The handle http://hdl.handle.net/1887/25180 holds various files of this Leiden University dissertation.

**Author:** Rietveld, K.F.D.  
**Title:** A versatile tuple-based optimization framework  
**Issue Date:** 2014-04-10
CHAPTER 9

Transformations for Automatic Data Structure Reassembly

9.1 Introduction

The first part of this thesis discussed how the forelem framework arose from the unification of code optimization in seemingly distinct fields of programming: transactional (database) applications and other (imperative) applications, and how this framework unifies these distinct fields of programming by expressing queries in an intermediate representation as a series of tuple accesses governed by simple loop control. Subsequently, this intermediate representation is optimized by traditional optimizing compiler techniques, accomplishing results similar to query optimization. In Chapter 7 the forelem framework was used to perform vertical integration of database applications, where queries in a database application are replaced with code segments that evaluate these queries using direct access to a local data store. Subsequently, the application and data access codes are optimized together.

In the second part of this thesis, the foundation of the forelem framework will be generalized and the use of the forelem framework for different code optimization problems will be discussed. Because the forelem framework was initially envisioned for database applications, its main features rely on viewing data as being stored as (multi)sets of tuples. The access of data through a tuple space is thus the main characteristic of the forelem framework. As a consequence, we propose problems from different application domains to be (automatically) expressed in terms of tuples, which enables the forelem framework to be used for optimization of these problems. For example, sparse matrix computations are characterized by the fact that next to the values, the column index and row index play an essential role. It is this relation, which can be naturally expressed as a tuple.

Next to accessing data as tuples, the forelem framework allows the execution order of tuple computations (transactions) to be out of order. This feature together with the possibility of presenting data access without having to specify the exact
data storage enables the forelem framework to automatically generate storage formats. Because of this out of order execution, application of compiler optimizations has to be carefully handled. As standard compiler optimizations rely on data dependence analysis and loop-carried dependencies, and these loop-carried dependencies are non-existing in forelem loop nests, the conditions under which the transformations can be applied have to be reconsidered, as has been discussed in Chapter 3.

In this chapter, the forelem framework is extended to define compiler transformations that operate on three levels: the tuple level, the materialized loop index level and the concretized data access level. The forelem tuple level provides an elegant representation method for expressing different data access codes such as database queries and sparse matrix algebra. Within the materialized loop index level, index sets on the tuple space that specify access patterns are being represented as array accesses. This is done without specifying how the tuples or arrays are actually stored. By giving the compiler transformation framework access to this second level of data access, the compiler can address the order of data access while the order of execution is not specified. Finally, within the concretized data access level, loops are expressed using regular (integer) iteration bounds. At this level, standard compiler optimizations can be applied taking into account the different semantics for data dependencies.

This chapter is organized as follows: in Section 9.2 the forelem intermediate representation is reiterated and generalized such that it applies to irregular computations as well. Section 9.3 demonstrates how Sparse BLAS routines are expressed in the forelem intermediate representation. Section 9.4 introduces the orthogonalization transformation, that can be used to impose a certain order on the iteration of the data. This is a preparatory step to materialization, discussed in Section 9.5. In this section, the process of transforming a loop to the materialized loop index level is defined and several transformations applicable at the materialized loop index level are described. Section 9.6 describes a number of transformations that can be applied on materialized forelem loops, influencing the data storage format that is generated. Section 9.7 outlines how loops are converted to the concretized data access level. In Section 9.8, the results of initial experiments performed with an important kernel, sparse matrix times $k$ vector multiplication, are presented. Section 9.9 concludes this chapter.

### 9.2 The Forelem Intermediate Representation

In this section, the basics of the forelem intermediate representation as introduced in Chapter 3 will be briefly reiterated and be generalized such that it applies to irregular computations as well. As has been discussed, the intermediate representation is centered around the forelem loop construct. Each forelem loop iterates over a specific array of structures. The subscripts of this array that are accessed are fetched from an “index set” that is associated with the array.

The arrays of structures that are iterated by forelem loops are modeled after database tables which are defined as multisets. The structure reflects the format of a database tuple. The intermediate representation operates at the tuple level, at
which is it not determined how the tuples are stored. For instance, the tuples are stored either row-wise or column-wise. In the latter case, a structure of arrays is iterated. In an array of structures \( A \) a tuple at index \( i \) is accessed with \( A[i] \) and a specific field \( \text{field1} \) in that tuple is accessed with \( A[i].\text{field1} \).

An index set is a set containing subscripts \( i \in \mathbb{N} \) into an array. Since each array subscript is typically processed once per iteration of the array, these subscripts are stored in a regular set. Index sets are named after the array they refer to, prefixed with “\( p \)”. For example, \( pA \) is the index set of all subscripts into an array \( A \): \( \forall s \in A : \exists i \in pA : A[i] = s \). Random access of an index set by subscript is not possible, instead all accesses are done using the \( \in \) operator.

The body of a \texttt{forelem} loop typically performs an action on the tuple subscripted by the current value of the loop iterator. When used in the context of database codes, the loop body often outputs tuples to a temporary or result set. Temporary sets are generally named \( T_1, T_2, ..., T_n \) and result sets \( R_1, R_2, ..., R_n \). In the context of, for example, sparse matrix codes a computation is typically performed also involving data from dense matrices or vectors. Results could be stored in a dense array.

Considering an array \( A \) with fields \( \text{field1} \) and \( \text{field2} \), a \texttt{forelem} loop that iterates all entries of \( A \), outputting the value of \( \text{field1} \) of each row, is written as follows:

\[
\text{forelem} \ (i; \ i \in pA) \\
R = R \cup (A[i].\text{field1})
\]

Although the \texttt{forelem} loop appears to be very similar to a \texttt{foreach} loop that exists in many common programming languages, \texttt{forelem} loops distinguish themselves with the use of the index sets. Every \texttt{forelem} loop iterates a single array, using subscripts from an index set that is associated with that array. Note that, the order of the subscripts in the index set is undefined. The only thing that is defined is which subscripts are to be iterated, but not in which order. As such, \texttt{forelem} loops do not have explicit looping structures and the exact semantics of the iteration of an array are determined in the course of the optimization process. Index sets are the essence of \texttt{forelem} loop nests as they encapsulate iteration and simplify the loop control so that aggressive compiler optimizations can be successfully applied.

Using conditions on index sets it is possible to narrow down the range of the array that is iterated. For example, the index set denoted by \( pA.\text{field2}[k] \) contains only those subscripts into \( A \) for which \( \text{field2} \) has value \( k \). This is expressed mathematically as follows:

\[
pA.\text{field2}[k] \equiv \{ i \mid i \in pA \land A[i].\text{field2} = k \}
\]

So, to only iterate entries of \( A \) in which the value of \( \text{field2} \) is 10, the following \texttt{forelem} loop is used:

\[
\text{forelem} \ (i; \ i \in pA.\text{field2}[10]) \\
R = R \cup (A[i].\text{field1})
\]

Note, that \( pA.\text{field2}[10] \) is not expressed more explicitly as the exact execution of the loop will be determined during the optimization process. This index set
```c
forelem (i; i ∈ pC)
{
    int sum = 0;
    forelem (j; j ∈ pA.row[C[i].index])
        forelem (k; k ∈ pB.index[A[j].col])
            sum += B[k].value * A[j].value;
    C[i].value = sum;
}
```

Figure 9.1: Matrix-Vector Multiplication with sparse vectors.

might be explicitly generated (at compile- or run-time), combined with other index sets, moved or eliminated. Alternatively, during the optimization process it may be decided to create a variant of array \( A \) only containing the tuples to be iterated.

More sophisticated index sets are possible, such as having conditions on multiple fields, in this case on field1 and field2:

\[
pA.(\text{field1, field2})[(k_1, k_2)] = \{ i \mid i ∈ pA ∧ A[i].\text{field1} = k_1 ∧ A[i].\text{field2} = k_2 \}
\]

Instead of a constant value, the values \( k_n \) can also be references to values from another array. To use such a reference, the array, subscript into the array and field name must be specified, e.g.: \( A[i].\text{field} \). To select values \( \text{field1} > 10 \) an interval is used: \((10, ∞)\).

### 9.3 Expressing Sparse BLAS routines in Forelem

In this section, we will demonstrate how Sparse BLAS routines are expressed in the `forelem` intermediate representation. Sparse structures are considered to be sets of tuples. A sparse matrix is represented using tuples of the form \((\text{row}, \text{column}, \text{value})\). Sparse vectors can be represented using \((\text{index}, \text{value})\). When consider tables to only contain a single tuple for every unique \((\text{row}, \text{column})\) pair or index.

As a first routine, we consider the Matrix-Vector Multiplication \( C = AB \). Figure 9.1 shows this multiplication where \( C \) and \( B \) are considered to be sparse vectors and are thus represented as tables\(^1\). Note the repeated use of index sets to define which tuples should be processed within an iteration. Figure 9.2 shows the `forelem` representation for the same operation, but with \( C \) and \( B \) are dense vectors.

Other BLAS routines can be similarly expressed. In Figure 9.3 an implementation of Triangular Solve \( Tx = B \) using `forelem` loops to access a matrix \( T \) is

\(^1\)For the interested reader, the SQL specification of this representation is:

```
select distinct A2.row, (select sum(B.value * A.value) from A, B where B.index = A.col and A.row = A2.row) from A A2;
```
9.4. Orthogonalization

```c
for (i = 1; i <= N; i++)
{
    int sum = 0;
    forelem (j; j ∈ pA.row[i])
    C[i] = sum;
}
```

Figure 9.2: Matrix-Vector Multiplication with dense vectors.

```c
for (i = N; i >= 1; i--)
{
    forelem (j; j ∈ pT.(col,row)[(i, i)])
        x[i] = b[i] / T[j].value
    forelem (j; j ∈ pT.col[i])
}
```

Figure 9.3: An implementation of Triangular Solve $Tx = b$ written in the `forelem` intermediate representation.

presented. As an additional example, Figure 9.4 shows an implementation of LU Factorization. Note that every loop over the same sparse matrix $A$ defines a different set of matrix elements to be iterated.

9.4 Orthogonalization

In `forelem` loops, iteration of a table of tuples is controlled by the index set. No order is defined on the index set, which has as consequence that the iteration order of the table is undefined. In this section, the orthogonalization transformation is introduced, which makes it possible to impose a certain order in which the table is iterated. This is achieved by partitioning the accesses to the array based on the values of one or more table fields. The orthogonalization transformation is used to control the order in which data is accessed as a preparatory step to Materialization, which is discussed in the next section.

Let $A$ be a table with `field1`, `field2`, ..., `fieldn`. Consider the loop:

```c
forelem (i; i ∈ pA)
    ... A[i] ...
```

In this loop, the tuples of $A$ can be iterated in any order. As an example, assume an iteration order is to be imposed on $A$ such that tuples $A$ are accessed in blocks with equal values for `field1`. The orthogonalization transformation is carried out to achieve this, resulting in the following loop nest:
for (i = 1; i <= N; i++)
{
    p = diag(i)
    forelem (j; j ∈ pA.(col,row)[(i, (i, ∞))])
    {
        forelem (l; l ∈ pA.(row,col)[(i, (i, ∞))])
        {
            fillin = True
            {
                fillin = False
            }
            if (fillin)
        }
    }
}

Figure 9.4: An implementation of LU Factorization written in the forelem intermediate representation.

forelem (ii; ii ∈ A.field1)
    forelem (i; i ∈ pA.field1[ii])
        ... A[i] ...

A.field1 in the outer loop denotes all possible values of field1 that occur in A. So, the iteration space of the outer loop consists out of every value of field1 in A.

The original loop iterates all tuples of A. The transformed loop nest will for every value of field1, iterate all tuples of A for which field1 equals this value. As a result, the transformed loop also iterates all tuples of A. Application of the orthogonalization transformation is not limited to a single field. An example of orthogonalization on two fields is:

forelem (ii; ii ∈ A.field1)
    forelem (jj; jj ∈ A.field2)
        forelem (i; i ∈ pA.(field1,field2)[(ii,jj)])
            ... A[i] ...

The outer loops that are introduced by the orthogonalization transformation iterate all values of a given table field. If it is possible to express this range of values as a subset of the natural numbers, i.e. $A.field1 \subseteq \mathbb{N}$, the encapsulation transformation can be applied, which replaces the loop over all table field values with a loop over a subset of the natural numbers. With the encapsulation transformation, a loop
forelem (ii; ii ∈ A.field1)

where $A.field1 = \{1, 2, 6, 7, 8, 10\}$, is replaced with:

forelem (ii; ii ∈ $N_{10}$)

with $N_{10} = [1..10]$. In the encapsulated loop, the values 3, 4, 5, 9 will be iterated, but note that no tuple will exist where field1 equals any of these values. As a result, the inner loop is not executed for these values, maintaining the iteration space of the original loop.

9.5 Materialization

In this section, the materialization transformation is described, which materializes the tuples iterated by a forelem loop using the accompanying index set to an array in which the data is represented in consecutive order and is accessed with integer subscripts. Although this can be seen as a simple normalization operation, it is an important enabling step that allows the compiler to address and modify the order of data access to these arrays. In fact, by materialization the execution order of an inner loop is fixed. (In the case of nested loops, orthogonalization fixes the order of the outermost loop). After two forms of materialization have been introduced, a number of transformations targeting the order in which data access takes place will be described.

A distinction is made between loop-independent and loop-dependent materialization. In loop-independent materialization, conditions in the index set of the loop to be materialized are not dependent on one of the outer loops. Materialization will result in a one-dimensional array. In loop-dependent materialization, the resulting array will get an additional dimension for each dependent loop. Both cases of materialization will now be discussed in turn.

9.5.1 Loop Independent Materialization

We first consider loop-independent materialization. The following loop iterates all tuples of $A$ whose field equals a value $X$:

forelem (i; i ∈ pA.field[X])

... A[i] ...

To be able to determine which tuples of $A$ to access, the index set is used. This is, in fact, a indirection level. This indirection can be removed by materializing the index into the tuple space as an array $PA$ which only contains the entries of $A$ that should be visited by this loop. This results in:

forelem (i; i ∈ $N_*$)

... PA[i] ...

with $N_* = [1, |PA|]$. The array $PA$ only contains elements from $A$ for which the condition $A[i].field == X$ holds. The compiler is now enabled to address the order
in which the data in PA is accessed, while the execution order of the loop is not specified. For example, using the transformations that can be applied on the materialization form, which are described below, the compiler can determine to put entries in PA in a specific order. The loop control is selected at the concretization stage, where the compiler can ensure the loop control for the loop will iterate the items of PA consecutively. For the general definition of loop-independent materialization, consider a loop iterating a sparse structure A:

```plaintext
forelem (i; i ∈ pA)
    ... A[i] ...
```

which is transformed to:

```plaintext
forelem (i; i ∈ N*)
    ... PA[i] ...
```

with $N^* = [0, |PA|)$. This transformation materializes the sparse structure A to an one-dimensional array PA.

The transformation can also be applied if the loop to be materialized is nested in another `forelem` loop and the posed condition in the index set of the loop to be materialized is independent of the outer loop. Consider, for example, where the outer loop could be the result of the application of the encapsulation transformation:

```plaintext
forelem (i; i ∈ N_n)
    forelem (j; j ∈ pA.field[X])
        ... A[j] ... B[i] ...
```

Materialization of the inner loop will enable the compiler to address the order of data access of A together with the other array or tuple space references. Materialization of the inner loop proceeds as explained above and the outer loop is untouched:

```plaintext
forelem (i; i ∈ N_n)
    forelem (j; j ∈ N*)
        ... PA[j] ... B[i] ...
```

with $N^* = [1, |PA|]$ and PA only containing items that satisfy the condition.

### 9.5.2 Loop Dependent Materialization

If a loop to be materialized is contained in a loop nest and the conditions of its index set have a dependency on another loop, then the above described loop-independent materialization cannot be applied. Instead, loop-dependent materialization must be used, which is described in this section. Because loop-dependent materialization will result in higher-dimensional arrays, this results in more opportunities for the compiler to address and modify the order of data access to these arrays. In general, a loop-dependent materialization has the form:
The index set iterated in the inner loop has a dependency on one or more of the outer loops. The iteration of \( A \) is materialized to an iteration of a multidimensional array \( PA \), in which each loop-dependent condition is represented as an additional dimension in \( PA \). The array \( PA \) only contains these items that are iterated by the original index set on \( A \):

\[
\text{forelem } (i; i \in N_o)
\]
\[
\text{forelem } (n; n \in N_n)
\]
\[
\text{forelem } (p; p \in PA.(\text{field}_i, \ldots, \text{field}_n)[(i, \ldots, n)])
\]
\[
\ldots A[p] \ldots
\]

with \( N_\ast = [0, |PA[i]| n] \). After this transformation, \( PA \) only contains entries that satisfy the conditions of the original index set. The dimensions of the materialized array correspond with the original conditions and thus with the loops on which the condition depended. Loop transformations, such as Loop Interchange, will thus have an effect on the order in which the data of \( PA \) is accessed. By taking this into account, the compiler can determine an efficient order in which to store the elements of \( PA \), which has at this point not been set in stone.

To illustrate the loop-dependent materialization, consider a simple nested loop:

\[
\text{forelem } (i; i \in N_n)
\]
\[
\text{forelem } (j; j \in \text{pA.row}[i])
\]
\[
\ldots A[j] \ldots
\]

The index set of the inner loop, \( \text{pA.row}[i] \) is dependent on iterator \( i \) of the outer loop. As a consequence, the array \( PA \) will obtain a dimension for this iterator \( i \). The result of the materialization transformation is as follows:

\[
\text{forelem } (i; i \in N_n)
\]
\[
\text{forelem } (j; j \in N_\ast)
\]
\[
\ldots PA[i][j] \ldots
\]

with \( N_\ast = [0, |PA[i]|] \). Because \( i \) was determining which row of \( A \) was iterated, in the transformed loop \( i \) still controls the order in which the rows of the original matrix \( A \) are accessed in the materialization \( PA \).

In case the index set has dependencies on two loops, a three-dimensional array is generated. Naturally, this has more degrees of freedom for optimization than the two-dimensional materialization. The application of the transformation is similar as in doubly-nested loops. In this example, the index set has dependencies on two different outer loops:
forelem \( (i; i \in \mathbb{N}_n) \)
forelem \( (j; j \in \mathbb{N}_m) \)
forelem \( (k; k \in pA.(\text{row, col})[(i,j)]) \)
\[ A[k].value = \ldots \]

This results in a three-dimensional array \( PA \):

forelem \( (i; i \in \mathbb{N}_n) \)
forelem \( (j; j \in \mathbb{N}_m) \)
forelem \( (k; k \in \mathbb{N}_\ast) \)
\[ PA[i][j][k].value = \ldots \]

with \( \mathbb{N}_\ast = [0, |PA[i][j]|) \).

### 9.5.3 Materialization Combined with Other Transformations

We will now demonstrate how the combination of materialization with other transformations within the \textit{forelem} framework leads to the automatic generation of different data storage formats for a particular problem. In this section, Matrix-Vector Multiplication is considered with a sparse matrix \( A \) and dense vectors \( B \) and \( C \):

forelem \( (i; i \in \mathbb{N}_m) \)
\[ C[i] = 0; \]

forelem \( (i; i \in \mathbb{N}_m) \)
forelem \( (j; j \in pA.\text{row}[i]) \)
\[ C[i] += B[A[j].\text{col}] \ast A[j].value; \]

In the remainder of this example we will focus on the second loop and consider that the first loop initializing \( C \) is left untouched. On the inner loop of this second loop nest, the materialization transformation will be applied. Because the argument to the index set of the inner loop depends on the outer loop, loop-dependent materialization will be performed. This will result in a two-dimensional array \( PA \):

forelem \( (i; i \in \mathbb{N}_m) \)
forelem \( (j; j \in \mathbb{N}_\ast) \)
\[ C[i] += B[PA[i][j].\text{col}] \ast PA[i][j].value; \]

Note that the arrays \( PA[i] \) will for every \( i \) only contain these elements of \( A \) that satisfied \( A[j].\text{row} = i \). As a next step, the loops are interchanged using the Loop Interchange transformation. This is possible because no loop-carried dependencies are present as the \textit{forelem} loops do not guarantee a specific iteration order:

forelem \( (j; j \in \mathbb{N}_\ast) \)
forelem \( (i; i \in \mathbb{N}_m) \)
\[ C[i] += B[PA[i][j].\text{col}] \ast PA[i][j].value; \]

The iterator \( j \) still controls which entry within the current row (indicated by \( i \)) is visited. These entries may not necessarily have the same column index, as entries
which are zero are not present in PA. So, this loop nest will for each column number (outer loop) iterate all rows. This is the essence of the jagged diagonal storage format, which in consecutive rows stores all first nonzero column entries of all rows, all second nonzero column entries of all rows, and so on. The corresponding column indices are stored in a separate array. Important is that this particular storage format has been deduced without any predefinition of this format in the framework. As will be described in Section 9.7, different variants of this jagged diagonal storage format can be concretized.

9.6 Transformations on the Materialized Form

After a forelem loop has been put in a materialized form, the data to be processed has been put in an array in consecutive order and is accessed with integer subscripts. At this stage, the compiler can modify the exact order of data access to these arrays and how this data is stored. In this section a number of transformations are described that affect the storage of the data processed by a loop nest.

9.6.1 Horizontal Iteration Space Reduction

The aim of Horizontal Iteration Space Reduction is to reduce unused fields from a table’s schema. In fact, it is possible to perform this transformation before the materialization stage.

Formally, the transformation is defined as follows. Let $T$ be a table with fields $field1, field2, field3, field4$, and $C$ a list of condition fields $C \subseteq (field1 field2)$ and $V$ a list of values. Consider the loop nest:

$$\text{forelem } (k; \ k \in pT.C[V])$$

$$R = R \cup T[k].field1 + T[k].field2$$

We define a new table $T' \subseteq T$ with fields $field1, field2$ and replace the use of $T$ with $T'$ in the loop.

9.6.2 Structure splitting

Before materialization tables are represented as multisets of tuples, accessible with integer subscripts. By default, the array that is the result of the materialization operation is an array of tuples or structures. In some cases, it is more efficient to use a structure of arrays, i.e. the structures are split [94, 26]. Within the forelem framework this is defined as the structure splitting transformation. Consider the materialized loop nest:

$$\text{forelem } (i; \ i \in N_m)$$

$$\text{forelem } (k; \ k \in N_*)$$

$$... \text{PA}[i][k].value ...$$

Structure splitting will modify the data storage of the array and convert the data accesses in the loop to:
forelem (i; i ∈ \mathbb{N}_m)
    forelem (k; k ∈ \mathbb{N}^*)
    ... PA.value[i][k] ...

9.6.3 \(\mathbb{N}^*\) materialization

Materialized loops use the \(\mathbb{N}^*\) index set as the set of integer subscripts to access the materialized array. How exactly these integer subscripts are stored is initially encapsulated within \(\mathbb{N}^*\) and can be made explicit using \(\mathbb{N}^*\) materialization. Consider the following loop, the result of a materialization to \(PA\):

forelem (i; i ∈ \mathbb{N}_m)
    forelem (k; k ∈ \mathbb{N}^*)
    ... PA[i][k] ...

As a prerequisite for the final code generation stage, \(\mathbb{N}^*\) must be made explicit. This can be achieved by converting \(\mathbb{N}^*\) to a set \(PA\_len\). There are different means in which this set can be defined. The first is to define the set as follows:

\[ PA\_len[q] = \max(\text{len}(PA[q])) \]

in which case all \(PA\_len[q]\) values are the same and a single set containing integers up to the maximum value can be stored for this loop nest. Padding is inserted in the array \(PA\) for the values \(PA[i][k]\) with \(k \geq PA\_len[i]\). The second way to create this array is to avoid inserting padding in \(PA\). In this case \(PA\_len[q] = \text{len}(PA[i])\).

Regardless of which implementation is chosen, the resulting loop after \(\mathbb{N}^*\) materialization is:

forelem (i; i ∈ \mathbb{N}_m)
    forelem (k; k ∈ PA\_len[i])
    ... PA[i][k] ...

Note that in this loop the iteration order is still undefined. Only \(\mathbb{N}^* = [0, \mathbb{N}^*]\) has been replaced with \(PA\_len[i] = [0, PA\_len[i]]\). In a subsequent concretization step the iteration order will be determined. For example, the loop:

forelem (k; k ∈ PA\_len[i])

is concretized to:

for (k = 0; k < PA\_len[i]; k++)

9.6.4 \(\mathbb{N}^*\) sorting

In case of loop-dependent materialization, \(\mathbb{N}^*\) encapsulates the sets of integer subscripts used for iteration of the inner loop. These sets are ordered irrespective of their cardinality. If the loop is to be parallelized, it is beneficial if the work is divided into blocks with evenly sized values for \(PA\_len\) (after \(\mathbb{N}^*\) materialization). One way to achieve this is by imposing an order on the iteration of \(\mathbb{N}^*\).

The aim of \(\mathbb{N}^*\) sorting is to find an order of the iterator values \(i\) such that the value of \(\mathbb{N}^*\) decreases with subsequent iterations of the outer loop on \(i\):
Consider that $N^* = [0, \text{len}(PA[i])]$. The goal is to iterate through $N^m$, such that $\text{len}(PA[i])$ decreases. Let $\text{perm}(N^m)$ store the permutation of $N^m$ for which this holds. Then, the loop is transformed to:

```plaintext
forelem (i; i ∈ perm(N^m))
  forelem (k; k ∈ N*)
    ... PA[i][k] ...
```

Note that this will affect the order of the data $PA$, which will be put in the corresponding sorted order at the concretization stage.

### 9.6.5 Dimensionality Reduction

Loop-dependent materialization results in a multi-dimensional array by default. If this array is concretized as a multi-dimensional array, padding may have to be inserted for the uneven lengths of the rows. It is possible to avoid the introduction of this padding by storing the rows back to back. This reduces the dimensionality of the materialized array. Consider the loop nest:

```plaintext
forelem (i; i ∈ N^m)
  for (k = 0; k < PA_len[i]; k++)
    ... PA[i][k] ...
```

to reduce the dimensionality of the materialized array $PA$ by one, this is transformed into:

```plaintext
forelem (i; i ∈ N^m)
  for (k = PA_ptr[i]; k < PA_ptr[i+1]; k++)
    ... PA[k] ...
```

Based on the $PA_len$ array, a new $PA_ptr$ array is introduced, which keeps track of the start and end of each row in $PA$. Note that the order of the iteration domain $[PA_ptr[i], PA_ptr[i + 1])$ does not have to be defined and could be in any order.

### 9.7 Concretization

Concretization is a simple one-to-one mapping from a given materialized loop to a C `for` loop that can be compiled by a regular C compiler. So, a `forelem` loop iterating a subset of integers is transformed into a regular `for` loop. This transformation selects a specific iteration order for the subset of integers. Essentially, at this point the data storage format is generated that has been chosen by the optimization process. Using the different transformations that can applied on a materialized loop, described in the preceding section, many different storage formats can be generated for a single loop nest.

To describe the basic concretization transformation, consider the following loop as an example, which is the result of a materialization transformation:
forelem (i; i ∈ N*)
... PC[i] ...

As a first step, N* materialization is applied, resulting in:

forelem (i; i ∈ PA_len)
... PC[i] ...

then the loop can be subsequently concretized to:

for (i = 0; i < PA_len; i++)
... PC[i] ...

At this point, a data storage format has been chosen by the optimization process. For this particular, single-dimensional, case, storage as an array of consecutive values is the most likely candidate. Note that, this storage format is the result of just merely a straightforward mapping of the materialized index set into a (multi)dimensional data structure. This cannot be compared to substituting coordinate storage by jagged diagonal storage, or the immediate selection of such a pre-defined format.

To better illustrate the possibilities within the concretization process, we will continue the Sparse Matrix-Vector Multiplication example from Section 9.5.3:

forelem (j; j ∈ N*)
forelem (i; i ∈ N_m)
C[i] += B[PA[i][j].col] * PA[i][j].value;

In this example the data access to PA is to be concretized. As a first step, the loop with iterator variable i is concretized:

forelem (j; j ∈ N*)
    for (i = 1; i <= m; i++)
        C[i] += B[PA[i][j].col] * PA[i][j].value;

Given that iterator variable j indicates which column number to process, the concretized inner loop will now iterate all rows to process in consecutive order. The outer loop can be concretized to iterate the column numbers in consecutive order. To do this, first N* materialization is applied. If N* is converted to a set PA_len such that PA_len[q] = max(len(PA[q])), all values in the set are the same, so a further conversion is possible to a single constant value, say k. The outer loop can then be concretized to result in:

for (j = 1; j <= k; j++)
    for (i = 1; i <= m; i++)
        C[i] += B[PA[i][j].col] * PA[i][j].value;

These steps have led to a certain storage scheme for PA. This storage scheme consists of a two dimensional array which has a row for each column number n, containing all nonzero column entries at position n in the different rows. Unused entries are padded with zero, so that every row has the same length. This enables
that a generic two dimensional array can be used as storage scheme. If code is
generated for the C language, which uses row-major order for array storage, then
the rows containing the column values and indices must be stored one after the
other. The resulting array should be accessed with $PA[column][row]$, which is in
fact different from the order of the subscripts in the current $forelem$ representation.
So, as a result the following C code will be produced for this storage format:

```c
for (j = 1; j <= k; j++)
    for (i = 1; i <= m; i++)
        C[i] += B[PA[j][i].col] * PA[j][i].value;
```

where $k$ will be a constant indicating the maximum number of non zero columns
in a row in the resulting array $PA$.

Different transformations could have been applied in between the materializa-
tion and concretization steps to influence the data storage format that is generated
in the end. For example, the row field that is part of the original tuples is not used
in the code fragment. Using Horizontal Iteration Space Reduction, such unused
fields are eliminated. It is also possible to store the col and value fields in sep-
arate arrays. To accomplish this, the structure splitting transformation must be
performed before concretization. The concretized result in C code will be:

```c
for (j = 1; j <= k; j++)
    for (i = 1; i <= m; i++)
        C[i] += B[PAcol[j][i]] * PAvalue[j][i];
```

Note that this generated storage scheme is described in the literature as simplified
Jagged Diagonal Storage, or ITPACK storage [12]. So, through the described trans-
formations, orthogonalization, materialization and concretization, many different
loops and accompanying data storage formats can be generated that achieve the
same result. Figure 9.5 illustrates how such data formats are generated, starting
from an unordered set of tuples. Established data storage formats, such as IT-
PACK and Jagged Diagonal Storage format [12], simply follow from the applica-
tion of the transformations described in this chapter. For example, the transforma-
tion sequence drawn in black in this figure is the sequence leading to the ITPACK
storage. The $N*$ materialization step is considered to be part of the concretization
step in this figure. Alternatively, when the structure splitting transformation in the
figure is followed by dimensionality reduction, Compressed Row Storage (CSR)
format is generated. Similarly, a transformation sequence that continues from or-
thogonalization on column can result in Compressed Column Storage (CCS) for-
mat.

Let us consider a different application of the transformations. If the alternative
form of $N*$ materialization is applied, a set $PA_{len}$ is generated such that no zeros
have to be inserted into the data structure as padding. Through the application of
dimensionality reduction, the rows stored in memory, that thus contain column
entries, will be stored back to back in a vector. As a consequence, an additional
data structure is added to record the start of each row. When $N*$ sorting is applied,
rows with similar number of entries are placed close to each other, for example
by reordering the rows of the matrix by sorting on the number of non-zeros per
Figure 9.5: An illustration of the application of orthogonalization, materialization and concretization on sparse matrix tuples in (row | col | value) format. The result of this concretization is commonly known as the ITPACK format (assuming the arrays are stored in column-major order). The arrows displayed in gray depict a non-exhaustive set of other possibilities.
row. This can be helpful as a form of load balancing in the case of parallelization. Although this transformation changes the order in which the rows are processed, this does not introduce a problem because before the concretization the iteration of the rows is specified as a \texttt{forelem} loop which does not impose a particular execution order. Note that this concretization leads to the Jagged Diagonal Storage format described in the literature [12]. This storage format follows from the application of the generic transformations, contrary to be devised by hand which was the only way to arrive this storage format up till now.

### 9.8 Initial Experimental Results

In this section, initial experimental results are presented of the performance of codes and data storage formats optimized with the \texttt{forelem} framework. It is demonstrated that these optimized codes are comparable in performance to hand-optimized sparse routines using the CUDA framework. As an example, the sparse Matrix-Vector Multiplication will be considered, with a sparse matrix $A$ and dense vectors $B$ and $C$. From the materialization of this loop that has been described in the previous section, three different concretizations have been generated for which CUDA code has been generated:

- **Simple JDS**: in this case, every row contains the nonzero column entries of a single row in consecutive order. This is thus the result of a concretization if no Loop Interchange was performed during the materialization.

- **Simple JDS 2**: in this case, every row contains the nonzero column entries at position $n$ of every row. This is the format generated in that previous section that is similar to ITPACK.

- **JDS**: Jagged Diagonal Storage format, as described in the previous section.

These generated implementations have been compared with Matrix-Vector Multiplication as implemented in the CUSP [14] library for different storage formats. The CUSP library provides several routines for performing sparse linear algebra on CUDA, which have been optimized for several, pre-defined, storage formats. The formats implemented in CUSP that we have benchmarked are: Coordinate format (COO), Compressed Sparse Row format (CSR) and a Hybrid (HYB) format. The Hybrid format employs a combination of the ELL format and COO format and is described in detail in [14].

For the benchmark, 15 matrices have been selected from The University of Florida Sparse Matrix Collection [28], also taking into account previous studies on sparse matrix times vector multiplication see [14, 98]. The selected matrices are listed in Table 9.1 and represent different problem classes.

The experiments have been performed on a workstation with a GeForce GTX 480 CPU with 1535MB of RAM. The multiplication operation has been repeated 1000 times. Figure 9.6 reports the execution time in milliseconds of 1000 Matrix-Vector Multiplications for the different matrices and implementations.

\footnote{In the next chapter, the search space of all possible concretizations will be explored and characterized}
Figure 9.6: Execution time in milliseconds of 1000 Matrix-Vector Multiplications for different matrices and implementations. Simple JDS, Simple JDS 2 and JDS are implementations generated with the \textit{forelem} framework. CUSP COO, CUSP CSR and CUSP HYB are CUSP implementations using different pre-defined storage formats.
Table 9.1: Details of the matrices that have been used in the benchmark.

The results indicate that the performance of the forelem generated kernels using automatically generated storage formats is in many cases comparable to the hand-optimized CUSP routines. Due to the varying characteristics of the different matrices, there is no all-round best storage format. For the majority of the matrices however, the Simple JDS 2 and JDS implementations clearly perform better than the Simple JDS implementation.

9.9 Conclusions

In this chapter, we have described the extension of the existing forelem framework with materialization techniques. These techniques enable the compiler to address the order of data access in a loop, while the order of execution of the loop is not specified. Through different applications of the materialization and concretization transformations, different data storage formats can be automatically generated.

As an application of these techniques, we demonstrated how Sparse BLAS routines can be expressed in the forelem representation and how the forelem framework automatically generates data storage formats and implementations of the data access codes. Initial experimental results demonstrated the effectiveness of this approach. Three forelem-generated CUDA implementations of sparse matrix-vector multiplication were compared to three hand-optimized CUDA implementations using three different, pre-defined, storage formats. The results show that the implementations generated with the forelem framework show performance that is comparable to hand-optimized code.

The Jagged Diagonal, ITPACK and ELLPACK data storage formats have been exemplified in quite some papers in the past. The main reason is that sparse matrix times vector multiplication is the main kernel for many large-scale simulations. In this chapter, we have seen that these storage schemes naturally arise from a basic
compiler transformation as Loop Interchange combined with index set materialization and therefore can be automatically derived.