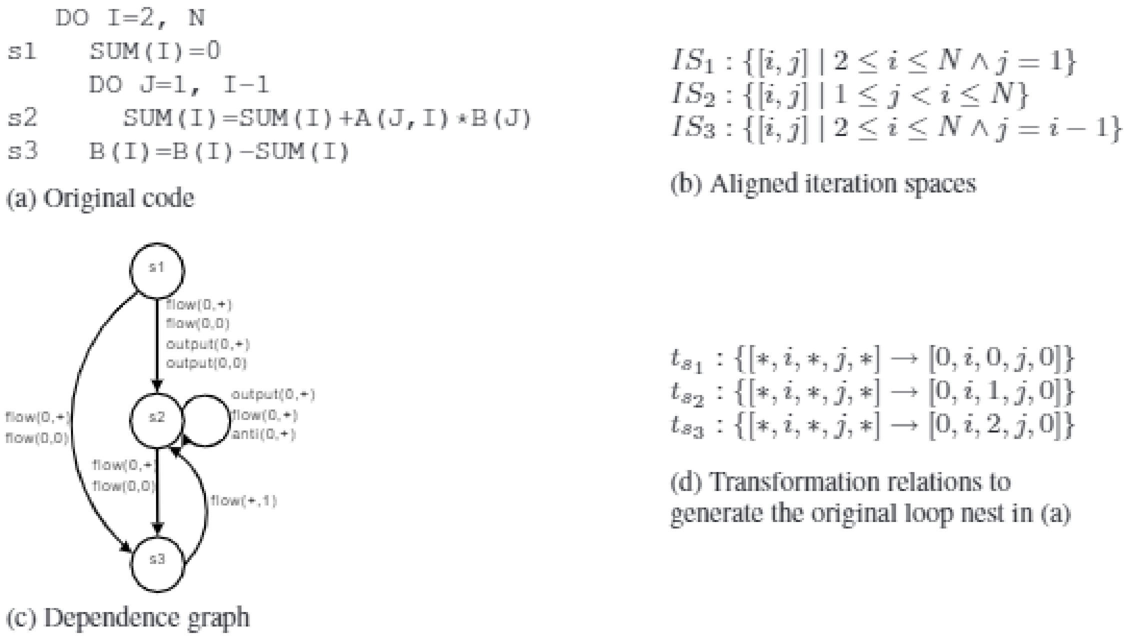
- Source-level loop transformations are still necessary for compilers to produce high-quality code[1]
- Manually applying transformations is tedious, and makes the source code harder to read
- Enter CHiLL, a source-to-source translator for Composing High-Level Loop transformations

[1] An empirical study of the effect of source-level transformations on compiler stability, Zhangxiaowen Gong, Zhi Chen, Justin Szaday, et al., CPC2018

Overview

- CHiLL lets users compose high-level loop transformations with ease
- CHiLL supports loops written in/with C/C++, CUDA and Fortran
- CHiLL's operations are driven by a user-supplied transformation script
- CHiLL verifies that all user-specified transformations preserve the dependences between the statements of the original code
- CHiLL uses a polyhedral loop representation with support for complex loop nests (via CodeGen+ and Omega+)
  - Thus, CHiLL does not need to generate intermediate code or rebuild the dependence graph between transformations

Loop Representation in CHiLL



```
   DO I=2, N
s1   SUM(I)=0
     DO J=1, I-1
s2     SUM(I)=SUM(I)+A(J,I)*B(J)
s3   B(I)=B(I)-SUM(I)
```

(a) Original code

$$IS_1 : \{[i, j] \mid 2 \leq i \leq N \wedge j = 1\}$$
$$IS_2 : \{[i, j] \mid 1 \leq j < i \leq N\}$$
$$IS_3 : \{[i, j] \mid 2 \leq i \leq N \wedge j = i - 1\}$$

(b) Aligned iteration spaces

(c) Dependence graph

$$t_{s_1} : \{[*, i, *, j, *] \rightarrow [0, i, 0, j, 0]\}$$
$$t_{s_2} : \{[*, i, *, j, *] \rightarrow [0, i, 1, j, 0]\}$$
$$t_{s_3} : \{[*, i, *, j, *] \rightarrow [0, i, 2, j, 0]\}$$

(d) Transformation relations to generate the original loop nest in (a)

Figure from: Chen, Chun & Chame, Jacqueline & Hall, Mary. (2008). *A Framework for Composing High-Level Loop Transformations*.

Transformation Script

- The transformation script is written in Python, and specifies the:
  - Location of the source file
  - Function and loops to modify
  - Known properties of variables
  - Transformations to apply
- The transformations' parameters include: sets of statements, loops, orders, factors, etc.

```
from chill import *
source('mm.c')
destination('mm_modified.c')
procedure('mm')
loop(0)
known(['ambn > 0', 'an > 0', 'bm > 0'])
permute([3,2,1])
print_code()
```

Example

Original:

```
void mm(float **A, float **B, float **C,
     int ambn, int an, int bm) {

  int i, j, n;

  for(i = 0; i < an; i++)

    for(j = 0; j < bm; j++) {

      C[i][j] = 0.0f;

      for(n = 0; n < ambn; n++)

        C[i][j] += A[i][n] * B[n][j];

    }

}
```

mm.py:

```
from chill import *

source(’mm.c’)

procedure(’mm’)

loop(0)

known([’ambn > 0’, ’an >
   0’, ’bm > 0’])

distribute([0,1],1)

print_code()
```

Transformed:

```
void mm(float **A, float **B, float **C,
     int ambn, int an, int bm) {

  int i, j, n;

  for(i = 0; i <= an - 1; i++)

    for(j = 0; j <= bm - 1; j++)

      C[i][j] = 0.0f;

  for(i = 0; i <= an - 1; i++)

    for(j = 0; j <= bm - 1; j++)

      for(n = 0; n <= ambn - 1; n++)

        C[i][j] += A[i][n] * B[n][j];

}
```

```
distribute(set<int> stmts, int loop)
```

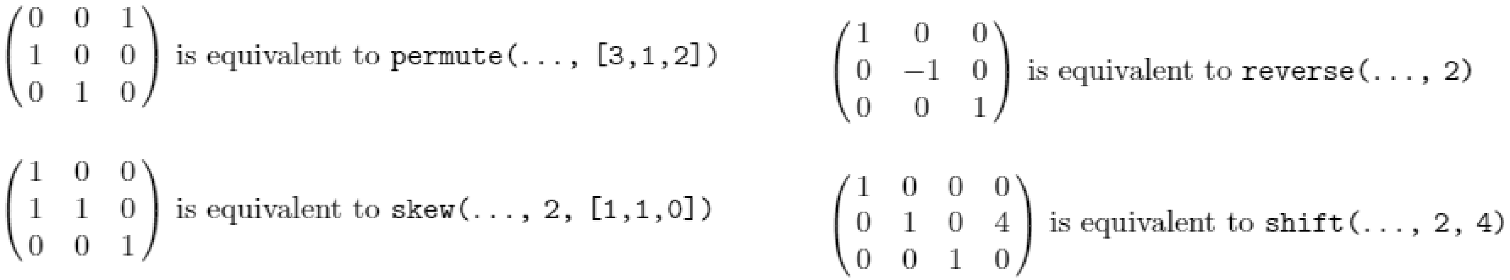
Transformations

```
distribute(set<int> stmts, int loop)

fuse(set<int> stmts, int loop)

nonsingular(matrix transform)

peel(int stmt, int loop,
     int amount = 1)

permute(set<int> stmts, vector<int> p)

reverse(set<int> stmts, int level)

scale(set<int> stmts, int loop,
      int amount)

shift(set<int> stmts, int loop,
      int amount)

shift_to(int stmt, int loop, int amount)

skew(set<int> stmts, int loop,
     vector<int> amounts)

split(int stmt, int loop, string expr)

tile(int stmt, int loop, int tile_size)

unroll(int stmt, int loop, int unroll_amount)
```

Transformations (details)

`nonsingular(matrix transform)`

Applies a unimodular or nonunimodular transformation on a perfect loop nest, affecting all statements in the loop nest. The only requirement for the matrix is that it be invertible.



$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$ is equivalent to `permute(..., [3,1,2])`

$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ is equivalent to `reverse(..., 2)`

$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ is equivalent to `skew(..., 2, [1,1,0])`

$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ is equivalent to `shift(..., 2, 4)`

Figure from: *The Composable High Level Loop Source-to-Source Translator*.

Transformations (details)

```
split(int stmt, int loop, string expr)
```

Divide a loop’s iteration space using the condition specified by *expr*, only one expression is allowed (and it cannot contain logical operators).

In

```
for(i = 0; i < an; i++)
  …
```

Transform

```
split(…, n, “Ln < 5”)
```

Out

```
for(i = 0; i < min(an, 5); i++)
  …
for(i = 5; i < an; i++)
  …
```

Transformations (details)

`unroll(int stmt, int loop, int amount)`

Unrolls a statement by a specified number of iterations. Adds cleanup code if necessary.

In

```
for(i = 0; i < an; i++)
  S1(i)
```

Transform

```
known('an % 2 == 0')
unroll(1, n, 2)
```

Out

```
for(i = 0; i < an; i += 2)
  S1(i)
  S1(i+1)
```

Other Features of CHiLL

- Users can bypass CHiLL's dependence analysis by removing dependences from dependence graph with: `remove_dep(int stmt1, int stmt2)`
- Users can print the dependences between all statements with: `print_dep()`
- Users can display the iteration spaces for each statement with: `print_space()`

Loop and Statement Identification

- The outermost loop of a nest is always loop level 1
- Individual loops within a loop nest are identified by their nesting level and the statement(s) that they surround
- Statements are numbered in the order they appear from top to bottom starting with zero
- The identification of a statement will not change after a transformation

```
for (i ...) {
  S0
  for (j ...) {
    S1
    for (k ...)
      S2
  }
  for (j ...)
    S3
}
```

Limitations of CHiLL

- Changes to the source code may change the identifications assigned the loops and statements
  - Breaks pre-existing CHiLL scripts, an alternative would be to have users tag loops with invariant tags
- As a source-to-source translator, CHiLL has limited bearing on the code produced by compilers
  - It cannot, for example, insert pragmas or prefetch instructions into the generated code
  - Compilers may undo transformations performed by CHiLL
- Requires enough knowledge of the underlying hardware to generate an optimization strategy

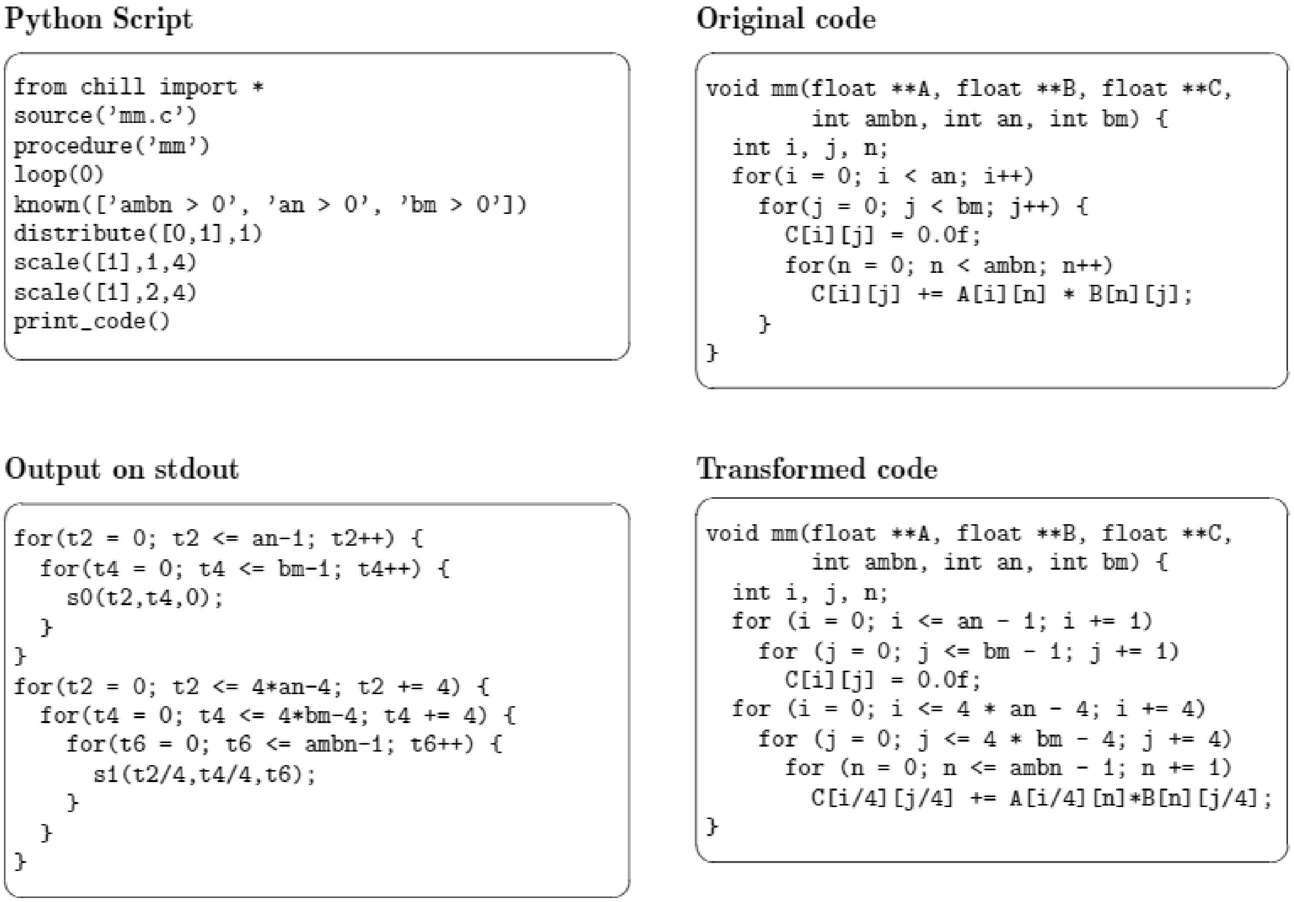
Applications of CHiLL

- CHiLL has been used as the backend for auto-tuning frameworks, such as Active Harmony (Chen, 2009)
- Employs empirical search to identify a variation that best meets a specific optimization criteria, usually performance
- Active Harmony with CHiLL auto-tuned:
  - Matrix Multiply (MM), for a 2.36x speedup
  - Triangular Solver (TRSM), for a 3.62x speedup
  - Jacobi, for a 1.35x speedup
- MM performed within 20% of ATLAS (a self-tuning library)

Conclusions

- Transformations can improve the performance of programs
- CHiLL allows users to apply transformations to their programs in an easy way, that does not affect readability
- CHiLL automatically verifies the correctness of user-specified transformations using dependence analysis
- CHiLL has a reasonably complete set of transformations, encompassing most of the common transformations

Multi-Transform Example



Python Script

```
from chill import *
source('mm.c')
procedure('mm')
loop(0)
known(['ambn > 0', 'an > 0', 'bm > 0'])
distribute([0,1],1)
scale([1],1,4)
scale([1],2,4)
print_code()
```

Original code

```
void mm(float **A, float **B, float **C,
        int ambn, int an, int bm) {
  int i, j, n;
  for(i = 0; i < an; i++)
    for(j = 0; j < bm; j++) {
      C[i][j] = 0.0f;
      for(n = 0; n < ambn; n++)
        C[i][j] += A[i][n] * B[n][j];
    }
}
```

Output on stdout

```
for(t2 = 0; t2 <= an-1; t2++) {
  for(t4 = 0; t4 <= bm-1; t4++) {
    s0(t2,t4,0);
  }
}
for(t2 = 0; t2 <= 4*an-4; t2 += 4) {
  for(t4 = 0; t4 <= 4*bm-4; t4 += 4) {
    for(t6 = 0; t6 <= ambn-1; t6++) {
      s1(t2/4,t4/4,t6);
    }
  }
}
```

Transformed code

```
void mm(float **A, float **B, float **C,
        int ambn, int an, int bm) {
  int i, j, n;
  for (i = 0; i <= an - 1; i += 1)
    for (j = 0; j <= bm - 1; j += 1)
      C[i][j] = 0.0f;
  for (i = 0; i <= 4 * an - 4; i += 4)
    for (j = 0; j <= 4 * bm - 4; j += 4)
      for (n = 0; n <= ambn - 1; n += 1)
        C[i/4][j/4] += A[i/4][n]*B[n][j/4];
}
```