The Lambda Loop Transformation Toolkit (User's Reference Manual)

Wei Li
Department of Computer Science
University of Rochester

Keshav Pingali
Department of Computer Science
Cornell University

August 1994
Cornell Theory Center Technical Report CTC94TR189

Abstract

Loop transformations are becoming critical to exploiting parallelism and data locality in parallelizing and optimizing compilers. This document describes the Lambda loop transformation toolkit, an implementation of the non-singular matrix transformation theory, which can represent any linear one-to-one transformation.

Lambda has a simple interface, and is independent of any compiler intermediate representation. It has been used in parallelizing compilers for multiprocessor machines as well as optimizing compilers for uniprocessor machines.

Keywords: Parallel programming, parallelizing compilers, loop transformations, linear transformations, nonsingular transformations.

[0]This work was partly supported by Cornell University, by a grant from the Hewlett-Packard Corporation, and by the University of Rochester.

Contents

1 Introduction

Loop transformations are becoming critical to exploiting parallelims and data locality in parallelizing and optimizing compilers. This document describes the Lambda loop transformation toolkit. The toolkit is an implementation of the non-singular matrix transformation theory described in [6, 8].

The main benefit of this loop transformation theory is that it provides an approach to tackling the so-called 'phase-ordering problem', i.e. for many problems it is difficult to decide the sequence of the *primitive* transformations such as loop interchange, loop skewing, loop reversal [13], and loop scaling [8] should be performed. The loop transformation theory, generalizing the unimodular matrix approach [2, 12], provides a framework to represent *compound* transformations of these primitive transformations. In fact, this theory can represent any linear one-to-one transformation, of which the primitive transformations mentioned above are special instances. This linear transformation theory has been extended to handle loop aligment [1]. Other research on compound transformations can be found in recent publications [4, 9, 10, 11]. This toolkit has been used in parallelizing compilers for multiprocessor machines [3, 7] as well as optimizing compilers for uniprocessor machines [5].

Lambda has a simple interface, and is independent of the intermediate representation used in the compiler. There are four modules: the data dependence module, the transformation module, the code restructuring module, and the utility module.

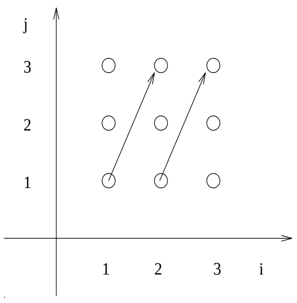
Lambda is a research software, and distributed freely in the interest of advancement of science. Due to its experimental nature, no warranty or liability are provided. A reasonable level of support approriate to a research project may be expected.

The document is organized as follows. Section 2 gives a brief review of the non-singular transformation theory. The data types and routines for handling data dependences are described in Section 3. The routines for constructing loop transformations are presented in Section 4. Section 5 shows the functions to restructure loops with a non-singular transformation. The utility functions such as print routines in Section 6 are helpful in the debugging process. Some examples of using these routines are given in Section 7. We summarize in Section 8.

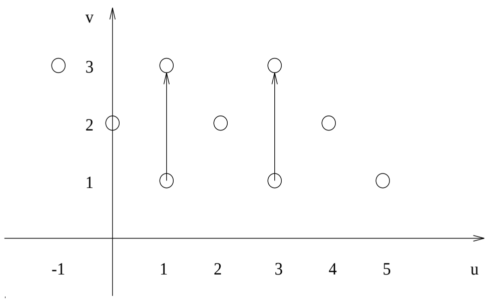
2 A Linear Loop Transformation Theory

In this section, we briefly review the linear transformation framework developed in [8]. This transformation theory is based on the use of integer lattices as the model of loop nests and the use of non-singular matrices as the model of loop transformations.

Consider the iteration space in Figure 1(a), which is derived from the loop nest in Figure 1(c) where i is the outer loop and j is the inner loop. The outer loop is not parallel because of the data dependence $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$. For coarse-grain parallelism, it is preferable to have a parallel outer loop, i.e. an outer loop that does not carry data dependences. This can be



(a) Iteration Space



(b) New Iteration Space

```
for i = 1, 3
  for j = 1, 3
    A[i, j] = A[i-1, j-2] + 1;
```

(c) Source code

```
for u = -1, 5
  for v = -u+2max(⌈(u+1)/2⌉,1),
        -u+2min(⌊(u+3)/2⌋,3), 2
    A[(u+v)/2, v] = A[(u+v)/2 -1), v-2] + 1
```

(d) Transformed code

Figure 1: Loop transformation for coarse-grain parallelism

accomplished by the loop transformation below:

$$\begin{pmatrix} 2 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix}$$

The iteration space in Figure 1(b) is the transformed iteration space; in this iteration space, loop u is a parallel loop as shown in Figure 1(d). It is possible to obtain this transformation by a sequence of simple loop transformations like skewing and interchange, but it is non-trivial to determine the order in which these transformations must be applied to achieve the desired effect. The matrix-based approach permits the generation of the transformation matrix from the dependences, and thereby produces the composite transformation directly.

Given a loop nest and a non-singular matrix that represents a legal transformation of the loop nest, it is non-trivial to generate the code for the transformed program. Fortunately, we have shown that the Hermite normal form of the transformation matrix can be used to solve this problem. The details of code generation may be found in a related paper [8]. This paper also gives a *completion procedure* which, given some initial rows of a desired transformation matrix, produces a complete transformation matrix that represents a legal restructuring of the original loop nest.

This framework can be used in a number of applications such as restructuring for machines exploiting coarse-grain or fine-grain parallelism, for exploiting spatial and temporal data locality and for enhancing data reuse.

3 Data Dependences

3.1 Data Types

A data dependence can be specified with either *distance* or *direction*. A distance is represented by an integer constant, and a direction is represented by the following symbols.

```
typedef enum {
  LA_dK,
  LA_dLT,
  LA_dLEQ,
  LA_dEQ,
  LA_dGEQ,
  LA_dGT,
  LA_dDOT,
  LA_dLG,
  LA_dSTAR,
  LA_dSIZE
  } LA_DIR_T;
```

LA_dK represents distance, where its value is stored in another variable. The data type for a dependence is a structure with two fields.

```
typedef struct _la_dep {
  LA_DIR_T dir;
  int dist;
} LA_DEP_T;

#define LA_DEP_DIR(dep)     ((dep).dir)
#define LA_DEP_DIST(dep)    ((dep).dist)
```

A data dependence vector is a vector (array) of dependences pointed by the field *vector*. The field *next* is used to form a dependence matrix with a linked list.

```
typedef struct _vector {
  LA_DEP_T * vector;
  struct _vector *next;
} *LA_DEP_V_T;

#define LA_DEP_V_VECTOR(v)  ((v)->vector)
#define LA_DEP_V_NEXT(v)    ((v)->next)
```

And a dependence matrix is a linked list of dependence vectors. The matrix has a header that contains the dimension (*dim*) and the size (*size*) of the dependence matrix, i.e. the number of dependence vectors in the matrix, in addition to the linked list of dependence vectors.

```
typedef struct _la_dep_m {
  LA_DEP_V_T vectors;
  int dim;
  int size;
} *LA_DEP_M_T;

#define LA_DEP_M_VECTORS(D)    ((D)->vectors)
#define LA_DEP_M_DIM(D)        ((D)->dim)
#define LA_DEP_M_SIZE(D)       ((D)->size)
```

3.2 Routines

There are routines provided to operate on the dependences, some of which are listed here. A legal dependence vector should be lexicographically positive. A data dependence analyzer may return a dependence vector with illegal components, i.e. *not* lexicographically positive. *la_dep_legal* deletes the illegal components. For example, the dependence vector $(*, 1)$ becomes $(<, 1)$ and $(=, 1)$. The illegal component $(>, 1)$ is deleted.

- Creates a dependence vector.

  ```
  LA_DEP_V_T la_dep_vec_new( int dim );
  ```

  dim is the length of the dependence vector, i.e. the number of loops. This routine allocates an array of size *dim*, and returns the pointer to the vector (array).

- Frees a dependence vector.

  ```
  void la_dep_vec_free( LA_DEP_V_T d );
  ```

- Creates a dependence matrix.

  ```
  LA_DEP_M_T la_dep_matrix_new(int dim, int size);
  ```

  This routine allocates the header for the dependence matrix. The dimension (*dim*) and size (*size*, usually zero for initialization) must be provided. The pointer to the vectors is nulled.

- Deletes the illegal components from a dependence matrix.

  ```
  LA_DEP_M_T la_dep_legal(LA_DEP_M_T D);
  ```

- Eliminates redundant data dependences.

  ```
  LA_DEP_M_T  la_dep_no_redundant(LA_DEP_M_T D);
  ```

  This routine is quite useful. The dependence vectors in a loop nest tend to be similar; in fact many of them will be the same. The size of the dependence matrix tends to decrease significantly after the redundant vectors are eliminated.

4 Constructing Transformations

4.1 Data Types

A matrix type is a two dimensional *matrix*, which is an array of arrays. The *row_size* is the number of rows, and the *col_size* is the number of columns in the matrix. The type can represent any rational matrix A/d, where A is an integer matrix, and d is the denominator. The field *denom* contains the integer denominator.

```
typedef struct _la_matrix {
  int ** matrix;
  int row_size;
  int col_size;
  int denom;
} *LA_MATRIX_T;

#define LA_MATRIX(T)            ((T)->matrix)
#define LA_MATRIX_ROWSIZE(T)    ((T)->row_size)
#define LA_MATRIX_COLSIZE(T)    ((T)->col_size)
#define LA_MATRIX_DENOM(T)      ((T)->denom)
```

- To create a matrix structure with *rowsize* and *colsize*, we can use the following routine. It allocates memory space for the two dimensional matrix, and sets the right values to fields *row_size* and *col_size*.

  ```
  LA_MATRIX_T la_matrix_new(int rowsize, int colsize);
  ```

- Create a matrix struct. No space is allocated for the matrix pointed by the field *matrix*.

```
LA_MATRIX_T la_matrix_allocate( void );
```

- Free the matrix struct. Any space pointed by *matrix* is not freed, and must be freed before *la_matrix_free* is called.

  ```
  void la_matrix_free( LA_MATRIX_T matrix);
  ```

4.2 Nonsingularity

This set of routines are useful in the construction of a non-singular transformation matrix.

- Checks if the transformation matrix is nonsingular.

  ```
  int la_is_nonsingular(LA_MATRIX_T T);
  ```

  It returns 1 if the square matrix T is non-singular, 0 if the matrix is singular.

- Checks if the transformation matrix has full rank.

  ```
  int  la_is_fullrank(LA_MATRIX_T pT);
  ```

  pT can be any partial transformation, i.e. the number of rows is not equal to the number of columns. The function returns 1 if the matrix has full rank, i.e. its rank equals to the minimum of row size and column size, 0 otherwise.

- Computes the rank of a matrix.

  ```
  int  la_rank(LA_MATRIX_T pT);
  ```

  It returns the rank of the partial transformation pT.

- Computes a base matrix.

  ```
  LA_MATRIX_T la_base(LA_MATRIX_T pT);
  ```

  A row base is computed by deleting the rows that are linearly dependent on the base rows. The rows are processed in the top-down fashion, i.e. the first row is considered first, then the second row, and so on.

- Extends a base matrix to a nonsingular matrix.

  ```
  LA_MATRIX_T  la_padding(LA_MATRIX_T pT);
  ```

  Once the row base matrix is obtained in pT, additional rows are added to form a non-singular square matrix.

- Computes the inverse of a transformation.

  ```
  LA_MATRIX_T  la_inverse(LA_MATRIX_T T);
  ```

  This routine is useful for computing the new loop body, where the inverse of the transformation must be applied to the expressions in the loop body.

4.3 Data Dependences

This set of routines assists the construction of a legal transformation.

- Checks if a full transformation is legal with respect to the data dependences.

  ```
  int  la_is_legal(LA_MATRIX_T T, LA_DEP_M_T D);
  ```

  It returns 1 if the complete transformation T does not violate the dependences in D, 0 otherwise.

- Checks if a partial transformation is legal.

  ```
  int  la_is_legal_par(LA_MATRIX_T pT, LA_DEP_M_T D);
  ```

  It returns 1 if the partial transformation T does not violate the dependences in D, 0 otherwise.

- Computes a legal base matrix by deleting the rows that violate the dependences.

  ```
  LA_MATRIX_T  la_base_legal(LA_MATRIX_T pT, LA_DEP_M_T D);
  ```

  A base matrix may violate data dependences. The rows that violate dependences will be deleted from the matrix to form a legal partial transformation. The data dependences that are satisfied by the the partial transformation are deleted from D.

- Extends the legal base matrix with respect to the dependence matrix.

  ```
  LA_MATRIX_T  la_padding_legal(LA_MATRIX_T pT, LA_DEP_M_T D);
  ```

  Similarly, the legal base matrix will be extended to a legal complete non-singular matrix by adding additional rows.

- A data dependence will be transformed to a new dependence vector. The function *la_dep_trans* returns the transformed dependence vector of the dependence vector d under the transformation T.

  ```
  LA_DEP_V_T la_dep_trans(LA_DEP_V_T d, LA_MATRIX_T T);
  ```

- Dot product of an integer vector and a dep vector.

  ```
  LA_DEP_T la_vec_X_dep(la_vect vec1, LA_DEP_V_T d, int dim);
  ```

- An integer matrix times a dependence matrix.

  ```
  LA_DEP_M_T  la_matrix_X_depM(LA_MATRIX_T A, LA_DEP_M_T D);
  ```

5 Code Restructuring

A loop nest is represented by a list of loops, where each loop contains a lower bound, an upper bound, and a step size. Each bound is a list of linear expressions. A lower bound can be the maximum of a set of linear functions, and an upper bound can be the minimum of a set of linear functions.

Symbolic variables are allowed in the loop bounds as long as they are loop invariants. We call them *blobs*. There are design decisions in choosing blobs. For example, if the expression $2x + 3y$ is one of the loop bounds, where both x and y are loop invariants. One can make the whole expression a blob, then the number of blobs is the number of distinct expressions that contain these variables. Or one can make each variable a blob, e.g. x is a blob, and y is another blob, then the number of blobs is the number of such variables. In general, a loop bound can have a linear expression of these blobs as a part of the bound.

We explain these data structures in the bottom-up fashion, i.e. from expression, loop, to loop nest.

5.1 Computing Loop Bounds

First, simple integer vectors and matrices are defined. An integer vector is an array of integers, and an integer matrix is an array of integer vectors.

```
typedef int * la_vect;
typedef la_vect * la_matrix;
```

5.1.1 Linear Expressions

A linear expression is a linear expression of the loop index variables and symbolic variables (blobs). The linear expression type has a field for the coefficients (*coef*) of the loop index variables and the coefficients (*blob_coef*) of the blobs. *c0* is the constant in the expression. Again, *denom* can be used to represent a rational linear expression in the same way as a rational matrix. The pointer *next* enables the representation of a linked list of linear expressions.

```
typedef struct _la_expr{
  la_vect coef;
  int c0;
  la_vect blob_coef;
  int denom;
  struct _la_expr *next;
} *LA_EXPR_T;
```

```
#define LA_EXPR_COEF(expr)        ((expr)->coef)
#define LA_EXPR_C0(expr)          ((expr)->c0)
#define LA_EXPR_BLOB_COEF(expr)  ((expr)->blob_coef)
#define LA_EXPR_DENOM(expr)       ((expr)->denom)
#define LA_EXPR_NEXT(expr)        ((expr)->next)
```

For example, let i, j be the loop index variables and x, y be two blobs in the linear expression $(i+j+2+x+y)/3$. The coefficient vector contains the linear coefficients ($coef =$ (1 1) for $i+j$). The integer constant is in *c0* (*c0* $= 2$). The blob coefficients are (1, 1), since there are only two blobs and we consider x as blob 1, and y as blob 2. The denominator is 3.

We can allocate and free an expression with the dimension *dim* and total number of blobs *blobs* using the following routines.

- `LA_EXPR_T la_expr_new(int dim, int blobs);`

  This function allocates space for not only the loop structure but also the two coefficient vectors. The size of the loop index coefficients is *dim*, and the size of the blobs coefficients is *blobs*.

- `void la_expr_free( LA_EXPR_T expr );`

5.1.2 Single Loops

A loop has a lower bound, an upper bound, and a step size. The lower bound can be the maximum of a set of linear expressions defined in the previous section, and a linear expression with an integer denominator may have an implicit ceiling function since only integer functions are allowed. An upper bound can be the minimum of a set of linear expressions, where the floor function is implicit, when necessary.

For example,

```
DO i = max( ceil((i+j)/2), ...), min( floor((2i)/3), ...), step 2
```

The loop type has a list of expressions for both lower and upper bounds. The offset expression (a linear expression of loop index variables) is used in both lower and upper bounds.

```
typedef struct _la_loop{
  LA_EXPR_T low;
  LA_EXPR_T up;
```

```
  int step;
  LA_EXPR_T offset;
} *LA_LOOP_T;

#define LA_LOOP_LOW(loop)    ((loop)->low)
#define LA_LOOP_UP(loop)     ((loop)->up)
#define LA_LOOP_STEP(loop)   ((loop)->step)
#define LA_LOOP_OFFSET(loop) ((loop)->offset)
```

The new loop bounds after a nonsingular transformation can be more complex. The bounds may have a linear offset, and an integer factor that equals to the step size. The offset and factor are the same for both lower and upper bound in the same loop. The integer factor is always the same as the loop step size.

For example,

```
DO v = u + 2*max( ceil((u+v)/2), ...), u + 2*min( floor(..),...), step 2
```

These two routines allocate and free a loop structure.

- `LA_LOOP_T la_loop_new( void );`
- `void la_loop_free( LA_LOOP_T loop );`

5.1.3 Loop Nests

A loop nest is an ordered array of loops with the first being the outermost loop and the last being the innermost. *depth* is the depth of the loop nest. *blobs* is the total number blobs.

```
typedef struct _la_loopnest{
  LA_LOOP_T *loops;
  int depth;
  int blobs;
} *LA_LOOPNEST_T;

#define LA_NEST_LOOPS(nest) ((nest)->loops)
#define LA_NEST_DEPTH(nest) ((nest)->depth)
#define LA_NEST_BLOBS(nest) ((nest)->blobs)
```

These routines allocate and free a loop nest.

- `LA_LOOPNEST_T la_nest_new(int depth, int blobs );`

- `void la_nest_free( LA_LOOPNEST_T nest );`

la_nest takes a loop nest, and the transformation matrix and returns the new loop nest.

- `LA_LOOPNEST_T la_nest(LA_LOOPNEST_T nest, LA_MATRIX_T T);`

5.2 Computing Loop Body

The loop body can be considered as a set of *simple vectors*. Each vector can be transformed using *la_vector*.

A simple vector is a piece-wise linear function.

```
typedef struct _la_vector {
  la_vect coef;
  int size;
  int denom;
} *LA_VECTOR_T;

#define LA_VECTOR_COEF(v)      ((v)->coef)
#define LA_VECTOR_SIZE(v)      ((v)->size)
#define LA_VECTOR_DENOM(v)     ((v)->denom)
```

The following routines allocate and free a simple vector.

- `LA_VECTOR_T la_vector_new( int size );`
- `void la_vector_free( LA_VECTOR_T v );`

la_vector computes the new expression in the transformed loop nest. The input is the inverse of the transformation.

- `LA_VECTOR_T la_vector(LA_MATRIX_T invT, LA_VECTOR_T v);`

6 Utility Routines

There are many other routines available in the toolkit. These routines include vector and matrix operations, algebraic operations on dependences, print routines for debugging, and so on.

6.1 Print Routines

Here is the list of print routines that are helpful for debugging.

- Print a matrix.

  ```
  void la_matrix_print( LA_MATRIX_T mat );
  ```

- Print a loop nest. If the *start* symbol is i, then index variables are i, j, k and so on.

  ```
  void la_nest_print( LA_LOOPNEST_T nest, char start);
  ```

- Print a loop.

  ```
  void la_loop_print( LA_LOOP_T loop, int depth, int blobs, char start);
  ```

- Print an expression.

  ```
  void la_expr_print( LA_EXPR_T expr, int depth, int blobs, char start);
  ```

- Print a dependence.

  ```
  void la_dep_print(LA_DEP_T d);
  ```

- Print a dependence vector.

  ```
  void la_dep_vec_print(LA_DEP_V_T d, int dim);
  ```

- Print a dependence matrix.

  ```
  void la_dep_matrix_print( LA_DEP_M_T D);
  ```

6.2 Vector Operations

A vector is an array of integers *la_vect*.

```
typedef int * la_vect;
```

In this section, unless specified otherwise, n is the length of the vector. Storage space for all vector parameters must be allocated before calling the routines.

- Negate the vector *vec1*, and the output is in *vec2*.

```
void la_vecNegate (la_vect vec1, la_vect vec2, int n);
```

- Multiply the vector *vec1* by the constant c, and store the result in *vec2*.

```
void la_vecConst (la_vect vec1, la_vect vec2, int n, int c);
```

- Add two vectors *vec1* and *vec2* of size n, and the output is in *vec3*.

```
void la_vecAdd(la_vect vec1, la_vect vec2, la_vect vec3, int n);
```

- Add two vectors *vec1* and *vec2* of size n. *vec3* is *f1*∗*vec1* + *f2*∗*vec2*.

```
void la_vecAddF(la_vect vec1, int f1, la_vect vec2, int f2,
                la_vect vec3, int n);
```

- Concatenate two vectors *vec1* of size *n1* and *vec2* od size *n2*. The output is in *vec3*.

```
void  la_vecConcat(la_vect vec1, int n1,
                   la_vect vec2, int n2,
                   la_vect vec3);
```

- Make a copy of the vector *vec1* of size n in *vec2*.

```
void la_vecCopy(la_vect vec1, la_vect vec2, int n);
```

- Check if every entry is zero. Returns 1 if yes, 0 otherwise.

```
int la_vecIsZero(la_vect vec1, int n);
```

- Clear a vector.

```
void la_vecClear(la_vect vec1, int n);
```

- Check if two vectors are equal. Returns 1 if yes, 0 otherwise.

```
int la_vecEq(la_vect vec1, la_vect vec2, int n);
```

- Free a vector.

```
void  la_vecFree(la_vect vec);
```

6.3 Matrix Operations

A matrix *la_matrix* is an array of vectors.

```
typedef la_vect * la_matrix;
```

In this section, unless specified otherwise, m is the number of rows; and n is the number of columns. Storage space for all matrix and vector parameters must be allocated before calling the routines.

- Allocate a matrix of m by n.

  ```
  la_matrix la_matNew(int m, int n);
  ```

- Elementary operation: exchange two columns *col1* and *col2*.

  ```
  void la_eExchange (la_matrix mat, int m, int col1, int col2 );
  ```

- Elementary operation: add an integral multiple (*fact*) of column *col1* to column *col2*.

  ```
  void la_eAdd (la_matrix mat, int m, int col1, int col2, int fact);
  ```

- Elementary operation: negate the column *col*.

  ```
  void la_eNegate (la_matrix mat, int m, int col);
  ```

- Elementary operation: multiply the column *col* by the const *fact*.

  ```
  void la_eColConst(la_matrix mat, int m, int col, int fact);
  ```

- Find the minimum non-zero in the vector from the position *start* to position n. Return the array index of the minimum element.

  ```
  int la_eMinInVec(la_vect vec, int n, int start);
  ```

- Return the first non-zero in the vector segment *vec[i..n]*. Return n if all are zeroes.

  ```
  int la_eFirstNonZero(la_vect vec, int n, int i);
  ```

- Apply the elementary column operations to decompose *mat* to the product of a lower triangular matrix H and an unimodular matrix U. All matrices are of size n by n.

```
void la_matHermite(la_matrix T, int n, la_matrix H, la_matrix U);
```

- Initialize the matrix *mat* of size $n \times n$ to identity.

```
void la_matId(la_matrix mat, int n);
```

- Copy the matrix *mat1* to the matrix *mat2*.

```
void la_matCopy(la_matrix mat1, la_matrix mat2, int m, int n);
```

- The matrix *mat1* is negated and the output matrix is in *mat2*.

```
void la_matNegate(la_matrix mat1, la_matrix mat2, int m, int n);
```

- Compute the transpose of the matrix *mat1* and the output matrix is in *mat2*.

```
void la_matT(la_matrix mat1, la_matrix mat2, int m, int n);
```

- Is an identity matrix? Returns 1 if yes, 0 otherwise.

```
int la_matrix_is_id(LA_MATRIX_T M);
```

- Add two matrices *mat1* and *mat2*, and the result is in *mat3*.

```
void la_matAdd(la_matrix mat1, la_matrix mat2,
               la_matrix mat3, int m, int n);
```

- Add two matrices with constant factors. *mat3* is *f1*$*$*mat1* + *f2*$*$*mat2*.

```
void la_matAddF(la_matrix mat1, int f1, la_matrix mat2, int f2,
                la_matrix mat3, int m, int n);
```

- Matrix multiplication. *mat1* is $m \times r$; *mat2* is $r \times n$; and *mat3* is $m \times n$. *mat3* is the product of *mat1* and *mat2*.

```
void la_matMult(la_matrix mat1, la_matrix mat2, la_matrix mat3,
                int m, int r, int n);
```

- Get a column from the matrix. The column *col* of the matrix *mat* is returned in the vector *vec*. *m* is the number of rows in the matrix, and also the length of the column vector.

```
void la_matGetCol(la_matrix mat, int m, int col, la_vect vec);
```

- *la_matConcate* concatenates two matrices by row. *m1* is the number of rows in *mat1*; and *m2* is the number of rows in *mat2*. The resulting matrix is in *mat3*, which shares the storage space of rows with *mat1* and *mat2*. *la_matConcat_newM* concatenates two matrices, and allocates new storage space for the new matrix.

```
void la_matConcat(la_matrix mat1, int m1,
                  la_matrix mat2, int m2,
                  la_matrix mat3);

void la_matConcat_newM(la_matrix mat1, int m1, int n,
                       la_matrix mat2, int m2,
                       la_matrix mat3);
```

- Exchange two rows *i1* and *i2*.

```
void la_rowExchange (la_matrix mat, int i1, int i2 );
```

- Add a multiple (*fact*) of row *i1* to row *i2*.

```
void la_rowAdd (la_matrix mat, int m, int i1, int i2, int fact);
```

- Computes the inverse of the matrix. The result is in *inv*. Both matrices are $n \times n$. The function returns the determinant.

```
int la_matInv(la_matrix mat, la_matrix inv, int n);
```

- Free a matrix.

```
void  la_matFree(la_matrix mat, int m, int n);
```

- Multiply the matrix *mat* by the column vector *vec*, and the output vector is in *vec-out*.

```
void la_matVecMult(la_matrix mat, int m, int n,
                   la_vect vec, la_vect vec-out);
```

- Multiply the row vector *vec* by the matrix *mat*, and the output vector is in *vec-out*.

```
void la_vecMatMult(la_vect vec, int m,
                   la_matrix mat, int  n,
                   la_vect vec-out);
```

6.4 Integer Operations

- Return the *greatest common divisor* of two integers.

```
int la_gcd(int a, int b);
```

- Return the *greatest common divisor* of an array of integers.

```
int la_gcdV(la_vect v, int n );
```

- Return the *least common multiple* of two integers.

```
int la_lcm(int a, int b);
```

6.5 Data Dependences

- Copy the dependence vector *d* of size *dim* to *d-out*.

```
void la_dep_vec_copy(LA_DEP_V_T d, LA_DEP_V_T d-out, int dim);
```

- Add the dependence vector *d* to the matrix as the first column.

```
void la_dep_add_to_list(LA_DEP_V_T d, LA_DEP_M_T D);
```

- Add the dependence vector *d* to the matrix as the last column. *last* is the pointer to the last dependence vector in the linked list.

```
void la_dep_add_to_list_last(LA_DEP_V_T d, LA_DEP_M_T D, LA_DEP_V_T last);
```

- Check if every element in the *direction* vector *d* is negative or zero. Return 1 if yes, 0 otherwise.

```
int la_allNegEq(LA_DIR_T *d, int n);
```

- Check if every element in the *direction* vector *d* is positive or zero. Return 1 if yes, 0 otherwise.

```
int la_allPosEq(LA_DIR_T *d, int n);
```

- Negate the *direction* vector.

```
void  la_flagNegate(LA_DIR_T *d, int n);
```

- Compute the dot product of the integer vector *vec* and the dependence vector *d*. Both vectors have size *n*.

```
LA_DEP_T la_vec_depV_mult(la_vect vec1, LA_DEP_V_T d, int dim);
```

- Multiply the integer vector *vec* by the dependence matrix *D*, and convert the resulting dependence vector to *direction* vector in *dir-out*.

```
void  la_depMult(la_vect vec, LA_DEP_M_T D, LA_DIR_T *dir-out);
```

- Delete the dependences with positive flags. The direction vector *flagV* has the same size as the number of dependence vectors (columns) in the dependence matrix. *flagV* is the vector of flags, each of which is a flag for the corresponding column in the dependence matrix. When the flag is positive (*LA_dLT*), the corresponding dependence vector is deleted from the matrix.

```
void la_delPosDep(LA_DEP_M_T D, LA_DIR_T * flagV);
```

- Check if two dependence vectors are equal. Return 1 if yes, 0 otherwise.

```
int la_dep_vec_eq( LA_DEP_V_T d1, LA_DEP_V_T d2, int size);
```

- Check if two dependences are equal. Return 1 if yes, 0 otherwise.

```
int la_dep_eq( LA_DEP_T d1, LA_DEP_T d2);
```

7 How to Use Lambda?

This section describes how to use the toolkit.

7.1 Compiling and Linking

The file "Lambda.h" must be included, and the library "Lambda.a" must be linked. The toolkit can be called from both C and C++. A typical Makefile looks like:

```
LIBS     = -lLambda
demo:   demo.o
        $(CC) $(CFLAGS) -o demo demo.o $(LDFLAGS) $(LIBS)
```

7.2 An Example

This example shows how Lambda toolkit is used to transform a loop nest. The dependence matrix, transformation matrix and loop nest (loop bounds) are all stored in the Lambda format after being read from the data files. In a compiler, they should be extracted from the intermediate representation of the compiler. The following is the set of variables used in the example.

```
int depth;                    /* the nesting depth of loop nest */
int blobs;                    /* number of blobs */
int i, j, k, r, n;
FILE *fp;                     /* data file */
LA_MATRIX_T T;                /* Transformation matrix */
LA_DEP_M_T D;                 /* Data dependence matrix */
LA_LOOPNEST_T nest, new_nest; /* Loop nests */
LA_LOOP_T loop;               /* A loop */
LA_EXPR_T expr;               /* An expression */
LA_DEP_V_T d;                 /* A dependence vector */
int depSize;                  /* number of dependence vectors */
```

7.2.1 Data Dependences

The data dependence presentation in Lambda may be different from that is used in other compilers. The conversion from other representation to the Lambda representation is quite straightforward.

```
fscanf(fp, "%d", &depSize);
D = la_dep_matrix_new(depth, depSize);
for (i=0; i<depSize; i++)
  {
    d = la_dep_vec_new(depth);
    LA_DEP_V_NEXT(d) = LA_DEP_M_VECTORS(D);
    LA_DEP_M_VECTORS(D) = d;
    for (j=0; j<depth; j++)
      {
        fscanf(fp, "%s", temp);
        if (!strcmp(temp, "<") )
          LA_DEP_DIR( LA_DEP_V_VECTOR(d)[j] ) = LA_dLT;
        else if (!strcmp(temp, "=") )
          LA_DEP_DIR( LA_DEP_V_VECTOR(d)[j] ) = LA_dEQ;
        else if (!strcmp(temp, ">") )
          LA_DEP_DIR( LA_DEP_V_VECTOR(d)[j] ) = LA_dGT;
        else if (!strcmp(temp, "*") )
```

```
          LA_DEP_DIR( LA_DEP_V_VECTOR(d)[j] ) = LA_dSTAR;
        else if (!strcmp(temp, "<=") )
          LA_DEP_DIR( LA_DEP_V_VECTOR(d)[j] ) = LA_dLEQ;
        else if (!strcmp(temp, ">=") )
          LA_DEP_DIR( LA_DEP_V_VECTOR(d)[j] ) = LA_dGEQ;
        else
          {
            LA_DEP_DIR( LA_DEP_V_VECTOR(d)[j] ) = LA_dK;
            sscanf(temp,
                   "%d",
                   &(LA_DEP_DIST( LA_DEP_V_VECTOR(d)[j] ))
                   );
          }
      }
  }
printf("The data dependence matrix is :\n");
la_dep_matrix_print(D);
```

7.2.2 Constructing Transformations

The transformation T is represented by a two-dimensional array with the row size and column size stored in the same structure. The print routine is always useful.

```
/* read the transformation matrix from the file */
T = la_matrix_new(depth, depth);
/* read in row size and col size */
fscanf(fp, "%d", &(LA_MATRIX_ROWSIZE(T)));
fscanf(fp, "%d", &(LA_MATRIX_COLSIZE(T)));
for(i=0; i<depth; i++)
  for(j=0; j<depth; j++)
    fscanf(fp, "%d", &(LA_MATRIX(T)[i][j]));
printf("The transformation matrix is :\n");
la_matrix_print(T);
```

A matrix may not contain a legal transformation. To be a transformation, the matrix has to be non-singular and does not violate data dependences. If these conditions are not satisfied, a legal transformation can be constructed using the routines for computing the base matrix and for completing the partial transformation.

```
/* compute the legal dependence matrix */
if( la_rank(T) == LA_MATRIX_COLSIZE(T) )
  {
    printf("\n\nThe transformation is non-singular :-) \n");
```

```
    }
  else
    {
      printf("\n\nThe transformation is singular or partial !! \n");
      printf("Completing... \n");
      /*  computes the base matrix from parT */
      T = la_base(T);
      printf("\nThe base matrix is:\n");
      la_matrix_print(T);
      /* computes the legal base matrix */
      /* dependences satisfied by T are deleted from D */
      T = la_base_legal(T, D);
      printf("\nThe legal base matrix is:\n");
      la_matrix_print(T);
      printf("\nThe unsatisfied dependences:\n");
      la_dep_matrix_print(D);
      /*  padding of the legal matrix w.r.t the dependence matrix.  */
      T = la_padding_legal(T, D);
      printf("\nThe legal partial matrix after padding w.r.t depM is:\n");
      la_matrix_print(T);
      /*  padding of the  matrix to a legal non-singular matrix */
      T = la_padding(T);
      printf("\nThe non-singular matrix is:\n");
      la_matrix_print(T);
    }
  if( la_is_legal(T, D) )
    printf("\n\nThe transformation is legal :-) \n\n");
  else
    {
      printf("\n\nThe transformation is illegal :-( \n");
      exit(0);
    }
```

7.2.3 Code Restructuring

The loop bounds from the perfectly nested loops are extracted from the intermediate representation of the compiler, and stored in the Lambda format. The following code segment reads in the bounds.

```
  /* Create a structure for the loop nest. */
  /* Number of blobs should be computed from the program. */
  fscanf(fp, "%d", &blobs);
  nest = la_nest_new(depth, blobs);
```

```
/* e.g DO k = max(1, i), min(j+b, M) */
for(i=0; i<depth; i++)
  {
    /* create a structure for the loop */
    loop = la_loop_new();
     /* k==0 lower bound;
        Note: there is an implicit CEIL on each expression,
              and an implicit MAX on the set of expressions. */
     /* k==1 upper bound;
        Note: there is an implicit FLOOR on each expression,
              and an implicit MIN on the set of expressions. */
    for(k=0; k<2; k++)
      {
        /* total number of expressions in the lower/upper bounds */
        fscanf(fp, "%d", &n);
        for(r=0; r<n; r++)
          {
            expr = la_expr_new(depth, blobs);
            /* read in the coef */
            for(j=0; j<depth; j++)
              fscanf(fp, "%d", &(LA_EXPR_COEF(expr)[j]));
            /* read in the constant */
            fscanf(fp, "%d", &(LA_EXPR_C0(expr)));
            /* read in the blob coef */
            for(j=0; j<blobs; j++)
              fscanf(fp, "%d", &(LA_EXPR_BLOB_COEF(expr)[j]));
            /* read in the denominator */
            fscanf(fp, "%d", &(LA_EXPR_DENOM(expr)));
            if(k==0)
              {
                /* add the expr to the low bound list */
                LA_EXPR_NEXT(expr) = LA_LOOP_LOW(loop);
                LA_LOOP_LOW(loop) = expr;
              }
            else
              {
                /* add the expr to the upper bound list */
                LA_EXPR_NEXT(expr) = LA_LOOP_UP(loop);
                LA_LOOP_UP(loop) = expr;
              }
          }/* end of r-loop */
      }/* end of k-loop */
    /* read in the stepsize */
    fscanf(fp, "%d", &(LA_LOOP_STEP(loop)));
    LA_NEST_LOOPS(nest)[i] = loop;
```

```
    }
  LA_NEST_BLOBS(nest) = blobs;
  LA_NEST_DEPTH(nest) = depth;
  printf("The original loop nest: \n");
  /* The loop index variables are i, j, k... */
  la_nest_print( nest, 'i');
  /* The transformation is T. */
  new_nest = la_nest(nest, T);
  printf("The new loop nest: \n");
  /* The loop index variables are u, v, w... */
  la_nest_print( new_nest, 'u');
```

The above code segment explained the data structure for storing the loop bounds, where each lower bound can be the *max* of a set of linear expressions, and each upper bound can be the *min* of a set of linear expressions. The function CEIL is assumed for each expression in the lower bound, and the function FLOOR is assumed for each expression in the upper bound.

However, the new loop bounds can be more complex. We display the loop type below again for convenience, and use a simple example to illustrate the loop bounds being represented.

```
typedef struct _la_loop{
  LA_EXPR_T low;
  LA_EXPR_T up;
  int step;
  LA_EXPR_T offset;
} *LA_LOOP_T;
```

For example, the field *low* is a linked list of two expressions $(i+j)/2$ and $(2i-j)/3$; the field *up* is a linked list of three expressions i, $j+3$ and $(5i+7j)/2$; the step size is 2; and the *offset* [1] is a linear expression $2i+3j$. Then the loop (call it loop k) has the following bounds:

```
for k = 2*i+3*j + 2*max( ceil((i+j)/2), ceil((2*i-j)/3) ),
        2*i+3*j + 2*min( i, j+3, floor((5*i+7*j)/2) ),
        step 2
```

Note that the offset expression is in both the lower bound and the upper bound. Both the lower bound and the upper bound are multiplied by the step size as well.

[1] The offset is always one single expression.

8 Summary

This document describes the Lambda loop transformation toolkit, an implementation of the non-singular matrix transformation theory described in [8] that can represent any linear one-to-one transformation. The toolkit has a simple interface, and is independent of the intermediate representation used in the compiler.

9 Acknowledgments

We would like to thank Donna Bergmark and David Presberg of the Cornell Theory Center, and Richard Schooler and Bob Gottlieb of Hewlett-Packard for their helpful feedback during the development of the toolkit and the writing of this document.

References

[1] E. Ayguade and J. Torres. Partitioning the statement per iteration space using non-singular matrices. In *Proceedings of The 1993 ACM International Conference on Supercomputing*, July 1993.

[2] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, August 1990.

[3] D. Bergmark and D. Presberg. Initial experiments in the integration of Parascope and Lambda. Technical Report CTC93TR136, Cornell Theory Center, 1993.

[4] W. Kelly and W. Pugh. Generating schedules and code within a unified reordering transformation framework. Technical Report CS-TR-2995, Dept. of Computer Science, University of Maryland, College Park, November 1992.

[5] W. Li. Compiler optimizations for cache locality and coherence. Technical Report 504, Department of Computer Science, University of Rochester, April 1994.

[6] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proc. 5th Annual Workshop on Languages and Compilers for Parallelism*, August 1992.

[7] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 1993.

[8] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallelism, Lecture Notes in Computer Science 757*. Springer-Verlag, 1993.

[9] L. Lu. A unified framework for systematic loop transformations. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 28–38, April 1991.

[10] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. of Supercomputing*, 1992.

[11] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations. In *SIGPLAN'92 Programming Language and Implementation Conference*, June 1992.

[12] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[13] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman Publishing, London, 1989.

Index