Leiden University

Sparse code optimization

Automatic transformation of linked list pointer structures

Sven Groot
2006-10-31

Contents

1 Introduction

One of the major problems in the area of restructuring compilers is that of optimizing sparse code. Preferably we wish to be able to automatically select a sparse structure that is best suited to the input matrices. The MT1 compiler, developed at Leiden University [1; 2], is a compiler that can take dense code and automatically generate sparse code suited to the non-zero structure of the sparse matrices used in the code.

In order to be able to transform it, the code used as input to such a compiler may usually not use indirections, and such is the case with MT1. The compiler needs to do deep data dependency and zero-structure analysis of the data structures used in the code, and if these structures are accessed irregularly it is often not possible to draw any meaningful conclusions from this analysis, thus prohibiting the transformation of the code. This presents a problem when the code one wishes to optimize is already using a sparse representation, since this code will use indirection.

While there exist approaches to automatically restructure irregular code into regular (dense) code [3], these are meant for languages such as FORTRAN, where indirect addressing is done by using index arrays; for example a certain array might be accessed directly using A(n) or indirectly using A(B(n)). Here B serves as an index into A, and the access pattern of A is governed by B. The irregular code transformation presented in [3] is applicable only to these types of irregular code.

In languages such as C, irregularity is typically created by the use of pointers. For this and other reasons, optimizing C code is incredibly complex. Pointers do not only allow irregular access, but they can point to arbitrary locations, multiple pointers can point to the same location (aliasing), they can be arbitrarily manipulated, and they allow more complicated structures such as a linked list or tree, all of which present considerable challenges. Some of the challenges involved in optimizing C code are outlined in chapter 12 of [4].

One very common pointer structure is the linked list. The linked list provides significant optimization challenges even outside the realm of sparse matrix computation. Linked lists represent a sequence of elements where each element is in a completely unrelated location, making it difficult for the compiler to optimize the memory access patterns and the CPU cache cannot use locality to speed up access to the list. It is also impossible to vectorize computations that involve them. Additionally, the presence of the linked list in a loop, which means the use of pointers, will often prevent simple automatic transformations such as loop interchange which can help to optimize the code.

In this thesis, we have devised a method whereby a linked list, created using pointers and used to represent sparse data, can be automatically transformed into code operating on dense data structures. The linked list will have been replaced by a regularly accessed array, allowing further optimization.

To evaluate the possibilities of further optimization and sparse data structure selection, we will use the sparse compiler MT1 to transform the generated dense code into sparse code optimized for a certain matrix structure. Because MT1 can only operate on FORTRAN code, and not on C code, we automatically transform our C code into FORTRAN. This transformation imposes some requirements on the used C code so it is not generally applicable, but it suffices for the code we need to transform here. The performance of the different versions will then be measured and compared.

The techniques presented here are generic; they can be applied to any C code using linked lists as long as it meets the requirements stated later. Even when the linked list was not used to

represent sparse matrices the transformation can yield some benefits by enabling optimizations such as vectorization on the generated code.

The approach used here is quite unique. For the past few decades, the major focus of pointer optimization research has been in the area of dependence analysis in the presence of pointers, something which will be covered further in Section 3.5.1. Although around a hundred papers and theses have been written on pointer analysis, surprisingly little research exists on the application of these techniques. What little work does exist tends to focus on more conventional optimization techniques such as common sub-expression elimination, loop-invariant elimination, etc. [5], on parallelization [6; 7] or on optimizing memory access and allocation [8]. No research in the area of sparse computations involving pointers or even vectorization of code involving pointers (other than pointers to arrays) could be found.

Instead of focusing on pointer dependence analysis, this thesis focuses on a specific pointer usage pattern and attempts to enable more complex optimizations, for example loop interchange and data structure transformation, such as those done by sparse compilers, an approach that appears to be unprecedented.

2 Example code: sparse matrix multiplication

In order to illustrate the techniques used for linked list transformation, we will use an example of a sparse matrix computation. In this example, the sparse matrix is represented using a linked list. The data structure for a sparse matrix consists of two linked lists of column and row headers. Each header item contains a linked list for the column or row respectively. The type definitions used by this representation are given below.

```
struct Cell
{
    float Value;
    int ColIndex;
    int RowIndex;
    struct Cell *RowNext; // Cell in the next row
    struct Cell *ColNext; // Cell in the next column
};

struct RowHead
{
    int RowIndex;
    struct Cell *Cell;
    struct RowHead *Next;
};

struct ColHead
{
    int ColIndex;
    struct Cell *Cell;
    struct ColHead *Next;
};

struct Matrix
{
    int Dimensions;
    struct ColHead *Col;
    struct RowHead *Row;
};
```

As you can see the row and column linked lists both use the same type to represent the cells. Indeed, every instance of the Cell structure will be part of the linked list for both its row and its column.

To allow sparseness, cells which have a zero value will be omitted from the linked lists. This means the RowIndex and ColIndex can skip values if a Cell has been omitted. Similarly, if a row or column consists entirely of zeros, the header may be omitted as well. This representation can be easily constructed reading the matrix from a file that uses either a dense or sparse notation.

Figure 1 shows an example of how a 3x3 sparse matrix would be represented in this format.

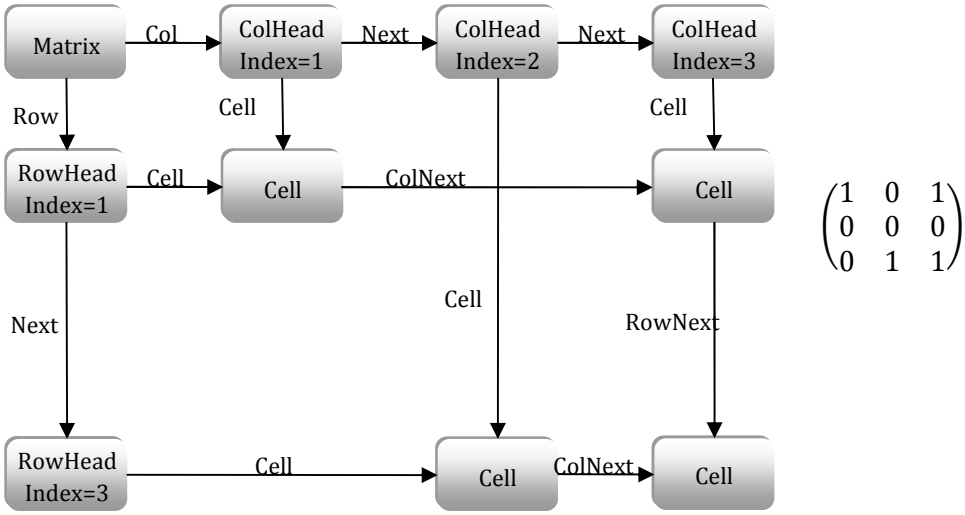

Figure 1 A sparse matrix using linked list representation.

For demonstration purposes, we use the following matrix multiplication algorithm which uses this representation for one of the matrices involved; the others use a traditional multi-dimensional array representation that is fully dense, i.e. all values are stored, even zeroes:

```
int MatrixMultiply(Matrix left, float** right, int rightDimensions,
                   float **result, int resultDimensions)
{
  struct RowHead *leftRow = left.Row;
  struct Cell *leftCell;
  int dimensions = left.Dimensions;
  int col;
  int row;
  int x;

  if( !(left.Dimensions == rightDimensions &&
        left.Dimensions == resultDimensions) )
    return -1;

  for( col = 0; col < dimensions; ++col )
  {
    leftRow = left.Row;

    for( row = 0; row < dimensions; ++row )
    {
      if( leftRow != NULL && leftRow->RowIndex < row )
        leftRow = leftRow->Next;
```

```
      if( leftRow != NULL && leftRow->RowIndex == row )
      {
        leftCell = leftRow->Cell;
        for( x = 0; x < dimensions; ++x )
        {
          if( leftCell != NULL && leftCell->ColIndex < x )
            leftCell = leftCell->ColNext;

          if( leftCell != NULL &&
              leftCell->ColIndex == x &&
              leftCell->RowIndex == row )
          {
            result[row][col] += leftCell->Value * right[x][col];
          }
        }
      }
    }
  }

  return 0;
}
```

There are a few things to note about this algorithm. The result and right matrices both use an "array-of-arrays" representation instead of a typical multi-dimensional array as used in C, which would be strided. This is to allow for dynamic allocation of the arrays. Because these are pointers, they present all the problems listed in [4]. For the purposes of our transformation, we must assume that these pointers point to distinct arrays. If they were to overlap, reads of "right" are dependent on writes of "result", which breaks data dependence analysis. This is one the aliasing problems which is mentioned in section 3.5.1 and it cannot be completely solved here. Transformation directives will be used to indicate which sections of the code do not violate such restrictions and are therefore safe to transform.

Another thing to note is that since the left matrix is walked in row order only, the ColHead linked list is not used. This does not matter to the algorithm, and indeed if it were used (as would be the case if right was sparse as well) the linked list transformation would be executed in exactly the same way.

Finally, you should note that the loop used to walk the linked list is not typical for linked list code. Usually you see linked list loops that use a condition such as `while( node != NULL )`. This matrix multiplication code uses what we call a semi-dense loop; this is elaborated on in Section 3.3. It is indeed possible to write matrix multiplication using the more common type of linked list code, and it would in fact be a great deal faster for matrices with low density. The choice of initial code however does not affect the algorithm; this version however is chosen in particular because it demonstrates more aspects of the transformation. Appendix B shows the alternative version of the algorithm along with results of transforming it.

3 Linked list transformation

In this section, we will cover the method used for linked list transformation. We will provide a set of automatic transformations that a compiler could execute to transform irregular sparse code, which uses linked list pointers, into regular dense code. The transformation will attempt to move all linked list style pointer accesses from the main loop being transformed into separate initialization loops, removing the linked list from the main loop leaving a linearly accessed array in its place in the main loop.

Currently, no real implementation of the system exists. It is however described using steps which can be performed automatically and is also presented in a pseudo code implementation. Although the transformed code examples were done by hand, all could be done automatically. It should be noted that the examples are not fully normalized as described in section 3.5; normalization is necessary for automatic processing, but some of it has been omitted in favor of keeping the examples readable.

3.1 Sublimation vs. Annihilation

The goal of the transformation is, in essence, to regularize the linked list into an array. There are two basic techniques by which we can do this, sublimation and annihilation.

With **sublimation**, we transform a sparse linked list into a dense array by filling in all the omitted values with a suitable fill-in value, e.g. zero or one; which value exactly will depend on the source data (for details on how the value to use is determined, see section 3.4.5).

With **annihilation**, the linked list is still transformed into an array, but leaving it sparse. The only values in the array will be the values that were in the linked list; values that were omitted in the linked list will still be omitted in the array, so no fill-in value is needed. If the code being transformed uses any other data structures apart from the linked list, such as a dense array, it may be necessary to transform those as well so that they match the transformed array.

Let us illustrate this with an example. Assume you have a vector of ten values, which looks like this:[1, 1, 3, 1, 1, 7, 2, 1, 9, 1]. In this case, we are only interested in the values that have a value other than one, so only four values are relevant. These are stored in a sparse linked list, which omits the values that are one; this means the linked list will contain four nodes. Besides the value, each node also contains a member which indicates the original index that value had in the vector. The (zero-based) indices of the relevant values are 2, 5, 6 and 8.

Figure 2 A simple linked list representing a sparse array

Figure 2 shows the structure of this linked list. We now look at a simple reduce algorithm that operates on this linked list:

```
int product = 1;
/***DENSE_INDEX(node, node->Index)***/
/***DENSE_DIMENSION(node, 10)***/
while( node != NULL )
{
  product *= node->Value;
  node = node->Next;
}
```

Besides the simple algorithm, this code snippet also contains two transformation directives, one which tells it how to determine the dense index for a node in the linked list, and one that tells it the original dense size of the vector (note that this does not need to be a constant, any expression will do). Directives influencing the translation are covered in more detail in section 3.6.

When translating the linked list using sublimation, the goal is to create an array that contains ten elements, i.e. all of the original elements in a dense representation. An initialization loop is

generated to copy the linked list contents into such an array, and the main loop is transformed to use the dense array instead of the linked list. The fill-in value we will use is one, since that was the omitted value, and it can be shown that using that value will not cause the semantics of the algorithm to change, since multiplying by one does nothing (since this thesis deals mainly with sparse matrices, the fill-in value will nearly always be zero; this example was designed specifically to show that it does not need to be; the techniques presented can deal with any fill-in value).

```
int product = 1;
int x;
// Initialisation
int *nodeArray = malloc(10 * sizeof(int));
for( x = 0; x < 10; ++x )
{
  if( node != NULL && node->index == x )
  {
    nodeArray[x] = node->Value;
    node = node->Next;
  }
  else
    nodeArray[x] = 1; // Fill-in value
}
// Main loop, transformed
for( x = 0; x < 10; ++x )
{
  product *= nodeArray[x];
}
free(nodeArray);
```

As you can see, new loop bounds have been determined using the dimensions of the dense array which were specified with the transformation directive. Both the initialization loop and the transformed main loop use this loop bound. The initialization loop uses a check against the dense index to see whether it should copy the value from the linked list or use the fill-in value (note that this is not the only way the initialization loop can be created in this case, more on that in section 3.4.6). The transformed main loop loops over the entire dense array, and uses the values from the array instead of the linked list. The linked list is not present in the main loop at all anymore.

After the initialization loop, nodeArray looks exactly like the original vector: [1, 1, 3, 1, 1, 7, 2, 1, 9, 1]

As you can see, it has ten elements, like the original dense data, and contains the used values at their proper indices. All the other values are one, the fill-in value.

Now let us see what this code would look like transformed using annihilation instead. Here the goal is to create an array that contains only the dense values. No fill-in value is needed because nothing is filled in.

```
int product = 1;
int x;
// Initialisation
int *nodeArray = malloc(10 * sizeof(int));
int nodeDimensions = 0;
while( node != NULL )
{
  nodeArray[nodeDimensions] = node->Value;
  ++nodeDimensions;
  node = node->Next;
```

```
}
// Main loop, transformed
for( x = 0; x < nodeDimensions; ++x )
{
  product *= nodeArray[x];
}
free(nodeArray);
```

Because the actual number of elements in the linked list cannot be known until runtime, the system takes the safe approach and still allocates an array of ten elements, which it knows is the maximum possible number (alternatively, you could dynamically grow the array but this is not done here for the sake of simplicity). The actual number of elements is determined as the linked list is being traversed in the initialization loop. The main loop, other than using a different upper bound, is the same as for sublimation (note that this is not always the case; in certain circumstances, annihilation can cause additional transformations to take place).

The actual location of the relevant values is not important here; the dense index expression is not used anywhere. If it is used, there are still ways to deal with that; these are covered in Section 3.4.5.

After initialization, nodeArray contains the following data: [3,7,2,9]

The array is actually more than four elements long, but the rest of its contents are irrelevant as they are never read.

In the following sections, whenever there is a difference in the approach taken for sublimation or annihilation, this will be explicitly mentioned.

3.2 Pre- and post-initialization

In the example above, the only additional work that needed to be done came before the transformed main loop. The only thing done after the loop is freeing the memory associated with the array, which is not relevant here; it is simply clean-up code, and its omission would not change the semantics of the program (but it would introduce a memory leak).

Whenever the transformation needs to generate code that uses one of the data structures from the main loop, and this code is executed before the main loop, it is called **pre-initialization**. There are situations when such code needs to be executed after the main loop. This is called **post-initialization**.

A common case where this would be necessary is when the contents of the linked list are not read, but written to. Let us look at an example of a simple algorithm, using the same linked list structure as in the previous section, where each of the elements is assigned a value from some other source (in this case a constant is used for simplicity, but normally this would be the result of some computation; commonly this would involve a read of the same member).

```
/***DENSE_INDEX(node, node->Index)***/
/***DENSE_DIMENSION(node, 10)***/
while( node != NULL )
{
  node->Value = 42;
  node = node->Next;
}
```

For sublimation, we have an additional problem to solve: we need to know which indices of the dense array to assign to. As we will see in later sections, it does not matter what value is assigned to the unused indices, since these will never be read. You can see that in the code below the post-initialization loop will only read the original dense indices. If the statement that

determines the value is expensive, it may be desirable to perform it only for valid indices. In this example that is not the case; it is after all only a constant. Therefore, we simply execute the statement every iteration.

```
// Pre-initialisation
int *nodeArray = malloc(10 * sizeof(int));
int x = 0;
// Main loop, transformed
for( x = 0; x < 10; ++x )
{
  nodeArray[x] = 42;
}
// Post-initialisation
while( node != NULL )
{
  node->Value = nodeArray[node->Index];
  node = node->Next;
}
free(nodeArray);
```

For annihilation, we will need a pre-initialization loop to find out the proper upper bound to use. We will also need to keep a counter to know what index to use for indexing the nodeArray. Below is the same code, transformed with annihilation.

```
// Pre-initialisation
int *nodeArray = malloc(10 * sizeof(int));
int nodeDimensions = 0;
int x = 0;
Node *nodeCopy = node;
while( nodeCopy != NULL )
{
  ++nodeDimensions;
  nodeCopy = nodeCopy->Next;
}
// Main loop, transformed
for( x = 0; x < nodeDimensions; ++x )
{
  nodeArray[x] = 42;
}
// Post-initialisation
nodeCopy = node;
x = 0;
while( nodeCopy != NULL )
{
  nodeCopy->Value = nodeArray[x];
  nodeCopy = nodeCopy->Next;
  ++x;
}
free(nodeArray);
```

It will often be the case that the linked list value is both read and written. In this case, pre-initialization for the read access is generated first. The post-initialization will then be able to use the same array as the pre-initialization code. For example:

```
/***DENSE_INDEX(node, node->Index)***/
/***DENSE_DIMENSION(node, 10)***/
while( node != NULL )
{
  node->Value = node->Value * 2;
```

```
  node = node->Next;
}
```

Using sublimation, this will be transformed to the following, with fill-in value zero:

```
// Pre-initialisation
int *nodeArray = malloc(10 * sizeof(int));
int x;
Node *nodeCopy = node;
memset(nodeArray, 0, 10 * sizeof(int));
while( nodeCopy != NULL )
{
  nodeArray[nodeCopy->Index] = nodeCopy->Value;
  nodeCopy = nodeCopy->Next;
}
// Main loop, transformed
for( x = 0; x < 10; ++x )
{
  nodeArray[x] = nodeArray[x] * 2;
}
// Post-initialisation
nodeCopy = node;
while( nodeCopy != NULL )
{
  nodeCopy->Value = nodeArray[nodeCopy->Index];
  nodeCopy = nodeCopy->Next;
}
```

Transformation of this code with annihilation is analogous to this so the result of this is not given.

Note that because there is more than one initialization loop, which all depend on the initial value of node, a copy of node is made and used instead.

This example also shows something else. Because the fill-in value is zero, there is a more efficient way to set it (using the memset function, which on most architectures is much faster than setting the values manually in a loop), which allows us to use the original loop construct for the pre-initialization loop, potentially drastically reducing the number of iterations needed.

3.3 Two types of loops

As indicated earlier the loops in the matrix multiplication sample are different from typical linked list loops such as the loops in the examples in the previous two sections.

Recall this example:

```
int product = 1;
/***DENSE_INDEX(node, node->Index)***/
/***DENSE_DIMENSION(node, 10)***/
while( node != NULL )
{
  product *= node->Value;
  node = node->Next;
}
```

The transformation is told, via the DENSE_DIMENSION directive, that the dense data structure represented here had ten values. Since the linked list is sparse, it likely has less than ten nodes (only four in the example list), since certain values were omitted. This also means that the loop will iterate fewer than ten times. In other words, the loop body will be executed fewer times than there are values in the original structure.

Now consider the same example, but written in an alternative way.

```
int product = 1;
int x;
/***DENSE_DIMENSION(node, 10)***/
for( x = 0; x < 10; ++x )
{
  if( node != NULL && node->Index < x )
    node = node->Next;
  if( node != NULL && node->Index == x )
    product *= node->Value;
}
```

This code has the same effect as the earlier sample[1], but it has a form that is very similar to the loops in the matrix multiplication example. In particular, despite the fact that the linked list contains less than ten members, the loop body will execute exactly ten times. A guard is used to ensure action is taken only when the linked list element is valid for the current iteration. So the loop body will be executed exactly as many times as there are values in the original structure.

A loop which iterates only over the elements over the sparse list, as in the first case, is called a **sparse loop**. A loop which executes as many times as the dimensions of the original data, regardless of the number of elements in the sparse list, is called a **semi-dense loop**.

The treatment of the two types of loops is mostly the same, but there are some slight differences. You can immediately observe that when performing sublimation on a sparse loop, the transformed loop will have more iterations than the original, but for a semi-dense loop the number of iterations will remain the same. Conversely, when performing annihilation the number of iterations stays the same for a sparse loop, whereas it decreases for a semi-dense loop. It is mainly this difference in the annihilation process where it is necessary to make the distinction between sparse and semi-dense loops.

Another difference is the determination of the dense index. The dense index is an important piece of information for the transformation; it is the index that a value in the sparse linked list had in the original dense representation in the data. For a sparse loop, the dense index is not implicitly retrievable from the code, so it must be specified in some way. As we have already seen, the DENSE_INDEX transformation directive is used to indicate what expression to use to determine the dense index. In a semi-dense loop, the dense index is implicit in the progress of the loop, so the loop counter can be used. The DENSE_INDEX directive is not necessary in this case.

The transformation method will treat a loop as semi-dense whenever the DENSE_INDEX directive is not present for that loop or the expression it specifies is independent of the linked list expression. Otherwise is will treat it as a sparse loop.

3.4 Linked list transformation algorithm

We will now present the algorithm used to transform a linked list into a dense array. The linked list transformation consists of the following steps, which will be discussed separately:

[1] The fact that the two examples are semantically identical can be shown as follows: because Index is a proper dense index member, for any nodes x and y where y succeeds x in the list, x->Index < y->Index. This means that the expression node->Index < x will evaluate true in the iteration immediately following one where node->Index == x was true. Then after evaluating node = node->Next, node->Index must be equal to or higher than x. This means that all nodes in the linked list are visited, and that for each of them the condition node->Index == x will be true exactly once, leading to the same sequence of multiplications as in the original sample.

1. Find candidate structures that could be part of a linked list
2. Analyze usage of these structures in the code to look for linked list access patterns.
3. Determine whether the loop containing linked-list access can be safely transformed.
4. Identify data members in the linked list structure.
5. Generate appropriate replacement dense data structures.
6. Replace linked list accesses with dense structure accesses, and generate the appropriate initialization.
7. Attempt to move the initialization loops so that they are not in any outer loops

As indicated in the introduction, pointer access types vary wildly, so the transformation must be capable of determining which pointers are actually linked lists. Step one and two deal with a heuristic approach of finding pointer usage patterns that indicate a linked list. In step three and four, the identified code sections are validated to see if they are safe to transform. Step five and six generate the initialization code and replace the original code with the transformed code, and step seven tries to move the initialization loops out of the way as much as possible.

Before these steps can be executed, as we will see, it is necessary for certain features in the code to be normalized so that they can be processed. This is described in section 3.5.

In the following sections, we will look at each of these steps in more detail.

3.4.1 Finding linked list structure candidates

To begin the process of linked list transformation, user defined types that can be used as a node in a linked list must be identified. All structure definitions in the source file will be examined. A structure X can potentially be part of a linked list when it contains a member that is a pointer to the type X, because this member could be used as a next pointer. A **linked list candidate** is a pair of values, one being the name of the structure that meets these requirements, and the other being the name of the potential "next" member.

For instance, it will look at the following structure from the matrix multiplication example:

```
struct Cell {
    int Value;
    int ColIndex;
    int RowIndex;
    struct Cell *RowNext;
    struct Cell *ColNext;
};
```

This structure contains two members that are a pointer to the defining structure type, namely RowNext and ColNext. This means that this structure has not one, but two members that would allow it to be a linked list. Therefore two linked list candidates will be generated from this structure; one for RowNext, and one for ColNext. Note that it is not necessary to look at the names of the members; although the use of the word "next" is a definite hint, the next step makes it unnecessary, as it is the usage pattern of the structure that determines whether it is actually a linked list. By not paying any attention to identifier strings, we can transform code that uses, say, Chinese identifiers just as easily as English.

Often a structure containing two members of the defining type is actually part of a binary tree and not of a list. Indeed, any linked list candidate found by this step might not actually be a linked list. The next step will be able to determine whether this is actually a linked list. The nice thing about this approach is that even if this structure were part of a binary tree, if there is any place in the code where some loop iterates over for instance the left-most branch of this tree, in a fashion that looks like a linked list even though it is really not, it would still be possible to perform the transformation.

In the matrix multiplication example, the following linked list candidates would be found: Cell::RowNext, Cell::ColNext, RowHead::Next, and ColHead::Next.

Below is the algorithm for step one in pseudo-code.

```
function FindCandidateStructs(translationUnit)
  candidateList = {}
  foreach structDefinition in translationUnit
    foreach member in structDefinition
      if GetType(member) = pointer to GetType(structDefinition)
        candidate = [ struct-definition, member ]
        Add(candidateList, canditate)
      endif
    next
  next
  return candidateList
```

3.4.2 Analyzing candidate structure usage

Once the linked list candidates are identified, we will proceed to look for locations in the code where these structures are used, and will try to determine if this usage is eligible for transformation. It can do this on a case-by-case basis; each loop or nested loop in the code that uses a linked list candidate can be evaluated separately from any other usages, and even if there are usages that cannot be transformed, that does not mean that other usages of the same candidate in other places of the code cannot be transformed.

Each linked list candidate found in step one, consisting of a candidate structure and member, is examined to look for loops in the code that have a statement that takes a variable, whose type is a pointer to the candidate structure, and assigns it the value of the candidate member of that same variable, e.g. it looks for statements of the form `variable = variable->candidate_member`. This is called the **linked list iteration statement**. Looking at the MatrixMultiply function presented earlier, it can be seen that two of the candidates have a valid usage: leftRow = leftRow->Next is an iteration statement for RowHead::Next, while `leftCell = leftCell->ColNext` is one for Cell::ColNext. This tells us that the loops containing these statements might be a candidate for transformation. The loops that contain these statements will be marked as **candidate linked list traversals**. Note that only the directly containing loops are candidates; so in the case of the Cell::ColNext candidate only the innermost loop is a candidate traversal, not the middle and outer loops. Similarly, the candidate traversal for RowHead::Next is the middle loop, not the outer loop.

It is this step that is the most important part of the heuristic used to find linked lists. As noted above, looking for this pattern makes it unnecessary to consider the naming of the pointer members, and the fact that only ColNext is used for rightCell means that this is not a tree (actually, it might still be a tree, but as noted above this does not matter as long as this particular usage looks like a linked list).

If a loop is a candidate traversal for more than one linked list candidate, each must be evaluated separately. The presence of other variables used in the loop is not an obstruction, provided none of the rules in the next section are violated.

The C language provides some difficulty here if we wish to be flexible about how the linked list is used. So far we have talked about a linked list *variable*, while in fact it would be more desirable to consider this in terms of *expressions*. After all, if a loop uses `someArray[x] = someArray[x]->next` in a consistent fashion, we would want our algorithm to recognize this as well. In this case, someArray is not actually a candidate variable, because it is not of itself the correct type. However someArray[x] is an expression whose result type is of the correct type, so

we could call this a candidate expression. So the linked list iteration statement would be formally identified by an l-value expression which resolves to a pointer to a candidate structure, which is assigned a value that is found by taking that same expression, and accessing the candidate member for this linked list candidate.

The difficulty in implementing it so that expressions instead of variables are considered lies with the fact that both expressions would need to be equivalent. To this end, a pre-processing step that normalizes all expressions so that they can be directly compared is performed. Normalization is further covered in section 3.5.

Step two can be written in pseudo code as follows.

```
function FindCandidateTraversals(function, candidateList)
  loopStack = {}
  traversalList = {}
  statements = FlattenStatements(GetBody(function))
  foreach statement in statements
    if IsInsideSafeRegion(statement)
      if IsLoopStart(statement)
        # check for init loops generated
        if IsInitLoop(statement)
          SkipLoop(statement)
        else
          Push(loopStack, statement)
        endif
      elseif not Empty(loopStack) and
             statement like "expression = expression->member"
        testCandidate = [ GetType(expression), member ]
        if Contains(candidateList, testCandidate)
          traversal = [ Peek(loopStack), testCandidate, statement ]
          # check if this traversal was not already processed
          if not IsProcessed(traversal)
            Add(traversalList, traversal)
          endif
        endif
      endif
      if IsLastLoopStatement(Peek(loopStack), statement)
        Pop(loopStack)
      endif
    endif
  next
  return traversalList
```

The FlattenStatements function recursively flattens all compound statements in the function body into a single list containing all statements, marking the beginning and end of compound statements so it remains possible to identify them. IsLoopStart returns true if the statement is a “do” or “while” statement (for-loops are not considered because they are transformed into while loops during normalization). IsLastLoopStatement returns true if the statement after the given statement is outside the loop body of the specified loop.

The result of this function is a list of sets of a loop, linked list candidate and linked list iteration statement identifying the candidate traversals in this function.

3.4.3 Transformation evaluation

Before we can continue with transforming the loop, we must evaluate if the candidate traversal is safe to transform. Because of the incredible expressive power of C this is not an easy task; it is easy to accidentally forget to mention some situation that would prevent a correct

transformation. For this reason the conditions are, as much as possible, written so that they demand the code fits a certain form that is known to be safe instead of demanding it does not have a form that is not safe.

In the section below, when we refer to the linked list expression, this is the expression whose type is a pointer to the candidate structure and which is used in the linked list iteration statement. As indicated in the previous section, this can be a simple variable or a more complicated expression.

We must first define what we consider to be a modification of an expression. An expression is modified when it, or any sub-expression of it, is used on the left-hand side of an assignment expression; when the dereferencing of any sub-expression of the expression is used on the left-hand side of an assignment; and when a reference to the expression or any sub-expression of it is passed to a function. The last case does not necessarily mean the value will be modified, but it can be, and without analyzing the function it is impossible to determine if it will; therefore, we will assume worst case. Whenever the conditions below mention an expression or variable is modified (e.g. a condition like "x may not be modified") this definition of modification is used.

The following examples illustrate this. Consider the expression `x->y`. Here x is a pointer to some structure containing a member named y.

```
x->y = z;
```

The expression is directly assigned to, so it obviously means that the value is modified.

```
x = z;
```

X is a sub-expression of `x->y`. The value of x changed, so the next time `x->y` is evaluated it will yield different results. Therefore, we count this is a modification to `x->y`.

```
*x = z;
```

Although x itself did not change, the contained value of x (which is the structure it points to) is overwritten. Since z probably held a different value y, the value of `x->y` will have changed by this statement.

```
func(&x);
```

The function receives a pointer to x, so it can potentially modify the value of x; since the function is not analyzed, we will assume it does, and count this is a modification of x, and thus of `x->y`.

```
x->a = z;
```

This does *not* count as a modification. Although x is dereferenced, the result of this is not actually assigned to; it is used as part of an expression that is assigned to, but that does not constitute an assignment to x itself.

```
*(x->y) = z;
```

This does not count as a modification either. The value of `x->y` itself did not change.

Now consider the expression `x[y]`. Here, x is an array.

```
x = z;
```

As above, this counts as a modification of `x[y]`, since x changed.

```
x[y][z] = a;
```

This again does not count as a modification. The value of `x[y]` itself is only read and then used in a further expression. Although the result of that expression is assigned to, the value of `x[y]` itself is not changed.

An expression is considered to be **loop-invariant** if it is not modified anywhere inside the loop body. If an expression A is modified, but is assigned the value of an expression B that is loop-invariant, A is considered loop-invariant as well.

The **root non-invariant expression** for an expression A that is not loop-invariant and that is assigned the value of expression B is the root non-invariant expression of the non-invariant sub-expression of B. If the sub-expression of B that is not loop invariant is equal to A, the non-invariant expression of A is A itself.

Consider the following example:

```
const int N;
int x = 0, y, z;
while( x < N )
{
  x = x + 1;
  y = x;
  z = N;
  function(x, y, z);
}
```

This loop uses three variables: x, y and z. All three are assigned to in the loop body. Variable z is modified, but the value it is assigned is a constant. Every time when the value of z is read it has the same value, so z is loop-invariant despite of the assignment. The variable x is assigned a value that depends on its value from the previous iteration, so x is not loop-invariant. The non-invariant part of the expression that is assigned to x is x itself, so the root non-invariant expression of x is x. The variably y is assigned the value of x, which is not loop-invariant, so y is also not loop-invariant. The root non-invariant expression of y is equal to the root non-invariant expression of x, which is x.

Related, but not equal, to this is the notion of side-effects. An expression is said to have side-effects if it can modify *any* other expression. An expression that is – or contains – an assignment expression (such as `x = y`) has side-effects because it modifies the left-hand side of that assignment. Any expression containing a function call (e.g. `x + func()`) can also have side-effects, since the function can modify global or static variables. Because we do not analyze the function, we again assume the worst and say that any function call will have side-effects.

We can now state the conditions that must be met. In order for it to be possible to automatically transform a loop iterating over a linked, it must meet the following conditions:

1. The linked list expression must not have side effects.
2. Loop termination control must be trivial; if determining whether or not the loop should terminate is a large part of the computational cost of the loop, then doing the transformation will move – or more likely, multiply – this since it becomes part of the initialization loop. To make it easier to reason about this the normalization step will transform all loop structures into while loops or do-while loops (see section 3.5.2), although this is not done in the examples posed in this thesis. The loop termination guard must meet the following conditions:
    - It may not have any side-effects.
    - If the guard uses any variables that are not loop-invariant other than those that are part of the linked list expression (these are **loop control variables**; typically this will be a counter or similar), then there may be exactly one

modification of these variables and this modification must occur on every loop iteration (i.e. it may not be guarded).
    - The loop termination condition must be the only factor controlling termination of the loop; therefore the use of goto and break statements is prohibited.

  Typically, a linked list iteration loop will use a condition similar to `while( candidate_variable != NULL )`, or sometimes a counter such as in the matrix multiplication example. This means that this condition will not often pose a problem.
3. The linked list iteration statement may be the only statement in the loop body that modifies the linked list expression.
4. The "next" pointer member may not be identified as a data member in step 4.
5. Any expression, other than the linked list iteration statement, that might be moved to an initialization loop may not have side effects. In an implementation, it is simpler to check this while generating the initialization loops instead of here in a separate step.
6. Any expression that might be moved to the initialization loop may only use constants, loop-invariant values, members of the linked list structure, and loop control variables.
7. Any expression that dereferences the linked list expression may not be modified.
8. The linked list expression may not be passed to a function, because a function could violate condition 7 and the function could use members of the structure which means such usages need to be included in step four and also transformed in step six, neither of which can be done.
9. If the linked list expression is guarded, it must be possible to move that entire guard, including both the true and false parts, to the initialization loop.
10. When performing annihilation on a semi-dense loop, there must be a single guard that covers all statements in the loop body except for the linked list iteration statement and its guard, and statements related to loop control (such as those that increment the counter). This guard must meet the conditions for code that can be moved to the initialization loop, and it must not have an else-clause. There may be no statements (besides the linked list iteration statement and its guard) outside this guard. This ensures that there are no statements that need to be executed in iterations that would be eliminated by the annihilation process.

Below is the algorithm for checking these conditions:

```
function ContainsModifications(statement, expression, checkDereferenced)
  if statement like "expression = ..." or
     ContainsFunctionCallWithArgument(statement, pointer to expression) or
     (checkDereferenced and statement like "*expression = ...")
    return true
  endif
  foreach subExpression in expression
    if ContainsModifications(statement, subExpression, true)
      return true
    endif
  next
  return false

function ContainsFunctionCallWithArgument(statement, expression)
  foreach functionCall in statement
    foreach actualParameter in functionCall
      if actualParameter = expression
        return true
```

```
      endif
    next
  next
  return false

function HasSideEffects(expression)
  if ExpressionContainsFunctionCall(expression) or
     ExpressionContainsAssignment(expression)
    return true
  else
    return false
  endif

function FindModificationStatements(compoundStatement, expression)
  assignments = {}
  statements = FlattenStatements(compoundStatement)
  foreach statement in statements
    if ContainsModifications(statement, expression)
      Add(assignments, statement)
    endif
  next
  return assignments

# This function should not be called on the linked list iteration statement,
# the result would be wrong.
function CanMoveToInitLoop(loop, statement)
  foreach subExpression in statement
    if HasSideEffects(subExpression) or
       not (IsLoopInvariant(loop, subExpression) or
            IsLoopControlVariable(loop, subExpression)
      return false
    endif
  next
  return true

function CanMoveToInitLoop(loop, compoundStatement)
  statements = FlattenStatements(compoundStatement)
  foreach statement in compoundStatement
    if not CanMoveToInitLoop(loop, statement)
      return false
    endif
  next
  return true

function IsLoopControlTrivial(loop, linkedListExpression)
  loopCondition = GetLoopCondition(loop)
  if HasSideEffects(loopCondition)
    return false
  endif
  loopControlStatements = {}
  foreach variable in loopCondition
    if not IsLoopInvariant(loop, variable) and
       not ContainsVariable(linkedListExpression, variable)
      assignments = FindModificationStatements(GetBody(loop), variable, false)
      if Count(assignments) <> 1 or IsGuarded(assignments[0]) or
         IsInNestedLoop(assignments[0])
        return false
      else
        Add(loopControlStatement, assignments[0])
```

```
      endif
    endif
  next
  statements = FlattenStatements(GetBody(loop))
  foreach statement in statements
    # Check for goto or break statements; condition 2
    if statement = "break;" or statement like "goto ..."
      return false
    endif
  next
  # store the loop control statements for later use
  SetLoopControlStatements(loop, loopControlStatements)
  return true

function EvaluateCandidateTraversal(candidateTraversal)
  linkedListExpression = GetLinkedListExpression(candidateTraversal)
  loopCondition = GetLoopCondition(candidateTraversal)
  loop = GetLoop(candidateTraversal)
  iterationStatement = GetLinkedListIterationStatement(candidateTraversal)
  # Check for disallowed side effects; condition 1
  if HasSideEffects(linkedListExpression)
    return false
  endif
  # Check loop condition variables for validity; condition 2
  if not IsLoopControlTrivial(loop, linkedListExpression)
    return false
  endif
  loopControlStatements = GetLoopControlStatements(loop)
  # check function calls using the linked list expression; condition 8.
  foreach statement in statements
    if ContainsFunctionCallWithArgument(statement, linkedListExpression)
      return false
    endif
    foreach subExpression in linkedListExpression
      if IsPointer(subExpression) and
         ContainsFunctionCallWithArgument(statement, subExpression)
        return false
      endif
    next
  next
  # Check linked list expression assignments; condition 3 and 7
  assignments = FindModificationStatements(GetBody(loop),
                                           linkedListExpression, true)
  if Count(assignments) <> 1 or assignment[0] <> iterationStatement
    return false
  endif
  # Check if iteration statement guard and all associated statements
  # can be moved; condition 9
  iterationGuard = GetGuardAroundStatement(iterationStatement)
  if iterationGuard <> null
    if not (CanMoveToInitLoop(loop, iterationGuard) and
       CanMoveToInitLoop(loop, GetTrueStatement(iterationGuard)) and
       CanMoveToInitLoop(loop, GetFalseStatement(iterationGuard)))
      return false
    endif
  endif
  # Condition 10
  if transformationMode = annihilation and LoopType(loop) = semi-dense
    # Note: compound statement is not flattened here, so we only check
```

```
    # statements directly inside the loop, not further nested statements.
    guardFound = false
    foreach statement in GetBody(loop)
      if statement <> iterationGuard and statement <> iterationStatement
        if statement like "if( expression ) ..." and not guardFound
          if not CanMoveToInitLoop(loop, expression)
            return false
          endif
          guardFound = true
        else if not Constains(loopControlStatements, statement)
          return false
        endif
      endif
    next
  endif
  return true
```

The code above checks all conditions except 4, 5 and 6. Before condition 4 can be checked, it is necessary to find data members, which will be done in the next step. And as indicated, conditions 5 and 6 are more easily checked when the code sections to which they apply are identified in later steps. The code for the next steps will use some of the functions defined here to aid in those checks.

There are three other global conditions that are not checked but assumed to be true for any code marked safe for transformation. The first is that there is no aliasing, or at least that all aliasing done is safe. If there is another variable that actually aliases the linked list expression, the above conditions would not consider a write to such a variable harmful even though it actually is. Trying to determine whether this is the case is a very difficult problem since it must be done globally. It is possible that a global variable aliases the linked list expression, and that a function, called in the loop body, ends up modifying this value. In the example, the Matrix structure is passed by value, but it contains two pointers, each of which might have a global alias created before the MatrixMultiply function is even called. Indeed, any of the Cells pointed to in the matrix may have some global variable pointing to it, which means that in some iterations the current value of leftCell might have an alias and in others not. One can mitigate some of this by posing some strict requirements, such as prohibiting global variable accesses, including in any functions called. That would still leave aliasing through function parameters, which cannot be analyzed at all unless you know every possible invocation point of the function (and even then it is very difficult) and local aliases. Some steps can be taken to reduce the problem of aliasing, which are taken in the normalization described in Section 3.5.1, but we cannot completely solve the aliasing problem.

The second condition is that no external processes may modify any value. A small check could be done in the program which would disqualify the use of variables marked volatile, but even if none are marked so it does not exclude external interference (memory boundaries may be enforced in other ways such as operating system-specific critical sections or other synchronization primitives).

The third condition is that usages of the linked list members are independent. If they are only read, the first and second condition will ensure that this is true (since the member may not be modified by aliasing or by external code). If a member is written to, a write in one iteration may not affect a read or write in a later iteration. Unless the linked list contains cycles, this is unlikely to happen anyway.

We must simply assume that these conditions are met, and require the user to provide only input code that meets them. If code that breaks these rules is transformed, transformation will

succeed but likely yield code that does not do the same as the original code anymore, or, worse, that sometimes behaves the same and other times does not depending on synchronization, initial values, or other external factors.

For safety, it is assumed that by default that all code in the translation unit is unsafe to transform. It will look for transformation directives that denote safe code sections or individual loops, and only attempt to transform those that are indicated safe. Code marked safe is only assumed to meet the criteria that cannot be checked. All the requirements that can be checked will be checked, so marking a loop as safe to transform is no guarantee that it can or will be transformed. Transformation directives are further covered in section 3.6.

The normalization mentioned earlier and discussed in further detail in section 3.5 is of great importance for this step, since it needs to be able to reliably determine if expressions are used in safe ways in the loop body. If equivalent expressions are not of the same form, unsafe usages may be missed, leading to unsafe transformations.

In order to evaluate this step, data dependence and flow analysis must be performed on all variables. For every usage of a variable, it must be known what other values that variable depends on, i.e. those values used when that variable was last assigned to. This is a fairly straight-forward process that most compilers will do anyway to determine where instructions can be reordered or removed. Here, the information is used to find loop-invariant variables and to find indirect usages of important values. Therefore, whenever the above list of conditions mentions a usage of a variable (or expression), this may also mean that a variable (or expression) is used that depends on that variable.

The list of conditions in this step contains only those conditions that are strictly necessary for this transformation. In this thesis, we will further transform the code by translating it to FORTRAN. Some additional conditions apply to make this translation possible, which will be covered in the relevant section.

3.4.4 Find data members

In order to generate replacement dense data structures, it must be known what data this dense data structure should contain. In other words, it needs to determine what members of the linked list candidate structure are used to store either input or output data relevant to the computation. To find this, it must consider all members besides the linked list next pointer already identified. If a structure contains more than one “next” pointer, we will exclude only the one that is actually used in this candidate traversal.

This analysis must be done for each candidate linked list traversal; it cannot be stored per structure and re-used. This is because we want to have the smallest possible number of data members, and not each usage of the linked list structure needs to actually use all the remaining members.

The first part of this step is to look at the loop body, including any nested loops, and find all members of the structure that are used. Obviously members that are not used at all need not be given any further consideration.

However, it is not the case that all members that are used must automatically be data members. Members that are written to will always be considered data members, but for a member that is read to be considered as a data member the read operation must affect some variable that is used after the current loop iteration. That means it is either assigned to a variable that will be read after the iteration, or it participates in a guard of a modification of a variable that is used after the iteration. Because function calls can have side-effects, guarding a function call also counts.

Consider the following example:

```
int prev = 0;
while( node != NULL )
{
  node->Value2 = prev;
  prev = node->Value1;
  node = node->Next;
}
```

This loop sets the Value2 member to the value of the Value1 member of the previous node in the list (this is not a particularly likely operation to perform on a sparse list, but that is not the point). The value of prev is not used after the loop, but it is used in the loop body *before* it is assigned to again. This means that, in the flow of execution of the loop, the value of prev will be used after the current iteration of the loop ends. This means that the value of node->Value1 affects something that happens after the iteration in which it is read, so node->Value1 must be a data member.

```
int remainder;
while( node != NULL )
{
  remainder = node->Value % 2;
  if( remainder == 1 )
    Foo();
  node = node->Next;
}
```

This example calls the function Foo() for odd values of node->Value. Here, node->Value is assigned to the variable remainder, which is not used after the loop or across iterations. So at first glance, Value would not qualify as a data member. However, the value of remainder is used in an if-statement which guards a function call. This function call might set global variables, write output to a file or the console, make a network connection, etc.; i.e. it can have any number of side effects that last beyond the scope of the loop. This means that node->Value is a data member after all.

As you can see, data dependence analysis is absolutely vital to finding data members.

If a potential data member is used only in a guard, there is still the possibility that it does not need to be a data member. If the guard that uses the data member will be moved to the initialization loop and completely eliminated from the transformed main loop, the member would not count as a data member. A guard will be moved to the initialization loop only if:

- It is used to guard the linked list iteration statement. This guard construct will be moved in its entirety to the initialization loop, so it is not needed in the transformed main loop.
- It guards the usage of other identified data members, and the guard that this value is part of uniquely determines whether the other data member is used or not (if the member is used in the true-part of the guard that uses the data member, it may not be used in the false-part, or vice versa, and it may not be used outside the if-statement). In this case, the fill-in value, or if necessary an additional validity check (as indicated in the next section) will replace this guard, so it will no longer be present in the main loop after transformation.

Naturally, the guard expressions must meet the conditions set in the previous section for code that will be moved to the initialization loop.

Again, data dependence analysis is used:

```
int product = 1;
int temp;
while( node != NULL )
{
  temp = node->Value;
  if( node->Index % 2 == 0 )
    product *= temp;
  node = node->Next;
}
```

This computes the product of those values that have an even numbered dense index. Without using dependency analysis, we would say that node->Index is used to guard an operation with side-effects, and it does not uniquely guard a data member, so it must be a data member. However, although node->Value is read outside the guard, it is in actuality only used if the guard evaluates to true, because it is assigned to temp, and temp is only used inside the guard. So using data dependence analysis, we can show that node->Index uniquely guards the use of node->Value, which means it can be removed if there is a fill-in value for node->Value or replaced with a validity check if there is not. This means that this expression will be removed from the transformed loop so node->Index is not a data member.

Furthermore, if the dense index expression supplied by the DENSE_INDEX directive uses a member of the linked list structure, if that member is read in the loop it is still not a data member, as its usage can be replaced by the counter for the new loop.

Let us look at the innermost loop of the matrix multiplication algorithm as an example:

```
for( x = 0; x < dimensions; ++x )
{
  if( leftCell != NULL && leftCell->ColIndex < x )
    leftCell = leftCell->ColNext;

  if( leftCell != NULL &&
      leftCell->ColIndex == x &&
      leftCell->RowIndex == row )
  {
    result[row][col] += leftCell->Value * right[x][col];
  }
}
```

The linked list traversal we are looking at is that of leftCell, a variable of type Cell. We see that the following members are used: the leftCell pointer value itself (in a comparison to NULL), and the members Value, ColIndex and RowIndex (note that once again, we are not looking at names, so we are ignoring the fact that one of the members is conveniently named Value; even if we did this, it would not guarantee that the other members could be discarded, so it would not help). Value is used on the right-hand side of an assignment, and is assigned to something that was passed by reference to this function, which means it can be used after the function returns, which is after the loop iteration, so Value is most certainly a data member. The pointer value, ColIndex and RowIndex are used in guards. The first guard is the guard of the linked list statement, which will be moved entirely to the initialization loop, so according to the first point above, we can ignore it. The second guard uniquely guards the use of Value, and all usages meet the criteria of the previous section (since NULL is a constant, x is the loop control and row is loop-invariant). This means that these three guard expressions can be replaced by a validity check or removed entirely if a suitable fill-in value is found (see the next section), and that none of these three members are data members, leaving only Value.

Note that it is perfectly valid for the pointer value itself to be identified as a data member. However, if this has not been eliminated after transformation, it will obstruct translation to FORTRAN. Fortunately, that is not the case in this example, and it would not likely often be the case either. Still, it is important to realize that this would not be a problem for the transformation itself, only for the follow-up steps we wish to take in this thesis.

The algorithm for step four:

```
function WillBeRemoved(guardStatement)
  ; Find which data members are used when the guard evaluates true or false.
  trueDataMembers = GetDataMembersForCompound(GetTrueStatement(guardStatement)
  falseDataMembers =
       GetDataMembersForCompound(GetFalseStatement(guardStatement)
  ; If the two are not the same, there is a member that is used
  ; in one but not the other, thus this guard will be removed
  return trueDataMembers <> falseDataMembers

function FindDataMembers(traversal)
  dataMembers = {}
  listStruct = GetLinkedListType(traversal)
  listExpression = GetLinkedListExpression(traversal)
  loopCondition = GetLoopCondition(traversal)
  loop = GetLoop(traversal)
  iterationStatement = GetLinkedListIterationStatement(traversal)
  loopBody = GetBody(loop)
  compound = GetInnermostUnprocessedCompound(loop)
  while compound <> null
    ; Statements are not flattened, so it does not process
    ; nested compound statement
    foreach statement in compound
      foreach member in listStruct
        if ContainsModifications(statement, "listExpression->member", false)
          ; Write always means a data member
          Add(dataMembers, [ member, Write ])
          AddDataMemberForCompound(compound, member)
        else if not IsDenseIndexExpression("listExpression->member") and
             ((statement like "lvalue = rvalue" and
             ContainsDependantExpression(rvalue, "listExpression->member") and
             IsUsedAfterLoopIteration(loop, lvalue)) or
             (statement like "if( expr )" and
             ContainsDependantExpression(expr, "listExpression->member") and
             not (statement = GetGuard(iterationStatement) or
             WillBeRemoved(statement))))
          ; this is a read that has effects outside the iteration or is a guard
          ; that is not the guard for the iteration statement and not a guard
          ; that can be moved.
          Add(dataMembers, [ member, Read ])
          AddDataMemberForCompound(compound, member)
        endif
      next
    next
    SetProcessed(compound)
    compound = GetInnermostUnprocessedCompound(loop)
  end while
  if Contains(dataMembers, GetNextPointerMember(traversal)
    return null
  endif
  return dataMembers
```

The algorithm processes compound statements in nesting order, starting with the most deeply nested statement that has not yet been processed (for the purposes of this algorithm, if a "for" or "if" or similar statement has only a single nested statement instead of a compound, this is still treated like a compound). This way whenever an if-statement is encountered it is already known what data members are used in the statements it guards. The GetDataMembersForCompound function, which is used in determining if a guard uniquely guards a data member and will therefore be removed or replaced in the transformed loop, returns only those data members that are used in all code paths of the given compound, ensuring that nested if-statements are correctly handled.

Data dependence analysis results are used by the function ContainsDependantExpression; this function returns true if the expression in the first parameter contains any expression that is dependent on the expression in the second parameter. IsUsedAfterLoopIteration also uses data dependence analysis to determine if the value of the expression in the second parameter is used beyond a single iteration of the specified loop, as was explained in the beginning of this section.

3.4.5 Generate dense data structures

Once the data has been identified, a dense array must be generated that holds this data. Important points in this step are determining the element type and the bounds of this array.

The element type is relatively straight-forward. It is ideal if there is only a single data member, because then this can be used as the array type (provided a validity flag is not needed, which is explained below). Otherwise a structure must be generated that can hold all the identified data members.

The sparse linked list does not include all the values of the original data. If the original loop was a sparse loop, the transformed loop will have more iterations than the original. The operations that are performed on the data members in the loop body cannot be blindly executed in the added iterations; this might change the semantics of the code. We must make sure that for those values that were omitted in the original linked list, the operation is either not executed or has no effect. For semi-dense loops the same thing applies, but only for operations on data members that are not executed in all possible code paths through the loop body. If they are executed in all code paths, they can safely be executed in all iterations of the new loop as well, since for a dense loop the number of iterations stays the same for sublimation.

With annihilation this is slightly different. Since the loop will not gain iterations, there is no need to prevent anything from being executed. In a semi-dense loop there will be fewer iterations after transformation, so anything that is executed on all code paths will actually be executed fewer times; because there is no good way to deal with this, annihilation is impossible in these cases; this is expressed by condition 10 in section 3.4.3.

But for sublimation, we must find a way to nullify these operations. There are two ways to do this: for each identified data member, we can try to find a fill-in value that causes these operations to have no effect, or we must introduce a flag that indicates whether the value at that position in the array is valid, which can be used to guard the relevant operations. This must be done separately for each data member; so if there is more than one data member, we might end up needing more than one flag. Finding a fill-in value is preferable to using a flag, since it will greatly simplify the transformed main loop, easing further optimization.

Obviously, data members that are only written to do not need a fill-in value or a guard. Post-initialization loops only read the valid values, so it does not matter what is written to the array indices that represent fill-in positions.

If we are to use a fill-in value for a data member, we must be certain that it eliminates all side-effects from the operation that use that data member or are otherwise affected by it (for example via a dependant variable). That means that no variables whose values would normally change may change their value now. If the data member is part of a guard, the fill-in value must be such that the guard evaluates to a value that causes any statements with side effects not to be executed. If the data member is used directly in an assignment, the value being assigned to may not change. Finding a semantically correct fill-in value automatically is however not a trivial matter. As has been noted in [3], certain operations are known to have no effect, such as multiplying by one or adding zero to a variable. This means that if the data member participates in a multiplication or addition, one or zero respectively could be the correct fill-in value. In the case of the matrix multiplication example, if the guard evaluates true, a value is added to the result, and that value is the data member of the linked list variable multiplied by another value, and if the guard evaluates false, nothing is added. It is therefore conceivable that an implementation of the transformation could determine that zero is a safe fill-in value, because it would cause zero to be added to the result, which is the same as doing nothing.

Alternatively, the programmer can use directives to indicate what the fill-in value should be. In this case it falls on the programmer to ensure that the fill-in value is correct.

If it is not possible to find a fill-in value, a guard must be put in place that prevents these operations from happening. In these cases, a validity flag for that data member is added to the list of data members. The initialization loop will make sure that this flag is set to true for those array indices where the value of the data member comes from the linked list, and false when it does not. The guard in the transformed loop that is placed around the operation will check this flag. If there is already a suitable guard in the original loop, as will often be the case in semi-dense loops, it may be replaced by this new guard.

Of course finding a fill-in value is preferable, since that means no new guards need to be added and some of the old guard might be removed completely.

For a semi-dense loop, where a guard is used in the original loop to distinguish between valid and omitted iteration values, it may be the case that this guard is responsible for other statements with side effects as well, that do not involve any data member (such statements can be in either the true or false part of the guard). This means that, even if a fill-in value for the data member is known, not all side effects related to the guard are eliminated, so the guard must be maintained. Similarly, in a sparse loop, any statements that do not involve the linked list expression will need to be prevented from execution on iterations that were introduced by the transformation, so a validity flag is needed. In this case, if a fill-in value is known, the guard can use a test for this fill in value so no validity flag needs to be added.

If a guard uniquely guards more than one data member, the guard can be omitted only if fill in values for all data members can be found that eliminate the side-effects from all guarded statements.

Let us once again recall the innermost loop of matrix multiplication example.

```
for( x = 0; x < dimensions; ++x )
{
  if( leftCell != NULL && leftCell->ColIndex < x )
    leftCell = leftCell->ColNext;

  if( leftCell != NULL &&
      leftCell->ColIndex == x &&
      leftCell->RowIndex == row )
  {
```

```
    result[row][col] += leftCell->Value * right[x][col];
  }
}
```

This loop is semi-dense; “dimensions” is the original dense upper bound, and even though the sparse linked list contains less than “dimensions” values, there are still “dimensions” iterations. As we noted in the previous section, the single data member “Value” is uniquely guarded by a guard that can be completely moved into the initialization loop. Since there are no other statements in the guard body besides the one that uses the data member, finding a fill-in would be ideal since it would mean we can remove the guard entirely.

There is indeed a fill-in value here, but let us first take a look at what would happen if the compiler was unable to find it. In that case, a validity flag is necessary to indicate whether “Value” should be used. This means that a new structure is needed for the dense array which contains this validity flag. The transformation would generate the following structure:

```
struct CellData
{
  float Value;
  int ValueValid; /* boolean flag */
};
```

ValueValid is the flag for the Value data member; it is a Boolean value but has type int since C has no Boolean type. You will notice that the name of this dense structure is the name of the original linked list structure followed by “Data”. Naturally, the name does not actually matter; in a real implementation, some steps would need to be taken to ensure a unique name, but other than that, the name can be arbitrary.

The transformation will create an array of CellData values (which we will call “leftCellArray” because it replaces the leftCell linked list variable), and initialize each ValueValid flag to false. The initialization loop will use the original guard around “Value”, and when it evaluates true it will set “Value” and set “ValueValid” to true.

The guard can for the transformed loop will be replaced by the guard `if( leftCellArray[x].IsValid )`. More details about how this transformation actually takes place will follow in the next section.

As we indicated, there is in fact a fill-in value here, namely the value zero. This value causes the statement involving “Value” to have no effect; the value of “result[row][col]” does not change if “Value” is zero (zero is probably the most common fill-in value for sparse data structures, especially sparse matrices, however it need not always be; we have already seen an example where it was one instead).

If it can be determined that zero is the fill-in value, either automatically or from a directive, this can lead to a significant simplification of the resulting code. The “Value” member is now the only data member, so a new structure is not necessary. Instead, the element type for the dense array will simply be float, the type of the value member. Because there are no other statements with side effects, the guard around the statement using “Value” can be eliminated entirely.

We also need to determine the dimensions of the dense array. It would be most preferable if the dimensions are a constant, but that is unlikely. Even if they are not, it is preferable that a simple expression, whose value can be determined a priori at runtime, determines the dimensions. If the loop we are transforming is a semi-dense loop, and the loop is countable, we iteration count of the loop is the size needed for the array. In the matrix multiplication example, the loop is countable and the number of iterations depends on the value of “dimensions”, so the

value of that variable is the size of the array. This expression will be used to allocate the array, and in the next step also as the upper bound for the dense loop.

In the case of a sparse loop, we cannot automatically determine if such an expression exists. In this case, the DENSE_DIMENSIONS transformation directive can be used to indicate what expression to use.

But in certain situations there is no such expression, for example when using annihilation where the dimensions will depend on the density of the data, and there might not be any way to find that other than walking the linked (in the matrix example there is not). We must also be prepared with a situation where even though there is such an expression, we cannot determine what it is and no directive is specified either. In these cases, we can use a dynamically growing array. To minimize the overhead of dynamically allocating and reallocating an array, we will use the typical approach used by many dynamic array implementations (such as C++'s std::vector class in most STL implementation) where the array is doubled in size each time it must grow. A counter is kept during execution of the initialization loop which indicates the length of the array. After the initialization loop is complete, the value of this counter can be used as the upper bound for the new transformed main loop.

For a post-initialization loop this is more troublesome, as the array it uses is allocated before the main loop so the size cannot be determined during the initialization loop. In some cases a post- and pre-initialization loop share the same array, which solves the problem, but otherwise an additional pre-initialization loop is needed that does nothing except walk the linked list and count the number of iterations if the bounds cannot be determined at compile time.

If we are doing annihilation and the dense index expression is used (which for semi-dense loops means the loop counter), the original index value will be needed. If this value is read anywhere in the loop (with the exception of the loop control statement for a semi-dense loop), an array with the relevant values of the index expression will be generated. This dense data structure is a simple array where the element type is the same as the type of the index expression (usually int). In order to prevent introducing irregularity in the loop, we will make a special case for when the expression is used to index an array. Here creating an array for the index expression would introduce indirection for the array it was indexing, so instead a new array will be generated with the same element type as the original array, which will be mapped to the original array during pre- or post-initialization. This can only be done if all other components of that expression are loop-invariant and safe to move to the initialization loop.

For example, in the matrix multiplication example, the loop variable x of the innermost loop is used to index the "right" array. That means if annihilation is performed, a new array to replace "right" is generated, and because the resulting type of "right[x]" is int*, the replacement array's element type will be int* as well.

The pseudo code algorithm for step five is given below.

```
function NeedsFillIn(loop, dataMember)
  return not (AccessType(dataMember) = Write or
         (IsSemiDense(loop) and
         IsUsedOnAllCodePaths(GetBody(loop), dataMember))

function FindAdditionalExpressionsNeedingTransformation(loop,
                                                        linkedListExpression)
  expressions = {}
  indexExpression = GetIndexExpression(loop)
  foreach variable in loopControlVariables
    transformVariable = false
    foreach statement in FlattenStatements(GetBody(loop))
```

```
      if not (ContainsExpression(linkedListExpression, variable) or
         IsLoopControlStatement(statement)) and
         ContainsExpression(statement, variable)
        # this statement reads the variable but is not a loop control
        # statement like x++
        foreach subExpression in statement
          if ContainsExpression(subExpression, variable)
            if subExpression is array index expression
              # This is an array subscripted by this variable,
              # transform the array
              Add(expressions, subExpression)
              if IsModification(subExpression)
                AddUnique(AccessTypes(subExpression), Write)
              else
                AddUnique(AccessTypes(subExpression), Read)
              endif
            else
              transformVariable = true
            endif
          endif
        next
      endif
    next
    if transformVariable
      Add(expressions, variable)
      AccessTypes(variable) = { Read }
    endif
  next
  return expressions

function GenerateDataStructures(traversal, dataMembers)
  listExpression = GetLinkedListExpression(traversal)
  loop = GetLoop(traversal)
  iterationStatement = GetLinkedListIterationStatement(traversal)
  denseArrays = {}
  accessTypes = {}
  foreach dataMember in dataMembers
    if NeedsFillIn(loop, dataMember)
      fillIn = GetFillIn(loop, dataMember)
      if fillIn <> null
        SetFillIn(dataMember, fillIn)
      else
        Add(dataMembers, CreateFlag(dataMember))
      endif
    endif
    AddUnique(accessTypes, AccessType(dataMember))
  next
  if Count(dataMembers) = 1
    arrayType = TypeOf(dataMembers[0])
  else
    arrayType = new struct
    foreach dataMember in dataMembers
      AddMember(arrayType, dataMember)
    next
  endif
  arrayLength = GetDenseDimensions(traversal)
  Add(denseArrays, [linkedListExpression, arrayType, arrayLength, accessTypes])
  if GetTransformationType() = annihilation
    foreach expression in FindAdditionalExpressionsNeedingTransformation(loop,
```

```
                        listExpression)
      Add(denseArrays, [ expression, GetType(expression), arrayLength,
                        AccessTypes(expression) ])
    next
  endif
  return denseArrays
```

The GenerateDataStructures code will return a list of triplets, each indicating the source, type and length of a dense array that will need to be generated. The function GetDenseDimensions is used to determine the length; it will read the DENSE_LENGTH transformation directive or use automatic means to try and determine the length. If it cannot determine it, its return value is -1, which will be a cue for the transformation algorithm to use a dynamic array.

The GetFillIn function uses the FILL_IN directive and perhaps some of the procedures indicated above to find the fill-in value.

3.4.6 Generate initialization loop and transform main loop

We can now categorize every statement in the original main loop based on the information gathered in the previous steps. Statements will be classified as one or more of the following:

1. Data member access: these statements access the identified data members of the linked list, and must thus be transformed to use the new dense structure. The algorithm will also mark whether it is being read or written.
2. Loop control variable access (direct): for annihilation on a semi-dense loop, these are statements that are not directly part of the loop control itself (so the statement that increments the value is not considered) but they do use the loop control variable for some purpose other than array indexing. Guard statements that will be moved to the initialization loop will not be marked this way. This will always be a read, since this value may not be written to outside the loop control due to the requirements stated in section 3.4.3.
3. Loop control variable access (indexing): same as above, only the variable is used to index an array. Read or write access to the referenced array element is marked.
4. Linked list iteration statement: this statement must be moved to the initialization loop.
5. Guard for the linked list iteration statement: idem. This includes all statements under that guard.
6. Guard where the compound statement in the true or false part contains a statement of category one, two or three. Statements will be marked category six separately for each value that is read or written in such a statement in the true or false part. If the value is accessed in all possible control paths in both the true and false parts of the guard, the guard need not be present in the initialization loop and thus will not be marked category 6.
7. Loop control statements. For example, this is the statement that increments the loop counter. A sparse loop typically does not have any loop control statements.
8. Other statements: all statements involving the linked list variable must necessarily be of one of the first six categories, so this leaves statements that do not involve the linked list variable at all. These do not need to be transformed and will be left in the transformed loop unmodified.

In section 3.4.7 we will see that there is another category of statement that can be present in the loop, namely previously generated initialization loops that we are attempting to extract from

this loop. These statements are ignored here; they are not added to the initialization loops and they are not retained in the transformed main loop either. They are dealt with separately.

Using this classification, the transformation can generate the initialization loops and the transformed main loop.

Below is the innermost loop of the matrix multiplication example, with the categories for each statement indicated in the comments.

```
for( x = 0; x < dimensions; ++x /* 7 */ )
{
  if( leftCell != NULL && leftCell->ColIndex < x ) // 5
    leftCell = leftCell->ColNext; // 4

  if( leftCell != NULL &&
      leftCell->ColIndex == x &&
      leftCell->RowIndex == row ) // 6 (for leftCell->Value and x)
  {
    result[row][col] += leftCell->Value * right[x][col]; // 1 (read), 3 (read)
  }
}
```

The assignment is both category one and three: one for the use leftCell->Value, and three for the use of x (which is only relevant for annihilation). The guard statement uniquely guards both of these, so it is marked category six for both of them.

The assignment is category three, not two, because it uses x to index the array "right". This means that the value that will be copied to the replacement array is that of "right[x]".

In the real transformation, this for-loop would have been transformed into a while loop by the normalization. This means that the "++x" expression would have been a separate statement at the end of the loop, which is category seven.

3.4.6.1 Loop termination

Several loops will be generated during the transformation process – the transformed main loop and one or more pre- and post-initialization loops – and they will all use one of two possible termination guards: the original guard from the untransformed main loop, or the new guard for the dense loop. The new loop statement will always be a for-loop, e.g. `for( x = 0; x < dense_length; ++x )` where x is a new counter variable introduced by the algorithm, and dense_length is the total number of elements in the dense array.

When dealing with sublimation on a semi-dense loop, the new and old guard will often be the same.

3.4.6.2 Pre-initialization

The pre-initialization loops can use either the original or the new loop guard. The original guard must be used if the length of the dense array is not known yet (which will always be the case for annihilation). In other cases, the new guard can be used. Which it will use then depends on whether there is a fill-in value, and if it can be set using a single operation. For example, if the fill-in value is zero, it is most efficient to initialize the whole dense array to that value using the memset function in C. If that is not possible, the new guard must be used. If any of the statements that need to be moved to the initialization loop use any of the original loop control variables, the loop must also use the original loop guard.

The transformation algorithm will now generate statements for the initialization loop based on the categories of the original statements, maintaining their relative order from the original loop.

The first statements in the loop will be statements to set the data members are to their fill-in value or their flags to false. If the fill-in was set in advance using memset, this is not necessary and these statements are omitted

For statements that fall into category one, two or three where the value is being read, a statement is generated that copies the value into the array that will replace it. If the statement has more than one such value, it will generate more than one statement. These statements will have the following form:

```
array[index_expression]->member = original_value;
```

Here, `array` is the dense data structure generated in the previous step that matches the data member or other value represented by `original_value`. The `index_expression` will be either the counter variable if one is available, or the expression that can be used to retrieve the original dense index which was determined earlier (specified by a directive). If the element type of the dense array is a structure, the member that matches the value being read is specified. Otherwise, that part of the statement is omitted. An example of such a statement for the matrix multiplication sample would be:

```
leftCellArray[x] = leftCell->Value;
```

If a validity flag is used, this statement will be immediately followed by a statement that sets the validity flag associated with that value to true.

Category four statements, i.e. the linked list iteration statement, will be copied verbatim to the initialization loop. The same is true for its guard (category five).

Category six statements uniquely guard some data member; these guards are copied to the initialization loop.

Category seven statements are copied to the initialization loop if the original loop guard is used; they are needed to ensure the same loop progression.

Category eight statements are left alone; these will not be used in the initialization loop.

If the new loop guard is used for the initialization loop and the original loop was sparse, an additional guard of the following form will be added:

```
if( linked_list_expression != NULL && index_expression == counter )
```

Here `index_expression` is the expression used to retrieve the dense index specified using a directive, and `counter` is the counter variable of the new loop guard. This guard simply checks if the dense index of the current linked list position matches the loop index. The entire loop body will be placed under this guard.

For the matrix transformation example, this leads to the following pre-initialization loop for the leftCell value in the inner loop, using sublimation:

```
leftCellArray = malloc(sizeof(float) * dimensions);

memset(leftCellArray, 0, sizeof(float) * dimensions);
for( x = 0; x < dimensions; ++x )
{
  if( leftCell != NULL && leftCell->ColIndex < x )
    leftCell = leftCell->ColNext;

  if( leftCell != NULL &&
      leftCell->ColIndex == x &&
      leftCell->RowIndex == row )
  {
    leftCellArray[x] = leftCell->Value;
```

```
  }
}
```

Using annihilation, we get a slightly different loop for leftCell, and an additional loop for right[x], which as noted earlier is necessary because the progression of x will change in the main loop. Below is the loop for right[x], which uses a dynamically growing array.

```
rigthArrayLength = 100;
rightArray = malloc(sizeof(float*) * rightArrayLength);
newDimensions = 0;
// Initialisation loop
for( x = 0; x < dimensions; ++x )
{
  if( newDimensions >= rightArraySize )
  {
    leftCellArraySize *= 2;
    rightArray = realloc(
           rightArray, sizeof(float*) * rightArraySize);
  }
  if( leftCell != NULL && leftCell->ColIndex < x )
    leftCell = leftCell->ColNext;

  if( leftCell != NULL &&
      leftCell->ColIndex == x &&
      leftCell->RowIndex == row )
  {
    rightArray[newDimensions] = right[x];
    ++newDimensions;
  }
}
```

As said earlier, there are a few situations where the upper bound to use for the new loop guard will not yet be known. A counter will be added to an initialization loop that counts the number of valid elements to determine what this bound is (called newDimensions in this sample). This counter can then also be used for the `index_expression` mentioned above. If there are no pre-initialization loops (for instance because only post-initialization is needed) and it is necessary to get this upper bound, a pre-initialization loop will be generated that does nothing but get this count. The final value of this counter will be used as the upper bound for the new loop guard.

Although no values from leftCell are being read, it is still necessary to use it in the initialization loop to determine which values of the "right" array need to be used.

Sections 3.1 and 3.2 show additional annihilation examples where a count is used in the pre-initialization loop.

All our examples so far have directly used the linked list expression (e.g. leftCell) in the initialization loops. If there is more than one initialization loop, they will all depend on the initial value of leftCell being what it was in the original code, so this is not safe. Instead, a new variable will be used that is initialized to the value of the original linked list expression and substituted for it in the initialization loops. After the final post-initialization loop – or directly after the main loop if there is no post-initialization – the original linked list expression will be set to the value of the copy so that its value after the transformed code matches that of the original code.

The algorithm for generating the pre-initialization loops is as follows:

```
function GeneratePreInitLoop(traversal, denseArray)
  initStatements = {}
```

```
localVariables = {}
length = GetLength(denseArray)
arrayType = GetType(denseArray)
source = GetSource(denseArray)
sourceType = GetType(source)
originalLoop = GetLoop(traversal)
canBulkInit = false
if source = GetLinkedListExpression(traversal)
  canBulkInit = true
  foreach dataMember in arrayType
    if GetFillIn(dataMember) <> 0
      canBulkInit = false
    endif
  next
endif
Add(localVariables, "arrayType *sourceArray;")
Add(localvariables,
    "GetType(linkedListExpression) linkedListExpressionCopy;")
Add(initStatements, "linkedListExpressionCopy = linkedListExpression");
newGuard = false
if length = -1
  # unknown length means dynamic array is needed
  Add(initStatements, "sourceArray = InitDynamicArray(sizeof(arrayType));"
  Add(localVariables, "int linkedListeDenseLength;")
  Add(initStatements, "linkedListDenseLength = 0;")
  Add(initStatements, GetLoopExpression(originalLoop));
else
  Add(initStatements, "sourceArray = malloc(length * sizeof(arrayType));"
  if canBulkInit
    Add(initStatements, "memset(sourceArray, 0, length * sizeof(arrayType));"
    Add(initStatements, GetLoopExpression(originalLoop));
  else if LoopControlVariableNeeded(originalLoop)
    if IsSemiDense(originalLoop)
      Add(initStatements, GetLoopExpression(originalLoop))
    else
      # if the original loop is sparse, the original loop guard must be
      # used and bulk initialisation is not possible, we must abort
      return null
    endif
  else
    Add(initStatements,
        "for( sourceCounter = 0; sourceCounter < length; ++sourceCounter )")
    newGuard = true
  endif
endif
Add(initStatements, "{")
# GetDensePosition returns an expression that can be used to determine
# the dense array index; for a semi-dense loop or if the new loop guard
# is used, this is the counter, otherwise it is the expression specified
# by the DENSE_INDEX directive
densePos = GetDensePosition(traversal)
if length = -1
  # Insert statements to grow the dynamic array and initialise skipped
  # positions if necessary
  AddArrayGrowthStatements(initStatements, traversal, denseArray, densePos)
  Add(initStatements, "++linkedListDenseLength;")
endif
if not canBulkInit and source = GetLinkedListExpression(traversal)
  if arrayType is struct
```

```
      foreach dataMember in arrayType
        fillIn = GetFillIn(dataMember)
        if fillIn = null
          # set validity flag to false
          Add(initStatements, "sourceArray[densePos]->dataMemberValid = 0;")
        else
          Add(initStatements, "sourceArray[densePos]->dataMember = fillIn;")
        endif
      next
    else
      # if it is not a struct it must have a fill-in; if not there would have
      # been a flag so it would have been a struct
      fillIn = GetFillIn(dataMember)
      Add(initStatements, "sourceArray[densePos] = fillIn")
    endif
  endif
  if not IsSemiDense(originalLoop) and newGuard
    Add(initStatements,
        "if( linkedListExpression != NULL && densePos == sourceCounter ) {"
  endif
  foreach statement in FlattenStatements(originalLoop)
    # Get the category this statement has for this source expression
    categories = GetStatementCategories(statement, source)
    foreach category in categories
      switch category
        case 1, 2, 3
          if GetAccessType(statement, source) = Read
            # Get the actual expression for this access, e.g. node->Value.
            accessExpression = GetAccessExpression(statement, source)
            accessExpression = Replace(accessExpression,
                                       "source", "sourceCopy")
            if not CanMoveToInitLoop(accessExpression)
              abort
            endif
            if arrayType is struct
              member = GetMemberForAccess(statement, source)
              Add(initStatements,
                  "sourceArray[densePos]->member = accessExpression;")
              if HasFlag(member)
                # set validity flag to true
                Add(initStatements, "sourceArray[densePos]->memberValid = 1;"
              endif
            else
              Add(initStatements, "sourceArray[densePos] = accessExpression;")
            endif
          endif
        case 4, 5
          # It was already checked whether these can move to the init loop,
          # no need to do it again.
          Add(initStatements, Replace(statement, "linkedListExpression",
                                       "linkedListExpressionCopy"))
        case 6
          if CanMoveToInitLoop(statement)
            Add(initStatements, Replace(statement, "linkedListExpression",
                                       "linkedListExpressionCopy"))
          else
            abort
          endif
        case 7
```

```
          if not newGuard
            Add(initStatements, statement)
          endif
        case 8
          # no action
      endswitch
    next
  next
  if not IsSemiDense(originalLoop) and newGuard
    # end the if-statement
    Add(initStatements, "}")
  endif
  # end the loop
  Add(initStatements, "}")
  return [ initStatements, localVariables ]

function GeneratePreInitLoops(traversal, denseArrays)
  initStatements = {}
  localVariables = {}
  foreach denseArray in denseArrays
    if Contains(AccessType(denseArray), Read)
      [ init, vars ] = GeneratePreInitLoop(traversal, denseArray)
      if init = null
        return null
      endif
      AddRange(initStatements, init)
      # add variables eliminating duplicates (e.g. the linked list copy)
      AddRangeUnique(localVariables, localVariables)
    endif
  next
  return [ initStatements, localVariables ]
```

There are a few places in this code where it can return null. Here conditions 5 and 6 from section 3.4.3 are checked because the code to which these conditions apply has finally been identified. If the conditions fail, transformation is impossible after all so it is aborted.

3.4.6.3 Post-initialization

Post-initialization loops will always use the original loop guard. Its goal is to walk the linked list precisely as the original loop did, and it will never take any action for the omitted values (fill-in or validity checking is not relevant here), so using the new loop guard would serve no purpose.

Every value that is written to in a category one or three statement will cause a statement to be generated in the post-initialization loop of the following form:

```
original_value = array[index_expression];
```

For sublimation, `index_expression` is always the dense index expression indicated using a directive. For annihilation, an additional counter is added to the loop whose value is used here.

Category four, five, six, seven and eight statements are treated the same way as for pre-initialization loops.

The matrix multiplication example does not need post-initialization when doing either sublimation or annihilation, but examples of post initialization have already been shown in section 3.2.

This is the algorithm in pseudo code:

```
function GeneratePostInitLoop(traversal, denseArray)
```

```
  initStatements = {}
  localVariables = {}
  originalLoop = GetLoop(traversal)
  arrayType = GetType(denseArray)
  source = GetSource(denseArray)
  sourceType = GetType(source)
  Add(localVariables, "arrayType *sourceArray;")
  Add(localvariables, "sourceType linkedListExpressionCopy;")
  Add(initStatements, "linkedListExpressionCopy = linkedListExpression;");
  densePos = GetDensePosition(traversal)
  Add(initStatements, GetLoopExpression(originalLoop));
  Add(initStatements, "{")
  foreach statement in FlattenStatements(originalLoop)
    # Get the category this statement has for this source expression
    categories = GetStatementCategories(statement, source)
    foreach category in categories
      switch category
        case 1, 3
          if GetAccessType(statement, source) = Write
            # Get the actual expression for this access, e.g. node->Value.
            accessExpression = GetAccessExpression(statement, source)
            accessExpression = Replace(accessExpression, "linkedListExpression"
                                       , "linkedListExpressionCopy")
            if not CanMoveToInitLoop(accessExpression)
              return null
            endif
            if arrayType is struct
              member = GetMemberForAccess(statement, source)
              Add(initStatements,
                  "accessExpression = sourceArray[densePos]->member;")
            else
              Add(initStatements, "accessExpression = sourceArray[densePos];")
            endif
          endif
        case 4, 5
          # It was already checked whether these can move to the init loop,
          # no need to do it again.
          Add(initStatements, Replace(statement, "linkedListExpression",
                                                 "linkedListExpressionCopy"))
        case 6
          if CanMoveToInitLoop(statement)
            Add(initStatements, Replace(statement, "linkedListExpression",
                                                   "linkedListExpressionCopy"))
          else
            abort
          endif
        case 7
          Add(initStatements, statement)
        case 2, 8
          # no action
      endswitch
    next
  next
  Add(initStatements, "}")
  return [ initStatements, localVariables ]

function GeneratePostInitLoops(traversal, denseArrays)
  initStatements = {}
  localVariables = {}
```

```
  foreach denseArray in denseArrays
    if Contains(AccessType(denseArray), Write)
      [ init, vars ] = GeneratePostInitLoop(traversal, denseArray)
      if init = null
        return null
      endif
      AddRange(initStatements, init)
      AddRangeUnique(localVariables, localVariables)
    endif
  next
  return [ initStatements, localVariables ]
```

3.4.6.4 Main loop transformation

Now the main loop can be transformed. First, the original loop guard for this loop is replaced with the new loop guard. For category one, two and three statements, the values that were read or written are replaced with their respective array values. So any occurrence of `linked_list_expression->data_member` will be replaced with `replacement_array[counter]` for category one, any occurrence of `original_counter` is replaced with `replacement_array[counter]` for category two, and any occurrence of `original_array[original_counter]` is replaced with `replacement_array[counter]` for category three.

Category four and five statements are removed completely; they deal only with the linked list which is no longer used in the transformed loop, so they serve no more purpose.

Category six statements are removed if a fill-in value is known, and there are no other statements under this guard that would block this removal as indicated in section 3.4.4. If they cannot be removed, the guard expression is replaced with a check against the validity flags of all members used in the guarded statements.

Category seven statements are removed; they deal with flow control of the original loop so they are no longer needed.

Category eight statements are left unaltered.

If the transformed loop was a sparse loop and validity flags are used or there are any category 8 statements not guarded by a category 6 guard, the entire loop body is wrapped in a guard that checks the fill-in value or validity flag for all data members.

This leads to the following transformed inner loop for the matrix example when using sublimation:

```
for( x = 0; x < dimensions; ++x )
{
    result[row][col] += leftCellArray[x] * right[x][col];
}
```

Because this was a semi-dense loop, the original and new loop statements are the same. The reference to leftCell->Value has been replaced with a reference to leftCellArray[x] which was filled in the pre-initialization loop.

For annihilation, the transformed loop looks like this:

```
for( x = 0; x < newDimensions; ++x )
{
  result[row][col] += leftCellArray[x] * rightArray[x][col];
}
```

Here the loop guard is different: the new upper bound that was determined in the pre-initialization loops is used instead of the original. Again, references to leftCell->Value have been

replaced with a reference to leftCellArray[x], and this time references to right[x] have been replaced with rightArray[x], which is the condensed copy of right that was created to allow for the changed progression of the loop counter.

The pseudo code algorithm for this is as follows.

```
function GenerateGuardExpression(dataMembers, negateDataMembers, flagsOnly)
  newGuardExpression = ""
  first = true
  foreach dataMember in dataMembers
    if first
      first = false
    else
      newGuardExpression += " && "
    endif
    fillIn = GetFillIn(dataMember)
    if Contains(negateDataMembers, dataMember)
      newGuardExpression += "!("
    endif
    if GetType(linkedListExpressionArray) is struct
      if fillIn = null
        newGuardExpression +=
            "linkedListExpressionArray[counter]->dataMemberValid"
      else if not flagsOnly
        newGuardExpression +=
            "linkedListExpressionArray[counter]->dataMember == fillIn"
      endif
    else if not flagsOnly
      newGuardExpression += "linkedListExpressionArray[counter] == fillIn"
    endif
    if Contains(falseDataMembers, dataMember)
      newGuardExpression += ")"
    endif
  endif
  return newGuardExpression

function TransformMainLoop(traversal, dataMembers, denseArrays)
  transformedStatements = {}
  originalLoop = GetLoop(traversal)
  length = GetDenseLength(traversal)
  if length = -1
    # use computed length
    upperbound = "linkedListDenseLength";
  else
    upperbound = "length";
  endif
  Add(transformedStatements,
      "for( counter = 0; counter < upperbound; ++counter ) {"
  if ContainsCategory7Statements(GetBody(originalLoop))
    # additional guard needed
    guard = GenerateGuardExpression(dataMembers, {}, false)
    Add(transformedStatements, "if( guard ) {")
  endif
  foreach statement in FlattenStatements(originalLoop)
    # Get the category this statement has for this source expression
    categories = GetStatementCategories(statement, source)
    foreach category in categories
      switch category
        case 1, 2, 3
```

```
          # Get the actual expression for this access, e.g. node->Value.
          accessExpression = GetAccessExpression(statement)
          denseArray = GetDenseArrayFor(accessExpression, denseArrays)
          if arrayType is struct
            member = GetMemberForAccess(statement, accessExpression)
            newExpression = "denseArray[counter]->member"
          else
            newExpression = "denseArray[counter]"
          endif
          Add(transformedStatements,
              Replace(statement, accessExpression, newExpression)
        case 4, 5, 7
          # No action
        case 6
          # find data members used when the guard expression evaluates
          # true or false.
          trueDataMembers = DataMembersForCompound(GetTruePart(statement))
          falseDataMembers = DataMembersForCompound(GetFalsePart(statement))
          # iterate over those data members used in true or false,
          #but not in both.
          uniquelyGuardedMembers = (trueDataMembers union falseDataMembers -
                                    trueDataMembers intersect falseDataMembers)
          # If there are no cat. 7 statements guarded, only flags are
          # needed, not fill in checks.
          flagsOnly = Count(GetCategory7Statements(GetTruePart(statement) union
                            GetFalsePart(statement)) = 0
          newGuardExpression = GenerateGuardExpression(uniquelyGuardedMembers,
                                                       falseDataMembers, flagsOnly)
          if newGuardExpression <> ""
            Add(transformedStatements, "if( newGuardExpression )"
          endif
        case 8
          Add(transformedStatements, statement)
      endswitch
    next
  next
  if ContainsCategory7Statements(GetBody(originalLoop))
    # end the if statement
    Add(transformedStatements, "}"
  endif
  # end the loop
  Add(transformedStatements, "}"
  return transformedStatements
```

3.4.6.5 The transformation as a whole

To do a complete transformation of the loop, the generation of pre-initialization and post-initialization loops and transforming the main loop are combined leading to a set of statements from all three that constitute the result of the transformation. Besides just performing the operations from the previous three subsections, some additional steps may be necessary. If there is a post-initialization loop for an array that has no pre-initialization, an allocation statement for that array still needs to be created before the transformed main loop. If there are no pre-initialization loops and the dense length is unknown (for instance with annihilation), an empty initialization loop must be generated that determines the dense length.

The following function is used to perform the transformation for step six. Its output is a list of generated variables and a list of statements containing all pre-initialization, post-initialization, and transformed main loop statements.

```
function TransformLoop(traversal, dataMembers, denseArrays)
  linkedListExpression = GetLinkedListExpression(traversal)
  localVariables = {}
  preInitStatements = {}
  postInitStatements = {}
  transformedStatements = {}
  [ preInitStatements, vars ] = GeneratePreInitLoops(traversal, denseArrays)
  if preInitStatements = null
    return null
  endif
  AddRangeUnique(localVariables, vars)
  [ postInitStatements, vars ] = GeneratePostInitLoops(traversal, denseArrays)
  if postInitStatements = null
    return null
  endif
  AddRangeUnique(localVariables, vars)
  length = GetDenseLength(traversal)
  if length = -1 and preInitStatements is empty
    # need loop to determine length
    [ preInitStatements, vars ] = GeneratePreInitLoop(traversal, null)
    AddRangeUnique(localVariables, vars)
    length = "linkedListDenseLength"
  endif
  foreach denseArray in denseArrays
    accessTypes = GetAccessTypes(denseArray)
    if Contains(accessTypes, Write) and not Contains(accessTypes, Read)
      # this means the array is used in the transformed loop and post-init
      # loops but not the pre-init loops, so we need to allocate it seperately
      arrayType = GetType(denseArray)
      Add(preInitStatements, "denseArray = malloc(length * sizeof(arrayType))
    endif
  next
  # Add a statement to set the linked list variable to its final
  # value if needed
  if ReadAfterLoop(loop, linkedListExpression)
    Add(postInitStatements, "linkedListExpression = listListExpressionCopy")
  endif
  transformedStatements = TransformMainLoop(traversal, dataMembers,
                                            denseArrays)
  result = preInitStatements
  AddRange(result, transformedStatements)
  AddRange(result, postInitStatements)
  return [ result, localVariables ]
```

3.4.7 Moving the initialization loops

In order to further optimize the code, it can be beneficial to move the initialization loops away from the main loop. We can observe the following basic rules:

1. If a block of code does not write to any of the expressions read in a pre-initialization loop A, and does not read any expressions written to in A, then A can be moved ahead of the block. Conversely, if A is a post-initialization loop is can be moved after the block under the same conditions.
2. If an initialization loop A is contained in an outer loop O, and all variables read in A are invariant over O, then A can be moved out of O. This is called **trivial loop extraction**.

Whenever this section mentions "initialization loop", it means all statements such as assignment statements and variable copying statements associated with the actual loop as well.

As we saw in the previous section, a post-initialization loop for an array that has no corresponding pre-initialization loop will also create an additional allocation statement before the main array. This statement can be moved in precisely the same way as the pre-initialization loops.

There are two situations where, even if the conditions for the second rule are not met, we can still move an initialization loop out of a containing loop.

The first situation is when the expressions that are not root invariant all have the same root non-invariant expression (see section 3.4.3) is the counter of the outer loop. Consider the following example, which performs the algorithm from one of the examples in section 3.1 on a number of linked lists instead of just one.

```
int product[M];
int x;
for( x = 0; x < M; ++x )
{
  product[x] = 1;
  /***DENSE_INDEX(node[x], node[x]->Index)***/
  /***DENSE_DIMENSION(node[x], 10)***/
  while( node[x] != NULL )
  {
    product[x] *= node[x]->Value;
    node[x] = node[x]->Next;
  }
}
```

In this example, node[x] is the linked list expression, leading to the following code after initial transformation:

```
int product[M];
int *nodeArray;
struct Node *nodeCopy;
int x, y;
for( x = 0; x < M; ++x )
{
  product[x] = 1;

  // pre-initialisation
  nodeArray = malloc(10 * sizeof(int));
  nodeCopy = node[x];
  for( y = 0; y < 10; ++y )
  {
    if( nodeCopy != NULL && nodeCopy->index == x )
    {
      nodeArray[y] = nodeCopy->Value;
      nodeCopy = nodeCopy->Next;
    }
    else
      nodeArray[y] = 1; // Fill-in value
  }

  // Transformed main loop
  for( y = 0; y < 10; ++y )
  {
    product[y] *= nodeArray[y];
  }
  node[x] = nodeCopy;
}
```

The pre-initialization loop (which as was said earlier includes the two statements before the actual loop) in this sample is not invariant over the outer loop, because it uses node[x], which depends on x, which is not loop invariant. The root non-invariant expression for node[x] is x, and because x is the counter for the outer loop, we can still move the loop by duplicating the outer loop structure and adding a dimension to nodeArray that represents the outer loop iterations. This leads to the following code after moving the loop.

```
int product[M];
int **nodeArrayArray;
struct Node **nodeArray2;
struct Node *nodeCopy;
int x, y;
// pre-initialisation
nodeArrayArray = malloc(M * sizeof(int*));
nodeArray2 = malloc(M * sizeof(struct Node*));
for( x = 0; x < M; ++x )
{
  nodeArrayArray[x] = malloc(10 * sizeof(int));
  nodeCopy = node[x];
  for( y = 0; y < 10; ++y )
  {
    if( nodeCopy != NULL && nodeCopy->index == x )
    {
      nodeArrayArray[x][y] = nodeCopy->Value;
      nodeCopy = nodeCopy->Next;
    }
    else
      nodeArrayArray[x][y] = 1; // Fill-in value
  }
  nodeArray2[x] = nodeCopy;
}

for( x = 0; x < M; ++x )
{
  product[x] = 1;

  // Transformed main loop
  for( y = 0; y < 10; ++y )
  {
    product[y] *= nodeArrayArray[x][y];
  }
  node[x] = nodeArray2[x];
}
```

The procedure for this is simple: create a new loop before (or after with a post-initialization loop) the outer loop that uses the same loop guard as the outer loop. Add a dimension to the array used in the initialization loop. Move the initialization loop into the new loop, modifying all references to the array to include this new dimension. Then modify all references to this array in the transformed main loop as well. This is called **simple loop extraction**.

In addition, an array is created to hold the final values of nodeCopy so that these can be assigned to node[x] after the transformed loop. If it can be determined that node[x] is not read after the loop this may be omitted.

For a post-initialization loop, a statement must be added before the outer loop that allocates the array.

While it may seem at first a bad idea to extract the loop like this – after all, it increases the amount of work because it introduces an extra loop – it may lead to additional optimization opportunities for the main loop making it beneficial in the long run.

The second situation in which we can still extract the loop is if the root non-invariant expression is itself a linked list candidate for the outer loop. In this case we can start a process called **linked list loop extraction**.

For this process, all the previous steps are performed to transform the outer loop exactly as before. However, in addition to the normal creation of an array for data members, an additional dimension is added to the array for the initialization loop we are moving, allowing us to move the loop into the outer loop is initialization loop in a similar fashion as above.

This situation occurs in the matrix multiplication example. In the previous section, we saw the initialization loop and transformed main loop for the innermost loop when performing sublimation. If we put those loops into context in the middle loop, we get the following (the outermost loop has been omitted):

```
for( row = 0; row < dimensions; ++row )
{
  if( leftRow != NULL && leftRow->RowIndex < row )
    leftRow = leftRow->Next;

  if( leftRow != NULL && leftRow->RowIndex == row )
  {
    leftCell = leftRow->Cell;

    leftCellArray = malloc(sizeof(float) * dimensions);
    memset(leftCellArray, 0, sizeof(float) * dimensions);
    leftCellCopy = leftCell;
    for( x = 0; x < dimensions; ++x )
    {
      if( leftCellCopy != NULL && leftCellCopy->ColIndex < x )
        leftCellCopy = leftCellCopy->ColNext;

      if( leftCellCopy != NULL &&
          leftCellCopy->ColIndex == x &&
          leftCellCopy->RowIndex == row )
      {
        leftCellArray[x] = leftCellCopy->Value;
      }
    }

    for( x = 0; x < dimensions; ++x )
    {
      result[row][col] += leftCellArray[x] * right[x][col];
    }

    free(leftCellArray);
  }
}
```

Note that since leftCell is a local variable that is not read after the loops it is not necessary to set it to its final value after the transformed main loop.

In this case, the initialization loop we want to move uses one value that is not invariant over the outer loop, namely leftCellCopy, which depends on leftCell, which depends on leftRow->Cell which depends on leftRow which is not invariant. The variable leftRow is a pointer to a RowHead structure, and step one had identified RowHead as a potential linked list candidate.

The algorithm will now check to see if this loop is a candidate linked list traversal for leftRow, checking the list returned by step two. This is indeed the case here, so we proceed with evaluating the conditions from step three, which are all met. Step four will identify one data member for this loop, namely Cell. Since data dependence analysis shows that the value of Cell is used only in the initialization loop, it is not necessary to create a separate array for it. Since the final value of leftCellCopy is also not used, no array for that is needed either. So step five does nothing, and the algorithm for step seven will extend leftCellArray with an extra dimension. Step six will perform transformation as usual, and afterwards the initialization loop for leftCell will be moved into a new initialization loop created for leftRow, adjusting it accordingly. Since there are no data members for leftRow outside of the leftCell initialization loop, the fill-in value for leftCell is assumed to apply here, which means the second if-statement in the loop (which is category six) can be removed from the transformed loop. Creation and initialization of the array for leftCell is done outside that if-statement in the leftRow initialization loop because it also needs to be done for the fill-in case. Note that if direct initialization of the fill-in value using memset as is done here is not possible, an else-clause must be added to this if-statement that fills the array with fill-in values.

This leads to the following code after transformation:

```
leftCellArrayArray = malloc(sizeof(float*) * dimensions);

leftRowCopy = leftRow;
for( row = 0; row < dimensions; ++row )
{
  if( leftRowCopy != NULL && leftRowCopy->RowIndex < row )
    leftRowCopy = leftRowCopy->Next;

  leftCellArrayArray[row] = malloc(sizeof(float) * dimensions);
  memset(leftCellArrayArray[row], 0, sizeof(float) * dimensions);

  if( leftRowCopy != NULL && leftRowCopy->RowIndex == row )
  {
    leftCell = leftRowCopy->Cell;

    leftCellCopy = leftCell;
    for( x = 0; x < dimensions; ++x )
    {
      if( leftCellCopy != NULL && leftCellCopy->ColIndex < x )
        leftCellCopy = leftCellCopy->ColNext;

      if( leftCellCopy != NULL &&
          leftCellCopy->ColIndex == x &&
          leftCellCopy->RowIndex == row )
      {
        leftCellArrayArray[row][x] = leftCellCopy->Value;
      }
    }
  }
}

for( row = 0; row < dimensions; ++row )
{
  for( x = 0; x < dimensions; ++x )
  {
    result[row][col] += leftCellArrayArray[row][x] * right[x][col];
  }
```

```
}

// Cleanup
for( row = 0; row < dimensions; ++row )
  free(leftCellArrayArray[row]);
free(leftCellArrayArray);
```

In this case it is immediately clear what the benefit of doing this is: although we have increased the overhead of initialization by adding an extra loop, the transformed loop now looks like a normal, dense matrix multiplication algorithm which allows further optimizations.

Note that when doing annihilation we can perform this step using annihilation as well in exactly the same manner as described in the previous sections.

To complete the transformation of the matrix multiplication example with sublimation, we also want to move this leftRow initialization loop out of the outermost loop. Since it is completely invariant, it can be moved out of the loop using trivial loop extraction. Note that the cleanup code will be moved out of the loop as well. This leads to the code in Appendix A. There you can also find the completely transformed code when using annihilation.

Step seven can be represented using the following pseudo code.

```
function GetLoopIterationCount(loop)
  loopControlStatements = GetLoopControlStatements(loop, null)
  # returns -1 if there is no directive
  length = GetLengthFromDirective(loop)
  if length = -1
    # these are the conditions under which the upper bound can be determined
    # and thus the needed array size
    if Count(loopControlStatements) = 1 and
       loopControlStatements[0] like "x = x + 1" and
       GetLoopCondition(loop) like "x < upperBound"
      length = upperBound
    endif
  endif
  return length

function AddDimension(declaration, length)
  type = GetType(declaration)
  name = GetName(declaration)
  newDeclaration = "type *nameArray;"
  allocation = "nameArray = malloc(length * sizeof(type*));"
  return [ newDeclaration, allocation ]

# Loop extraction for when the non-invariant value is the counter.
# As mentioned, initLoop includes the malloc statement for the loop,
# and can also be just a malloc statement.
function DoSimpleLoopExtraction(outerLoop, initLoops, localVariables)
  preInitStatements = {}
  postInitStatements = {}
  transformedStatements = {}
  if IsLoopControlTrivial(outerLoop)
    length = GetLoopIterationCount(outerLoop)
    if length = -1
      name = GenerateUniqueName()
      # add a pre-init loop to count the length
      Add(localVariables, "int nameArrayLength;")
      Add(preInitStatements, "nameArrayLength = 0;")
      Add(preInitStatements, GetLoopGuard(outerLoop))
      Add(preInitStatements, "{ ++nameArrayLength; ")
```

```
      AddRange(preInitStatements, loopControlStatements)
      Add(preInitStatements, "}")
      length = "nameArrayLength"
    endif
    foreach initLoop in initLoops
      RemoveStatements(outerLoop, initLoop)
      # get the control variable from the outer loop that the init loop is
      # using
      controlVariable = GetDependantControlVariable(initLoop)
      # get the declaration statement for the array used by this init loop
      initVariables = GetInitVariables(initLoop)
      newNames = {}
      foreach variable in initVariables
        declaration = GetDeclaration(localVariables, variable)
        # check if the declaration is already processed for this loop; this
        # can happen e.g. if an pre- and post-init loop share an array.
        if not DeclarationIsAlreadyProcessed(declaration, outerLoop)
          # the original declaration in localVariables is replaced
          [ declaration, allocation ] = AddDimension(declaration, length)
          Add(preInitStatements, allocation)
        endif
        Add(newNames, GetName(arrayDeclaration))
      Endif
      if IsPostInit(initLoop)
        initStatements = postInitStatements
      else
        initStatements = preInitStatements
      endif
      Add(initStatements, GetLoopGuard(outerLoop))
      # swap in the array for all statements of the init loop
      foreach name in newNames
        originalName = name - "Array"
        initLoop = Replace(initLoop, originalName, "name[controlVariable]")
        outerLoop = Replace(outerLoop, originalName, "name[controlVariable]")
      endif
      Add(initStatements, initLoop)
      Add(initStatements, "}")
      # remove the init loop statements from the outer loop
      RemoveRange(outerLoop, initLoop)
    next
    result = preInitStatements
    AddRange(result, outerLoop)
    AddRange(result, postInitStatements)
    return result
  else
    return null
  endif

# generates an init loop for the traversal in outerloop placing the
# specified statements in the loop at their proper relative position
# or at the end if they are new statements. InitStatements typically
# is an init loop that is being extracted which means these statements
# are not actually in outerLoop anymore since they were removed earlier.
function GenerateExtractedLoop(outerLoop, initLoop, length, newArrayNames)
  result = {}
  Add(result, GetLoopGuard(outerLoop))
  traversal = GetTraversal(outerLoop)
  densePos = GetDenseIndex(traversal)
  allocation = GetAllocationStatements(initLoop)
```

```
  AddRange(result, allocation)
  # Generate the memset statement or the loop needed to set
  # the entire array to the fill-in value. This is needed because
  # the init loop we are moving might be guarded so cannot be guaranteed
  # to take care of the fill-in for us.
  GenerateFillInStatements(result, initLoop)
  foreach statement in FlattenStatements(GetBody(outerLoop))
    # only loop control and linked list iteration statements are needed
    switch GetCategoryForLoopExtraction(statement, initLoop)
      # cat 1 is the init loop statements to be moved, with the exclusion
      # of statements to allocate and fill-in initialisation of the array,
      # the rest is the same as for regular init loop generation
      case 1, 4, 5, 7
        Add(result, statement)
    endswitch
  next
  return result

# outerLoop is completely original and untransformed
# transformedOuterLoop is the outer loop after regular linked list
# transformation; the init loops have already been removed. It is passed
# to this function so the new array names can be swapped in.
function DoLinkedListLoopExtraction(outerLoop, initLoops, localVariables,
                                    transformedLoop)
  preInitStatements = {}
  postInitStatements = {}
  length = GetLoopIterationCount(outerLoop)
  if length = -1
    name = GenerateUniqueName()
    # add a pre-init loop to count the length
    Add(localVariables, "int nameArrayLength;")
    Add(preInitStatements, "nameArrayLength = 0;")
    initStatements = { "++nameArrayLength;" )
    # create empty loop purely for the counter
    loop = GenerateExtractedLoop(outerLoop, initStatements, -1, null)
    Add(preInitStatements, loop)
    length = "nameArrayLength"
  endif
  foreach initLoop in initLoops
    # get the variables that this loop is initialising
    initVariables = GetInitVariables(initLoop)
    newNames = {}
    foreach variable in initVariables
      declaration = GetDeclaration(localVariables, variable)
      # check if the declaration is already processed for this loop; this
      # can happen e.g. if an pre- and post-init loop share an array.
      if not DeclarationIsAlreadyProcessed(declaration, outerLoop)
        [ arrayDeclaration, allocation ] = AddDimension(declaration, length)
        Add(preInitStatements, allocation)
      endif
      Add(newNames, GetName(arrayDeclaration))
    endif
    if IsPostInit(initLoop)
      initStatements = postInitStatements
    else
      initStatements = preInitStatements
    endif
    traversal = GetTraversal(outerLoop)
    densePos = GetDenseIndex(traversal)
```

```
    foreach newArrayName in newNames
      originalName = newArrayName - "Array"
      initLoop = Replace(initLoop, originalName, "newArrayName[densePos]")
      # update the transformed loop as well
      transformedOuterLoop = Replace(transformedOuterLoop, originalName,
                                     "newArrayName[densePos]")
    next
    loop = GenerateExtractedLoop(outerLoop, initLoop, length, newNames)
    AddRange(initStatements, loop)
  next
  return [ preInitStatements, postInitStatements ]

function ExtractInitLoops(outerLoop, candidateTraversals, localVariables)
  # get all pre- and post-init loops and associated code
  # in the loop
  initLoops = GetInitLoops(outerLoop)
  candidateTraversal = null
  loopExtractionCandidates = {}
  simpleExtractionCandidates = {}
  trivialExtractionCandidates = {}
  # unmoveableLoops is used only for simple extractions; a loop that ends up
  # in this collection because it uses more than one control variable might
  # still be moveable by linked list loop extraction.
  unmoveableLoops = {}
  foreach initLoop in initLoops
    # get the variables that this init loop is initialising
    initVariables = GetInitVariables(initLoop)
    nonInvariantVariableCount = 0
    foreach variable in initLoop
      if not (Contains(initVariables) or IsLoopInvariant(outerLoop, variable))
        nonInvariantVariableCount += 1
        rootExpression = GetRootNonInvariantExpression(variable)
        # see if the outerloop has a candidate traversal for the type of this
        # variable, the member and statement do not matter
        traversal = GetItem(candidateTraversals,
                            [ outerLoop, [ GetType(rootExpression), * ], * ])
        # the outer loop may have more than one traversal; we consider only
        # one.
        if traversal <> null and candidateTraversal = null or
           traversal = candidateTraversal
          traversal = candidateTraversal
          Add(loopExtractionCandidates, initLoop)
        else
          if not Contains(unmoveableLoops, initLoop)
            foreach controlVariable in GetLoopCondition(outerLoop)
              if rootExpression = controlVariable
                if Contains(simpleExtractionCandidates, [ initLoop, * ])
                  # simpe extraction is impossible, linked list loop extraction
                  # may still be possible
                  Add(unmoveableLoops, initLoop)
                  Remove(simpleExtractionCandidates, initLoop)
                else
                  Add(simpleExtractionCandidates, [initLoop, controlVariable])
                endif
              else
                # loop extraction is guaranteed impossible
                Add(unmoveableLoops, initLoop)
                Remove(simpleExtractionCandidates, initLoop)
                Remove(loopExtractionCandidates, initLoop)
```

```
            endif
          next
        endif
      next
      if nonInvariantVariableCount = 0
        Add(trivialExtractionCandidates, initLoop)
      endif
    next
    # now all init loops that can do simple loop extraction or linked list loop
    # extraction are known. Only one type of extraction is performed, then true
    # is returned so that the caller knows changes were made and it can
    # check the loop again
    if Count(loopExtractionCandidates) > 0
      return TransformTraversal(candidateTraversal, loopExtractionCandidates,
    else if Count(simpleLoopExtractionCandidates) > 0
      outerLoop = DoSimpleLoopExtraction(outerLoop,
                                         simpleLoopExtractionCandidates,
                                         localVariables)
      return true
    else if Count(trivialLoopExtractionCandidates) > 0
      return DoTrivialLoopExtraction(outerLoop, trivialLoopExtractionCandidates)
    endif
    return false
```

For simplicity, the code for trivial loop extraction and loop movement (when the first condition from the beginning of this section is met) is not provided, since these cases are both trivial to implement.

The main function for this step is ExtractInitLoops. It finds all init loops that can be extracted using linked list loop extraction, simple loop extraction or trivial loop extraction. It will then perform one of those transformations, on all applicable initialization loops, and return. The caller can use the return value to determine if changes were made. If so, ExtractInitLoops can be called again to analyze the modified situation and see if more loops can be extracted using perhaps a different technique.

Linked list loop extraction is done by calling on the main transformation since it is a linked list transformation just like we have done before only with some extra steps. This main transformation function is given in the next section, which includes some of the code for this step.

3.4.8 Putting it all together

In the previous sections, pseudo code is given for the individual steps, defining functions that perform the steps and return the results given by that step. What is left then is a function that calls these functions in the correct order so it performs the entire transformation. This is given below.

```
function TransformTranslationUnit(translationUnit)
  candidateStructs = FindCandidateStructs(translationUnit)
  foreach function in translationUnit
    do
      candidateTraversals = FindCandidateTraversals(function, candidateStructs)
      # get the candidate with the highest nesting level
      if Count(candidateTraversals) > 0
        candidateTraversal = GetMostNestedCandidate(candidateTraversals)
        if not TransformTraversal(candidateTraversal)
          # transformation was unsuccessful, make sure not to try it again
          MarkProcessed(candidateTraversal)
```

```
        endif
      endif
    loop while Count(candidateTraversals) > 0
    # do the simple movement for loops matching the first condition
    # for init loop movement.
    foreach initLoop in function
      MoveInitLoop(function, initLoop)
    next
  next

function TransformTraversal(traversal, initLoopsToMove, candidateTraversals)
  if EvaluateCandidateTraversal(traversal)
    loop = GetLoop(traversal);
    dataMembers = FindDataMembers(traversal)
    foreach dataMember in dataMembers
      # remove those data members that are only used in the init loops
      # we are going to be moving by linked list loop extraction.
      if DataMemberOnlyUsedInInitLoops(dataMember, initLoopsToMove)
        Remove(dataMembers, dataMember)
      endif
    next
    if Count(dataMembers) = 0 and Count(initLoopsToMove) = 0
      # nothing to do
      return false
    endif
    denseArrays = GenerateDenseDataStructures(traversal, dataMembers)
    function = GetFunction(traversal)
    [ result, vars ] = TransformLoop(traversal, dataMembers, denseArrays)
    if result = null
      return false
    endif
    AddDeclarations(function, vars)
    # perform linked list loop extraction
    if initLoopsToMove <> null
      [ pre, post ] = DoLinkedListLoopExtraction(loop, initLoopsToMove,
                                          GetDeclarations(function), result)
    endif
    transformed = Concat(pre, result, post)
    ReplaceStatements(function, loop, transformed)
    # See if the loop was contained inside another loop
    do
      madeChanges = false
      outerLoop = GetContainingLoop(loop)
      if outerLoop <> null
        localVariables = GetDeclarations(function)
        madeChanges = ExtractInitLoops(outerLoop, candidateTraversals,
                                       localVariables)
      endif
    loop while madeChanges
    return true
  endif
  return false
```

TransformTranslationUnit is the main function driving the transformation of a single translation unit (a pre-processed C code file). It will find the candidate traversals using steps one and two, and then call TransformTraversal which performs the remaining steps. Step seven, as was noted in the previous section, can recursively call TransformTraversal to perform linked list loop extraction on a containing loop.

As a final step some cleanup can be performed on the code. The transformation can leave statements in the code that have no effect anymore after transformation; these can be found using data dependence analysis and be safely removed.

3.5 Normalization

As indicated earlier, before any of the transformation steps above take place, the code needs to be normalized so that it can be reliably transformed and analyzed. Normalization consists of several parts which are covered in the following sections. Since the separate normalization techniques can affect other techniques it may be necessary to repeat normalization until no further changes occur in the code.

Most of the examples in this thesis have not had full normalization applied to ease readability.

3.5.1 Aliasing

When trying to do any kind of analysis of C code, pointer aliasing forms a major obstacle in reasoning about it. Pointer analysis has been the subject of intense study in the past decades, and there are a great number of different approaches. These approaches can be categorized by flow-sensitivity and context-sensitivity. A flow-insensitive algorithm ignores the order of statements when it calculates pointer information, whereas a flow-sensitive algorithm takes control flow within a procedure into account. A context-insensitive algorithm does not distinguish the different calling contexts of a procedure, whereas a context-sensitive does. Hind and Pioli do a comparative analysis of several pointer analysis methods in [9], and Hind examines some of the remaining problems in [10]. An introduction to possible applications of pointer analysis is given in [5].

Especially context-sensitive pointer analysis, which is required to solve the global aliasing problem, can be extremely time-consuming and generally grow exponentially with the size of the program, a problem which is compounded in the presence of function pointers [11]. In [12; 13] Zhu and Calman give an approach that can efficiently do a whole program pointer analysis.

Although global aliasing is an extremely hard problem, how to deal with aliasing when it involves only the scope of a single function is relatively straight-forward and the normalization step can take some steps to easy data dependence analysis, which are outlined below.

A variable is an alias for another variable if modifications of either the original or the alias would affect the other. This means that an alias must necessarily always be a pointer (except when pointers are cast to numerical values). Aliases can be created when a pointer to an existing variable is assigned to a variable, or when an existing pointer is copied to a new variable. By looking for assignments of a variable that has a pointer type we can be recognize where aliases are created. In the case of using a higher level of indirection, e.g. `x = &y`, any change in y will affect x and any change in x where x is dereferenced at least once will affect y. In the case of the same level of indirection, e.g. `x = y`, any change in either x or y where they are dereferenced affects the other.

In either case, we will normalize all uses of x and the aliased value &y or just y to use the identical expression, up to the point where x itself (not dereferenced) is modified again. For the same level of indirection, a change in y will also break the aliasing and thus be the end of the replacements, except if y is changed to a value that also depends on y (for instance, y = y + 1), we should try to re-express x in the new value of y.

A special case is the situation where a variable is assigned the result of a pointer manipulation, e.g. `x = y + 5` if x and y are both pointers. Although you are not directly aliasing y, there is a case of aliasing here because it is now possible to access the memory location y+5 in

two ways. In other words, x has become equivalent to saying y+5. Therefore, we will also normalize these occurrences. In this case, if y is modified, the same rules apply as above, so we must try to re-express the value if the new value of y depends on y, or break the relationship if it is changed independently.

The left-hand side of the assignment need not be a variable, but can be any l-value that has a suitable pointer type. For instance in the situation that `*x = &y`, we must consider *x an alias for y, and treat it as such. In this case, changing x as well as *x will break the relation.

In C it is unfortunately possible to cast a pointer to an integer, and then later back to a pointer, possibly after manipulating it. If such a cast occurs, we cannot safely determine aliases and must assume safety if we are in a SAFE_CODE region.

As can be clearly seen, aliasing is a very complex problem, and solving it in its entirety is beyond the scope of this thesis, especially considering that if you want to prove that the transformation is safe, you must also prove that no corner cases exist in which a potentially unsafe alias is undetected and left in the code. To this end we will always assume that any code marked safe using directives simply does not use any unsafe aliasing to begin with, and if it does transformation would either fail or lead to semantically incorrect results.

Nevertheless, in the future it will be worthwhile to investigate how this transformation can be improved by applying some of the more powerful pointer analysis methods. Some of the transformation steps, such as locating the linked list iteration statement in Section 3.4.2, which are expressed in terms of syntax in this thesis, can possibly be expressed in terms of pointer dependencies. Additionally, context-sensitive pointer analysis can allow us to actually verify some of the conditions for SAFE_CODE regions, and would also allow us to deal with function calls more effectively.

3.5.2 Loop structure normalization

In order to make processing loops simpler, for-loops will be transformed into while-loops. A for loop has the following structure:

```
for( initialisation; condition; loop-expression )
{
  /* … */
}
```

This can be transformed to a while-loop with the following structure:

```
initialisation;
while( condition )
{
  /* … */
  loop-expression;
}
```

Again it must be noted that this has not been done in the examples in this thesis for readability.

Do-loops and existing while-loops will be left unchanged.

3.5.3 Expression normalization

As indicated in sections 3.4.2 and 3.4.3, it is important to be able to consider semantic equivalence of expressions in order to correctly determine whether values are used in a safe way. By normalizing expressions used in the code, equivalent expressions take the same form so they can be more easily detected.

The following is a list of normalizing steps that could be taken (note that this list is probably not exhaustive):

- Remove any unnecessary parentheses, e.g. `(x)` becomes `x` if the parenthesis do not affect evaluation order.
- Pre-compute the value of any constant expressions, e.g. `5+7` becomes `12`. The values of constant identifiers (variables declared as const) will be substituted as well.
- Remove any zero-effect operations. This includes unnecessary referencing/dereferencing (e.g. `&*x` becomes `x`), addition of zero (e.g. `x+0` becomes `x`), multiplication or division by one (e.g. `x*1` and `x/1` become `x`)
- Transform left and right bit-shift operations into multiplications and divisions respectively, e.g. `x<<1` becomes `x*2`.
- Pointer arithmetic followed by dereferencing is transformed into array indexing, e.g. `*(pointer + x)` becomes `pointer[x]`.
- Distribute any distributive operators.
- Normalize the operand order of commutative operators. For array indexing, we make sure the pointer variable comes first (e.g. `x[pointer]` becomes `pointer[x]`) and for all other commutative operators we sort the operands alphabetically, e.g. `y+x` becomes `x+y`.
- Expand shorthand operators, e.g. `++x` becomes `x = x+1`, and `x *= 2` becomes `x = x * 2`. The special case for `x++` can be solved by introducing temporaries.
- Extract operations with side effects into separate statements, e.g. `if( (x=x+1) > y )` becomes `x=x+1; if( x > y )`
- Etc.

The steps in this list are repeated until no more changes are possible.

There are some more esoteric expressions that are actually equivalent that cannot be reliably detected. You can, for instance, access a struct member using pointer arithmetic. We must once again assume that this is not done in sections marked as safe.

Some of these modifications can have performance implications (for instance swapping shift operations with multiplications). In that case the original form can be stored and put back in after the transformation has completed.

3.6 Transformation directives

In several of the preceding sections we have made mention of transformation directives, and we have also seen some examples of them.

Transformation directives serve to fill in the gaps where the transformation cannot discover the required information automatically by analyzing the code. Because the directives often apply to only a small section of the code they must be part of the source code itself, and cannot be replaced with e.g. command-line parameters for the compiler. A hypothetical implementation of this transformation will look for these directives in the source and use them accordingly.

Unlike some programming languages (such as C#), C does not have built-in support for attributes or directives. Instead, we will use specially formatted comments to represent them. Comments are well suited to this because they do not interfere with the ability of a regular compiler to process the code. Alternative approaches such as #pragma directives might clash with those used by another compiler and may require conditional compilation to ensure cross-platform functioning; comments do not have these drawbacks.

A transformation directive will be any text between /*** and ***/. The directive is case-sensitive, and any white space within the directive comment is insignificant, unless it is within a

string literal that serves as an argument for the directive. If the text between those delimiters cannot be understood as a directive, it must be assumed it is a normal comment and be ignored.

Below is a list of some of the directives that would be used by our transformation. This is not meant to be a complete list.

SAFE_CODE

> By default, the transformation assumes all code is unsafe to transform unless explicitly marked safe. All code following this directive is considered to be safe, until an UNSAFE_CODE directive is encountered. Code between a SAFE_CODE and UNSAFE_CODE directive is called a safe code section. A safe code section is assumed not to violate any of the rules that cannot be automatically checked; it is still checked for violations of the other rules.

UNSAFE_CODE

> This directive indicates that all code following this directive is unsafe, until a SAFE_CODE directive is encountered. Code between a SAFE_CODE and UNSAFE_CODE directive is called a safe code section.

SAFE_LOOP

> This indicates that the loop directly following the directive may be transformed, even if it is not in a safe code section. If the loop contains nested loops, they are *not* considered safe as well.

UNSAFE_LOOP

> This indicates that the loop directly following the directive may not be transformed, even if it is in a safe code section. This directive does not affect loops nested in the loop it applies to.

DENSE_INDEX(linked_list_expression, index_expression)

> This applies to the loop directly following it and the indicated *linked_list_expression.* It indicates that *index_expression* can be used inside the loop body to retrieve the original dense index. Both parameters can be any valid C expression.

DENSE_DIMENSIONS(linked_list_expression, dimension_expression)

> This directive applies to the loop directly following it and the indicated *linked_list_expressions.* It indicates that *dimension_expression* can be used to determine the dimensions of the original dense data.

FILL_IN(linked_list_expression[, fill_in_value])

> This directive applies to the loop directly following it and the indicated *linked_list_expressions.* It indicates the fill-in value to use for those values that have been omitted when generating the dense data structure. If the *fill_in_value* parameter is omitted, the transformation assumes there is no fill in value and must use a separate validity check. If this directive is not present, the transformation will attempt to determine a fill in value by itself.

In all cases, it is up to the programmer to ensure that the information specified by the directives is actually true. Although a section of code that is marked safe is still subject to the

checks indicated in section 3.4, the transformation will make no effort to verify whether things such as aliasing or pointer arithmetic are not present or safe if they are present. Similarly, the transformation will make no effort to determine if the index expression actually returns the index, if it is safe to be used as such (e.g. if it has side-effects) or even if it has the right type. Nor will it do this for the fill in value, or anything else. If the programmer chooses to lie to the compiler with these directives, transformation will succeed but likely yield incorrect results.

4 Experimentation

In the previous section, we have described the linked list transformation process, and applied it to the matrix multiplication example from section 2. Appendix A contains the complete result of that transformation for both sublimation and annihilation. Here, all possible traversals have been transformed and the initialization loops have been moved as much as possible. This has left us with a very clean main loop that looks exactly like a normal dense matrix multiplication.

As indicated in the introduction, we will evaluate the results of this transformation by generating new, optimized sparse code using some sample matrices from the Harwell Boeing collection.

The sparse compiler MT1 will be used to generate this new sparse code. This will be done for the sublimation results only; the resulting data structures of the annihilation process are no longer sparse so there is little MT1 could do with it, and the array structures do not lend themselves to translation to FORTRAN.

Almost everything in this section is specific to the use of matrices. The linked list transformation algorithm itself is not limited to matrices, but can be used on any linked list structure.

4.1 Translation into FORTRAN

The next thing to do is translate the code into FORTRAN so it can be transformed into sparse code again by the MT1 compiler.

Of course, not all C code can be easily translated into FORTRAN. For this translation to work there may be no more pointers (besides those that are actually arrays) left in the code that must be translated, and no structures. Some of this can be worked around; structures can be split into separate variables and arrays of structures into separate arrays.

Fortunately, the transformed main loop of the matrix multiplication sample contains no pointers and no structures, so it can be transformed without issue.

Translation into FORTRAN is done using an automated process. This C to FORTRAN conversion program was developed for this research, and can handle only a very limited subset of C; it can handle exactly those parts of C that are used in the loops that need to be translated.

In order to feed this translation process, the transformed main loop is extracted into a separate function, and some additional directives are added. For the loop that results from sublimation, this looks like this:

```
/***ARRAY_BOUNDS(result,dimensions,dimensions)***/
/***ARRAY_BOUNDS(leftCellArrayArray,dimensions,dimensions)***/
/***ARRAY_BOUNDS(right,dimensions,dimensions)***/
void Mult(int dimensions, float **result, float **leftCellArrayArray,
          float **right)
{
4  int col;
  int row;
```

```
  int x;

  for( col = 0; col < dimensions; ++col )
  {
    for( row = 0; row < dimensions; ++row )
    {
      for( x = 0; x < dimensions; ++x )
      {
        result[row][col] = result[row][col] +
                           leftCellArrayArray[row][x] * right[x][col];
      }
    }
  }
}
```

The directives used here are not listed in the previous section because they are not part of the linked list transformation process, but belong to the C to FORTRAN translation specifically. The ARRAY_BOUNDS directive specifies that a certain pointer variable is in fact an array, and specifies the expression to use for the size of each of its dimensions.

The resulting FORTRAN code looks like this:

```
      subroutine Mult(dimensions, result, leftCellArrayArray, right)
      integer dimensions
      real result(dimensions,dimensions)
      real leftCellArrayArray(dimensions,dimensions)
      real right(dimensions,dimensions)
      integer col
      integer row
      integer x
      do col = 1, dimensions
        do row = 1, dimensions
          do x = 1, dimensions
            result(row,col) = result(row,col) +
     +      leftCellArrayArray(row,x) * right(x,col)
          end do
        end do
      end do
      end
```

It is worth noting that the original loops in the C code ran from 0 to dimensions-1, while these loops run from 1 to dimensions. Because FORTRAN arrays are one-based (whereas C arrays are zero-based) and the row, col and x variables are used *only* in an array index expression, this is safe.

The code above is not precisely the code we will use. We will in fact use a slight variation of the algorithm. So far, we have assumed both matrices are square and of the same size. Because the right hand side and result matrices are not sparse need to be stored in memory completely this becomes prohibitive for large matrices. Instead, the algorithm we will use multiplies the sparse matrix with one that is tall and narrow. This does not change much for the algorithm; all that needs to be changed is the upper bound for the "col" loop, everything else remains the same.

4.2 Using the sparse compiler

The subroutine above cannot be input in MT1 as-is. A program declaration must be added as well as annotations that instruct MT1 about the sparse matrices used in the program. As indicated in [2], there are multiple types of directives that can be used to indicate the non-zero

structure of the matrix used. We will be using the automatic non-zero structure analysis by using file annotations as described in section 4.2 of [2].

We assume here that the used matrix or at least one with identical structure is available at compile time.

After performing the transformation, we will modify the initialization loop for the extended data structure leftCellArrayArray to explicitly generate the matrix file in the coordinate system format used by the sparse compiler.

```
FILE *file = fopen("matrix.cs", "r");
for( row = 0; row < dimensions; ++row )
{
  if( leftRowCopy != NULL && leftRowCopy->RowIndex < row )
    leftRowCopy = leftRowCopy->Next;

  if( leftRowCopy != NULL && leftRowCopy->RowIndex == row )
  {
    leftCell = leftRowCopy->Cell;

    leftCellCopy = leftCell;
    for( x = 0; x < dimensions; ++x )
    {
      if( leftCellCopy != NULL && leftCellCopy->ColIndex < x )
        leftCellCopy = leftCellCopy->ColNext;

      if( leftCellCopy != NULL &&
          leftCellCopy->ColIndex == x )
      {
        fprintf(file, "%i %i %f\n", row, x, leftCellCopy->Value);
      }
    }
  }
}
```

This loop is then executed at compile time, after which it can be removed from the transformed code. Now the matrix file has been generated, we can generate a program to complete the FORTRAN code:

```
      program main
      integer N
      parameter (N=5005)
      real left(N,N)
C_SPARSE(ARRAY(left), FILE('matrix.cs'))
      real right(N,N)
      real result(N,N)
      call Mult(N, result, left, right)
      end
```

This code is run through MT1. Appendix C shows the result of this transformation.

Although we will not run the annihilation code through MT1 in this case we can still do some optimizations, and if the entire transformation is implemented for C these could easily be done automatically. Because of the simple structure of the transformed loop, we can easily use techniques such as loop interchange to optimize memory access patterns and enable vectorization. The restructured loop looks like this:

```
for( row = 0; row < newRowDimensions; ++row )
{
  for( x = 0; x < newDimensionsArray[row]; ++x )
```

```
  {
    array = rightArrayArray[row][x];
    for( col = 0; col < dimensions; ++col )
    {
      tempResult[row][col] += leftCellArrayArray[row][x] * array[col];
    }
  }
}
```

The array variable was introduced to simplify the index expression in the main loop; without it the expression was deemed too complex to vectorize by the compiler.

4.3 Compilation

To ensure a fair comparison between the original linked list code and the code generated by MT1, the latter is transformed back into C using the f2c utility. At this point, a few manual modifications must be made. The C→FORTRAN→C conversion has caused our two-dimensional arrays, which were originally "array-of-arrays" style, to be changed into FORTRAN-style column-major order strided arrays. We must manually change this back to use jagged arrays, and correct the indexing order so that optimal memory access efficiency is obtained. If we can eventually implement the entire process for C, these manual corrections will not be necessary.

The final resulting C code is integrated with the test harness which loads all the necessary matrices and also measures the time taken. This code is then compiled using the Intel C++ Compiler for Windows 9.1, using the /O3 /QxN /Qipo compiler options. These options tell the compiler to use maximum optimization and to target the Pentium 4 enabling it to use the MMX, SSE and SSE2 vector instructions.

Because the inefficiencies and complex expressions introduced by the f2c transformation make it very difficult to effectively optimize the resulting C code (a problem which would not exist if the entire sparse matrix transformation is done with C in mind) we will also include the FORTRAN code in the results. This has been compiled with the Intel Visual FORTRAN Compiler for Windows 9.1, using the same compiler options as used with the C compiler.

4.4 Results

The steps above were performed for a number of matrices from the Harwell Boeing collection, of varying size and density. The results of each algorithm were timed on a system using an Intel Xeon 3.06GHz CPU. In each case, the right hand side matrix was 1000 columns wide.

	sherman3	e40r5000	af23560
Size	5005x5005	17281x17281	23560x23560
Non-zero elements	20033	553965	484256
Density	0.080%	0.186%	0.087%
Results (seconds)			
Original algorithm	22.813 s	266.404 s	443.982 s
Alternative algorithm	2.138 s	46.108 s	29.940 s
Annihilation	0.195 s	1.742 s	2.744 s
MT1 (Fortran)	0.095 s	5.228 s	2.216 s
MT1 (C)	0.124 s	5.153 s	3.314 s

It can be seen that the original algorithm is, as expected, quite inefficient. It always executes the full number of iterations, regardless of how dense the matrix actually is. The alternative

linked list algorithm presented in appendix B does much better at exploiting the sparseness of the matrix.

The C versions of the MT1 algorithms are outperformed by their FORTRAN equivalents, mainly because of the optimization difficulties noted above. The FORTRAN compiler is able to vectorize far more of the loops; in fact, the C compiler most often can vectorize none of them.

Figure 3 shows the speed increases relative to the original linked list algorithm, where higher is better.

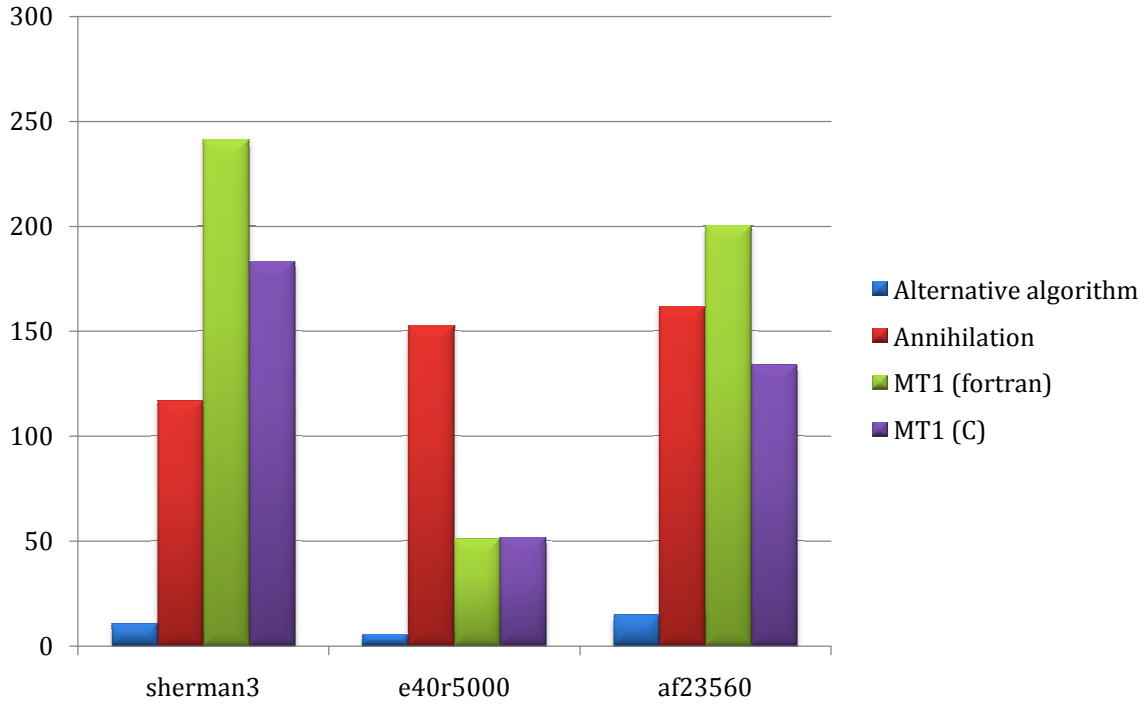

Figure 3 Relative speed increase

The odd one out is obviously e40r5000, as it is the only one where annihilation performs better than MT1. The reason is that this matrix is not recognized by MT1 as having a diagonal structure, but instead a band structure. In this case, MT1 uses a single array to store the non-zero values and uses index arrays to locate them. The multiplication still accesses the matrix in row order, but only the indices containing non-zero values are used. This is extremely like what annihilation does, but annihilation does it without indirection in the inner loop, and with a more efficient nesting order of the loops, allowing better vectorization and improved memory access patterns. The C and FORTRAN versions of the MT1 algorithm for this matrix are very close together owing to the fact that vectorization is not as efficient here.

The absolute times for the af23560 matrix are actually lower for most of the algorithms than they were for the smaller e40r5000 matrix. This is because af23560 has much lower density; it has a lower total number of non-zero elements than e40r5000. The original algorithm, which does not exploit the sparsity, is still much slower, and, perhaps surprisingly, so is annihilation. It would appear that annihilation performs better with a relatively denser matrix, whereas the alternative linked list algorithm and MT1 perform better with a less dense matrix. Sherman3, which has nearly all its values on the main diagonal, is clearly best suited to the transformations MT1 can perform.

It is interesting to note that the annihilation code was generated from the original, inefficient algorithm. Even without using MT1, with just a few simple loop interchanges, and with the overhead of the initialization loops included, we have succeeded in transforming a very

inefficient algorithm into one that outperforms an optimal linked list algorithm and in a few cases even the sparse code generated by MT1.

Bibliography

1. **Bik, Aart C.J.** *Compiler Support for Sparse Matrix Computations.* s.l. : PhD Thesis, Leiden University, 1996.

2. **Bik, Aart J.C., Brinkhaus, Peter J.H. and Wijshoff, Harry A.G.** *The Sparse Compiler MT1: A Reference Guide.* s.l. : LIACS.

3. **Zhao, L. and Wijshoff, H.** *A Case Study in Automatic Data Structure Selection for Optimizing Sparse Matrix Computation.* s.l. : LIACS.

4. **Kennedy, Ken and Allen, Randy.** *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* s.l. : Morgan Kaufmann Publishers, 2001.

5. *Putting pointer analysis to work.* **Ghiya, Rakesh and Hendren, Laurie J.** New York, NY, USA : ACM Press, 1998. Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 121-133.

6. *Parallelizing graph construction operations in programs with cyclic graphs.* **Hwang, Yuan-Shin.** 9, Amsterdam, The Netherlands : Elsevier Science Publishers B. V., 2002, Parallel Computing, Vol. 28, pp. 1307-1328.

7. *Identifying parallelism in programs with cyclic graphs.* **Hwang, Yuan-Shin and Saltz, Joel H.** 3, Orlando, FL, USA : Orlando, FL, USA, 2003, Journal of Parallel and Distributed Computing , Vol. 63, pp. 337-355.

8. *Optimizing memory accesses for spatial computation.* **Budiu, Mihai and Goldstein, Seth C.** Washington, DC, USA : IEEE Computer Society, 2003. Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. pp. 216-227.

9. *Which pointer analysis should I use?* **Hind, Michael and Pioli, Anthony.** New York, NY, USA : ACM Press, 2000. Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis . pp. 113-123.

10. *Pointer analysis: haven't we solved this problem yet?* **Hind, Michael.** New York, NY, USA : ACM Press, 2001. Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 54-61.

11. *Context-sensitive interprocedural points-to analysis in the presence of function pointers.* **Emami, Maryam, Ghiya, Rakesh and Hendren, Laurie J.** New York, NY, USA : ACM Press, 1994. Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation. pp. 242-256.

12. *Symbolic pointer analysis.* **Zhu, Jianwen.** New York, NY, USA : ACM Press, 2002. Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design . pp. 150-157.

13. *Symbolic pointer analysis revisited.* **Zhu, Jianwen and Calman, Silvian.** New York, NY, USA : ACM Press, 2004. Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation. pp. 145-157.

Appendix A. Transformed matrix multiplication code.

Sublimation:

```
int MatrixMultiplyTransformed(Matrix left, float** right, int rightDimensions,
                              float **result, int resultDimensions)
{
  RowHead *leftRow = left.Row;
  Cell *leftCell;
  int dimensions = left.Dimensions;
  int col;
  int row;
  int x;

  float **leftCellArrayArray;
  Cell *leftCellCopy;
  RowHead *leftRowCopy;

  if( !(left.Dimensions == rightDimensions &&
      left.Dimensions == resultDimensions) )
    return -1;

  leftRow = left.Row;

  leftCellArrayArray = malloc(sizeof(float*) * dimensions);

  leftRowCopy = leftRow;
  for( row = 0; row < dimensions; ++row )
  {
    if( leftRowCopy != NULL && leftRowCopy->RowIndex < row )
      leftRowCopy = leftRowCopy->Next;

    leftCellArrayArray[row] = malloc(sizeof(float) * dimensions);
    memset(leftCellArrayArray[row], 0, sizeof(float) * dimensions);

    if( leftRowCopy != NULL && leftRowCopy->RowIndex == row )
    {
      leftCell = leftRowCopy->Cell;

      leftCellCopy = leftCell;
      for( x = 0; x < dimensions; ++x )
      {
        if( leftCellCopy != NULL && leftCellCopy->ColIndex < x )
          leftCellCopy = leftCellCopy->ColNext;

        if( leftCellCopy != NULL &&
            leftCellCopy->ColIndex == x &&
            leftCellCopy->RowIndex == row )
        {
          leftCellArrayArray[row][x] = leftCellCopy->Value;
        }
      }
    }
  }

  // Main loop
  for( col = 0; col < dimensions; ++col )
  {
    for( row = 0; row < dimensions; ++row )
    {
```

```
      for( x = 0; x < dimensions; ++x )
      {
        result[row][col] += leftCellArrayArray[row][x] * right[x][col];
      }
    }
  }

  // Cleanup
  for( row = 0; row < dimensions; ++row )
    free(leftCellArrayArray[row]);
  free(leftCellArrayArray);

  return 0;
}
```

Annihilation:

```
int MatrixMultiplyAnnihilation(Matrix left, float **right, int rightDimensions,
                               float **result, int resultDimensions)
{
  RowHead *leftRow = left.Row;
  Cell *leftCell;
  int dimensions = left.Dimensions;
  int col;
  int row;
  int x;

  float **leftCellArrayArray;
  int leftCellArraySize;
  float ***rightArrayArray;
  int rightArraySize;
  int *newDimensionsArray;
  int newRowDimensions;
  float **resultArray;
  Cell *leftCellCopy;
  RowHead *leftRowCopy;
  float *test;
  int leftCellArrayArraySize;
  int rightArrayArraySize;
  int resultArraySize;

  if( !(left.Dimensions == rightDimensions &&
      left.Dimensions == resultDimensions) )
    return -1;

  // Initialisation
  newRowDimensions = 0;
  leftCellArrayArraySize = 100;
  leftCellArrayArray = malloc(sizeof(float*) * leftCellArrayArraySize);
  newDimensionsArray = malloc(sizeof(int) * leftCellArrayArraySize);
  leftRowCopy = leftRow;
  for( row = 0; row < dimensions; ++row )
  {
    if( newRowDimensions >= leftCellArrayArraySize )
    {
      leftCellArrayArraySize *= 2;
      leftCellArrayArray = realloc(leftCellArrayArray,
                                   sizeof(float*) * leftCellArrayArraySize);
      newDimensionsArray = realloc(leftCellArrayArray,
```

```
                                    sizeof(int) * newDimensionsArray);
    }
    if( leftRowCopy != NULL && leftRowCopy->RowIndex < row )
      leftRowCopy = leftRowCopy->Next;

    if( leftRowCopy != NULL && leftRowCopy->RowIndex == row )
    {
      leftCell = leftRowCopy->Cell;

      leftCellArraySize = 100;
      leftCellArrayArray[newRowDimensions] = malloc(sizeof(float) *
                                                    leftCellArraySize);
      newDimensionsArray[newRowDimensions] = 0;
      leftCellCopy = leftCell;
      // Initialisation loop
      for( x = 0; x < dimensions; ++x )
      {
        if( newDimensionsArray[newRowDimensions] >= leftCellArraySize )
        {
          leftCellArraySize *= 2;
          leftCellArrayArray[newRowDimensions] =
                           realloc(leftCellArrayArray[newRowDimensions],
                                   sizeof(float) * leftCellArraySize);
        }
        if( leftCellCopy != NULL && leftCellCopy->ColIndex < x )
          leftCellCopy = leftCellCopy->ColNext;

        if( leftCellCopy != NULL &&
            leftCellCopy->ColIndex == x &&
            leftCellCopy->RowIndex == row )
        {
leftCellArrayArray[newRowDimensions][newDimensionsArray[newRowDimensions]] =
leftCellCopy->Value;
          ++newDimensionsArray[newRowDimensions];
        }
      }
      ++newRowDimensions;
    }
  }


  newRowDimensions = 0;
  rightArrayArraySize = 100;
  rightArrayArray = malloc(sizeof(float**) * rightArrayArraySize);
  leftRowCopy = leftRow;
  for( row = 0; row < dimensions; ++row )
  {
    if( newRowDimensions >= leftCellArrayArraySize )
    {
      rightArrayArraySize *= 2;
      rightArrayArray = realloc(rightArrayArray,
                                sizeof(float*) * rightArrayArraySize);
    }
    if( leftRowCopy != NULL && leftRowCopy->RowIndex < row )
      leftRowCopy = leftRowCopy->Next;

    if( leftRowCopy != NULL && leftRowCopy->RowIndex == row )
    {
      leftCell = leftRowCopy->Cell;
```

```
      rightArraySize = 100;
      rightArrayArray[newRowDimensions] = malloc(sizeof(float*) *
rightArraySize);
      newDimensionsArray[newRowDimensions] = 0;
      leftCellCopy = leftCell;
      // Initialisation loop
      for( x = 0; x < dimensions; ++x )
      {
        if( newDimensionsArray[newRowDimensions] >= rightArraySize )
        {
          leftCellArraySize *= 2;
          rightArrayArray[newRowDimensions] =
                                realloc(rightArrayArray[newRowDimensions],
                                       sizeof(float*) * rightArraySize);
        }
        if( leftCellCopy != NULL && leftCellCopy->ColIndex < x )
          leftCellCopy = leftCellCopy->ColNext;

        if( leftCellCopy != NULL &&
            leftCellCopy->ColIndex == x &&
            leftCellCopy->RowIndex == row )
        {
rightArrayArray[newRowDimensions][newDimensionsArray[newRowDimensions]] =
right[x];
          ++newDimensionsArray[newRowDimensions];
        }
      }
      ++newRowDimensions;
    }
  }

  leftRowCopy = leftRow;
  // Create temp result array
  resultArraySize = 100;
  resultArray = malloc(sizeof(float*) * resultArraySize);
  // Initial values are read, so we need another initialisation loop
  newRowDimensions = 0;
  for( row = 0; row < dimensions; ++row )
  {
    if( newRowDimensions >= leftCellArrayArraySize )
    {
      resultArraySize *= 2;
      resultArray = realloc(resultArray,
                                sizeof(float*) * resultArraySize);
    }
    if( leftRowCopy != NULL && leftRowCopy->RowIndex < row )
      leftRowCopy = leftRowCopy->Next;

    if( leftRowCopy != NULL && leftRowCopy->RowIndex == row )
    {
      resultArray[newRowDimensions] = result[row];
      ++newRowDimensions;
    }
  }


  for( col = 0; col < cols; ++col )
  {
```

```
    for( row = 0; row < newRowDimensions; ++row )
    {
      for( x = 0; x < newDimensionsArray[row]; ++x )
      {
        tempResult[row][col] += leftCellArrayArray[row][x] *
                                rightArrayArray[row][x][col];
      }
    }
  }

  // cleanup
  free(tempResult);
  for( row = 0; row < newRowDimensions; ++row )
  {
    free(leftCellArrayArray[row]);
    free(rightArrayArray[row]);
  }
  free(leftCellArrayArray);
  free(rightArrayArray);
  free(newDimensionsArray);

  return 0;
}
```

Appendix B. Alternative matrix multiplication algorithm

```
int MatrixMultiply(Matrix left, float **right, int rightDimensions,
                   float **result, int resultDimensions)
{
  RowHead *leftRow = left.Row;
  Cell *leftCell;
  int dimensions = left.Dimensions;
  int col;

  if( !(left.Dimensions == rightDimensions &&
      left.Dimensions == resultDimensions) )
    return -1;

  for( col = 0; col < dimensions; ++col )
  {
    leftRow = left.Row;

    while( leftRow != NULL )
    {
      leftCell = leftRow->Cell;
      while( leftCell != NULL )
      {
        result[leftCell->RowIndex][col] += leftCell->Value *
                                           right[leftCell->ColIndex][col];
        leftCell = leftCell->ColNext;
      }
      leftRow = leftRow->Next;
    }
  }

  return 0;
}
```

Transformed using sublimation:

```
int MatrixMultiplyTransformed(Matrix left, float **right, int rightDimensions,
                              float **result, int resultDimensions)
{
  RowHead *leftRow = left.Row;
  Cell *leftCell;
  int dimensions = left.Dimensions;
  int col;
  int leftRowCounter;
  int leftCellCounter;

  float **leftCellArrayArray;

  if( !(left.Dimensions == rightDimensions &&
        left.Dimensions == resultDimensions) )
    return -1;

  leftRow = left.Row;

  leftCellArrayArray = malloc(dimensions * sizeof(float*));
  /* bulk initialisation not possible so use new loop guard */
  for( leftRowCounter = 0; leftRowCounter < dimensions; ++leftRowCounter )
  {
    leftCellArrayArray[leftRowCounter] = malloc(dimensions * sizeof(float));
```

```
    memset(leftCellArrayArray[leftRowCounter], 0, dimensions * sizeof(float));

    if( leftRow != NULL && leftRowCounter == leftRow->RowIndex )
    {
      leftCell = leftRow->Cell;
      while( leftCell != NULL )
      {
        leftCellArrayArray[leftRowCounter][leftCell->ColIndex] =
                                                          leftCell->Value;
        leftCell = leftCell->ColNext;
      }
      leftRow = leftRow->Next;
    }
  }

  for( col = 0; col < dimensions; ++col )
  {
    for( leftRowCounter = 0; leftRowCounter < dimensions; ++leftRowCounter )
    {
      for( leftCellCounter = 0; leftCellCounter < dimensions; ++leftCellCounter
)
      {
        result[leftRowCounter][col] +=
leftCellArray[leftRowCounter][leftCellCounter] * right[leftCellCounter][col];
      }
    }
  }

  for( leftRowCounter = 0; leftRowCounter < dimensions; ++leftRowCounter )
  {
    free(leftCellArrayArray[leftRowCounter]);
  }
  free(leftCellArrayArray);

  return 0;
}
```

Transformed using annihilation:

```
int MatrixMultiplyAnnihilation(Matrix left, float **right, int rightDimensions,
                               float **result, int resultDimensions)
{
  RowHead *leftRow = left.Row;
  Cell *leftCell;
  int dimensions = left.Dimensions;
  int col;

  float **leftCellArrayArray;
  float ***rightArrayArray;
  Cell *leftCellCopy;
  int *newDimensionsArray;
  int newRowDimensions;
  int leftRowCounter;
  int leftCellCounter;
  RowHead *leftRowCopy;
  float **resultArray;

  if( !(left.Dimensions == rightDimensions &&
      left.Dimensions == resultDimensions) )
```

```
    return -1;

  leftRow = left.Row;

  leftCellArrayArray = malloc(dimensions * sizeof(float*));
  leftRowCopy = leftRow;
  newRowDimensions = 0;
  while( leftRowCopy != NULL )
  {
    leftCell = leftRowCopy->Cell;

    leftCellArrayArray[newRowDimensions] = malloc(dimensions * sizeof(float));
    newDimensionsArray[newRowDimensions] = 0;
    leftCellCopy = leftCell;
    while( leftCellCopy != NULL )
    {
leftCellArrayArray[newRowDimensions][newDimensionsArray[newRowDimensions]] =
leftCellCopy->Value;
      leftCellCopy = leftCellCopy->ColNext;
      ++newDimensionsArray[newRowDimensions];
    }
    leftRow = leftRow->Next;
    ++newRowDimensions;
  }

  rightArrayArray = malloc(dimensions * sizeof(float**));
  leftRowCopy = leftRow;
  newRowDimensions = 0;
  while( leftRowCopy != NULL )
  {
    leftCell = leftRowCopy->Cell;

    rightArrayArray[newRowDimensions] = malloc(dimensions * sizeof(float*));
    newDimensionsArray[newRowDimensions] = 0;
    leftCellCopy = leftCell;
    while( leftCellCopy != NULL )
    {
      rightArrayArray[newRowDimensions][newDimensionsArray[newRowDimensions]] =
      right[leftCellCopy->ColIndex];
      leftCellCopy = leftCellCopy->ColNext;
      ++newDimensionsArray[newRowDimensions];
    }
    leftRow = leftRow->Next;
    ++newRowDimensions;
  }

  resultArray = malloc(dimensions * sizeof(float*));
  leftRowCopy = leftRow;
  newRowDimensions = 0;
  while( leftRowCopy != NULL )
  {
    resultArray[newRowDimensions] = result[leftRow->RowIndex];
    leftRow = leftRow->Next;
    ++newRowDimensions;
  }

  for( col = 0; col < dimensions; ++col )
  {
    for( leftRowCounter = 0; leftRowCounter < newRowDimensions;
```

```
          ++leftRowCounter )
    {
      for( leftCellCounter = 0; leftCellCounter <
            newDimensionsArray[newRowDimensions]; ++leftCellCounter )
      {
        resultArray[leftRowCounter][col] +=
           leftCellArrayArray[leftRowCounter][leftCellCounter] *
           rightArrayArray[leftRowCounter][leftCellCounter][col];
      }
    }
  }

  return 0;
}
```

Appendix C. Optimized matrix multiplication

```
      PROGRAM MAIN
      REAL RIGHT(5005,1000),RESULT(5005,1000)
      INTEGER I_,J_,M_,N_,NNZ_,K_
      REAL V_
      INTEGER TMP__(1)
      REAL DN1_LEFT,DN2_LEFT,DN3_LEFT,DN4_LEFT,DN5_LEFT
      COMMON /LEFT___/DN1_LEFT(386:5005),DN2_LEFT(4620),DN3_LEFT(
     +36:5005),DN4_LEFT(4970),DN5_LEFT(5005,(-1):1)

      OPEN (1,FILE='matrix.mtx',STATUS='OLD')
      READ (1,*) M_,N_,NNZ_
      IF ((M_.NE.5005).OR.(N_.NE.5005)) STOP 'Incorrect size'
      DO K_ = 1, NNZ_, 1
        READ (1,*) I_,J_,V_
        IF (((I_-J_).EQ.(-385)).AND.((387.LE.(I_+J_)).AND.(((I_+J_).LE.
     +  9625).AND.((386.LE.J_).AND.(I_.LE.4620))))) THEN
          DN1_LEFT(J_) = V_
        ELSE IF ((I_-J_).EQ.385) THEN
          DN2_LEFT(J_) = V_
        ELSE IF (((I_-J_).EQ.(-35)).AND.((37.LE.(I_+J_)).AND.(((I_+J_)
     +    .LE.9975).AND.((36.LE.J_).AND.(I_.LE.4970))))) THEN
          DN3_LEFT(J_) = V_
        ELSE IF ((I_-J_).EQ.35) THEN
          DN4_LEFT(J_) = V_
        ELSE IF (((-1).LE.(I_-J_)).AND.((I_-J_).LE.1)) THEN
          DN5_LEFT(J_,I_-J_) = V_
        ELSE IF (((-34).LE.(I_-J_)).AND.((I_-J_).LE.(-2))) THEN
          STOP 'Entry occurs in zero region'
        ELSE IF ((2.LE.(I_-J_)).AND.((I_-J_).LE.34)) THEN
          STOP 'Entry occurs in zero region'
        ELSE IF (((-384).LE.(I_-J_)).AND.((I_-J_).LE.(-36))) THEN
          STOP 'Entry occurs in zero region'
        ELSE IF ((36.LE.(I_-J_)).AND.((I_-J_).LE.384)) THEN
          STOP 'Entry occurs in zero region'
        ELSE IF ((I_-J_).LE.(-386)) THEN
          STOP 'Entry occurs in zero region'
        ELSE IF (386.LE.(I_-J_)) THEN
          STOP 'Entry occurs in zero region'
        END IF
      ENDDO
      CLOSE (1)
      CALL MULT_000LEFT0(5005,1000,RESULT,RIGHT)
      STOP

      END

      SUBROUTINE MULT_000LEFT0(DIM,WIDTH,RESULT,RIGHT)
      INTEGER DIM,WIDTH
      REAL RESULT(5005,1000)
      REAL RIGHT(5005,1000)
      INTEGER ROW,COL,X
      REAL DN2_LEFT,DN1_LEFT,DN3_LEFT,DN4_LEFT,DN5_LEFT
      COMMON /LEFT___/DN1_LEFT(386:5005),DN2_LEFT(4620),DN3_LEFT(
     +36:5005),DN4_LEFT(4970),DN5_LEFT(5005,(-1):1)

      DO ROW = 1, 1000, 1
        DO X = 1, 4620, 1
```

```
            RESULT(X+385,ROW) = RESULT(X+385,ROW)+(DN2_LEFT(X)*RIGHT(X,ROW
     +      ))
            ENDDO
            DO X = 1, 4970, 1
              RESULT(X+35,ROW) = RESULT(X+35,ROW)+(DN4_LEFT(X)*RIGHT(X,ROW))
            ENDDO
            DO COL = -1, 1, 1
              DO X = MAX0(1,COL+1), MIN0(5005,COL+5005), 1
                RESULT(-COL+X,ROW) = RESULT(-COL+X,ROW)+(DN5_LEFT(X,-COL)*
     +          RIGHT(X,ROW))
              ENDDO
            ENDDO
            DO X = 36, 5005, 1
              RESULT(X-35,ROW) = RESULT(X-35,ROW)+(DN3_LEFT(X)*RIGHT(X,ROW))
            ENDDO
            DO X = 386, 5005, 1
              RESULT(X-385,ROW) = RESULT(X-385,ROW)+(DN1_LEFT(X)*RIGHT(X,ROW
     +      ))
            ENDDO
          ENDDO
          RETURN

          END

          BLOCK DATA
          REAL DN1_LEFT,DN2_LEFT,DN3_LEFT,DN4_LEFT,DN5_LEFT
          COMMON /LEFT___/DN1_LEFT(386:5005),DN2_LEFT(4620),DN3_LEFT(
         +36:5005),DN4_LEFT(4970),DN5_LEFT(5005,(-1):1)

          DATA DN1_LEFT /4620*0./
          DATA DN2_LEFT /4620*0./
          DATA DN3_LEFT /4970*0./
          DATA DN4_LEFT /4970*0./
          DATA DN5_LEFT /15015*0./

          END
```