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 11

Handling Data Dependencies In The Forelem Framework

11.1 Introduction

Irregular computations are characterized by non-strided, unpredictable memory access. This defeats hardware features such as caching based on spatial locality and memory pre-fetching. Commonly, these irregular accesses are caused by the use of pointer-linked data structures or, even worse, a data structure that is unsuitable for the computation that is carried out. Static compiler analysis breaks down on pointer-linked data structures, as the order in which pointer-linked entities are accessed cannot be determined at the time of compilation. As the compiler cannot analyze the access pattern, it is refrained from applying effective techniques to optimize memory access or to parallelize the program. Because compilers can in most cases not exert extensive control on how data is stored, barely any opportunity is left to improve on this situation.

To overcome these problems, a different programming and optimization paradigm is needed. In this paradigm, irregular computations should be expressed in a different manner, breaking down the used data structures and capturing the essence of the computation. This will expose more information to the compiler about the order in which data is visited and dependencies between data that were encapsulated in data structures. Furthermore, this paradigm must enable the compiler to restructure the storage of data, next to restructuring the computation. As a result, the number of opportunities to effectively optimize and parallelize the computation will be greatly increased.

In this chapter, the forelem framework is extended so that data dependencies can be handled properly allowing irregular computations to be fully optimized and parallelized. Irregular computations can be naturally described within the forelem framework as can be seen in the previous chapters. Any data structure that is used by the computation is reduced to tuples. Dependencies that are encapsulated in loop nests are made explicit as dependencies between tuples. Com-
Computations to be carried out are expressed as the iteration of a (sub)set of tuples and an operation based on these tuples. The major benefit of expressing computation on the elementary tuple level, rather than on the (complicated) data structure level, is that any obstructions introduced by the use of complicated data structures or unnatural encoding of dependencies are eliminated. As a consequence, the compiler is provided with more opportunities to automatically optimize the code. Furthermore, the compiler can explicitly control the way data is stored by modifying the structure and organization of the tuples. This leads to modifications to the data structure(s) used by the computation and, hence, the actual data structure(s) are constructed during the code generation phase.

Because dependency information is made explicit as dependencies between tuples in the tuple pool on which is operated, it is trivial to deduce which operations on tuples can be executed at the same time. This makes the proposed extension to the forelem framework especially suited for the automatic parallelization of codes. Using this extension, the central action for parallelization is to map a given specification of the computation onto an execution model. Two execution models will be described: affine embedding and static scheduling. The forelem framework that is extended is equipped with transformations that can be applied before and after the mapping onto a particular execution model, or execution scheduling. These transformations can be applied in many different ways, giving rise to a large optimization space. Effective application of these transformations leads to the generation of codes that are competitive with hand-optimized codes.

To take advantage of this paradigm, it is not necessary to rewrite existing codes to codes that operate on tuples. Rather, current implementations of irregular computations in for instance the C programming language can be automatically mapped into the tuple-based programming model. Consequently, sophisticated parallel codes can be generated from a starting point provided in the C programming language. We will demonstrate that from an ordinary triangular solve code written in C, parallelized implementations can be automatically produced that up till now could only be derived by hand. The performance of these automatically generated implementations is comparable to that of hand-optimized implementations.

This chapter is organized as follows: Section 11.2 introduces the programming model in which generic computations can be expressed in terms of tuples. Section 11.3 discusses how tuple-based expressions of computations are mapped onto an execution model. In Section 11.4 transformations are described that can be applied before and after execution scheduling. Section 11.5 discusses the applicability, versatility, university and transferability of the proposed programming model. In Section 11.6 a case study is presented, in which through the use of the proposed extension of the forelem framework implementations of triangular solve are derived from a starting point written in the C programming language. Section 11.7 presents our conclusions and plans for future work.
11.2 Handling Data Dependencies Between Tuples

In this section, the expression of dependencies between tuples is discussed. A method is introduced to explicate the dependencies between tuples in the tuple iteration structure, allowing for irregular applications to be naturally expressed in terms of loops processing tuples. First, we will give a small review on how tuples are handled by the forelem framework.

11.2.1 Iteration of Tuples

Data to be processed is specified as (multi)sets of tuples. The computation is expressed in terms of loops processing the tuples. Different transformations are implemented in the framework, ranging from standard compiler optimizations, such as Loop Interchange [4], Loop Fusion [52], Scalar Expansion and Def-Use analysis [2, 50], to transformations that address the order in which tuples are executed and stored.

The principal syntactic construct in the forelem framework is the forelem loop. A forelem loop iterates (a subset of) a multiset of tuples and performs an operation on these tuples. As an example, consider a multiset $T$ containing tuples with fields $\text{field1}$ and $\text{field2}$: $(\text{field1, field2})$. Then, the following loop sums the values of $\text{field1}$ of tuples of which $\text{field2}$ equals the value 9:

```
sum = 0;
forelem (i; i ∈ pT.field2[9])
    sum += T[i].field1;
```

Iteration of the forelem loop is controlled with the “index set” $\text{pT.field2[9]}$, which in this case contains all subscripts into $T$ for tuples of which $\text{field2}$ equals 9. The index set specifies which tuples will be visited, but does not specify the order in which these tuples are visited, which is undefined.

Naturally, forelem loops can also be nested. The value of a tuple in a tuple pool can be used to access tuples in another tuple pool, say $S$:

```
forelem (i; i ∈ pT.field2[9])
    forelem (j; j ∈ pS.field1[T[i].field1])
        sum += S[j].field2;
```

The index set conditions are designed such that they can be rewritten to a conditional clause of if statements. This property is used to rewrite loop nests into a form with the conditions tested in the innermost loop, enabling a variety of loop transformations to be performed. When this is done for the above loop nest, the result is:

```
forelem (i; i ∈ pT)
    forelem (j; j ∈ pS) {
        if (T[i].field2 == 9 &&
            S[j].field1 == T[i].field1)
            sum += S[j].field2;
    }
```
This loop nest will produce equivalent results, since the statement in the inner loop is executed for the same set of tuples from T and S. By moving the conditions to the innermost loop, it has been made possible to apply the loop interchange transformation, after which the conditions can be moved back from the inner loop to the corresponding index sets.

11.2.2 The Ready Clause

The most important property of the forelem loop construct is that through the use of index sets the tuples that should be visited are specified, but not in which order. Iteration of the selected tuples may happen in any order. In other words, forelem loops are inherently parallel.

For irregular codes, this is a problematic property, as statement instances in an irregular code commonly have a dependency on another statement instance to be executed first. In the case of linked list traversals, a certain element must be visited before it is known what the next, or previous, item to visit is. Matrix computation codes, such as triangular solvers, need to ensure writes to rows \( k \in [0, i) \) are completed before column \( i \) can be processed.

To accommodate the specification of such dependencies, we propose to extend the forelem framework, or in particular the index set capabilities, with a ready clause. The ready clause is an expression that for a given tuple \( t \) in tuple pool T specifies which tuples \( r \) in tuple pool T must have been visited. Using this method, dependencies can be set up between tuples in a tuple pool. As will be discussed in Section 11.5.1, a bijection can be set up between the iteration space of an original loop and tuples, which enables the ready clause to express dependencies between statement instances as well.

The ready clause naturally extends the index sets that are used to control iteration in forelem loops. An example of the syntax for this clause is:

```c
forelem (q; q \in pT.ready(r)[\n(T[r]) = T[q]])
SEQ;
```

where SEQ denotes a sequence of statements and \( \n(T[r]) = T[q] \) is the ready expression. For a more formal treatment of the ready expression, see Section 11.3. The index set \( pT \) will contain subscripts \( q \) into tuple pool T, for which all tuples \( T[r] \) in T that meet the specified ready condition have been processed. Additionally, the subscript \( q \) may not have been processed already. As a definition, this evaluation takes place before the forelem loop is entered. As a consequence, no further subscripts will be added to the index set while the loop is in progress, including new tuples that have become ready after any modifications to T that may have occurred in the body of the forelem loop.

An example of a ready expression is \( T[r] \cdot z = T[q] \cdot y \) (see Section 11.3 how this can be expressed formally), which specifies that in order to be able to process \( T[q] \), all tuples \( T[r] \) must have been visited that have a \( z \) field equal to the \( y \) field of \( T[q] \). In general, in ready clause expressions, the tuples addressed by \( r \) and the iterator variable of the loop in which the clause is embedded (in the case of the example \( q \)) are used as operands, and standard C operators such as \( ==, !=, | |, && \) and \! are used as operators.
11.2. Handling Data Dependencies Between Tuples

Note that the specification of dependencies in this manner allows the compiler to find a suitable execution schedule for the computation at hand, without being bound to redundant constraints. This is contrary to existing approaches, where the dependencies are encoded in a particular nesting and ordering of loops. In that case, a compiler may modify the loop nesting and ordering, as long as any dependencies in the loop are not broken. So, the compiler has to deduce the actual dependencies from an encoding in the loop structure and may find redundant dependencies that are an artifact of encoding the actual dependency in this structure.

11.2.3 Tuple Marking

For a given tuple, the \textit{ready} construct specifies a condition for tuples that must have been processed already. This implies that the possibility must exist to make a distinction between tuples that have been processed/visited and tuples that have not been processed/visited.

To be able to make this distinction, we introduce the possibility of “marking” tuples in a tuple pool. The following operations are defined:

1. \texttt{reset(tuple\_pool)}. Resets all marks in the given tuple pool.
2. \texttt{mark(tuple)}. Mark the specified tuple.
3. \texttt{unmark(tuple)}. Unmark the specified tuple.
4. \texttt{marked?}(tuple). Returns whether the specified tuple is marked.
5. \texttt{unmarked?}(tuple\_pool). Returns whether the tuple pool contains any un-marked tuples.

As we will see in the next subsection, with these operators it is possible to come to a formalization of how \texttt{forelem} loops with a \textit{ready} clause are executed.

11.2.4 Specification of the ready clause

So far, a number of requirements for the execution of \texttt{forelem} loops with a \textit{ready} clause have been put forward. An index set containing a \textit{ready} clause is evaluated before the \texttt{forelem} loop is entered. Through this requirement, it is guaranteed that no new tuples can become ready during execution of the loop. A second requirement is that only subscripts into a tuple pool $T$ are considered that have not yet been visited and are thus not marked.

Similar to the ability to move the conditions tested in index sets to the inner loop, it is possible to move the testing of the \textit{ready} inside the loop. This must be done while taking the requirements for the execution of these loops into consideration. By making use of the tuple marking operations, these requirements can be met and the result is a formal specification of how \texttt{forelem} loops with a \textit{ready} clause are executed:

\begin{verbatim}
visited = ∅;

forelem (q; q ∈ pT)
  if (!marked?(T[q]) && ready(r)[∇(T[r]) = T[q]]) {

\end{verbatim}
SEQ;
visited = visited ∪ q;
}
for (q'; q' ∈ visited)
mark(T[q']);

The first loop is a regular \textit{forelem} loop which visits all subscripts of \( p_T \) once and tests whether these are marked and are ready for execution. This is equivalent to a loop which computes which subscripts are to be visited before execution of the actual loop. The actual execution of the statements \( \textit{SEQ} \) has been merged into this loop. Note that the requirement to disallow new tuples to become ready during execution of the loop, resulting in these being added to the index set, is enforced by updating the marks for tuples visited in the loop after this loop has completed execution.

Note that this is only a formal, algebraic, specification of how \textit{forelem} loops with a \textit{ready} clause should be executed. This specification is necessary to be able to define transformations on these loops. In practice, loops are not executed in this manner, rather the code is transformed to a suitable execution model as will be discussed in Section 11.3.

11.2.5 Ensuring All Tuples Are Processed

Commonly, when a \textit{ready} condition is present not all tuples of a tuple pool \( T \) are to be processed in a single execution of the loop. Rather, a execution of the loop will ready subsequent tuples, that are to be visited in subsequent executions of the same \textit{forelem} loop. So, to process all tuples in a tuple pool \( T \) (provided a ready condition is specified such that execution of all tuples can indeed be reached), a \textit{while} loop is to be added. The tuple marking as described above ensures that tuples are only visited once, even when the \textit{forelem} loop is executed multiple times:

\begin{verbatim}
reset(T);
while (unmarked?(T))
    forelem (q; q ∈ pT.ready(r)[∇(T[r]) = T[q]])
    SEQ;
\end{verbatim}

As a shorthand for this pattern, a \textit{whilelem} notation is introduced:

\begin{verbatim}
whilelem (q; q ∈ pT.ready(r)[∇(T[r]) = T[q]])
    SEQ;
\end{verbatim}

so, when iteration of the index set is completed, the index set is re-instantiated as long as \( T \) contains unmarked nodes.

In fact, the \textit{whilelem} construct can be viewed as an synchronized execution of all the tuples, where first all tuples are to be processed which can at first be processed, then all tuples will be processed which were enabled by the previous tuples, etc. One could also imagine a more dynamic implementation of \textit{whilelem} which allows the execution of a tuple at any stage. More formally, let us call this \textit{whilever} (whatever tuple can be processed is processed), and
whileever (q; q ∈ pT.ready(r)[∇(T[r]) = T[q]])
  SEQ;

implies that at each iteration mark[T[q]] will be executed as well as a “new” evaluation of the ready clause. In this case, the number of possible execution orders is greatly enhanced. However, at the basis of the forelem loop concept is the assumption that the index set pT can be precomputed, thereby still allowing the index set to be precomputed whilst still allowing a random order of the tuples to be executed. Therefore, this whileever construct is not further elaborated on in this chapter. Also, note that whenever the number of tuples ready at each “stage” equals one, then whileever is equal to whilelem.

11.3 Execution Models

The forelem loop with ready clause provides an algebraic means to specify the computations that must be carried out on the tuples as well as the dependencies between these tuples. To come to an executable code, this abstract specification must be mapped onto an execution model. In this section, this mapping process is described together with two execution models: affine embedding and static scheduling.

Consider tuples (field1, field2, ..., fieldn) with field1 ∈ F1, field2 ∈ F2, ..., fieldn ∈ Fn. The full tuple space TS that is set up by these tuples is defined as F1 × F2 × ... × Fn. Let a tuple pool T be a subset of TS. Now, consider the following forelem loop:

forelem (q; q ∈ pT.ready(r)[T[r].y == T[q].z])
  SEQ;

The expression in the ready clause gives rise to projections Δin and ∇out that are defined as follows:

∇ : F1 × F2 × ... × Fn → F1' × F2' × ... × Fm

with m ≤ n. The ready expression in the loop is then replaced as follows:

forelem (q; q ∈ pT.ready(r)[∇out(T[r]) = Δin(T[q])])
  SEQ;

in other words, all tuples T[r] that project onto the same set of tuples as Δin(T[q]) must have been visited before T[q] can be visited. In case Δin is invertible, then one obtains Δin⁻¹(∇out(T[r])) = T[q] or in short ∇(T[r]) = T[q]. This forelem loop is still an algebraic specification of the computation to be performed and can be executed through dynamic execution. In dynamic execution, the index set with ready clause is evaluated dynamically at runtime. In the two subsequent sub-sections, two execution models are now defined that reformat the forelem loop to a loop without ready clause. This reformatted loop can be subjected to further code transformations and is used as a starting point to generate efficient executable code. We refer to this process as “execution scheduling”.

11.3.1 Affine Embedding

Whenever tuples contain integer fields, these tuples can be iterated by enumerating all possible values of these integer fields in all possible combinations. In affine embedding, an enumeration order of these integer values is determined that satisfies any ready condition, or tuple dependency, that is defined.

Given tuples \((x, y, z)\) in a tuple pool \(T\), where \(y\) and \(z\) are integer fields. The notation \(T.y\) denotes the set of all values \(t.y\) of all tuples \(t\) in \(T\). So, \(T.y\) and \(T.z\) contain all values that occur for these fields in all tuples in the tuple pool. Suppose these fields have values within the interval \([0, N)\), then these fields have integer ranges \(T.y \subseteq I_y = [0, N)\) and \(T.z \subseteq I_z = [0, N)\).

These integer ranges must be transformed such that the tuples are visited in an order that does not violate the tuple dependencies. To accomplish this, a unimodular matrix \(U\) (see for instance [13, 99] for a treatment of the use of unimodular matrices for performing loop transformations) is defined such that:

\[
\forall t, r \in T : \nabla_{\text{out}}(r) = \Delta_{\text{in}}(t) \Rightarrow U(r) <^l U(t)
\]

where \(<^l\) denotes lexicographical ordering. From \(U\) functions \(f_y: I_y \rightarrow I_y, f_z: I_z \rightarrow I_z\) follow such that:

\[
\forall t, r \in T : \nabla_{\text{out}}(r) = \Delta_{\text{in}}(t) \Rightarrow (f_y(r.y), f_z(r.z)) <^l (f_y(t.y), f_z(t.z))
\]

Using these functions, an affine embedding can be written as follows:

\[
\text{for } (i; i \in f_y(T.y)) \\
\quad \text{for } (j; j \in f_z(T.z)) \\
\quad \text{SEQ;}
\]

As an example, consider a simple triangular solve loop:

\[
\text{whilelem } (q; q \in pT.ready(r)[T[r].y == T[q].z]) \\
\]

on tuples \((x, y, z)\) containing values \((A[j][i], j, i)\), as will be described in Section 11.5.1. Consider

\[
U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}
\]

which implies a loop interchange. Then it can be found that \(f_y\) and \(f_z\) are the identity function. To show that the ready condition is satisfied, consider tuples \((y, z)\) (just the integer fields) at time \(t\): \((y, z, t)\). For all tuples holds that: (1) \(y > z\), the tuple pool only contains the elements of the lower triangle; (2) for two tuples \(a, b\) if \(a.t < b.t\) then \(a.y \leq b.y\), the \(y\) values denoting the row number \(j\) in \(A[j][i]\) are processed in order. The ready function is defined such that \(\nabla_{\text{out}}((j, i)) = (j)\) and \(\Delta_{\text{in}}((j, i)) = (i)\).

Now, for all tuples \((k, m, t)\) holds that

\[
\{(n, p, s) : s > t \land \nabla_{\text{out}}((n, p)) = \Delta_{\text{in}}((k, m))\}
\]
is an empty set. Consider this is not the case, then \( \exists (n, p, s) : s > t \land \nabla_{out}((n, p)) = \Delta_{in}((k, m)) \). From the definitions of \( \nabla_{out} \) and \( \Delta_{in} \) we know that \( n = m \). (1) gives \( p \geq m \). From (2) we know that \( n > p \). So, \( n > p \geq m \). Contradiction.

As a result, we can write an affine embedding of the triangular solve loop as follows:

```plaintext
for (j; j \in f_y(T.y))
  for (i; i \in f_z(T.z))
    forelem (q; q \in pT.(y,z)[(j,i)])
```

Note that this is an intermediate representation of the loop nest. As will be further described in Section 11.6.1, many different variants of the triangular solve code can be generated from this loop nest.

### 11.3.2 Static Execution

Through a symbolic execution of a `forelem` loop with `ready` clause, it is possible to derive a static execution schedule in which the tuples can be visited without violating the `ready` clauses. This schedule can be stored and later be used to perform the actual computation, without disturbances caused by re-computation of the `ready` clause.

In this subsection, we will consider the following loop nest:

```plaintext
reset(T);
while (unmarked?(T)) {
  forelem (q; q \in pT.ready(r)[\nabla_{out}(T[r]) = \Delta_{in}(T[q])])
    SEQ;
}
```

To be able to derive a static execution schedule from this loop, all values used in the ready clause may not be written to by `SEQ`. This enables a symbolic execution of the `forelem` loop. Alternatively, a copy of the tuple pool can be used. Any other field may be changed by the statements in `SEQ` without loss of generality.

The derivation is carried out by determining which subscripts \( q \) of the specified index set can be processed at the same time. In other words, the ready clause is satisfied for these tuples. Groups of tuples are formed that can be executed at the same time, which are called “levels”. Tuples are tagged with the level they belong to, by adding a field name `level` to the tuple. This leads to the following loop, which tags all tuples with the correct level:

```plaintext
reset(T);
1 = 0;
while (unmarked?(T)) {
  forelem (q; q \in pT.ready(r)[\nabla_{out}(T[r]) = \Delta_{in}(T[q])])
    T[q].level = 1;
    1 = 1 + 1;
}
```

Note that the `forelem` loops within the `while` loop make use of the two important properties that have been defined in Section 11.2.2:
1. All visited tuples are marked after execution of the loop, such that no tuples that were newly made ready are visited.

2. The loop does not visit tuples that have been visited already.

After the tagging has been carried out, the loop performing the actual computation can be carried out as follows:

```plaintext
for (l' = 0; l' < l; l'++)
  forelem (q; q ∈ pT.level[l'])
  SEQ;
```

Due to the absence of a ready clause in this loop nest, no dynamic evaluation of the ready clause is necessary at run-time and the loop can be executed according to a static execution schedule.

### 11.4 Transformations For Ready Loops

Several transformations are defined in the forelem framework that target loop structure and arrangement of tuples. In this section, a number of transformations are described that are specific to forelem loops with a ready clause. These transformations can be divided into transformations that are applied before execution scheduling and transformations that are applied after execution scheduling.

#### 11.4.1 Before Execution Scheduling

An index set with a ready condition contains subscripts to tuples in a tuple pool which have not yet been processed and are ready for execution. Due to the nature of the forelem loop, the tuples referenced by these subscripts may be executed in any order. In certain cases, it is useful to group sets of tuples with certain similar properties for execution before scheduling takes place. This influences the final execution schedule of this loop. In this section, the Projection transformation is proposed to accomplish this. By applying the Projection transformation on a given loop on different properties, a transformation space is set up yielding different variants of the same loop with different performance characteristics. The Projection transformation is defined as follows. Consider:

```plaintext
forelem (q; q ∈ pT.ready(r)[\nabla_{out}(T[r]) = \Delta_{in}(T[q])])
  SEQ;
```

where in the ready expression the following operands are used: \(T[q].field1, T[q].field2, \ldots, T[q].fieldn\). For this particular loop, a grouped execution is defined as follows:

```plaintext
forelem (q; q ∈ pT.ready(r)[\nabla_{out}(T[r]) = \Delta_{in}(T[q])])
  forelem (p; p ∈ pT.(field1, field2, \ldots, fieldn)
         [T[q].field1, T[q].field2, \ldots, T[q].fieldn])
  SEQ;
```
To understand the validity of this transformation, it is important to note the following: if for a given tuple \( t \) in \( T \) all dependent tuples are ready based on \( \text{field}_1, \text{field}_2, \ldots, \text{field}_n \), then for all other \( t' \) in \( T \) with equal values for \( \text{field}_1, \text{field}_2, \ldots, \text{field}_n \) the ready sets are satisfied. Therefore, it is valid to process all of these tuples in the inner loop. In this case, execution is grouped in groups of tuples having equal values for \( \text{field}_1, \text{field}_2, \ldots, \text{field}_n \).

Next to conditioning the values of \( \text{field}_1, \text{field}_2, \ldots, \text{field}_n \) in the inner loop, conditions may be added to test on other properties of the tuples to define a further projection. Important is that the fields that are present in the \text{ready} expression may never be omitted from condition testing in the inner loop.

Because we have defined \text{forelem} loops with a \text{ready} condition to only mark tuples visited in the loop body as visited after the loop has finished execution, the transformation needs to make a further modification. To avoid processing tuples that are visited in the inner loop for a second time by the other loop, these tuples must be marked as visited immediately, or at least within the loop body of the outer loop.

With projection, tuples to be processed are grouped based on certain properties. This is similar to the Orthogonalization transformation [83] that is defined in the \text{forelem} framework. Orthogonalization imposes a specific iteration order on a loop, based on the values of one or more fields of the tuples. For example:

```plaintext
for (k; k ∈ \mathbb{N})
    \text{forelem} (q; q ∈ pT.y[k])
    \text{SEQ;}
```

executes \text{SEQ} for tuples with the same value for field \( y \), with \( k \)'s value in increasing order. In fact, the tuples to be executed are grouped into groups of tuples with equal values for field \( y \).

Although Orthogonalization seems similar to the Projection transformation that has just been proposed, there is a fundamental difference. In the orthogonalized loop, tuples are always ready for execution: there is no \text{ready} condition. To see why orthogonalization does usually not have the desired effect, consider the following:

```plaintext
for (k; k ∈ \mathbb{N})
    \text{forelem} (q; q ∈ pT.y[k].\text{ready}(r)[\nabla_{out}(T[r])] = \Delta_{in}(T[q]))
    \text{SEQ;}
```

Only tuples \( t ∈ T: t.y = k \) can be visited for which all dependent tuples are ready. This may not be the case for all tuples \( t \). So, after a full iteration of the outer loop, not all tuples may have been processed. As a consequence, this loop nest must be surrounded by a \text{while} loop that repeats execution of this loop nest until all tuples have been visited. Naturally, as a result, the tuples are no longer processed in exact increasing order of \( k \), defeating the goal of orthogonalization.

### 11.4.2 After Execution Scheduling

Loop nests that are the result of mapping a \text{forelem} loop with \text{ready} clause onto an execution model, can be subjected to further transformations. A number of
these transformations will be described for loop nests that are the result of static scheduling. An example is:

```plaintext
for (i = 0; i < L; i++)
  forelem (q; q ∈ pT.level[i])
  SEQ;
```

where L is the number of levels that have been identified during the computation of the schedule. Within this loop nest, execution of tuples contained in a certain level can again be grouped based on certain properties. To accomplish this, the Orthogonalization transformation can be used to group the tuples this example code on field y:

```plaintext
for (i = 0; i < L; i++)
  forall (j; j ∈ Nj)
    forelem (q; q ∈ pT.(y,level)[(j, i)])
    SEQ;
```

Since all tuples in a level can be processed in parallel, the loop with iteration j is made a `forall` loop to indicate all identified groups can be processed in parallel.

An important difference with the Projection transformation described in the previous section is that since Projection is performed before execution scheduling, the grouped tuples are placed in separate levels. When Orthogonalization is applied after scheduling, groups are identified within levels. Secondly, orthogonalization after scheduling is not bound to any restrictions on the conditions like is the case with Projection.

Another transformation that can be performed after scheduling is to ensure sets of tuples with a certain property are processed at the same time. For example, all tuples with equal values for the y field should be processed at the same time. However, due to the imposed `ready` clause these tuples may be spread among different levels. In order to still execute all tuples with equal y field at the same time, the highest level in which such a tuple is placed should be found. All tuples with equal y can be safely executed in that particular level.

This transformation can be implemented by analyzing and modifying the level tags on the tuples. A pseudocode to perform this modification is as follows:

```plaintext
forelem (j; j ∈ Nj) {
  max_level = 0;
  forelem (k; k ∈ pT.y[j])
    max_level = MAX(max_level, T[j].level)
  forelem (k; k ∈ pT.y[j])
    T[j].level = max_level;
}
```

## 11.5 Characteristics of the Extended Forelem Framework

The extended `forelem` framework that is proposed in this chapter has several compelling characteristics. These include applicability, as the techniques that have been
described can be applied to a wide variety of existing codes, universality as all data structures can be represented in terms of tuples, transferability allowing sparse codes to be generated from a dense specification of the computation, versatility leading to many different implementations of the same computation and optimality as an implementation optimal for a particular architecture can always be found in the different generated implementations. In this section, these characteristics are discussed in turn.

11.5.1 Applicability

The extended forelem framework that is proposed in this chapter is applicable to a large variety of existing codes. To demonstrate this, in this section we will show how a C code is translated to a tuple program. The C code that we will consider in this section is a simple triangular solver of lower triangular unit matrices:

```
for (int i = 0; i < N; i++)
    for (int j = i + 1; j < N; j++)
```

B is a vector consisting of N elements and A is an N × N matrix. First, the translation of data accessed by this loop into a tuple space is addressed. This translation is performed by forming tuples consisting of all array elements that are referenced by the statements in the loop. When a loop contains multiple statements, one tuple is formed containing all array elements accessed by all statements in the loop body. In fact, a bijection is set up between the iteration space of the original loop and the tuples.

In the case of the triangular solve example, the loop body only contains a single statement, resulting in the tuple:

```
(A[j][i], j, i)
```

The fields of the tuple are labeled with x, y and z. The matrix and vector are stored as arrays of doubles. To store all values of A, \( N^2 \) doubles are needed. This implies that \( N^2 \) tuples are stored in the tuple space, so \( N^2 \) doubles are needed to store all values for B[j] and another \( N^2 \) for B[i]. The total necessary storage capacity is \( 3N^2 \) doubles.

The translation process tries to decrease the amount of storage that is necessary. Because two fields of the tuple are values from an array B, it replaces the values of B at this position with subscripts into B:

```
(A[j][i], j, i)
```

Now, \( N^2 \) doubles are needed to store the values A, \( 2N^2 \) integers to store the values for fields y and z and \( N \) doubles to store B. A total of \( (N^2 + N) \) doubles and \( 2N^2 \) integers. Considering doubles are typically stored in 8 bytes and integers in 4 bytes, the required storage capacity has almost been cut in half.

When the values for A in the tuples are replaced with subscripts, no storage saving space is achieved:

```
((j, i), j, i)
```
To store these tuples, 4N^2 integers are needed, N^2 doubles and N doubles. Since this does not present a saving of storage space over the previous tuple, the process selected the previous tuple as “loop tuple”.

The loop bounds of the original loop indicate which of the loop tuples need to be stored in the tuple loop. In this case, the loop bounds are i ∈ [0, N) and j ∈ [i + 1, N). So, only tuples with y > z are to be stored in the tuple pool.

The next step in the translation process is to translate the loops to forelem loops, that operate on the tuple space defined by the selected loop tuple. This translation must preserve existing dependencies by translating these to corresponding ready clauses. The triangular solve loop that is being considered has a true dependency on array B: B[j] δ^t B[i]. This can be phrased as in order to execute the statement that reads from B[i], so z = i, all tuples that write to B[i], so y = i must have finished. So, the ready function is defined by the expression T[r].y == T[q].z. As a result, triangular solve can be expressed using a whilelem loop as follows:

```
forelem (q; q ∈ pT.ready(r)[T[r].y == T[q].z])
```

11.5.2 Universality

All possible data structures can be represented in terms of tuples. This follows from the fact that computer memory in which these data structures are located can be represented in terms of tuples: e.g. (addr, value) tuples. In this subsection we demonstrate the universality of our approach by expressing linked list and tree travels as whilelem loops. Consider linked list links defined as:

```c
struct List {
    void *data;
    struct List *next;
};
```

This linked list can be iterated in C with a regular while loop:

```c
struct List *l = start;
while (l != NULL) {
    operate_on(l->data);
    l = l->next;
}
```

The body of this while loop accesses the values l, l->next and l->data, giving rise to tuples (x, y, z), containing the values (l, l->data, l->next). A whilelem loop can be written, which performs exactly the same iteration:

```
whilelem (l; l ∈ pT.ready(r)[T[r].z == T[l].x])
    operate_on(T[l].y);
```

The ready expression states that a tuple T[l].x can be visited, once all tuples T[r].z (z is the next field) have been visited. Note that this automatically leads to the first link of the linked list to be visited. No tuple exists with a next field
equal to the address of the first tuple in the list, therefore the empty set of tuples that precedes the first tuple in the list have all been visited.

Note that when a static execution schedule is generated for this \texttt{whilelem} loop and the data storage is materialized, the linked list is automatically linearized into an array. So, this process can be performed automatically with the generic tools provided within this framework, instead of with specific frameworks that have been developed for this in the past [94].

As another example, breadth-first traversal of a binary tree can be elegantly expressed as a \texttt{whilelem} loop. Let \( T \) be a tuple pool with tuples \((w, x, y, z)\) corresponding to values \((n, n->\text{data}, n->\text{left}, n->\text{right})\) of a simple binary tree data structure. The following loop then visits the nodes of the tree in a breadth-first order:

\[
\text{\texttt{whilelem}} \ (n; \ n \in pT.\text{ready}(r)[T[r].y == T[n].w \ || \ T[r].z == T[n].w])
\]

\[
\text{operate}_\text{on}(T[n].x);
\]

The ready condition specifies that a node \( T[n] \), which is either the left or right child of a parent node, can be visited once the parent node has been visited. Because nodes are only marked as visited after execution of the \texttt{forelem} loop (which is embedded in the \texttt{whilelem} loop as discussed in Section 11.2.5), no new nodes become ready during the execution of the loop. As a consequence, there is the guarantee that nodes are indeed visited in a level-by-level order of the tree.

\subsection*{11.5.3 Transferability}

Sparse matrix codes are often developed separately from dense matrix codes. This is because sparse matrices are stored in custom, pre-defined, data structures, contrary to dense matrix codes that store the data as a regular multi-dimensional array. It is this large difference in data structures that leads to the existence of separate code bases for dense and sparse matrix algebra.

Programs that are expressed in the extended \texttt{forelem} framework proposed in this chapter, can operate on both dense and sparse data storage, since both of these can be stored into a set of tuples that can be operated on. As a consequence, when a dense linear algebra computation is translated to a \texttt{forelem} loop, a sparse version of this routine can be derived automatically. To see this, consider tuples containing \((A[j][i], j, i)\) from Section 11.5.1 and the loop body of the triangular solve code:

\[
\]

As has been observed in the literature [15], this statement is a no-op in case \( T[q].x == 0 \) and this implies that all tuples with \( T[q].x == 0 \) can be omitted without affecting the end result of the computation. This corresponds with removing all matrix elements \( A[j][i] == 0 \). The result is an expression of the original computation that operates on sparse storage.

As another example, consider sparse matrix times vector multiplication:
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        C[i] = C[i] + A[i][j] * B[j];

this yields a tuple space \( T \) with tuples \((i, A[i][j], j)\). Since this loop does not exhibit true dependencies, it can be expressed as a \textit{forelem} loop without having to make use of a \textit{ready} condition:

\[
\text{forelem} (m; m \in pT) \\
C[T[m].x] = C[T[m].x] + T[m].y * B[T[m].z];
\]

Also in this case, tuples with \( T[m].y == 0 \) are no-op statements, and such tuples can be deleted from the tuple space without affecting the final result. Again, a transfer is made to a sparse data storage. By applying different transformations that are supplied with the \textit{forelem} framework to this loop nest, different forms of this sparse data storage are automatically generated. For example, through Orthogonalization and Materialization the loop nest can be put into a form such that the computation is performed on a row-by-row basis and the tuples are explicitly organized in a row-by-row order in the array \( PT \):

\[
\text{forall} (i = 0; i < N; i++)
    \text{forelem} (m; m \in N^*) \\
    C[i] = C[i] + PT[i][m].y * B[PT[i][m].z];
\]

where \( N^* = [0, |PT[i]|) \). Further transformations may, for example, lead to a loop from which Compressed Row Storage is derived:

\[
\text{forall} (i = 0; i < N; i++)
    \text{forall} (m = PT.ptr(i); m < PT.ptr(i+1); m++) \\
    C[i] = C[i] + PT[m].y * B[PT[m].z];
\]

Other sparse data formats that can be generated automatically include, for example, Jagged Diagonal Storage, a sparse data storage format that up till now could not be derived automatically.

### 11.5.4 Versatility

The extended \textit{forelem} framework that is proposed in this chapter, is backed by a versatile transformation framework that contains transformations that restructure the computation (projection, scheduling, orthogonalization), data storage (orthogonalization and concretization) as well as transformations that address efficiency of the final code (loop blocking, loop unrolling). Using these transformations, a search space is set up that contains many different variants of the same loop with different performance characteristics.

Chapter 10 showed that this search space contains at least 130 principal forms of the sparse matrix times \( k \) vector multiplication and 76 principal forms of the triangular solve computation. A principal form is an instantiation of the computation with a different loop structure and different data storage derived from this loop structure. In Figure 11.1 we plot again the transformation tree leading to the
Figure 11.1: Transformation tree of the triangular solve computation. Through the application of transformations on the starting point 1, 76 different principal forms are generated.
76 different principal forms of triangular solve, see also page 178. The form in the center, labeled with 1, is the starting point. The edges represent transformations that result in many different principal forms. The nodes that are prefixed with “tmp” are intermediate stages of the transformation process for which no executable is generated and these do not count as principal form.

11.5.5 Optimality

For all computations that are expressed in the extended forelem framework, either directly, or indirectly through a mapping from an original program code written in for instance C, a very large search space of possible implementations is set up. As we have described, the implementations in this search space have different orders in which the computation is carried out, different execution schedules leading to different parallelizations of the computation, different data structures and are subjected to different final parametrized optimizations that tune the executable code.

In this large search space, implementations of the computation can be found that are optimal for a given target architecture and make best use of the architecture’s resources. The optimal implementation will vary for architecture, class of input data, etc. Note that because this search space is significantly larger than the search space that is exploited by contemporary compilers, optimal implementations can be found that are not found by contemporary compilers. The results of the work discussed in the previous subsection show that for the sparse matrix times $k$ vector(s) multiplication always an implementation is found that is faster than the implementations supplied by several hand-optimized sparse algebra libraries, speedups are observed as large as 46%. Automatically generated implementations for triangular solve achieve in the majority of cases a speedup, up to 30% to 56% and in some cases no speedup is reported but the performance of the generated implementation is at least on par with the implementations provided by the sparse algebra libraries. For a further discussion of optimality, see also the next section.

11.6 Case Study: Triangular Solve

In this section, we demonstrate that from an ordinary triangular solve code in C, within the extended forelem framework parallelized implementations are produced, that up till now could only be derived by hand. The performance of these parallelized implementations is compared with that of a hand-optimized triangular solver.

11.6.1 Transformation Process to Produce Parallelized Implementations

The starting point is a triangular solve code written in C. This is a lower triangular solve code for unit matrices:
for (int i = 0; i < N; i++)
    for (int j = i + 1; j < N; j++)

B is a vector consisting of N elements and A is an $N \times N$ matrix. As has been shown in Section 11.5.1, the optimization process will select $(A[j][i], j, i)$ as an appropriate loop tuple for this loop. The fields of the tuples are named $x$, $y$, and $z$. Derived from the loop bounds, a tuple space is created that contains tuples for $j > i$, so tuples with fields such that $y > z$.

whilelem (q; q ∈ pT.ready(r)[T[r].y == T[q].z])

Now that the computation is expressed within the extended forelem framework, transformations are applied and a mapping is done onto one of the execution models. From a different application of the transformations and different selections of execution models, many different implementations are generated. For example, in Section 11.3.1 a mapping of the triangular solve loop onto the affine embedding execution model was derived:

for (j; j ∈ $f_y(T.y)$)
    for (i; i ∈ $f_z(T.z)$)
        forelem (q; q ∈ pT.(y,z)[$(j,i)$])

From the facts that $f_y$ and $f_y$ are identity functions and that only tuples with $y > z$ are present in the tuple pool, can be deduced that the inner for loop with iteration variable $i$ can be executed in parallel. This loop is eliminated to give the forelem loop another degree of freedom:

for (j; j ∈ $f_y(T.y)$)
    forelem (q; q ∈ pT.y[j])

This way, the 76 different principal forms as described in Section 11.5.4 can be generated from different affine embeddings. Additionally, further transformations will lead to a loop that operates on an efficient sparse data storage, such as described in Section 11.5.3.

Also implementations will be produced by mapping the computation onto the static execution model. For example:

l = 0;
reset (T);
while (unmarked?(T)) {
    forelem (q; q ∈ pT.ready(r)[T[r].y == T[q].z])
        T[q].level = l;
    l = l + 1;
}
for \((l' = 0; l' < l; l'++)\)

\[
\textbf{forelem} (q; \ q \in pT.\text{level}[l'])
\]

\[
B[T[q].y] = B[T[q].y] - T[q].x \times B[T[q].z];
\]

In this implementation, the ready clauses are first processed to find out which tuples can be processed at the same time. The loop nest that follows no longer needs the ready clause in order to perform the computation correctly, but rather uses the resulting “level” information. A static schedule is beneficial when the second loop nest, that actually performs the computation, is carried out multiple times for the same tuples, but a different array \(B\).

This implementation is similar to implementations of triangular solvers described in the literature, that consist out of an processing step, or analysis phase, to analyze the structure of the sparse matrix and a solve phase that uses the result of this analysis to perform a highly parallel triangular solve computation [8, 72]. Contrary to the implementations described in the literature, which are derived by hand by an expert programmer, the implementation described in this section is produced automatically from an ordinary dense version of triangular solve using the extended \textit{forelem} framework described in this chapter.

Further transformations can be performed. For example, Materialization, as described in Chapter 9, can be used to materialize the tuples into a two-dimensional array, where the tuples are stored in a level-by-level order. This results in the following computation loop nest:

\[
\textbf{for} \ (l' = 0; l' < l; l'++)
\]

\[
\textbf{forelem} (q; \ q \in N^*)
\]

\[
B[PT[l'][q].y] = B[PT[l'][q].y] - PT[l'][q].x \times B[PT[l'][q].z];
\]

where \(N^* = [0, |PT[l']|]\).

11.6.2 Experimental Evaluation

We have conducted a preliminary experimental evaluation with codes that have been generated using the extended \textit{forelem} framework. Various implementations have been generated for the affine embedding execution model and one implementation for the static scheduling execution model. CUDA codes have been generated that were compiled with the CUDA 5.0 toolkit. The resulting executables have been timed on a machine containing an Intel Xeon E5-2650 CPU at 2.00GHz, hosting an NVidia Telsa K20m GPU with 4799MB of RAM. The experiments were run for 16 matrices, obtained from the University of Florida Matrix Collection [28]. These matrices were reformatted to only store the lower triangle of the matrix and ones were placed on the diagonal, the sparsity patterns were preserved.

In Table 11.1, the results are shown for 8 different implementations based on the affine embedding execution model. All reported execution times are in milliseconds. These 8 implementations differ both in loop structure, as well as in data storage format. The data storage formats are also the result of transformations carried out by the compiler, that have modified the tuples into a form suitable for the computation to be carried out. As can be seen in the table, all implementations
exhibit different performance characteristics. Some implementations are clearly faster than other implementations for all surveyed matrices. In this case, implementation VIII always presents the fastest execution time.

With the extended forelem framework, one implementation was generated that is based on the static execution model. This implementation is especially suited when the same matrix is processed multiple times, with different values for the vector B. We have conducted a preliminary comparison of the performance of the automatically generated implementation to the performance of the parallelized triangular solve provided with the CUDA 5.0 toolkit in the CUSPARSE library [72].

The implementation supplied by the CUSPARSE library consists of both an analysis and solve phase that run on the GPU. The automatically generated implementation performs the analysis phase on the CPU and runs the computation loop nest on the GPU. For the or2010 matrix, the automatically generated analysis phase, running on the CPU, is 20 times slower than the CUSPARSE analysis running on the GPU. Even though the execution times for the solve phase are the ones of prime interest since the analysis phase only has to be performed once per matrix, we intend to address this deficiency in future work.

The execution time of the solve phase of CUSPARSE and the automatically generated implementation using the extended forelem framework are shown for the 16 different matrices in Table 11.2. For a number of matrices, e.g. stomach, shipsec1 and G2_circuit, the automatically generated implementation is faster than the implementation provided by CUSPARSE. This exemplifies the strength of our approach, due to the use of versatile transformations, implementations can be automatically found that beat hand-optimized implementations. In most other cases, the performance of the automatically generated implementation is with a difference of a factor of 2 to 3 competitive with the hand-optimized version. Notable
Table 11.2: Execution times of the solve phase of the triangular solve algorithm of both CUSPARSE and the automatically generated implementation using the extended *forelem* framework. Times reported are in milliseconds.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>CUSPARSE</th>
<th>Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erdos971</td>
<td>0.136</td>
<td>0.193</td>
</tr>
<tr>
<td>mcf e</td>
<td>1.66</td>
<td>3.33</td>
</tr>
<tr>
<td>blckhole</td>
<td>0.456</td>
<td>1.19</td>
</tr>
<tr>
<td>OPF.10000</td>
<td>0.443</td>
<td>0.493</td>
</tr>
<tr>
<td>lhr71</td>
<td>0.881</td>
<td>8.03</td>
</tr>
<tr>
<td>stomach</td>
<td>22.1</td>
<td>16.5</td>
</tr>
<tr>
<td>3dtube</td>
<td>49.6</td>
<td>81.7</td>
</tr>
<tr>
<td>orsreg.1</td>
<td>0.279</td>
<td>0.461</td>
</tr>
<tr>
<td>shipsec1</td>
<td>24.3</td>
<td>15.6</td>
</tr>
<tr>
<td>shipsec5</td>
<td>32.0</td>
<td>20.4</td>
</tr>
<tr>
<td>pdb1HYS</td>
<td>100.0</td>
<td>128.0</td>
</tr>
<tr>
<td>or2010</td>
<td>0.86</td>
<td>0.937</td>
</tr>
<tr>
<td>G2_circuit</td>
<td>8.58</td>
<td>7.79</td>
</tr>
<tr>
<td>144</td>
<td>6.47</td>
<td>6.72</td>
</tr>
<tr>
<td>cop20k_A</td>
<td>2.93</td>
<td>2.83</td>
</tr>
<tr>
<td>consph</td>
<td>7.1</td>
<td>5.46</td>
</tr>
</tbody>
</table>

The exception is the *lhr71* matrix, for which a approximately factor of 10 slowdown is observed.

11.7 Conclusions

In this chapter, we have described an extension to the *forelem* framework for the expression of data dependencies and the optimization of irregular parallel computations. By expressing computations using the ready clause, more information is exposed to the compiler about the order in which data is visited. Furthermore, the compiler is enabled to reorganize the processed data in a more optimal form, next to the ability to restructure the computation.

Another major benefit of this programming model is that from dependency information that is expressed as dependencies between tuples in a tuple pool, it is trivial to deduce which operations on tuples can be executed at the same time. As a consequence, this framework is especially suited for the automatic parallelization of irregular codes. We have described that through the application of transformations that are implemented in the underlying optimization framework, many different implementations can be generated with different performance characteristics and that an effective exploitation of this search space leads to automatically generated implementations that are competitive with hand-optimized codes.

We have shown that from an ordinary (dense) triangular solve code written in C, a highly parallelized implementation can be generated automatically, that computes on sparse data storage. Preliminary experiments that have been conducted, demonstrate that the performance of this automatically generated implementation is competitive to the performance of hand-optimized codes.