CHiLL

The Composable High Level Loop Source-to-Source Translator
For version 0.2.3

This manual describes CHiLL (version 0.2.3), a source-to-source translator for optimizing loop based calculations.

Published by the University of Utah School of Computing
Compiler Technology to Optimize Performance Research Group
`ctop.cs.utah.edu/ctop/`
50 S., Central Campus Dr., Salt Lake City, UT 84112

Table of Contents

1 Introduction

CHiLL is a source-to-source translator for composing high level loop transformations to improve the performance of nested loop calculations written in C, C++ or Fortran. CHiLL's operations are driven by a script which is generated or supplied by the user that specifies the location of the original source file, the function and loops to modify and the transformations to apply. CHiLL can be configured to include support for the NVIDIA CUDA compiler. In this mode, CHiLL can generate source code for both host functions and device functions to be compiled and executed on NVIDIA GPUs.

1.1 Intended Audience

This manual is intended for C/C++ or Fortran programmers wishing to optimize loop based calculations. The user should have sufficient knowledge of the underlying hardware on which the code should execute to generate an optimization strategy.

1.2 Getting CHiLL

CHiLL is available in source form from `https://github.com/CtopCsUtahEdu` and requires the ROSE compiler from Lawrence Livermore National Laboratory (see `http://rosecompiler.org`)[1]. When ROSE is available CHiLL can be installed and tested by executing the following commands in the source direcory.

```
./configure --with-interface=python --prefix=<INSTALLDIR> \
            --with-rose=<ROSEINSTALLDIR> --with-boost=<BOOSTINSTALLDIR>
make -j`nproc`
make -j`nproc` install
cd test-chill; ./runtests
```

If you have problems with installation, find bugs or have comments, questions or suggestions for this document, please send mail to `chill-support@cs.utah.edu`.

1.3 Invoking CHiLL

The C program below is an implementation of matrix multiplication as a direct translation of an optimized Fortran program where all of the loops are ordered such that memory accesses to the arrays a, b and c are all in column order. Since C stores arrays in row major order there is an opportunity for better cache utilization if the arrays are accessed as rows and not columns. We will refer to this code often and assume it is in a file named `mm.c`.

```
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];
    }
}
```

[1] Building ROSE requires very specific versions of GNU autoconf, gcc and the boost libraries. If you do not have ROSE installed then please see and modify the script `buildall` which was used to install CHiLL on Blue Waters at NCSA.

Permuting the order of the loops from i, j, n to n, j, i results in the more cache centric C algorithm as shown below where all array accesses are in row major order.

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

The interchanges of the outer and innermost loops can be done in CHiLL with this simple Python script.

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

The first line of the script '`from chill import *`' loads the CHiLL interface into the python interpreter. The commands `source` and `procedure` identify the source file and the procedure to modify. The `loop` command specifies the loop nest to be transformed. The `known` command specifies constraints on parameters that are known and the transformation '`permute([3,2,1])`' exchanges the inner and outermost loops. Finally the command `print_code` prints the transformed loop nest in a C-like pseudo code showing the loops, indices and statements.

Assuming for this example that that the script above is in the file `mm.py`, the command '`chill mm.py`' would print to *stdout* pseudo code similar to that shown below and produce the transformed code in the file `rose_mm.c`[1]

```
for(n = 0; n <= ambn-1; n++)
  for(j = 0; j <= bm-1; j++)
    if (n <= 0)
      for(i = 0; i <= an-1; i++) {
        s0(i,j,n);
        s1(i,j,n);
      }
    else
      for(i = 0; i <= an-1; i++)
        s1(i,j,n);
```

[1] The code produced by the current version of CHiLL does not preserve loop variables names in the transformed code which makes it difficult to see the effects of a transformation. In this manual we have used the original loop variable names in the generated code to make it easier to understand.

2 Background

Before CHiLL applies a user specified transformation to the loop structure it first insures that the transformed code will produce the same results as the original code. It does this by determining all dependences between statements in the original program and then requiring that any and all transformations that are applied preserve the dependences between the statements in the original code.

Conceptually CHiLL treats each statement in a source program as one of three basic types; a loop, a conditional or a statement. When we refer to "a statement" in CHiLL, we are referring to a block of one or more actual program statements which have a single uninterrupted execution path through them and we notationally represent it as a function which is passed the values of the indices of all loops enclosing it.

For each statement we compute an *iteration vector* that encodes the the absolute execution order of the statement as a function of its lexical position in the source code and the index values of the enclosing loops. We then define the *iteration space* for the statement by joining the iteration vector with the constraints on each index that is in an enclosing loop.

Next we analyze the memory access patterns of the statements and loops. We compute the set of *dependences* by taking all statements pair-wise and finding those pairs of statements (S_1, S_2) where there exists an iteration vector i_1 and i_2 in the respective iteration spaces of S_1 and S_2 such that $S_1(i_1)$ and $S_2(i_2)$ both refer to the same memory location and one or both of them write to that location. The *distance vector* defined by $i_2 - i_1$ gives the execution distance from the source statement S_1 to the sink statement S_2.

If a dependence exists between statements S_1 and S_2 with the constraint that S_1 must execute before S_2 in the original code then that constraint must be preserved across any and all transformations. If a dependence exists that can not be preserved across a transformation then CHiLL alerts the user to this problem. Dependence information between S_1 and S_2 is maintained by a *dependence vector* which encapsulates the notion of all the distance vectors where statement S_1 must execute before S_2.

The diagnostic commands `print_space` and `print_dep` will print the iteration space of each statement and the dependences between all pairs of statements. The command `remove_dep` will force the removal of a dependence leaving responsibility for the correctness of the transformation to the user.

2.1 Iteration Vectors

Given a loop nest with a maximum loop depth of n, we define for each executed statement an iteration vector that encodes the time of execution of a statement executed with specific values for the loop indices which enclose the statement. This allows us to determine the relative order of execution of any two statements so that dependences between statements can be preserved.

We define an iteration vector for a nest of n loops as $i = \{c_0, l_1, c_1, l_2, ..., c_n, l_n, c_{n+1}\}$ where l_k is the value of the index[1] of the loop at nesting level k and c_k is an auxiliary loop used to track the lexigraphical ordering of statement executed within the loop nested at level

[1] with a suitable transformation such that the index is monotonically increasing as the loop progresses

k. The outermost loop level in the nest is 1 and c_0 lexigraphically orders any statements that precede the first loop.

At this level of abstraction we only care about loops and blocks of code between loops. The even numbered elements $\{c_0, c_1, ..., c_{n+1}\}$ are always constant integers that describe the static lexigraphical ordering of the statements in the original code. The odd numbered elements $\{l_1, ..., l_n\}$ represent the current values of the loop levels. This scheme allows a uniform method to both track both the progression of the loop indices as well as the execution order of statements within each loop.

Iteration vectors are ordered and thus can be used to enforce dependences between statements. Iteration vector i *precedes* iteration j, denoted $i < j$, if and only if $i[1 : n-1] < j[1 : n-1]$ or $i[1 : n-1] = j[1 : n-1]$ and $i[n] < j[n]$.

Given two statements S_0 which executes at a time specified by iteration vector i_0 and a statement S_1 which executes at a time specified by iteration vector i_1, then the execution of S_0 precedes that of S_1 if and only if $i_0 < i_1$.

2.2 Iteration Spaces

Consider the following loop nest below. There are three loop levels to track the three indices i, j and k and four auxiliary loop levels to track the relative execution of the statements within the loops.

```
S0
for (i ...) {
  S1
  S2
  for (j ...) {
    S3
    for (k ...) {
      S4
    }
    S5
  }
  S6
}
```

An iteration space is is a set of iteration vectors. It is usually specified in set notation with one or more values of l specified as an integer variable along with constraints on the variables. The iteration space for each statement is shown below as a set of integer tuples. In practice, the upper and lower bounds of each loop index would be specified in each set condition as well.

$$
\begin{array}{ll}
S_0: & \{[0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0]\} \\
S_1: & \{[1,\ i,\ 0,\ 0,\ 0,\ 0,\ 0]\} \\
S_2: & \{[1,\ i,\ 1,\ 0,\ 0,\ 0,\ 0]\} \\
S_3: & \{[1,\ i,\ 2,\ j,\ 0,\ 0,\ 0]\} \\
S_4: & \{[1,\ i,\ 2,\ j,\ 1,\ k,\ 0]\} \\
S_5: & \{[1,\ i,\ 2,\ j,\ 2,\ 0,\ 0]\} \\
S_6: & \{[1,\ i,\ 3,\ 0,\ 0,\ 0,\ 0]\}
\end{array}
$$

If we were were told that the current point of execution of the above loop nest was described by the iteration vector [1, 3, 2, 6, 2, 0, 0] we would know that statement S_5 was executing with the indices of the loops being i = 3 and j = 6.

The `print_space` command will print the iteration space for every statement (or block of statements). For example `print_space` applied to the following code.

```
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];
  }
```

gives the following results.

```
s0: {Sym=[bm,an,ambn] [t1,t2,t3,t4,t5,t6,t7] : t1=0 && t3=0 && t5=0 &&
     t7=0 && t6=0 && 0<=t2<an && 0<=t4<bm && 1<=ambnyes }
s1: {Sym=[ambn,bm,an] [t1,t2,t3,t4,t5,t6,t7] : t1=0 && t3=0 && t5=0 &&
     t7=0 && 0<=t2<an && 0<=t6<ambn && 0<=t4<bm }
```

2.3 Dependences

There are two general categories of dependences, control dependences and data dependences.

A *control dependence* exist when one statement is executed conditionally on the result of another. For example, in the statements below S_1 cannot be executed before S_0 and thus S_1 has a control dependence on S_0.

```
S0    if (x != 0)
S1      a /= x;
```

A *data dependence* exists between statements S_0 and S_1 (meaning S_1 depends on statement S_0) if and only if there is a plausible run-time execution path from S_0 to S_1, both statements access the same memory location and at least one of them stores to it. There are three types of data dependences:

A *true dependence* exists when S_0 writes to a location that is later read by S_1.

```
S0    x = ...
S1    ... = x
```

An *antidependence* exists when S_0 reads from a location that is later written to by S_1.

```
S0    ... = x
S1    x = ...
```

An *output dependence* exists when S_0 writes to a location that is later written to by S_1.

```
S0    x = ...
S1    x = ...
```

In the parlance of hardware design, a true dependence is known as a RAW (read after write) hazard, an antidependence is a WAR (write after read) hazard and an output dependence is a WAW (write after write) hazard. These dependences are fairly intuitively and are used instinctively by every programmer to determine the correct order of statements in sequential code. However, when loops and arrays are involved these same data dependences arise in more subtle ways.

2.4 Dependences with loops and arrays

It is useful to categorize data dependences in loops into two types. Consider a loop that contains two statements call them S_0 and S_1. If both S_0 and S_1 reference the same memory location within the same iteration as they do in this case, then the dependence is

loop-independent. If statement S_0 and S_1 reference the same memory location in different iterations, then the dependence is created by the loop and it is said to be *loop-carried*.

As an example the loop below which has two loop-carried dependences and two loop-independent dependences.

```
        for (i = 0; i < n; i++) {
S0        a[i + 1] = b[i];
S1        b[i + 1] = a[i];
S2        c[i] = a[i] + b[i];
        }
```

In every iteration other than the first, S_1 reads an element of `a[]` that was written to by S_0 in the previous iteration and thus there is dependence from S_0 to S_1. Because S_0 appears before S_1 in the loop, it is a *loop-carried forward* true dependence.

Likewise, for every iteration other than the first, S_0 uses a value of `b[]` that was written by S_1 in the previous iteration and thus there is also a dependence from S_1 to S_0 but because S_1 appears after S_0 in the loop, it is a *loop-carried backward* true dependence.

Finally S_2 reads a value of `a[]` that was written by S_0 and a value of `b[]` that was written by S_1 in the same iteration and thus there are two *loop-independent* true dependence, one is S_2 on S_0 and the other is of S_2 on S_1.

2.5 Distance Vectors

Given two statements S_1 and S_2 with iteration vector i_1 and i_2 respectively where S2 depends on S1 we define the distance vector d of the dependence as follows. Let

$$i_1 = \{c'_0,\ l'_1,\ c'_1,\ l'_2,\ ...,\ c'_n,\ l'_n,\ c'_{n+1}\} \texttt{ and}$$
$$i_2 = \{c_0,\ l_1,\ c_1,\ l_2,\ ...,\ c_n,\ l_n,\ c_{n+1}\}$$

then we define the distance vector as

$$d = \{l_1 - l'_1,\ l_2 - l'_2,\ ...,\ l_n - l'_n\}.$$

The only statement in the nested loop below has a loop-dependent dependence with itself.

```
for(i = 0; i < 2 * n; i++)
  for (j = 1; j < m; j++)
    a[i+1][j-1] = a[i][j] + b[i][j];
```

The right hand side of the assignment reads array a at iteration vector i_R below and the left hand side of the assignment writes to the same location of the array at the iteration vector i_W.

$$i_R = \{0,\ i,\ 0,\ j,\ 0\}$$
$$i_W = \{0,\ i+1,\ 0,\ j-1,\ 0\}.$$

Because $i_R < i_W$ this is a read/write dependence and the distance vector is

$$d = i_W - i_R = \{1,\ -1\}.$$

The command `print_dep` applied to this loop yields '`s0->s0: a:true(1, -1)`'

2.6 Direction Vectors

In the same way that an iteration space is a set of iteration vectors, a direction vector is a summarization of a set of distance vectors between two statements.

In the matrix multiplication loop nest below, statement S_0 initializes the value of `C[i][j]` and statement S_1 accumulates the inner product into it. S_1 has multiple dependences on S_0 as the initialization must occur before *every* accumulation statement S_1 executed in the loop.

```
      for(i = 0; i < an; i++)
        for(j = 0; j < bm; j++) {
S0        C[i][j] = 0.0f;
          for(n = 0; n < ambn; n++)
S1          C[i][j] += A[i][n] * B[n][j];
        }
```

In general, given a nest with n loops each direction vector has the form $(d_1, d_2, \ldots, d_n)$ where The value of d_i indicates the range of distances between the source and sink carried by loop i and is one of the following symbols below where n, n_l and n_u represent integer values in the iteration space of the loop.

Symbol	**Lower Bound**	**Upper Bound**
n	n	n
$n_l{\sim}n_u$	n_l	n_u
$*$	$-\infty$	$+\infty$
$-$	$-\infty$	-1
$n-$	$-\infty$	n
$+$	1	$+\infty$
$n+$	n	$+\infty$

Notice that neither the symbols + nor − includes the value of 0. The reason is that a distance of 0 means that the dependence is not loop-carried which CHiLL likes to separate from dependences that are loop-carried. This is shown below with output of the command `print_dep` on the above code.

```
s0->s1: C:true(0,0,+) C:true(0,0,0) C:output(0,0,+) C:output(0,0,0)
s1->s1: C:anti(0,0,+) C:output(0,0,+)
```

2.7 Legality of Transformations

From the definition of an iteration vector it is clear that transformations that permute the loop structure also permute the iteration vector and distance vectors of all dependences in exactly the same way. Because the distance vector is the distance from the source to the sink, any permutation of the loop structure that causes the permuted distance vector to be negative is illegal as the transformation has caused the sink to execute before the source.

Looking at the transformed elements of an iteration vector or distance vector from left to right (or outer loop to inner loop), if the first non-zero loop element is negative, then the distance vector is negative and the transformation is illegal.

3 The CHiLL Scripting Language

3.1 Loop and Statement Identification

The first two commands in every CHiLL script identify the source file and the procedure to modify. Only one source file and one procedure can be modified in any single script.

Individual loops within a loop nest are identified by the level that they are nested and the statement that they surround. The outermost loop of a nest is always loop level 1. Thus in the introductory example, the transformation '`permute([3,2,1])`' exchanged the inner and outermost loops.

It is very important to realize that every transformation has the capability to insert or reorder the loops and thus the identification of a specific loop will change after a transformation. In the example referenced above the j, k and i loops that were respectively at levels 1, 2 and 3 before the permutation are now at levels 3, 2 and 1 after the transformation. Any subsequent transformation must use these new loop levels to identify the individual loops.

Consider the pseudo code below.

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

The loop level alone insufficient to uniquely determine a specific loop within a nest unless a statement enclosed by the loop is also provided. Statements are initially numbered in the order they appear from top to bottom in the nest starting with zero. Transformations may also insert or reorder the statements in the nest but the identification of a specific statement will not change after a transformation.

3.2 Commands

`source` (*string* `filename`) [Command]

The `source` command specifies the filename of the original code to be transformed. There can only be one `source` command in a script and it must precede the `loop` command.

`procedure` (*string* `name`) [Command]

The `procedure` command specifies the procedure name in the file to transform. There can only be one procedure modified in a given script.

`loop` (*int* `level`) [Command]
`loop` (*int* `start`, *int* `end`) [Command]

The `loop` command specifies the loop nest to be transformed by specifying the top level loop of the nest. The first form of the command selects a nest contained in a single top level loop. The second form takes a range of top level loops and treats them as a single unified nest.

Top level loops are those loops at procedure scope and are numbered starting from zero. Once a loop nest is selected by the `loop` command the outermost loop of the nest is numbered from one.

The `loop` command can be issued multiple times in a script to select and modify different top level loops within the same file.

`print_code` () [Command]

The `print code` commands display C-like pseudo code of the nest showing the loops, indices and statements. Statements in the pseudo-code appears as a function indexed by the loop indices as if it were a call to the block of code.

`print_dep` () [Command]

The `print_dep` command displays the dependences between all statements in the current nest. Given a nest with n loops each individual dependence has the form:

$$var{:}type(d_1, d_2, \ldots, d_n)$$

where *var* is the variable name that is creating the dependence, *type* is the type of dependence which is one of *_quasi*, *true*, *anti*, *output*, *input*, *control* or *unknown*.

The value of d_i indicates the distance or range of distances where a dependence exists between the source and sink carried by loop i.

`print_space` () [Command]

The `print_space` command displays the iteration space for each statement in the current nest.

`exit` () [Command]

The `exit` command exists the script.

`known` (*string* `cond`) [Command]

The `known` command adds a condition as an expression. The value of *cond* can be a string or a list of strings.

`remove_dep` (*int* `stmt1`, *int* `stmt2`) [Command]

The `remove_dep` removes a dependence between two statements in the loop nest.

3.3 Transformations

Distribute

`distribute` (*set<int>* `stmts`, *int* `loop`) [Transform]

Distribute the set of statements in *stmts* such that each statement executes under its own clone of the common loop structure down to and including level *loop*.

The transformation is legal if and only if, all loop-carried dependences between the statements in *stmts* are contained entirely to loop levels less than *loop*.

Python Script

```
from chill import *
source('mm.c')
procedure('mm')
loop(0)
known(['ambn > 0', 'an > 0', 'bm > 0'])
distribute([0,1], 1)
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 <= an-1; t2++) {
  for(t4 = 0; t4 <= bm-1; t4++) {
    for(t6 = 0; t6 <= ambn-1; t6++) {
      s1(t2,t4,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 <= an - 1; i += 1)
    for (j = 0; j <= bm - 1; j += 1)
      for (n = 0; n <= ambn - 1; n += 1)
        C[i][j] += (A[i][n] * B[n][j]);
}
```

Fuse

`fuse` (*set<int>* `stmts`, *int* `loop`) [Transform]

Fuse the set of statements in *stmts* such that all statement executes under the common loop structure down to and including level *loop*.

Python Script

```
from chill import *
source('dist.c')
procedure('mm')
loop(0, 1)
known(['ambn > 0', 'an > 0', 'bm > 0'])
fuse([0,1], 1)
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 - 1; i += 1)
    for (j = 0; j <= bm - 1; j += 1)
      C[i][j] = 0.0f;

  for (i = 0; i <= an - 1; i += 1)
    for (j = 0; j <= bm - 1; j += 1)
      for (n = 0; n <= ambn - 1; n += 1)
        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);
    s1(t2,t4,0);
    for(t6 = 1; t6 <= ambn-1; t6++) {
      s1(t2,t4,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;
      C[i][j] += (A[i][0] * B[0][j]);
      for (n = 1; n <= ambn - 1; n += 1)
        C[i][j] += (A[i][n] * B[n][j]);
    }
}
```

Nonsingular

`nonsingular` (*matrix* `transform`) [Transform]

The `nonsingular` transformation applies a unimodular or nonunimodular transformation on a perfect loop nest. The only requirement for the matrix is that it be invertible. All statements in the loop nest are effected by the transformation.

Given a perfect loop nest of depth n, with original iteration indexes i and an n x n transformation matrix T, a new set of index vectors i' is formed as $i' = Ti$. If the transformation matrix T is an n x $n+1$ matrix, the last column vector is a constant shift as shown below.

$$\begin{pmatrix} i'_1 \\ i'_2 \\ \vdots \\ i'_n \end{pmatrix} = \begin{pmatrix} t_{11} & t_{12} & \dots & t_{1n} & c_{1,n+1} \\ t_{21} & t_{22} & \dots & t_{2n} & c_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ t_{n1} & t_{n2} & \dots & t_{nn} & c_{n,n+1} \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \\ 1 \end{pmatrix}$$

This transform has the ability to simultaneously compose the transforms of permutation, skew, reverse and shift. For example ...

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

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

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

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

The difference between `nonsingular` and `permute`, `skew`, `reverse` and `shift` is that `nonsingular` can apply combinations of all of the above transformations simultaneously but it must be applied to all statements in the nest. The individual transformations accept a set of statements depicted above with "...".

Python stores arrays in row major order (like C and unlike Fortran) so

the array $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ is written as [[1,0,0,0],[0,1,0,4],[0,0,1,0]].

Original

`original ()` [Transform]

The `original` transformation initializes the loop structure without permuting it.

Python Script

```
from chill import *
source('mm.c')
procedure('mm')
loop(0)
known(['ambn > 4', 'an > 0', 'bm > 0'])
peel(1,3,4)
print_code()
original()
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);
    s1(t2,t4,0);
    for(t6 = 1; t6 <= ambn-1; t6++) {
      s1(t2,t4,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;
      C[i][j] += (A[i][0] * B[0][j]);
      for (n = 1; n <= ambn - 1; n += 1)
        C[i][j] += (A[i][n] * B[n][j]);
    }
}
```

Peel

`peel` (*int* `stmt`, *int* `loop`, *int* `amount` = *1*) [Transform]

The `peel` transformation unrolls a specified number of iterations of a statement from the beginning or the end of a loop at level *loop*.

If *amount* is positive then statements are peeled from the start of the loop, if negative then the statements are peeled from the end.

Python Script

```
from chill import *
source('mm.c')
procedure('mm')
loop(0)
known(['ambn > 4', 'an > 0', 'bm > 0'])
peel(1,3,4)
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++) {
    s2(t2,t4,0);
    s3(t2,t4,0);
    s4(t2,t4,1);
    s5(t2,t4,2);
    s6(t2,t4,3);
    for(t6 = 4; t6 <= ambn-1; t6++) {
      s1(t2,t4,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;
      C[i][j] += (A[i][0] * B[0][j]);
      if(2 <= ambn) {
        C[i][j] += (A[i][1] * B[1][j]);
        if(3 <= ambn) {
          C[i][j] += (A[i][2] * B[2][j]);
          if(4 <= ambn)
            C[i][j] += (A[i][3] * B[3][j]);
        }
      }
    }
  }
}
```

Permute

The `permute` transformation interchanges the loops of a loop nest.

`permute` (*vector<int> p*) [Transform]
`permute` (*set<int> stmts*, *vector<int> p*) [Transform]

The loop nest to permute is specified by the statements in the set *stmts*. The loops in the nest are permuted according to the permutation vector *p*.

Python Script

```
from chill import *
source('mm.c')
procedure('mm')
loop(0)
known(['ambn > 0', 'an > 0', 'bm > 0'])
permute([3,1,2])
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 <= ambn-1; t2++) {
  for(t4 = 0; t4 <= an-1; t4++) {
    if (t2 <= 0) {
      for(t6 = 0; t6 <= bm-1; t6++) {
        s0(t4,t6,t2);
        s1(t4,t6,t2);
      }
    }
    else {
      for(t6 = 0; t6 <= bm-1; t6++) {
        s1(t4,t6,t2);
      }
    }
  }
}
```

Transformed code

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

Reverse

`reverse` (*set<int>* `stmts`, *int* `level`) [Transform]

The `reverse` transformation changes the direction of the iteration through the loop and is a shortcut for the transformation `scale(stmts, level, -1)`.

Python Script

```
from chill import *
source('mm.c')
procedure('mm')
loop(0)
known(['ambn > 0', 'an > 0', 'bm > 0'])
distribute([0,1],1)
reverse([1],1)
reverse([1],2)
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 = -an+1; t2 <= 0; t2++) {
  for(t4 = -bm+1; t4 <= 0; t4++) {
    for(t6 = 0; t6 <= ambn-1; t6++) {
      s1(-t2,-t4,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 = -an + 1; i <= 0; i += 1)
    for (j = -bm + 1; j <= 0; j += 1)
      for (n = 0; n <= ambn - 1; n += 1)
        C[-i][-j] += (A[-i][n] * B[n][-j]);
}
```

Scale

`scale` (*set<int>* `stmts`, *int* `loop`, *int* `amount`) [Transform]

The `scale` transformation multiplies the index variable for the loop at level *loop* by *amount* and is a shortcut for the transformation `skew`(*stmts*, *loop*, $[0, \ldots, 0, amount]$).

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];
}
```

Shift

`shift` (*set<int>* `stmts`, *int* `loop`, *int* `amount`) [Transform]

The `shift` transformation adjusts the index of the loop at level *loop* by adding *amount* to what the the non transformed index would be and then subtracting *amount* from the index when it is used by statements in *stmts*. The aim of this transformation is to add a constant offset to the index used when executing selected statements and it is accomplishes by either adjusting the staring point of the loop or using conditionals when there are statements in the loop that are not in *stmts*.

Python Script

```
from chill import *
source('mm.c')
procedure('mm')
loop(0)
known(['ambn > 0', 'an > 0', 'bm > 0'])
shift([1],1,4)
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+3; t2++) {
  for(t4 = 0; t4 <= bm-1; t4++) {
    if (an >= t2+1) {
      s0(t2,t4,0);
      if (t2 >= 4) {
        s1(t2-4,t4,0);
      }
      if (t2 >= 4) {
        for(t6 = 1; t6 <= ambn-1; t6++) {
          s1(t2-4,t4,t6);
        }
      }
    }
    else {
      if (t2 >= 4) {
        for(t6 = 0; t6 <= ambn-1; t6++) {
          s1(t2-4,t4,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 + 3; i += 1)
    for (j = 0; j <= bm-1; j += 1)
      if (i + 1 <= an) {
        C[i][j] = 0.0f;
        if (4 <= i)
          C[i-4][j] += A[i-4][0]*B[0][j];
        if (4 <= i)
          for (n = 1; n <= ambn-1; n += 1)
            C[i-4][j] += A[i-4][n]*B[n][j];
      }
      else if (4 <= i)
        for (n = 0; n <= ambn-1; n += 1)
          C[i-4][j] += A[i-4][n]*B[n][j];
}
```

Shift_to

`shift_to` (*int* `stmt`, *int* `loop`, *int* `amount`) [Transform]

The `shift_to` transformation adjusts the index of the loop at level *loop* by adding *amount* to the upper and lower bounds of the loop and then subtracting *amount* from every statement within the same loop structure as *stmt*.

Python Script

```
from chill import *
source('mm.c')
procedure('mm')
loop(0)
known(['ambn > 0', 'an > 0', 'bm > 0'])
shift_to(1,1,4)
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 = 4; t2 <= an+3; t2++) {
  for(t4 = 0; t4 <= bm-1; t4++) {
    s0(t2-4,t4,0);
    s1(t2-4,t4,0);
    for(t6 = 1; t6 <= ambn-1; t6++) {
      s1(t2-4,t4,t6);
    }
  }
}
```

Transformed code

```
void mm(float **A, float **B, float **C,
        int ambn, int an, int bm) {
  int i, j, n;
  for (i = 4; i <= an + 3; i += 1)
    for (j = 0; j <= bm-1; j += 1) {
      C[i-4][j] = 0.0f;
      C[i-4][j] += A[i-4][0]*B[0][j];
      for (n = 1; n <= ambn-1; n += 1)
        C[i-4][j] += A[i-4][n]*B[n][j];
    }
}
```

Skew

`skew` (*set<int>* `stmts`, *int* `loop`, *vector<int>* `amount`) [Transform]

The `skew` transformation changes the index variable of the loop at level *loop* to be a linear combination of the indexes that are less than or equal to the level being transformed.

Let $i_1, i_2, \ldots, i_{loop}$ be the original loop indexes and let $amount = (a_1, a_2, \ldots, a_{loop})$. The new index i'_{loop} will be $\sum_{l=1}^{loop} a_l i_l$.

The example below takes an algorithm with a negative loop-carried dependence and transforms it into one without a dependence.

Python Script

```
from chill import *
source('skew.c')
procedure('f')
loop(0)
known(['n > 0', 'm > 1'])
print_code()
print_dep()

skew([0], 2, [1, 1])
print_code()
print_dep()
```

Original code

```
void f(float **a, int n, int m) {
  int i, j;
  for (i = 1; i < n; i++)
    for (j = 0; j < m; j++)
      a[i][j] = a[i-1][j+1] + 1;
}
```

Output on stdout

```
for(t2 = 1; t2 <= n-1; t2++) {
  for(t4 = 0; t4 <= m-1; t4++) {
    s0(t2,t4);
  }
}

dependence graph:
s0->s0: a:true(1, -1)


for(t2 = 1; t2 <= n-1; t2++) {
  for(t4 = t2; t4 <= t2+m-1; t4++) {
    s0(t2,-t2+t4);
  }
}

dependence graph:
s0->s0: a:true(1, 0)
```

Transformed code

```
void f(float **a,int n,int m)
{
  int i, j;
  for (i = 1; i < n; i += 1)
    for (j = i; j < i + m; j += 1)
      a[i][j-i] = (a[i-1][j-i+1]+1);
}
```

Split

`split` (*int* `stmt`, *int* `loop`, *int* `expr`) [Transform]

The `split` transformation divides the iteration space of the loop at level *loop* using the condition specified in *expr* for the statement in *stmt*.

The condition in *expr* can refer to the value of the iteration of the loop nested at level *n* as "L<n>". Only one expression is allowed and it may not contain logical operators and/or or multiple formulas.

Python Script

```
from chill import *

source('mm.c')
procedure('mm')
loop(0)
known('ambn > 0')
known('an > 0')
known('bm > 10')
split(1, 2, "L2 < 5")
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 <= 4; t4++) {
    s0(t2,t4,0);
    s1(t2,t4,0);
    for(t6 = 1; t6 <= ambn-1; t6++) {
      s1(t2,t4,t6);
    }
  }
  for(t4 = 5; t4 <= bm-1; t4++) {
    s2(t2,t4,0);
    s3(t2,t4,0);
    for(t6 = 1; t6 <= ambn-1; t6++) {
      s3(t2,t4,t6);
    }
  }
}
```

Transformed code

```
#define __rose_lt(x,y) ((x)<(y)?(x):(y))

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<=__rose_lt(4,bm-1);j+=1){
      C[i][j] = 0.0f;
      C[i][j] += (A[i][0] * B[0][j]);
      for (n = 1; n <= ambn - 1; n += 1)
        C[i][j] += (A[i][n] * B[n][j]);
    }
    for (j = 5; j <= bm - 1; j += 1) {
      C[i][j] = 0.0f;
      C[i][j] += (A[i][0] * B[0][j]);
      for (n = 1; n <= ambn - 1; n += 1)
        C[i][j] += (A[i][n] * B[n][j]);
    }
  }
}
```

Tile

`tile` (*int* `stmt`, *int* `loop`, *int* `tile_size`, *int* `control_loop` = *1*, *TileMethod* `method` = *0*, *int* `alignment_offset` = *1*, *int* `alignment_multiple` = *1*) [Transform]

The `tile` transformation allows a loop dimension to be segregated into tiles, the execution of which are scheduled by a control loop placed outside the tiled loop. The statements *stmt* and surrounding loops inside the control loop will be executed a tile at a time along the tiled dimension.

The loop nest to tile is specified by *stmt* and *loop*. The argument *tile_size* specifies the tile size, a value of 0 indicates no tiling, a value of 1 is similar to loop interchange and a value greater than 1 sets the tile size to that value. The argument *control_loop* specifies the loop level where the controlling loop should be placed, the default is 1 or the outermost loop. The argument *method* specifies the tiling method, a value of 0 indicates that the index value of the control loop is the actual index to the start of the tile and is known as a "strided tile", a value of 1 indicates that the index value of control loop is the value of the tile and must be multiplied by *tile_size* to get the index to the start of the tile. The value of *alignment_offset* shifts the beginning of the area to tile consistent with the alignment constraint in *alignment_multiple*.

Python Script

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

```
chill test_tile.py
for(t2 = 0; t2 <= bm-1; t2 += 4) {
  for(t4 = 0; t4 <= an-1; t4++) {
    for(t6 = t2; t6 <= min(t2+3,bm-1); t6++) {
      s0(t4,t6,0);
      s1(t4,t6,0);
      for(t8 = 1; t8 <= ambn-1; t8++) {
        s1(t4,t6,t8);
      }
    }
  }
}
```

Transformed code

```
#define __rose_lt(x,y) ((x)<(y)?(x):(y))
void mm(float **A, float **B, float **C,
        int ambn, int an, int bm) {
  int i, j, n, jj;
  for (jj = 0; jj <= bm - 1; jj += 4)
    for (i = 0; i <= an - 1; i += 1)
      for (j=jj; j<=__rose_lt(bm-1,jj+3);
           j+=1) {
        C[i][j] = 0.0f;
        C[i][j] += (A[i][0] * B[0][j]);
        for (n = 1; n <= ambn - 1; n += 1)
            C[i][j] += (A[i][n] * B[n][j]);
      }
}
```

Unroll

`unroll` (*int* `stmt`, *int* `loop`, *int* `unroll_amount`, *int* `cleanup_split_level`) [Transform]

`unroll_extra` (*int* `stmt`, *int* `loop`, *int* `unroll_amount`, *int* `cleanup_split_level`) [Transform]

The `unroll` transformation unrolls a specified number of iterations of a statement inside the loop at level *loop*. `unroll_extra` is the same as unroll except the cleanup loop is fully unrolled whenever possible

Python Script

```
from chill import *
source('mm.c')
procedure('mm')
loop(0)
known('ambn > 0', 'an > 0', 'bm > 0')
distribute([0,1], 1)
unroll(1, 3, 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 <= an-1; t2++) {
  for(t4 = 0; t4 <= bm-1; t4++) {
    s2(t2,t4);
    for(t6 = 0; t6 <= -over1+ambn-1;
        t6 += 4) {
      s1(t2,t4,t6);
      s4(t2,t4,t6);
    }
    for(t6 = max(0, ambn-over1);
        t6 <= ambn-1; t6++) {
      s3(t2,t4,t6);
    }
  }
}
```

Transformed code

```
#define __rose_gt(x,y) ((x)>(y)?(x):(y))

void mm(float **A, float **B, float **C,
        int ambn, int an, int bm)
{
  int i, j, n, over1;
  over1 = 0;
  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 <= an - 1; i += 1)
    for (j = 0; j <= bm - 1; j += 1) {
      over1 = ambn % 4;
      for (n = 0; n <= -over1 + ambn - 1;
           n += 4) {
        C[i][j] += (A[i][n]*B[n][j]);
        C[i][j] += (A[i][n+1]*B[n+1][j]);
        C[i][j] += (A[i][n+2]*B[n+2][j]);
        C[i][j] += (A[i][n+3]*B[n+3][j]);
      }
      for (n = __rose_gt(0,ambn - over1);
           n <= ambn - 1; n += 1)
        C[i][j] += (A[i][n] * B[n][j]);
    }
}
```

3.4 Data operations

Datacopy

For the specified accesses, a temporary array copy construction is introduced. Those array accesses are replaced by appropriate temporary array accesses.

`Datacopy` (*int* `stmt`, *int* `loop`, *string* `array`, *bool* `allow_extra_read` = *False*, *int* `fastest_changing_dimension` = *-1*, *int* `stride_padding` = *1*, *int* `alignment_padding` = *0*, *int* `memory_type` = *0*) [Transform]

`Datacopy` (*set*<*tuple*<*int*, *vector*<*int*>>>> `refs`, *int* `loop`, *bool* `allow_extra_read` = *False*, *int* `fastest_changing_dimension` = *-1*, *int* `stride_padding` = *1*, *int* `alignment_padding` = *0*, *int* `memory_type` = *0*) [Transform]

array all array accesses in the subloop with this array name.

refs $[(stmt\#, [ref\#1, ref\#2, \ldots]), \ldots]$, array number followed by order. Such as $C = C * B$ have accessing order $C^0 = C^1 * B^2$.

loop inside which loop level the data footprint is copied and put right before this loop.

allow_extra_read
whether extra data copy is allowed to simplify read copy loop.

fastest_changing_dimension
-1: no change in array layout
d: contiguous array elements in the memory at d-th dimension

stride_padding
0: keep the original data layout in all dimensions
1: compressed array layout.
d: accessing the fastest-changing-dimension in stride d, while other array dimensions are compressed.

alignment_padding
0: keep the original data layout in all dimensions
d(> 1): the size of the fastest changing dimension is multiples of d
d(< -1): the size of the fastest changing dimension is coprime with $|d|$

Python Script

```
from chill import *
source('dist.c')
procedure('mm')
loop(0)
known(' ambn > 0 ')
datacopy(1,3, 'C')
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

```
if (bm >= 1) {
  for(t2 = 0; t2 <= an-1; t2++) {
    for(t4 = 0; t4 <= bm-1; t4++) {
      s0(t2,t4,0);
      s1(t2,t4,0);
      for(t6 = 1; t6 <= ambn-1; t6++) {
        s1(t2,t4,t6);
      }
      s2(t2,t4);
    }
  }
}
```

Transformed code

```
void mm( float **A, float **B, float **C,
    int ambn, int an, int bm )
{
  float newVariable0;
  int i, j, n;
  if (1 <= bm) {
    for (i = 0; i <= (an - 1); i += 1)
    for (j = 0; j <= (bm - 1); j += 1) {
      newVariable0 = 0.0f;
      newVariable0 += (A[i][0] * B[0][j]);
      for (n = 1; n <= (ambn - 1);
                            n += 1)
        newVariable0 += A[i][n]
                            * B[n][j];
      C[i][j] = newVariable0;
    }
  }
}
```

Concept Index

Function and Transformation Index