THESIS

IDENTIFICATION OF REGULAR PATTERNS WITHIN SPARSE DATA STRUCTURES

Submitted by

Travis Augustine

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2020

ABSTRACT

IDENTIFICATION OF REGULAR PATTERNS WITHIN SPARSE DATA STRUCTURES

Sparse matrix-vector multiplication (SpMV) is an essential computation in linear algebra. There is a well-known trade-off between operating on a dense or a sparse structure when performing SpMV. In the dense version of SpMV, useless operations are performed but the computation is amenable SIMD vectorization. In the sparse version, only useful operations are executed. However, an indirection array must be used, thus hindering the compiler's ability to perform optimizations that exploit the vector units available on the majority of modern processors. Our process automatically builds sets of regular sub-computations from the irregular sparse data structure. We mine for regular regions in the irregular data structure, grouping together non-contiguous points from the reorderable set of coordinates representing the sparse structure. The coordinates become partitioned into groupings of coordinates of pre-defined shapes using polyhedra. This partition models the exact same points from the input set of coordinates in a way that is specialized to the input's sparsity pattern. Once we have obtained a partition of the points into sets of polyhedra, we then scan these polyhedra to synthesize code that does not store any coordinates of zero-valued elements and does not require any indirection array to access data, thus making it amenable to SIMD vectorization.

ACKNOWLEDGEMENTS

This work was supported in part by the U.S. National Science Foundation award CCF-1750399 and CCF-1725611.

TABLE OF CONTENTS

Chapter 1

Introduction

Sparse matrices and other irregular data structures are an important component of modern computing. In particular, sparse matrix-vector multiplication (SpMV) is an essential computation in linear algebra. SpMV is widely used in a variety of scientific computations, including optimization, computer graphics, and structural problems. The sparse matrices used in SpMV are distinguished by their sparsity pattern defined by the nonzero coordinates in the matrix. The operation shown throughout this work is SpMV. However, our approach is not limited to SpMV and can handle any arbitrary irregular sparse data structure that is reorderable and can be represented as a stream of values.

There is a well-established trade-off between operating on a dense or a sparse structure when performing SpMV. In the dense version of SpMV, useless operations are performed. That is, either an if-conditional must be executed to know when to skip useless computations, or a multiplication by zero operation must be performed needlessly. Favorably, the dense computation leads to easy SIMD vectorization of the program. On the other hand, there are also favorable and non-favorable aspects to performing the computation using a sparse representation. Since the elements whose value is zero are not represented, only useful operations are executed, so there is no longer an extra conditional needed and useless multiplication by zero operations do not get performed. However, the trade-off for this is that an indirection array of the form A[B[i]] must be used. Thus, the index of the outer array is not known at compile time, which limits polyhedral analysis needed to perform optimizations and hinders optimizations regarding vectorization of the program.

An assortment of formats to represent sparse matrices have been explored, as described in detail in Chapter 2. However, there are limitations present in the sparse formats that have been previously explored. Sparse formats commonly require one array to store the data, and another array to locate the data. This leads to indirect array accesses. Other formats for representing sparse matrices may require storage for zero-valued elements. Typical formats for representing sparse matrices are generic to the sparsity pattern of the input matrix. Conversely, our approach is specialized to the sparsity of the matrix. Our process automatically constructs sets of regular sub-computations from the irregular sparse data structure. It does not require any storage of zero-valued elements. Additionally, there is no indirection array required by our approach, thus enabling SIMD vectorization opportunities.

In essence, we exploit the associativity and commutativity of the $+$ operation in SpMV in the case that a relative error margin is allowable to enable the reordering of the computation. We then mine for regular regions in the irregular data structure; i.e., we mine for regularity in the input sparse matrix. The matrix is represented as a trace, which is a set of tuples corresponding to the nonzero coordinates of the matrix. The key property that we exploit is the grouping of non-contiguous points from the reorderable input trace of coordinates into the same structure. Hence, each grouping of points is an independent component of the trace. The template shapes to be mined for are defined prior to analyzing the trace for regularity and can be chosen so that each component of the trace is amenable to vectorization. The tuples obtained from the trace of points are thus compacted using inequalities so that the exact same points from the input trace of points are captured. The trace becomes partitioned into sets of polyhedra defined by inequalities, from which the original trace can be reconstructed using the encoded information of origin point for the shape, as well as the extent and stride of the shape in each dimension.

Once we have obtained a partition of the input trace into sets of polyhedra, we synthesize code that is specialized to the sparsity of the input sparse matrix. Our approach generates a collection of loop-nests, corresponding to each polyhedra containing points from the trace, which iterate exclusively on the non-zero coordinates of the sparse input matrix. Wherever regularity in the sparsity pattern can be identified, a loop-nest is generated for the sub-computation that iterates on the non-zero coordinates which have been grouped together. The generated code does not require any indirection array to access data and does not store any coordinates of zero-valued elements. Thus, the synthesized code is amenable to SIMD vectorization and can be automatically vectorized by the compiler. We evaluate the generated code on traces of nonzero coordinates representing

sparse matrices with under 10 million nonzero elements from the SuiteSparse repository. We present results showing that the time to perform the entire reconstruction process takes under a few minutes and scales with the number of nonzero elements in the input matrix. We also present results showing an improvement in performance compared to standard approaches for some of the evaluated matrices.

Chapter 2

Background and Related Work

Now, we provide a formal description of the matrix-vector multiply computation. We then describe popular sparse formats, and present code which illustrates the issues arising when using these formats. We then describe existing approaches.

2.1 Description of Matrix-Vector Multiply

We now describe the sparse matrix-vector multiply kernel. SpMV is a matrix-vector product. The SpMV computation is of the form $\vec{y} = A.\vec{x}$. We have that the input to the computation is the matrix A and the vector $\vec{x}$. The input matrix A is immutable and sparse and is of size $N \times M$ and the input vector $\vec{x}$ is dense and is of size M. The output is the dense vector $\vec{y}$ of size N. We identify the position of an element in the matrix A by the coordinate (i, j) such that $0 \leq i < N$ and $0 \leq j < M$. When A is a dense matrix; i.e., the majority of its elements are nonzero, the computation using a dense kernel will iterate over all values (i, j) such that $0 \leq i < N$ and $0 \leq j < M$. When A is a sparse matrix; i.e., when the majority of the elements in A are zero, and a sparse representation is used, only the nonzero elements of A are stored and operated on. In the sparse case, only a subset corresponding to the nonzero elements of the (i, j) coordinates are represented.

2.2 Description of Existing Formats for Representing Sparse Matrices

We now provide a formal description of popular existing formats for representing static sparse matrices. The majority of the formats described in this section are applicable to general sparse matrices of any sparsity structure, with the exception being the Diagonal (DIA) format, which requires the matrix to have an underlying diagonal structure. However, there do exist formats besides DIA that are specialized to specific sparsity patterns.

Coordinate List (COO) First, we describe the Coordinate List Format (COO). COO is a simple storage format. It it used for representing general sparse matrices. COO uses three one-dimensional arrays, all of length NNZ. The first array, row, explicitly stores the row indices of the nonzero elements in the matrix. The second array, col, explicitly stores the column indices of the nonzero elements in the matrix. The third array, data, stores the values of the nonzero elements in the matrix. The values at a given index for each of these three arrays align with the values in the other two arrays. Figure 2.1 shows the three arrays used by the COO format to represent a sparse matrix. As illustrated by the structure of the COO representation, the storage required when using COO is proportional to the number of nonzero entries that exist in the sparse matrix. The merit of this format does not depend on the sparsity structure of the matrix, but rather the number of nonzero entries that exist in the matrix.

$$A = \begin{bmatrix} 0 & 0 & 9 & 0 & 0 \\ 1 & 3 & 4 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{bmatrix} \qquad row = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 3 \end{bmatrix}$$

$$col = \begin{bmatrix} 2 & 0 & 1 & 2 & 3 & 3 \end{bmatrix} \qquad data = \begin{bmatrix} 9 & 1 & 3 & 4 & 7 & 2 \end{bmatrix}$$

Figure 2.1: The arrays row, col, and data, used by the COO format to represent the 4×5 matrix A.

Essentially, this means the COO format can be viewed as a list of three-dimensional tuples that identify the row index, column index, and value for each nonzero entry in the sparse matrix. Thus, the tuples used in COO are of the form of $(row, column, value)$. COO explicitly encodes the row and column indices for each nonzero value, unlike most of the other sparse formats described here, which typically utilize some encoding to minimize the storage space while still implicitly encoding the row and column indices. An advantage of COO is that it can be quickly and easily converted into other formats such as compressed sparse rows and compressed sparse columns, described later in this chapter, by converting these explicit indices into the implicit formats used by the other representations.

In the general case, there is not a required ordering done to the tuples maintained by COO. However, a simple extension of this format can have the entries sorted by row index, then by column index. COO is the native format used by the Matrix Market Exchange format within the SuiteSparse repository and is used as the raw input to our program prior to parsing and analysis of the matrix. In addition to the COO representation, the Matrix Market Exchange format appends a header consisting of the number of rows, number of columns, and number of nonzero entries listed, although this is not technically part of the COO format.

Compressed Sparse Rows (CSR) We now describe the Compressed Sparse Rows (CSR) format. Similar to COO, CSR represents the sparse input matrix by using three one-dimensional arrays. In fact, CSR explicitly stores the column indices of the nonzero elements of the matrix in the exact same way as the COO format. These column indices are stored in the array col_indices, which is of length NNZ. Additionally, in the exact same way as COO, CSR explicitly stores the values of the nonzero entries of the sparse matrix in the array data, which is also of length NNZ.

The clear difference between COO and CSR is that CSR extends upon the sorted version of COO's representation of the row indices by applying a compression scheme to the repeated row indices, thus reducing the required storage. The name of the CSR format is derived from this row compression scheme being used. The third array used by CSR for implicitly encoding row indices which varies from COO is referred to as the ptr array. Figure 2.2 shows these three arrays that are used by the CSR format to represent the sparse matrix A from Figure 2.1.

$$A = \begin{bmatrix} 0 & 0 & 9 & 0 & 0 \\ 1 & 3 & 4 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{bmatrix} \qquad ptr = \begin{bmatrix} 0 & 1 & 5 & 5 & 6 \end{bmatrix}$$

$$col\_indices = \begin{bmatrix} 2 & 0 & 1 & 2 & 4 & 3 \end{bmatrix} \qquad data = \begin{bmatrix} 9 & 1 & 3 & 4 & 7 & 2 \end{bmatrix}$$

Figure 2.2: The arrays ptr, col_indices, and data, used by the CSR format to represent the sparse matrix A.

For representing an $N \times M$ sparse matrix, the ptr array will be of length $N + 1$; that is, one more than the number of rows of the input matrix. The first entry in the ptr array is always zero. For every subsequent entry, its value is defined as the previous entry's value plus the number of nonzero elements in the previous row. Thus, we have that for all $0 < i \leq N, ptr[i] = ptr[i-1]+$ the number of nonzero elements in row $(i-1)$ of the matrix. Although not part of the formal definition, a useful encoding to note is that the final value of the ptr array is always equal to the number of nonzero elements in the matrix.

Compressed Sparse Columns (CSC) The next format we describe is the Compressed Sparse Columns (CSC) format, which uses exactly the same concept as CSR, simply applying the compression scheme to the columns instead of the rows. Clearly, it also uses three one-dimensional arrays. It uses the same array data as was used in CSR and COO for storing the nonzero elements of the sparse input matrix. The data array is of length NNZ as before. A significant difference is that the order of the elements stored in the data array could differ from the order they were stored using CSR since the nonzero elements are now sorted by column instead of by row.

In contrast to CSR, which explicitly stores the column index of each nonzero element, CSC explicitly stores the row index of each nonzero element. These explicit row indices are stored in the array row_indices, which is of length NNZ. Instead of applying a compression scheme to the row indices as is done by CSR, CSC instead applies a compression scheme to the column indices. The third array used by CSC for preserving column indices which varies from COO is referred to as the ptr array. Shown in Figure 2.3, we have the arrays used by the CSC format to represent the previously shown sparse matrix A.

$$A = \begin{bmatrix} 0 & 0 & 9 & 0 & 0 \\ 1 & 3 & 4 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{bmatrix} \qquad ptr = \begin{bmatrix} 0 & 1 & 2 & 4 & 5 & 6 \end{bmatrix}$$

$$row\_indices = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 3 \end{bmatrix} \qquad data = \begin{bmatrix} 1 & 3 & 9 & 4 & 2 & 7 \end{bmatrix}$$

Figure 2.3: The arrays ptr, row_indices, and data, used by the CSC format to represent the sparse matrix A.

For representing an $N \times M$ sparse matrix, the ptr array will be of length $M + 1$; that is, one more than the number of columns of the input matrix. The definition of the ptr array is recursive and analogous to its definition for CSR. As with CSR, its first entry is always zero. For every subsequent entry, its value is defined as the previous entry's value plus the number of nonzero elements in the previous column. We have that for all $0 < i \leq M$, $ptr[i] = ptr[i-1] +$ the number of nonzero elements in column $(i - 1)$ of the matrix. As with CSR, the final value of the ptr array is always equal to the number of nonzero elements in the matrix.

ELLPACK (ELL) The next format we describe is the ELLPACK format (ELL), which takes a completely different approach to that of COO, CSR, and CSC. ELL requires two arrays to represent the sparse matrix, the two-dimensional data array and the two-dimensional col_indices array. The data array is used to store the values of the nonzero elements, and the col_indices array is used to explicitly store the column indices of the nonzero elements. For an input sparse matrix with N rows, both of the matrices used by ELL will have N rows. However, the number of columns for these matrices is determined by the row in the input matrix which contains the largest number of nonzero entries. This can result in ELL using matrices with only a single column up to as many columns as the number of columns as the input matrix. Due to this variance in column width, ELL is not an efficient approach if there is a medley of both rows with very few nonzero elements as well as rows with a large number of nonzero elements since storage is wasted on the rows with few nonzero elements. Therefore, unlike COO, the merit of ELL is heavily reliant on the sparsity structure of the matrix.

The value for each element is stored in the data array, such that the value is stored in the same row of the data array as it was in the original sparse matrix. However, the column where the nonzero value resides in the data array is not necessarily the same column as in the original matrix. The column indices for the nonzero components are identified using the col_indices array. The col_indices array has the same structure as the data array, but instead of containing the nonzero values of the matrix, it contains the explicit column index where each corresponding nonzero element resides in the original matrix. Shown in Figure 2.4 are the two two-dimensional arrays used by ELL to represent the previously shown sparse matrix A.

$$A = \begin{bmatrix} 0 & 0 & 9 & 0 & 0 \\ 1 & 3 & 4 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{bmatrix} \qquad col\_indices = \begin{bmatrix} 2 & * & * & * \\ 0 & 1 & 2 & 4 \\ * & * & * & * \\ 3 & * & * & * \end{bmatrix} \qquad data = \begin{bmatrix} 9 & * & * & * \\ 1 & 3 & 4 & 7 \\ * & * & * & * \\ 2 & * & * & * \end{bmatrix}$$

Figure 2.4: The two two-dimensional arrays, col_indices and data, used by ELL to represent the previously shown sparse matrix A, where * represents an element required for padding.

Hybrid (HYB) Next, we describe the Hybrid format (HYB), which combines properties of both COO and ELL. It consists of a regular structure through the ELL format and an irregular structure through the COO format. Since COO is a highly unstructured format and is invariant to the sparsity structure of the input matrix, it is chosen to be used in complement with ELL. The goal of the HYB format is to alleviate the issues arising in ELL when there are large differences in the number of nonzero elements in each row. As with ELL, there are two two-dimensional arrays maintained, the data array and col_indices array as before. However, the structure of these arrays is not the same as the structure using ELL. In particular, the number of columns in these arrays differs from the typical ELL representation. The ELL representation is used to store the nonzero values within a determined width. That is, rows with an extreme number of nonzero elements are not fully stored in the ELL representation used by HYB. Rather, the values from rows with an abnormally large number of nonzero elements are stored using the three arrays of the COO representation, as previously described. Figure 2.5 shows the sparse matrix B being represented using the HYB format, with its elements being split across storage in the ELL and COO formats.

$$B = \begin{bmatrix} 8 & 0 & 9 & 0 & 0 \\ 1 & 3 & 4 & 5 & 7 \\ 6 & 0 & 0 & 1 & 0 \\ 0 & 7 & 0 & 2 & 0 \end{bmatrix}$$

$$col\_indices = \begin{bmatrix} 0 & 2 \\ 0 & 1 \\ 0 & 3 \\ 1 & 3 \end{bmatrix} \qquad ell\_data = \begin{bmatrix} 8 & 9 \\ 1 & 3 \\ 6 & 1 \\ 7 & 2 \end{bmatrix}$$

$$row = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \qquad col = \begin{bmatrix} 2 & 3 & 4 \end{bmatrix} \qquad coo\_data = \begin{bmatrix} 4 & 5 & 7 \end{bmatrix}$$

Figure 2.5: The sparse matrix B is represented using the HYB format. The col_indices and ell_data arrays are the ELL component of the representation, and the row, col, and coo_data arrays are the COO component of the representation.

Diagonal Format (DIA) We now describe the Diagonal format (DIA), which specifically models sparse matrices which have diagonal patterns. Contrary to the previously described formats, DIA is not considered a general format for representing sparse matrices since it is specialized to matrices that have the majority of their nonzero values located along diagonals in the matrix. It can be used to model any sparse matrix, but the merit of this approach relies on the sparsity pattern of the matrix containing diagonal patterns. DIA requires a two-dimensional array data and a one-dimensional array offsets to represent the input matrix. The data array consists of the same number of rows as the input matrix. Its number of columns is determined by the number of diagonals in the input matrix which contain nonzero values. Each column contains values corresponding to a specific diagonal within the matrix. The column ordering is determined by the location of the diagonal in the matrix. The values which are located along diagonals that are below the main diagonal of the matrix are stored in the leftmost columns of the data array and the values located along diagonals above the main diagonal are stored in its rightmost columns. In the data array, the nonzero values are stored at the same row position as they were in the input matrix.

The offsets array is used to determine the location of each diagonal in the input matrix. It contains the same number of columns as the number of diagonals of nonzero points in the input matrix. Each column of this one-dimensional array stores the offset of the diagonal in relation to the main diagonal. For example, if column k in the data array corresponds to the main diagonal of values, then column k in the offsets array will have a value of zero. Columns in offsets corresponding to diagonals below the main diagonal will contain negative values representing the diagonal's distance from the main diagonal. Likewise, columns in offsets corresponding to diagonals above the main diagonal will contain the positive offsets representing distance from the main diagonal. Figure 2.6 shows an example of the DIA format being used to model the matrix C which contains the majority of its elements along diagonals.

$$C = \begin{bmatrix} 9 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 8 & 1 & 6 \\ 0 & 0 & 5 & 2 \end{bmatrix} \qquad data = \begin{bmatrix} * & 9 & 2 \\ 0 & 3 & 4 \\ 8 & 1 & 6 \\ 5 & 2 & * \end{bmatrix} \qquad offsets = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

Figure 2.6: The DIA format using the two-dimensional data array and the one-dimensional offsets array to model the matrix C which contains the majority of its elements along diagonals.

Blocked Compressed Sparse Rows (BCSR) Finally, we describe the Blocked Compressed Sparse Rows (BCSR) format, which extends on the CSR format described above. As in CSR, BCSR uses three arrays to represent the input matrix. A pre-defined block size $P \times Q$ is set, which identifies blocks of size P in the i-dimension and size Q in the j-dimension. The input matrix is divided into block rows, where each block row consists of P rows from the matrix. The two-dimensional data array used by BCSR will consist of P rows and its number of columns will be Q multiplied by the number of identified blocks. Each of the identified dense blocks are stored consecutively in the data array, including the zero-valued elements contained in the blocks. The one-dimensional col_indices array is used analogously to its use in CSR, explicitly storing the starting column index for each of the identified blocks in row-major order. Hence, the length of the col_indices array will be the number of identified blocks. There is an index in the one-dimensional ptr array for each of the divided block rows of the matrix, plus one additional index. In a comparable way to CSR, the ptr array stores the number of blocks along a block row in relation to the preceding block row, with its last element containing the total number of blocks stored in the data array. We show in Figure 2.7 the three arrays used by BCSR to represent the matrix D using a block size of 2×3.

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 & 3 & 6 & 9 \\ 5 & 0 & 0 & 0 & 2 & 8 & 0 \\ 2 & 8 & 1 & 0 & 0 & 0 & 0 \\ 4 & 3 & 6 & 9 & 0 & 0 & 0 \end{bmatrix}$$

$$ptr = \begin{bmatrix} 0 & 2 & 4 \end{bmatrix} \qquad col\_indices = \begin{bmatrix} 0 & 4 & 0 & 3 \end{bmatrix}$$

$$data = \begin{bmatrix} 0 & 0 & 0 & 3 & 6 & 9 & 2 & 8 & 1 & 0 & 0 & 0 \\ 5 & 0 & 0 & 2 & 8 & 0 & 4 & 3 & 6 & 9 & 0 & 0 \end{bmatrix}$$

Figure 2.7: The BCSR format used to represent the matrix D using a block size of 2×3.

2.3 Kernels of Existing Formats for Representing Sparse Matrices

We now provide C code for the SpMV computation for each of the previously described sparse formats. We start by showing C code for the dense computation, in the first case performing the multiplication by zero, and in the second case skipping the multiplications by zero by using an if-conditional. In these examples, the value N is the number of rows for the input matrix and the value M is the number of columns for the input matrix. When the parameter num_cols is used, it is referring to the number of columns in the sparse format's array and not the number of columns in the input matrix. For representations utilizing a two-dimensional array, we assume that the two-dimensional array has been converted to a one-dimensional array in column-major order prior to the SpMV kernel.

Classical dense We show the SpMV code for the dense computation which performs the multiplication by zero operations.

```
for (i = 0; i < N; ++i) {
  y[i] = 0;
  for (j = 0; j < M; ++j) {
    y[i] += A[i][j] * x[j];
  }
}
```

Classical dense skipping nonzero entries We show the SpMV code for the dense computation which uses an if-conditional to skip the multiplications by zero.

```
for (i = 0; i < N; ++i) {
  y[i] = 0;
  for (j = 0; j < M; ++j) {
    if (A[i][j] != 0) {
      y[i] += A[i][j] * x[j];
    }
  }
}
```

Coordinate List (COO) We show the SpMV code using the three arrays maintained by the COO format.

```
for (i = 0; i < NNZ; ++i) {
  y[row[i]] += data[i] * x[col[i]];
}
```

Compressed Sparse Rows (CSR) We show the SpMV code for the classical CSR approach.

```
for (i = 0; i < N; ++i) {
  for (j = ptr[i]; j < ptr[i+1]; ++j) {
    y[i] += data[j] * x[col_indices[j]];
  }
}
```

Compressed Sparse Columns (CSC) We show the SpMV code for the classical CSC approach.

```
for (i = 0; i < M; ++i) {
  for (j = ptr[i]; j < ptr[i+1]; ++j) {
    y[row_indices[j]] += data[j] * x[i];
  }
}
```

ELLPACK (ELL) We show the SpMV code for the classical ELL format with its two-dimensional arrays converted to one-dimensional arrays in column-major order.

```
for (j = 0; j < num_cols; ++j) {
  for (i = 0; i < N; ++i) {
    y[i] += data[j * N + i] * x[col_indices[j * N + i]];
  }
}
```

Hybrid (HYB) We now show the SpMV code for the HYB approach, first computing on the portion of data stored using the ELL format, then computing on the portion of data stored using the COO format.

```
// ELL
for (j = 0; j < num_cols; ++j) {
  for (i = 0; i < N; ++i) {
    y[i] += data[j * N + i] * x[col_indices[j * N + i]];
  }
}
// COO
for (i = 0; i < L; ++i) {
  y[row[i]] += data[i] * x[col[i]];
}
```

Diagonal Format (DIA) We now show the SpMV code for the DIA representation, with its two-dimensional arrays converted to one-dimensional arrays in column-major order.

```
for (d = 0; d < num_cols; ++d) {
  i_start = max(0, -offsets[d]);
  j_start = max(0, offsets[d]);
  K = min(N - i_start , M - j_start);
  for (k = 0; k < K; ++k) {
    y[i_start + k] += data[i_start + d * N + k] * x[j_start + k];
  }
}
```

Blocked Compressed Sparse Rows (BCSR) We now show the BCSR kernel for a block size of 2×3 with an unrolled inner computation. The inner computation consisting of six statements performs the operations corresponding to each point within the block of size 2×3.

```
// for each block row
for (i = 0; i < m; ++i) {
  // for each block in the row
  for (j = ptr[i]; j < ptr[i+1]; ++j) {
    y[i] += data[j] * x[col_indices[j]];
    y[i] += data[j+1] * x[col_indices[j]+1];
    y[i] += data[j+2] * x[col_indices[j]+2];
    y[i+1] += data[j+3] * x[col_indices[j]];
    y[i+1] += data[j+4] * x[col_indices[j]+1];
    y[i+1] += data[j+5] * x[col_indices[j]+2];
  }
}
```

2.4 Related Work

There has been research work done regarding the optimization of SpMV, with a variety of approaches described in this chapter. There has also been work done regarding compression and analysis of traces of values. An additional common approach involves work being done regarding inspector/executor approaches which inspect the sparsity of the matrix so that data can be packed for an executor program to run. Regarding SpMV optimization, substantial work has been done regarding improving the representation of the sparse matrix. These approaches are the most analogous to our approach since we manipulate the inherent way that the sparse matrix is depicted. However, these past approaches are fundamentally different from ours. Our approach achieves the objective of creating piecewise-regular representations of the irregular sparse structure at compile-time with a target of optimized code generation, which is not the objective achieved by past approaches.

2.4.1 Polyhedral Trace Compression

The Trace Reconstruction Engine (TRE) [1], described in detail in this work, iteratively analyzes a stream of memory addresses. In essence, TRE incrementally constructs a solution describing an input trace of memory addresses. It incrementally increases the size of this solution until the entire input trace is modeled by its generated solution. It has been expanded [2] to support reconstruction of general piecewise-affine domains, which partitions the irregular computation into a union of polyhedral pieces. TRE has also been expanded to support multiple statements inside a loop body. While our work focuses on identifying pre-defined rectangular subregions in the trace of accesses in order to generate code which is highly regular and amenable to vectorization, TRE contrarily models the trace using complex shapes that do not necessarily fit a rectangular criteria. Also in contrast to our approach, the form of TRE's solution is unknown until it has processed the trace, whereas our approach uses pre-defined shapes to identify regular regions in the input.

Work done by Clauss et al. [3, 4] models memory access information as loop nests. This work identifies patterns in the input memory access trace through detection of periodic behavior.

They produce loop nests which produce the exact original sequence of values when the loop nests are executed. Therefore, the loop nests can be stored instead of the input trace, thus achieving compression of the memory trace.

More recently, work by Ketterlin and Clauss [5] is analogous to this work in that they present a method for the modeling and compression of memory access traces. They also present a method for prediction of memory traces. Their work models memory access traces as loop nests consisting of affine bounds and subscripts. It relies on the detection of periodic behavior such that the exact same original memory access sequence is traversed when the loops are run. This work, relying on periodic behavior, could be used instead of TRE to model the input trace.

2.4.2 Sparse Formats

There has been prior work done optimizing SpMV on a variety of platforms. We describe some of the related approaches in this section. However, to the best of our knowledge, this is the first work which automatically mines for regularity in irregular sparse data structures by identifying regular sub-components and reconstructing a polyhedral representation for the sub-components towards the aim of enabling optimized polyhedral code generation for the sparse data structure [18]. Additionally, our work focuses specifically on small sparse matrices; that is, those containing below ten-million nonzero elements. In contrast, work typically done to optimize SpMV is focused on the efficient execution of very large sparse matrices. For reference regarding the size of the matrices that we operate on, the overhead of the simple initialization of the Intel Math Kernel Library is unreasonable for the small sparse matrices that we operate on.

While our approach does not perform parameter tuning, there are valuable approaches that operate on formats requiring additional tuning of parameters. Vuduc [6] presented a system that generates code for sparse matrix kernels that are specifically adjusted for the CPU platform. This automated approach searches a space of generated kernels and selects the one determined to be most efficient by both heuristic and empirical criteria. Also utilizing parameter tuning, the work of Williams et al. [7] focuses specifically on the optimization of SpMV on multicore platforms.

Their work performs optimizations using a variety of approaches, both focusing on single-core performance, as well as a focus on multicore performance through parallelization optimizations.

Although not directly related to our approach, since we focus on operating on the CPU and not the GPU, there has also been extensive work to optimize SpMV on the GPU platform. Notably, Bell and Garland [8] implemented sparse formats in CUDA, and proposed the hybrid (HYB) approach which combines properties of ELL and COO to alleviate the issues arising with ELL when there are large differences in the number of nonzero elements in each row. They also demonstrate efficient SpMV kernels for the GPU, including the CSR, COO, ELL, DIA, and PKT formats.

Also focusing on the execution of SpMV on GPUs, Choi et al. [9] developed hand-tuned SpMV implementations for GPUs and implemented variations of the classical Blocked Compressed Sparse Row (BCSR) format and developed the Blocked ELLPACK (BELLPACK) sparse format. The overall objective of their block-based formats are similar to our approach in the sense that they view the sparse data structure as a collection of regular blocks of matrix coordinates. However, unlike our technique, blocked approaches do not utilize the reorderability property of the input matrix. The blocked procedure groups contiguous points together, which leads to the common requirement that the blocks must be the same size across the matrix, as well as the need to store zero-valued elements. Our approach does not contain either of these limitations as it does not store any zero-valued elements and it does not require that blocks of points be the same size throughout the matrix.

Yang et al. [10] present a representation scheme and a tiling algorithm to develop SpMV optimizations on GPUs which account for architecture features of GPUs. It also accounts for characteristics of the graph-mining applications being focused on. Their approach targets sparse matrices which represent large graphs with power-law characteristics.

A variety of formats that are not considered standard formats for representing sparse matrices have been proposed. The work of Kourtis et al. [11] proposes a new storage format for sparse matrices targeting SpMV called Compressed Sparse eXtended (CSX). Their approach identifies substructures within the sparse matrix which can be one-dimensional substructures along a row, column, or diagonal, or two-dimensional substructures. It then encodes these patterns using a compression format expanding upon the compression scheme of CSR which is specialized to the identified substructures.

Ekambaram and Montagne [12] proposed an alternative storage format for sparse data structures that is inspired by the Jagged Diagonal Storage format, referred to as the Transposed Jagged Diagonal Storage format. It is a general format in that it handles arbitrary sparsity patterns within the input matrix. However, this format relies on a level of indirection arrays to access the rearranged nonzero matrix elements when performing the SpMV kernel.

The TACO compiler [13,14] is a technique that generates code for optimized sparse and dense tensor algebra computations. It is able to generate optimized code without relying on hand-written kernels. It could be possible to integrate our work within the TACO compiler.

2.4.3 Inspector/Executor Methods

The inspector/executor approach is done in two phases. The first phase is analysis. In the analysis phase, the data access pattern is inspected. So in the case of SpMV, the matrix sparsity pattern is inspected and changes to the structure of the matrix are applied. The second phase is execution. In the execution phase, the values are retrieved, and the loop computation is executed. The inspector/executor approach was developed initially by Saltz et al. [15].

Sympiler is a compiler developed by Chesmi et al. [16,17] which uses an inspector/executor approach to optimize sparse computations. Sympiler takes as input both the sparse matrix as well as the sparse algorithm being computed and performs symbolic analysis of sparse codes at compile-time. It generates optimized executor code that is specialized to the sparsity pattern of the input matrix.

Chapter 3

Folding Integer Sequences to Polyhedra

We now describe in detail the approaches used to represent sets of integer tuples as a partition of polyhedra. We first define concepts and definitions. In subsequent sections, we describe the algorithms that are used to mine for regularity in the irregular sparse data structure.

	i	j	k
1:	1	1	1
2:	4	1	2
3:	2	2	3
4:	5	2	4
5:	6	2	5
6:	3	3	6
7:	5	3	7
8:	6	3	8
9:	4	4	9

Figure 3.1: Shown is an excerpt of the accesses performed during the SpMV computation of the HB/nos2 matrix, where i is in the index into the y array, j is the index into the x array, and k is the index into the A array.

We now formally define the structures which are rebuilt from the input trace.

Definition 3.1. (Integer tuple). A point $\vec{p}$ is an integer tuple of dimension $dim(p)$. It is noted $\vec{p} = (p_1, ..., p_{dim(p)})$, such that $\forall i,\ p_i \in \mathbb{N}$.

Example 3.1. In Figure 3.1, the point located on line 4 is a tuple $\vec{p^4} = (5, 2, 4)$ such that $dim(\vec{p^4}) = 3$.

Definition 3.2. (Trace of points). A trace T is an ordered list of n points $T = \{\vec{p^1}, ..., \vec{p^n}\}$. The dimensionality of T is denoted as $dim(T) = dim(\vec{p^i})$. All points in T must have the same dimensionality.

Example 3.2. In the above example, we have that T is the list of all points in the trace, and $dim(T) = 3$. In the case that the trace is reorderable, T can be viewed as a set of points instead of a list.

Definition 3.3. (Reorderable trace of points). A reorderable trace of points T is an unordered set of n points $T = \{\vec{p^1}, ..., \vec{p^n}\}$. Its dimensionality is denoted as $dim(T) = dim(p^i)$, where all points in T must have the same dimensionality.

We have that a trace is deemed reorderable when the dependences in the computation allow the points in the trace to be executed in any order. If the associativity and commutativity of the $+$ operation are allowed, then the trace for SpMV is considered reorderable. Without the property of reorderability, there is a limitation that makes it so that only consecutive points in the trace may be grouped into the same polyhedron. Since we exploit the associative reordering of SpMV, we are able to consider the input trace as a reorderable trace, which enables skipping points in the trace and allows us to combine points distant in the trace into the same polyhedra by modeling strides between points that are grouped together. Without the ability to skip points in the trace, we would be limited to grouping only points which are consecutive, leading to fewer opportunities to identify regularity in the sparse matrix.

Now, we formally define the structures which are rebuilt from a trace of points.

Definition 3.4. (Affine inequality). Given an integer set of dimension d involving variables noted $(x_1, ..., x_d)$, an affine inequality is written as $\sum_{i=1}^{d} \alpha_i x_i + \beta \geq 0$, where $\forall i, \alpha_i \in \mathbb{Z}$ and $\beta \in \mathbb{Z}$.

Definition 3.5. (Integer polyhedron). An integer polyhedron $\mathcal{D}$ of dimension $dim(\mathcal{D})$ contains integer points $\vec{p^i}$, such that $dim(\mathcal{D}) = dim(\vec{p^i})$. It is defined by the intersection of finitely many half-planes defined by affine inequalities. We note a polyhedron by specifying its dimensions name followed by the inequalities defining it in the following way: $\mathcal{D} = \{[i_1, ..., i_{dim(\mathcal{D})}] : ineq_1, ..., ineq_p\}$

Example 3.3. In Figure 3.1, consider the point $\vec{p^2} = (4, 1, 2)$. The polyhedron $\mathcal{D}_1 = \{[i, j, k] : i = 4 \wedge j = 1 \wedge k = 2\}$ trivially captures this single point exactly. In practice, representing a single point using a single polyhedron is not ideal as it does not provide compression improvement. Note that we can always represent an equality as a set of two inequalities; for example, $i = 0$ can be represented equivalently as $i \leq 0 \wedge i \geq 0$.

Example 3.4. In Figure 3.1, consider the two points $\vec{p^1} = (1, 1, 1)$ and $\vec{p^2} = (4, 1, 2)$. The single polyhedron $\mathcal{D}_1 = \{[i, j, k] : 8j = 1 \wedge 1 \leq k \leq 2 \wedge i = 2k\}$ captures this set of two points exactly.

Example 3.5. In Figure 3.1, consider the two non-consecutive points $\vec{p^4} = (5, 2, 4)$ and $\vec{p^8} = (6, 3, 8)$. The single polyhedron $\mathcal{D}_1 = \{[i, j, k] : 5 \leq i \leq 6 \wedge j = i - 3 \wedge k = 4i - 16\}$ captures this set of two points exactly.

Definition 3.6. (Integer lattice). An integer lattice F is an integer multidimensional function $F : \vec{I} \rightarrow \vec{O}$. F must be represented exactly via a matrix $dim(\vec{O}) \times dim(\vec{I})$ consisting of only integer coefficients. We denote a lattice by specifying its input and output dimensions, followed by the equalities defining the function: $F = \{[i_1, ..., i_{dim(\vec{I})}] \rightarrow [o_1, ..., o_{dim(\vec{O})}] : o_1 = f_1(\vec{I}), o_2 = f_2(\vec{I}), ...\}$.

Definition 3.7. ($\mathcal{Z}$-polyhedron). A $\mathcal{Z}$-polyhedron is the intersection of an integer polyhedron and an integer lattice as defined above. Equivalently stated, the points represented by a $\mathcal{Z}$-polyhedron are the image of $\mathcal{D}$ by F.

Example 3.6. Consider the set of six one-dimensional points $\{3, 5, 7, 11, 13, 15\}$. They are captured exactly by the integer lattice $F = \{[i, j] \rightarrow [x] : x = 8i + 2j + 1\}$ and the polyhedron $\mathcal{D}_2 = \{[i, j] : 0 \leq i \leq 1 \wedge 1 \leq j \leq 2\}$. By taking the image of $\mathcal{D}_2$ by F; that is, $F(\mathcal{D}_2)$, we obtain the original values from the set, 3, 5, 7, 11, 13, and 15.

We use the standard notation, $\cap$, $\cup$ and $\#$ for representing union, intersection, and number of points, respectively.

Definition 3.8. ($\mathcal{Z}$-polyhedron origin). The origin of a $\mathcal{Z}$-polyhedron is the lexicographically first point in this $\mathcal{Z}$-polyhedron; that is, $\vec{p_{\text{orig}}} = lexmin(\mathcal{D} \cap F)$.

Example 3.7. The origin of the $\mathcal{Z}$-polyhedra from the previous example is 3.

Finally, we define a reconstructed trace of points as a union of $\mathcal{Z}$-polyhedra of potentially differing dimensions.

Definition 3.9. (Reconstructed trace). A reconstructed trace T_{poly} from the trace T of n elements is a list of finitely many $\mathcal{Z}$-polyhedra D_i noted $T_{poly} = \{D_1, ..., D_p\}$, such that $\sum_i \#D_i = n$ and $T \equiv T_{poly}$. That is, T_{poly} captures the exact same set of points as the trace T. Any two $\mathcal{Z}$-polyhedra D_i and D_j in T_{poly} may have different dimensionalities and different cardinalities.

Example 3.8. Consider the trace of points in Figure 3.1. We will describe a reconstructed trace that captures exactly the nine points in the original trace. Assume the original trace to be reorderable. Then we can group sets of points together in the same polyhedron, regardless if they are consecutive in the trace. Consider the points $\vec{p^1} = (1,1,1)$ and $\vec{p^2} = (4,1,2)$. The polyhedron $\mathcal{D}_1 = \{[i,j,k] : 8j = 1 \wedge 1 \leq k \leq 2 \wedge i = 2k\}$ models these two points. The polyhedron $\mathcal{D}_2 = \{[i,j,k] : 5 \leq i \leq 6 \wedge j = i - 3 \wedge k = 4i - 16\}$ models $\vec{p^4} = (5,2,4)$ and $\vec{p^8} = (6,3,8)$. The three points $\vec{p^5} = (6,2,5)$, $\vec{p^7} = (5,3,7)$, and $\vec{p^9} = (4,4,9)$ are modeled by the polyhedron $\mathcal{D}_3 = \{[i,j,k] : 2 \leq j \leq 4 \wedge k = 2j + 1 \wedge i = -j + 8\}$. Finally, the polyhedron $\mathcal{D}_4 = \{[i,j,k] : 2 \leq i \leq 3 \wedge j = i \wedge k = 3i - 3\}$ captures points $\vec{p^3} = (2,2,3)$ and $\vec{p^6} = (3,3,6)$. The complete reconstructed trace is the collection of polyhedra D_1, D_2, D_3, D_4, which describe exactly the nine points from the original reorderable trace. Note that some polyhedra may capture only a single point, which is the worst-case solution for the components of the reconstructed trace. Also, note that this set of polyhedra is only one possible solution for reconstructing this trace.

Example 3.9. As in the last example, consider the trace of points shown in Figure 3.1. We now describe an alternate reconstructed trace that captures exactly the 9 points in the original trace. We again assume the original trace to be reorderable, so that we can again group sets of non-consecutive points in the same polyhedra. As before, we model the two points $\vec{p^1} = (1,1,1)$ and $\vec{p^2} = (4,1,2)$ with the polyhedron $\mathcal{D}_1 = \{[i,j,k] : 8j = 1 \wedge 1 \leq k \leq 2 \wedge i = 2k\}$. Next, the two points $\vec{p^5} = (6,2,5)$ and $\vec{p^6} = (3,3,6)$ are modeled by the polyhedron $\mathcal{D}_2 = \{[i,j,k] : 2 \leq$

$i \leq 3 \wedge j = i \wedge k = 3i - 3\}$. The polyhedron $\mathcal{D}_3 = \{[i, j, k] : 5 \leq i \leq 6 \wedge j = 3 \wedge k = i + 2\}$ captures exactly the two points $\vec{p^7} = (5, 3, 7)$ and $\vec{p^8} = (6, 3, 8)$. We now model the single point $\vec{p^3} = (2, 2, 3)$ using the polyhedron $\mathcal{D}_4 = \{[i, j, k] : i = 5 \wedge j = 2 \wedge k = 4\}$. We also model the single point $\vec{p^4} = (5, 2, 4)$ using the polyhedron $\mathcal{D}_5 = \{[i, j, k] : i = 5 \wedge j = 2 \wedge k = 4\}$. Finally, the polyhedron $\mathcal{D}_6 = \{[i, j, k] : i = 4 \wedge j = 4 \wedge k = 9\}$ captures the last remaining point in the trace, $\vec{p^9} = (4, 4, 9)$. The complete reconstructed trace is now the collection of polyhedra D_1, D_2, D_3, D_4, which again describe the nine points from the original reorderable trace. This set of polyhedra provides another possible solution for reconstructing this trace. However, some single points are represented as a single polyhedra, which is not typically the ideal solution since compaction is not achieved. In the worst case regarding compaction criteria, we can always represent a trace of n points with n polyhedra, one for each point.

Finally, we ensure that the vertices and extents and strides of all polyhedra are explicitly defined when we reconstruct the trace into polyhedra. This enables code generation to be performed by scanning the data structures obtained representing polyhedra. Loop-based code can be generated by scanning the vertices of each polyhedra using our implemented code generator that performs a simple direct scan of the data structures for the polyhedra. Alternatively, code could be generated using a code generator such as CLooG. The integer points in the polyhedra represent iterations of the generated loop nests. A loop is created for every dimension of the polyhedra so that the executed code scans each of the integer points in the polyhedra.

3.1 Pattern Matching Approach

We now introduce the pattern matching approach and explain its inherent differences from geometric reconstruction using the Trace Reconstruction Engine (TRE). As described in depth in Chapter 4, we have that TRE operates within a limited window when reconstructing a trace and consumes consecutive points from the trace. The form of its solution is not known until it has reconstructed all of the points, whereas the pattern matching approach begins with defined shapes and attempts to find them in the trace of points. TRE also places emphasis on compaction, so there are instances where points are grouped together which cannot be vectorized. Also, points that are grouped together could have poor locality; for example, points that are executed consecutively but are stored distantly in memory.

In order to overcome the inherent challenges of using TRE, we use a pattern matching approach. This pattern matching approach can be used together with TRE or as an independent approach. Both of these options are described in detail in this chapter. The pattern matching approach requires that the input trace of values be a reorderable trace as defined previously. This is required because the pattern matching approach must be able to group points at arbitrary distances in the trace. The trace must be reorderable due to the pattern matching approach not operating in the same sequential window that TRE does.

3.2 Description of Micro-Codelets

The process takes as input a defined family of template shapes to be mined for in the reorderable input trace of values. These template shapes are partially defined $\mathcal{Z}$-polyhedra of specific shapes. The specific shapes are defined with the goal of enabling SIMD vectorization opportunities in the generated code; however, this is not a strict requirement and any hyper-rectangle shape can be listed. Below, we show template shapes which model traces of three-dimensional tuples applicable to reconstructing traces for three-dimensional sparse tensors. The three-dimensional templates depicted below can be easily generalized to handle shapes of arbitrary dimension.

The process mines for hyper-rectangles of arbitrary size within a given range of values. Additionally, the process allows for constant strides between points to be modeled. Thus, the prototype polyhedron shape being mined for in the three-dimensional case is: $\mathcal{D}_{3Dcodelet} = \{[i, j, k] : m_i \leq i \leq M_i \wedge m_j \leq j \leq M_j \wedge m_k \leq k \leq M_k\}$. The prototype integer lattice being mined for in the three-dimensional case is: $F_{3Dcodelet} = \{[i, j, k] \rightarrow [i', j', k'] : i' = s_i i \wedge j' = s_j j \wedge k' = s_k k\}$. The unknowns of these equations are obtained by mining the input trace. The unknowns defining the vertices of the polyhedron in the three-dimensional case are the variables $m_i, M_i, m_j, M_j, m_k, M_k \in \mathbb{N}$. The additional unknowns define the constant stride permitted between points in the $\mathcal{Z}$-polyhedron. These unknowns representing stride are the variables $s_i, s_j, s_k \in \mathbb{N}$.

If there is one dimension representing the coordinates of a data vector in the sparse representation; e.g., the third, $data$, dimension for SpMV, then by construction, this dimension is monotonically increasing. Thus, we must rely on building a function specifically for capturing the coordinates of this dimension. This is achieved by using an affine function of the other dimensions being modeled. For SpMV, this means a two-dimensional reconstruction must occur, with an affine function of the i and j dimensions used to capture the third, $data$, dimension.

Therefore, by construction, when operating on sparse tensors in the three-dimensional case, the polyhedron origin is $\vec{p_{orig}} = (m_i, m_j, m_k)$. The origin of the polyhedron is a key component to the reconstructed trace, because from the origin point and pre-defined shape, we are able to recover all of the original points that are modeled by the polyhedron. Thus, straightforward code generation is enabled, achieved by scanning all of the origin points, as described in Chapter 7.

3.3 Micro-Codelet Mining Algorithm

As described in Section 3.2, a set of template shapes is defined, which capture all rectangular shapes with constant stride. Now, the input trace of values can be mined to identify all places where the template shapes can be applied to model groups of points in the input trace. The trace is generally mined with priority placed on the largest rectangle shapes; i.e., those shapes which capture the largest number of points will be searched for with higher priority, down to the smallest rectangle shapes. However, since the user defines the specific order that shapes are mined for, this is not a strict requirement and may be disregarded. This requirement can especially be partially disregarded if the goal of finding larger shapes first conflicts with the goal of placing priority on identifying shapes which are amenable to SIMD-vectorization; i.e., rectangles whose size is a multiple of the vector length in one or more dimensions. These shapes can be given priority over other shapes which do not lead to vectorization opportunities. Thus, in practice, the mining process follows other orderings than simply the size of the rectangles from largest to smallest with a continuous decrease in the number of points captured due to the additional criteria of mining to enhance vectorizability. Also, the rectangles are typically mined with priority on smallest stride to largest stride; that is, the distance between points fitting the template rectangle shape.

Due partially to the high complexity of the base algorithm, in practice, the extent of the largest shapes explored is typically limited to 5 in each dimension, also with constant stride between points valued up to 5 in each dimension. However, this is not a strict requirement and the maximum extent and stride can be adjusted higher. Additionally, shapes containing less than two points are not listed as input templates. These single points that are not able to be grouped into any of the specified template shapes are handled as a special case, and are grouped together as “left over” points. Thus, the entire trace will always be reconstructed, with points not fitting into the candidate shapes still being captured in polyhedra of single points. We show in Algorithm 1 the complete algorithm for performing micro-codelet mining.

Algorithm 1 Micro-codelet mining

Input: Trace T, Msz_i, Msz_j, Msz_k the maximal polyhedron sizes, Mst_i, Mst_j, Mst_k the maximal strides

Output: List of $\mathcal{Z}$-polyhedra L

```
1: procedure $CodeletMiner$(Trace $T$, $Msz_i, Msz_j, Msz_k$ the maximal polyhedron sizes, $Mst_i, Mst_j, Mst_k$ the maximal strides)
2:     $L$ = emptyList
3:     $T_{reord}$ = lexicographicSort($T$)
4:     for $s_i$ in $[Msz_i..1]$, $s_j$ in $[Msz_j..1]$, $s_k$ in $[Msz_k..1]$,$st_i$ in $[1..Mst_i]$, $st_j$ in $[1..Mst_j]$, $st_k$ in $[1..Mst_k]$ do
5:         $\vec{p}$ = firstPointInTrace($T_{reord}$)
6:         $m_i = p_1,\ m_j = p_2,\ m_k = p_3$
7:         $M_i = m_i + s_i,\ M_j = m_j + s_j,\ M_k = m_k + s_k$
8:         $S = \emptyset$ invalid = false
9:         for i in $[m_i..M_i]$, j in $[m_j..M_j]$, k in $[m_k..M_k]$ do $\vec{p}_{searched} = (i * st_i, j * st_j, k * st_k)$
10:            if ExistsInTrace($\vec{p}_{searched}$, $T_{reord}$) then
11:                $S = S \cup \vec{p}_{searched}$
12:            else
13:                invalid = true
14:                break
15:        if invalid = false then
16:            $T_{reord}$ = removePointsFromTrace($T_{reord}$,$S$)
17:            $L$ = append($L$, ZpolyhedronFromVertices($m_i, M_i, m_j, M_j, m_k, M_k, st_i, st_j, st_k$))
18:    return $L$
```

3.4 Geometric Compression of Micro-Codelets

The previous sections described how to reconstruct a reorderable trace using polyhedra, using either the TRE approach as described thoroughly in the preceding Chapter 4 or the codelet-centric pattern matching approach described in Section 3.3. The final result using either of these techniques is the same. That is, a fully functional program can be synthesized by scanning the polyhedron which partition the input trace into a reconstructed trace.

In this section, we provide details of hierarchical reconstruction with the goal of finding ways to combine micro-codelets together under two separate approaches. The first approach combines both the power of multidimensional geometric reconstruction through TRE and the micro-codelet reconstruction technique by identifying micro-codelets, then calling TRE on the traces of micro-codelet origin points. The second approach, deemed macro-codelet mining, recursively extends the micro-codelet approach so that the micro-codelet mining algorithm gets called on the traces of codelet origins from the previous iteration.

3.4.1 TRE Used in Combination with Pattern-Matching

We now describe the first hierarchical reconstruction approach, which achieves trace reconstruction by applying TRE to the traces of micro-codelet origin points, thus identifying further compaction through TRE's geometrically guided analysis of micro-codelet origin points. The algorithm is essentially the exact same as the TRE algorithm that was previously described. However, instead of the input being a list of the matrix coordinates corresponding to nonzero elements, the input is now a list of the origin points for the micro-codelets that were found from the micro-codelet mining algorithm applied to the input trace. That is, the traces consist of the $\mathcal{Z}$-polyhedron origin points that were found by the pattern-matching process described in Section 3.3. New traces must be produced, one corresponding to the trace of origin points for each pre-defined micro-codelet shape. Then TRE is called on each of these traces of micro-codelet origins to reconstruct the traces of origin points.

TRE is used to reconstruct complex shapes beyond those fitting the hyper-rectangle criteria which are input to the micro-codelet mining algorithm. This is a useful property especially when applying TRE to the trace of leftover points; that is, those which were not able to be grouped together when performing micro-codelet mining. It is used to discover complex patterns that the macro-codelet mining procedure is not able to identify. However, the loop control for the solution generated by TRE may be complex compared to the codelet-centric approach, thus resulting in decreased performance.

3.4.2 Macro-Codelet Mining

This section now describes the second hierarchical reconstruction approach. This is the process of macro-codelet mining. It is a recursive extension of the micro-codelet mining algorithm. This process is achieved by searching for regularity in the micro-codelet origin points and consists of a recursive call of the micro-codelet mining algorithm, where we input the trace of micro-codelet origins for each shape to the micro-codelet mining algorithm. A polyhedron built from micro-codelet origins is referred to as a macro-codelet. In essence, this approach amounts to building a polyhedron of polyhedra.

The algorithm consists of calls to the micro-codelet mining algorithm. As with the hierarchical reconstruction technique using TRE, the inputs to the codelet mining algorithm are the traces made of the $\mathcal{Z}$-polyhedron origin points that were found by the pattern-matching process. Regarding input shapes to search for, we typically increase the strides between points in the pre-defined template shapes. This is done to increase the chances of grouping origins of micro-codelets together. Once hierarchical reconstruction has occurred, the macro-codelets are uniquely identified by their origins, together with their micro-codelet shape and macro-codelet shape. We show in Algorithm 2 the pseudocode for performing hierarchical reconstruction using macro-codelet mining so that two levels of codelet mining are performed, which is the number of levels used in practice when operating on sparse matrices. The algorithm can be easily extended to repeat the process on macro-codelet origin points; however, in practice it was observed that there were minimal opportunities to identify further regularity when reconstructing sparse matrices.

In practice, we use this macro-codelet mining approach prior to attempting hierarchical reconstruction with TRE. It is also a potential option to first use the macro-codelet approach, then allow TRE to group together micro-codelet origin points that were left over from not being able to be assembled together using the macro-codelet approach. Since the macro-codelet approach will result in a trace of zero or more leftover points for each micro-codelet shape, each of these traces of leftover points can either be reconstructed with TRE or executed as single statements.

Algorithm 2 Macro-codelet mining

Input: Trace T, micro-codelet maximal sizes, macro-codelet maximal sizes

Output: List of $\mathcal{Z}$-polyhedra L

```
1: procedure $HierarchicalMiner$($T$, micro- and macro-codelet max. sizes)
2:     $L$ = emptyList
3:     $L_{bottom}$ = CodeletMiner($T$, micro-codelet max. sizes)
4:     for each unique shape size $s$ in $L_{bottom}$ do
5:         $T^{s}_{origins}$ = buildTraceFromAllPolyOrigins($L$,$s$)
6:         $L_s$ = CodeletMiner($T^{s}_{origins}$, macro-codelet max. sizes)
7:         $L$ = append($L$,$L_s$)
8:     return $L$
```

Chapter 4

Geometric Reconstruction Examples

The Trace Reconstruction Engine (TRE) [1] operates on a sequence of memory addresses. These memory addresses can be composed of those accessed by loop-nests as well as single statements. It analyzes the trace element-by-element and is driven by the access stride between each pair of consecutive points. In contrast to the codelet-centric approach, which mines the input for pre-defined shapes, TRE is an exploratory approach in that it works sequentially to model the trace. It incrementally constructs a partial solution that models a growing section of the trace. TRE sometimes backtracks if the partial solution it had been developing no longer is able to incorporate new points from the trace. TRE models a trace of memory accesses by grouping points into polyhedra. In essence, TRE operates by starting with an empty $\mathcal{Z}$-polyhedron, which is the intersection between an integer polyhedron and an integer lattice. It attempts to enlarge the cardinality of the $\mathcal{Z}$-polyhedron by appending sequential points from the input trace of memory accesses. The engine reconstructs affine references, which when executed, enact the exact trace of memory accesses which were input. We now provide details for the solution constructed by TRE.

Definition 4.1. (Convex hull). The convex hull of a set of points is the smallest convex set that contains all of the points in the set.

Definition 4.2. (Program memory trace). The list of all memory addresses accessed during the execution of the program.

The input to TRE is a program memory trace, which we denote as $\mathcal{A}$. TRE iteratively constructs a solution, which is denoted as $\mathcal{S}_n^N$, that generates the program memory trace of N elements using n nested loops. The solution generated by TRE consists of four components. First, it contains a vector $\vec{c}$, which consists of loop index coefficients. That is, $\vec{c}$ is the vector projected onto the vector of loop iterators such that the coefficients of $\vec{c}$ get multiplied by the surrounding loop iterators, thus determining the memory access being performed.

The next two components to the solution provided by TRE are the bounds matrix U and the bounds vector $\vec{w}$. In the original version of TRE, we had that the bounds matrix U is square; that is, $U \in \mathbb{Z}^{n \times n}$, where n is the number of loop dimensions which were reconstructed by TRE. In this original representation, each row j of U, $0 \leq j < n$, stores the coefficients of the loop indices regarding the upper bound of that loop dimension's iterator. This is represented in the form $i_j \leq u_j(i_1, \ldots, i_{j-1})$, where u_j is an affine function being applied to the surrounding loop iterators. Only the upper bounds contained an explicit encoding. For the lower bounds, an implicit encoding was used, such that for each row j of U, we had the implicit bound of $i_j \geq 0$.

However, in the extended version of TRE that is used in our work, both the upper and lower bounds faces in the U matrix are explicitly encoded. The lower bounds faces are now of the form $i_j \geq l_j(i_1, \ldots, i_{j-1})$, where l_j is an affine function of the surrounding loop iterators. Thus, the bounds matrix is no longer square. Instead we have $U \in \mathbb{Z}^{F \times n}$, where F is the number of faces of the iteration polyhedron.

Similarly, in the original TRE, each of the n components of the bounds vector $\vec{w}$ contains the offset for this affine upper bound. In the extended TRE which we use, lower bounds are also explicitly encoded, so there are F components of the bounds vector $\vec{w}$, where F is again the number of faces of the iteration polyhedron. From the solution of matrix U and vector $\vec{w}$ generated by TRE, we obtain an iteration polyhedron of the form $Ui + \vec{w} \geq 0$.

The final component of the solution generated by TRE is the matrix $I^N = [\vec{i^1} \ \vec{i^2} \ \ldots \ \vec{i^N}]$ of iteration indices; however, this matrix can be regarded as a byproduct of the reconstruction process, since we can obtain the iteration points in the convex hull of the iteration polyhedron from the bounds matrix and bounds vector. The original input trace is recovered when the vector of iteration coefficients, $\vec{c}$, is multiplied by the iteration polyhedron.

The solution generated by TRE must adhere to two properties to be considered valid. First, each consecutive pair of reconstructed indices $\vec{i^k}$ and $\vec{i^{k+1}}$ must be lexicographically sequential. That is, $\vec{i^{k+1}}$ follows $\vec{i^k}$ lexicographically. Thus, a strict address order is forced on the accesses that are reconstructed. This condition is stronger than the condition that the accesses must be within the bounds of the iteration polyhedron. The second required condition is that the strides between consecutive points in the solution TRE reconstructs are consistent with the observed strides between the corresponding points in the input trace. This condition ensures that the trace TRE reconstructs is correct regarding the values in the input trace. Put formally, this requirement is that $\vec{i^k} \cdot \vec{c} = a^k$, where $\vec{i^k}$ is the iteration vector for the k^{th} loop iteration and a^k is the k^{th} access in the input trace.

4.1 Simple Geometric Reconstruction Example

We now describe the process of TRE regarding its construction of the solution $\mathcal{S}_n^N$, to represent a one-dimensional trace of N memory accesses $\mathcal{A}$ using n nested loops. Consider the following code example in Figure 4.1 which copies the eight elements of array A into the eight locations of array y. The memory trace generated by the access by array A on line four is $\mathcal{A} = \{0, 1, 2, 3, 4, 5, 6, 7\}$. Technically, the addresses will be in a hexadecimal format, but for clarity, we normalize these addresses into integer values. This normalization does not affect the process done by TRE, since the solution it generates for a stream of memory addresses is normalized in this same way. The final solution TRE generates in practice must be offset by the base address of the trace of memory addresses in order to reconstruct the exact stream of accesses.

```
1    #define N 8
2    double y[N], A[N];
3    for (i = 0; i < N; i++)
4      y[i] = A[i];
```

Figure 4.1: Source code for the application being analyzed by TRE. The access of array A on line four is the statement which generates a stream of hexadecimal addresses, which are then reduced to the integer values of $\mathcal{A} = \{0, 1, 2, 3, 4, 5, 6, 7\}$ for clarity of analysis. The trace $\mathcal{A}$ will then be the input to TRE for reconstruction.

To model the complete trace $\mathcal{A}$, TRE first constructs an initial solution that models the subtrace $\{a_1, a_2\} = \{0, 1\}$. It then incrementally enlarges this solution by processing and incorporating consecutive points from the complete trace $\mathcal{A}$. It continues incorporating points until the entire trace can be modeled by its generated solution. To construct the initial partial solution that incorporates points a_1 and a_2, TRE first computes the observed access stride between the first two points in the trace $\sigma^1 = a_2 - a_1 = 1$. TRE then determines which iteration index to append to the matrix of iteration indices I so that the generated access stride is coherent with the observed access stride σ^1. The viable solutions correspond to the various potential increments of the iteration indices. That is, an iteration index can iterate by incrementing its value at any one of its coordinates by one, and resetting all coordinates which are inner the incremented coordinate to zero. Alternatively, an iteration index can be modified by inserting a new loop at each possible nesting level. Of these options, the only considered solutions are those which adhere to the previously described conditions of indices being lexicographically sequential and having a reconstructed stride coherent with the observed stride. In the case that at least one of these options is acceptable under these conditions, we compute the next iteration index $\vec{i^{k+1}}$ by adding the stride solution computed by TRE to the iteration index $\vec{i^k}$.

In our example, this leads to the following initial partial solution being constructed which models two points using a single loop: $\mathcal{S}_1^2 = \{\vec{c} = [1], I^2 = [0\ 1], U = [\begin{smallmatrix} 1 \\ -1 \end{smallmatrix}], \vec{w} = [0, 1]\}$. This solution is equivalent to the following loop code being generated, shown in Figure 4.2.

```
for (i1 = 0, i1 <= 1; i1++)
  printf ("%d\n", 0 + 1 * i1);
```

Figure 4.2: The loop-code equivalent of the partial solution $\mathcal{S}_1^2$ that TRE has constructed which models the first two points in the trace $\mathcal{A}$ using one loop.

In the general case, at the point that TRE processes memory access a_{k+1}, it computes the observed stride $\sigma^k = a_{k+1} - a_k$ to determine if the memory access a_{k+1} can be added to the solution. If TRE is not able to generate a solution that aligns with the observed stride σ^k from the trace and access a_{k+1} cannot be added to the partial solution it had been developing, one of three things occur. First, the solution dimensionality can always be increased to incorporate the new access. Second, the bounds matrix U and bounds vector $\vec{w}$ are modified. This occurs in the case that TRE selects a solution with stride not matching the observed stride, and a different solution must be selected. This scenario requires an adjustment of the bounds matrix and bounds vector to ensure the next iteration index is lexicographically sequential to the previous iteration index. The final alternative is that TRE must backtrack and discard the path it had previously taken to incorporate points. It then attempts other partial solutions to incorporate the new access so that later points can be successfully incorporated without continued dimensionality increases.

Returning to the example, upon processing the third point, TRE calculates the observed access stride between accesses a_2 and a_3, which is $\sigma^2 = a_3 - a_2 = 1$. The observed stride between this next pair of accesses remains the same as the observed stride between the previous two points. TRE is able to incorporate the new point so that the solution grows to handle three points without requiring an adjustment of the solution dimensionality. The loop body simply gets expanded to iterate one more time, but the dimensionality of the solution stays the same. The solution is now $\mathcal{S}_1^3 = \{\vec{c} = [1], I^3 = [0\ 1\ 2], U = \left[\begin{smallmatrix}1\\-1\end{smallmatrix}\right], \vec{w} = [0, 2]\}$ The loop is now of the following form, shown in Figure 4.3.

```
for (i1 = 0, i1 <= 2; i1++)
    printf("%d\n", 0 + 1 * i1);
```

Figure 4.3: The loop-code equivalent of the partial solution $\mathcal{S}_1^3$ that TRE has constructed which models the first three points in the trace $\mathcal{A}$ using one loop.

Since all pairs of consecutive points in the complete trace $\mathcal{A}$ have the same access stride of 1, TRE incrementally processes each point in this same way. TRE is able to successfully expand the solution to incorporate each of the new points by extending the number of iterations of the single loop. Since the stride between points remains the same throughout the trace, TRE does not require an increase of the solution dimensionality and it does not require backtracking. Eventually, once the final point in the trace is processed, TRE has obtained a single-dimensional solution that accounts for all eight points from the original trace $\mathcal{A}$. The final loop body that is generated by TRE after it processes the final memory address is of the form of Figure 4.4. The loop corresponds to the solution $\mathcal{S}_1^8 = \{\vec{c} = [1], I^8 = [0\ 1\ 2\ 3\ 4\ 5\ 6\ 7], U = [\,{}^{1}_{-1}\,], \vec{w} = [0, 7]\}$.

```
for (i1 = 0, i1 <= 7; i1++)
   printf ("%d\n", 0 + 1 * i1);
```

Figure 4.4: The final loop code generated for the trace $\mathcal{A}$ by TRE.

4.2 Complex Geometric Reconstruction Example

We now consider a more complex example of the reconstruction process done by TRE. The following analysis involves TRE constructing a solution that requires increasing the solution dimensionality during reconstruction to model the complete trace. We describe the process of reconstructing the access of array A, A[i][k] on line 12 of Figure 4.5. The complete memory trace of this access is shown in Figure 4.6.

```
#define N 8
double A[N][N];
for (i = 0; i < N; ++i) {
  for (j = 0; j < i; ++j) {
    for (k = 0; k < j; ++k) {
      A[i][j] -= A[i][k] * A[k][j];
    }
    A[i][j] /= A[j][j];
  }
  for (j = i; j < N; ++j) {
    for (k = 0; k < i; ++k) {
      A[i][j] -= A[i][k] * A[k][j];
    }
  }
}
```

Figure 4.5: Source code for the application now being analyzed by TRE.

1:	0	35:	24	66:	40
2:	0	36:	25	67:	41
⋮	⋮	37:	26	68:	42
8:	8	38:	27	69:	43
9:	9	39:	24	70:	44
10:	8	40:	25	71:	45
11:	9	41:	26	72:	40
⋮	⋮	42:	27	73:	41
20:	16	⋮	⋮	74:	42
21:	17	51:	32	75:	43
22:	18	52:	33	76:	44
23:	16	53:	34	77:	45
24:	17	54:	35	78:	48
25:	18	55:	36	79:	49
⋮	⋮	⋮	⋮	80:	50
				81:	51
				82:	52
				83:	53
				84:	54

Figure 4.6: Program memory trace for the access of A[i][k] on line 12 of the application in Figure 4.5, which we refer to as $\mathcal{A}$.

In the same manner as the first example, TRE processes memory accesses sequentially. TRE first constructs a partial solution that models the first two elements in the trace. That is, the first partial solution models the subtrace $\{a_1 = 0, a_2 = a_1\}$. TRE first calculates the observed access stride σ^1 between the first two elements in the trace, $\sigma^1 = a_2 - a_1 = 0$. The partial solution obtained by TRE is $\mathcal{S}_1^2 = \{\vec{c} = [0], I^2 = [0\ 1], U = [\,{}^{\ 1}_{-1}\,], \vec{w} = [0, 1]\}$. This solution is modeled by the following loop shown in Figure 4.7 which simply iterates over the first two points in the memory trace $\mathcal{A}$.

```
for (i1 = 0, i1 <= 1; i1++)
  printf ("%d\n", 0);
```

Figure 4.7: The loop-code equivalent of the partial solution that TRE has constructed which models the first two points in the trace $\mathcal{A}$.

TRE continues its sequential processing of elements. The observed stride between each pair of consecutive elements remains zero until the access at line eight in the memory trace, a_8, is reached. At this point, TRE has constructed a one-dimensional solution that incorporates the first seven elements of the memory trace. The solution is now $\mathcal{S}_1^7 = \{\vec{c} = [0], I^7 = [0\ 1\ 2\ 3\ 4\ 5\ 6], U = [\,{}^{\ 1}_{-1}\,], \vec{w} = [0, 6]\}$, whose corresponding loop is of the form shown in Figure 4.8.

```
for (i1 = 0, i1 <= 6; i1++)
  printf ("%d\n", 0);
```

Figure 4.8: The loop-code equivalent of the partial solution that TRE has constructed which models the first seven points in the trace $\mathcal{A}$.

Upon processing a_8, the observed access stride now changes to $\sigma^7 = a_8 - a_7 = 8 - 0 = 8$. The loop body that TRE has constructed with $\vec{c} = 0$ cannot handle a stride other than 0. TRE verifies this by computing the solution $\vec{\delta^7} = \vec{i^8} - \vec{i^7}$. Since the current solution is only one-dimensional, there is only one way TRE can expand the loop without increasing dimensionality. It obtains $\vec{i^8}$ by adding 1 to the single component of $\vec{i^7} = [6]$ to obtain $\vec{i^8} = [7]$. Thus, we have a solution of $\vec{\delta^7} = 1$. TRE verifies that this solution does not produce a stride coherent with the observed stride by taking

$\vec{c} \cdot \vec{\delta^7} = 0 \cdot 1 = 0$. Thus, TRE is unable to obtain the observed stride of 8 without increasing the dimensionality of its partial solution.

Thus, it has been determined that the subtrace $\{a_1, \ldots, a_8\}$ cannot be generated using a single-level loop. The dimensionality of the solution must be increased, so the partial solution changes from being a one-dimensional solution that incorporates seven points to a two-dimensional solution that incorporates eight points. Two increase the dimensionality of the solution from one to two, there are two possible locations to insert the newly added loop. That is, either above or below the current loop. Heuristically, the more common solution is that newly discovered loops are outer the previously constructed loops, so TRE first explores adding a new loop at the outer level.

Hence, after increasing dimensionality by adding a loop at the outer level 0, we obtain a solution of the following form. The iteration domain I^7 is expanded from having one row to having two rows since there are now two nesting levels. A row of zeros gets appended to the top of the I^7. $\vec{i^8}$ gets appended to the right-hand side of the I^7 matrix to obtain I^8. Since we are adding a loop at level 0, $\vec{i^8}$ is computed by setting its outer component to 1 and all inner values to 0. In general, when a loop is inserted at level p, the new iteration vector $\vec{i^{k+1}}$ is computed from $\vec{i^k}$ by inserting a value of 1 at level p and resetting all inner components to 0, while keeping all outer indices the same as they were in $\vec{i^k}$. The iteration domain is now of the following form.

$$I^8 = \begin{bmatrix} 0 & 0 & 0 & \ldots & 0 & 1 \\ 0 & 1 & 2 & \ldots & 6 & 0 \end{bmatrix}$$

Additionally, the vector of index coefficients $\vec{c}$ must be transformed to handle the new dimension and stride. The new component of $\vec{c}$ is calculated as $\sigma^7 + i_1^7 \cdot \vec{c} = 8 + 6 \cdot 0 = 8$ so that we now have $\vec{c} = [8\ 0]$. In general, when the dimensionality of the solution is increased by adding a loop at level p, the new component c_p' of $\vec{c}$ can be obtained by using the formula $c_p' = \sigma^k + \sum_{r=p+1}^{n} i_r^k \cdot c_r$. Thus, after expanding the dimensionality to incorporate point a_8, the solution is now of the form $\mathcal{S}_2^8 = \{\vec{c} = [8\ 0], I^8 = \left[\begin{smallmatrix} 0 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 1 & 2 & \cdots & 6 & 0 \end{smallmatrix}\right], U = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ -6 & -1 \end{bmatrix}, \vec{w} = [0, 1, 0, 6]\}$. This solution corresponds to the following loop nest being generated, shown in Figure 4.9.

```
for (i1 = 0, i1 <= 1; i1++)
  for (i2 = 0, i1 <= 6 − 6 ∗ i1; i1++)
    printf ("%d\n", 0 + 8 ∗ i1);
```

Figure 4.9: The loop-code equivalent of the partial solution that TRE has constructed which models the first eight points in the trace $\mathcal{A}$.

At the point that TRE processes point a_9, the observed stride changes to $\sigma^8 = a_9 - a_8 = 9 - 8 = 1$. Since the partial solution is now two-dimensional, there are two possible ways to incorporate a_9 without increasing the dimension of the solution to three dimensions. That is, the next iteration index $\vec{i^9}$ can be formed from $\vec{i^8}$ by adding one to either of the components of $\vec{i^8}$ and resetting inner indices to zero. These correspond to either $\vec{i^9} = [2\ 0]$ or $\vec{i^9} = [1\ 1]$. However, neither of these options are able to achieve a stride of $\sigma^8 = 1$, so the solution's dimensionality must be increased. Since we are now operating on a two-dimensional partial solution, we now have three options for which nesting level to add a new loop. That is, a new loop can be inserted at level 0, 1, or 2.

TRE attempts to insert a new loop at outer levels first in the same way as when the solution grew to two-dimensions. However, for an insertion at level 0 and 1, TRE builds a solution that is not able to incorporate the next points in the trace. We assume TRE does not continually add new dimensions and instead backtracks and attempts an insertion at level 2 to incorporate point a_9. The new $\vec{c}$ is calculated to be $\vec{c} = [8\ 0\ 1]$ and we form the iteration domain by appending a row of zeros to the bottom of the matrix and appending $\vec{i^9}$ on the right-hand side. Thus, the iteration domain is now of the following form.

$$I^9 = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 & 1 & 1 \\ 0 & 1 & 2 & \dots & 6 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 \end{bmatrix}$$

TRE continues processing the remaining points in the trace. The solution continues to grow, with points gradually being added to its $\mathcal{Z}$-polyhedron without requiring an increase in dimensionality of the solution.

In Algorithm 3, we see the general form of TRE's reconstruction process. This is an extremely simplified version used to illustrate the concept of TRE; refer to [1] for the complete version of the algorithm which is implemented in the tool.

Algorithm 3 Simplified TRE Pseudocode

```
Input: trace $T$
Output: $\mathcal{Z}$-polyhedron modeling points in trace $T$
1: procedure TRE(trace $T$)
2:     for each point $a^{k+1}$ in the trace $T$ do
3:         compute observed stride $\sigma^k = a^{k+1} - a^k$
4:         generate $\vec{i^{k+1}}$ lexicographically sequential to $\vec{i^k}$
5:         compute $\vec{\delta^k} = \vec{i^{k+1}} - \vec{i^k}$
6:         if $\vec{c} \cdot \vec{\delta^k} = \vec{\sigma^k}$ then
7:             append $\vec{i^{k+1}}$ to $I$
8:         else
9:             increase solution dimensionality to incorporate $a^{k+1}$
```

4.3 Complexity Trade-Offs

There exists an important trade-off in the reconstruction process of TRE between the number of pieces it reconstructs and the maximum dimensionality of the polyhedra. It is always possible to rebuild a set of integer points as a set of polyhedra. In an extreme worst case, we have that each individual point is captured in a polyhedron of cardinality one, but this is useless in practice since it results in single-statement code being emitted and no loops generated. Also, this would result in no compaction improvement. The dimensionality of a piece corresponds to how many variables are required to model the sequence of values. A series of points which cannot be modeled by a two-dimensional loop-nest may be able to be modeled by a three-dimensional loop-nest. For example, the points $1, 3, 5, 7$ can be modeled by the one-dimensional affine function $f(i) = 2i + 1 : 0 \leq i \leq 3$. However, the sequence of points $1, 3, 7, 9$ cannot be modeled by a one-dimensional affine function. Instead, a two-dimensional affine function $f(i, j) = 6i + 2j + 1 : 0 \leq i \leq 1 \wedge 0 \leq j \leq 1$ is able to model these four points.

The solution that TRE constructs when operating on its version of a reconstruction with multiple pieces can be bounded by the maximum allowed dimensionality of the pieces it builds. For example, a maximum allowed dimension of 8 would allow an eight-dimensional loop-nest to be created. Contrarily, a maximum dimension of 1 would allow only single-level loop-nests to be formed, which would typically result in a larger number of separate loop-nests being generated so that all the points from the trace could be modeled. In practice, both extreme scenarios typically result in a decrease in performance. This trade-off is clearly shown by the matrix HB/nos1 from the SuiteSparse collection, which consists of 1,017 nonzero elements. The entire matrix can be modeled using a single eight-dimensional polyhedron, but the execution time dramatically increases due to the loop control required to model it using a highly complex shape. We show in Table 4.1 the different reconstruction choices corresponding to different allowed maximum dimensionalities for HB/nos1. We show reconstruction statistics for reconstruction in the form of a single eight-dimensional polyhedron down to 312 disjoint pieces with each piece being bounded to be two-dimensional.

$\max_d$	2	3	4	5	6	7	8
pieces	312	159	81	4	3	2	1
cycles	11373	11583	9938	35730	34116	39306	50371
LoC	772	1004	671	195	368	165	101

Table 4.1: The evolution of number of pieces as a function of their maximum allowed dimensionality for matrix HB/nos1 from the SuiteSparse repository.

We observe a clear sparsity pattern when observing the zoomed-in view of the pattern of nonzero elements for HB/nos1 in Figure 4.10 [18], which consists of elements located near the main diagonal. In contrast to this clear sparsity pattern, the matrix HB/can_1072 from the SuiteSparse repository is shown in Figure 4.11 [18]. The matrix HB/can_1072 is a 1,072 by 1,072 sparse matrix consisting of 12,444 nonzero elements. This lack of evident regularity results in many disjoint complex loop nests being required to model the matrix. Thus, the reconstructed code consists of 870 disjoint pieces with dimensionality of the generated loop nests being up to 8.

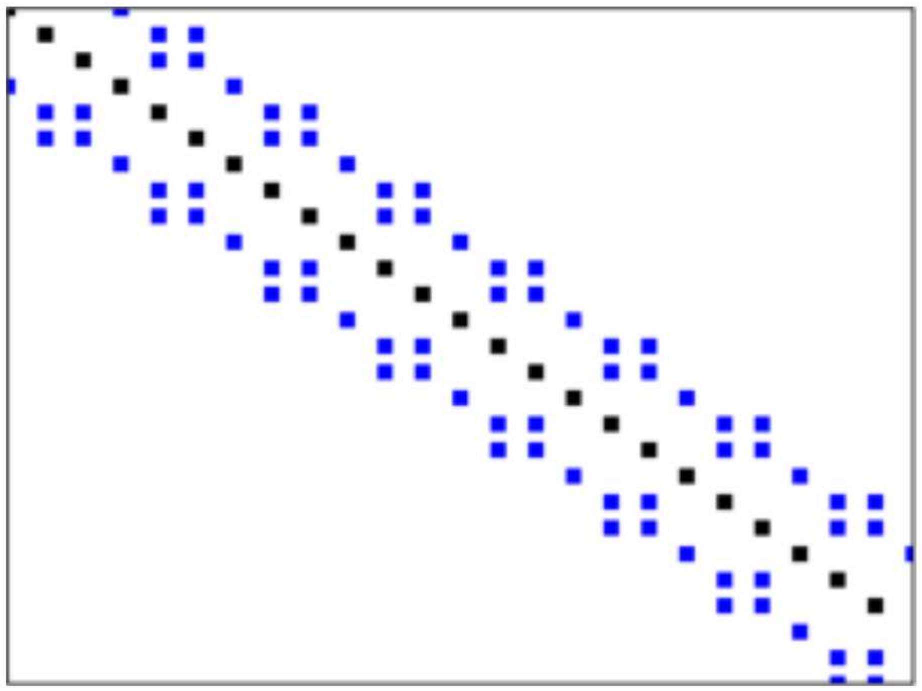

Figure 4.10: The sparsity pattern for the matrix HB/nos1 when zooming in on its main diagonal. [18]

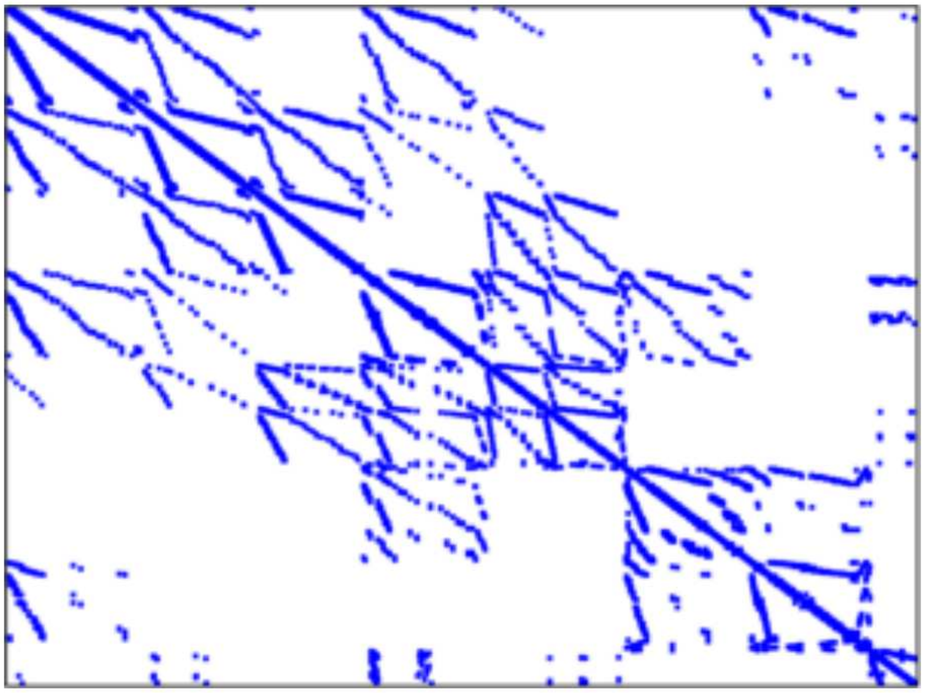

Figure 4.11: The nonzero elements in HB/can_1072, a 1,072 $\times$ 1,072 matrix consisting of 12,444 nonzero elements which does not exhibit any apparent regularity. The reconstructed code for this matrix consists of 870 pieces of up to 8 dimensions. [18]

Chapter 5

Implementation Details

5.1 Parsing the Sparse Input Matrix

We now describe how the structure of a sparse matrix is captured by polyhedra. We show the process of mining the sparse input matrix for regularity and converting its structure into a union of polyhedra describing the matrix. We first scan the input sparse matrix to emit a trace of all of the (i, j) coordinates that are the coordinates of nonzero elements in the matrix. This trace will be of length NNZ. In general, we take as input a matrix in any format, and scan its nonzero values to emit a trace of all of the nonzero (i, j) coordinates. The typical raw input is a trace of matrix coordinates from the SuiteSparse repository in Matrix Market form which is in coordinate format; that is, the matrix is represented as a list of three-dimensional tuples: $(i, j, value)$, where $value$ is the value of the nonzero element corresponding to the listed coordinates (i, j). This list of three-dimensional tuples is parsed prior to analysis into a new trace of length NNZ of three-dimensional tuples of the form: $(i, j, index)$, where $index$ is the row index of the tuple in the list of all three-dimensional tuples, so that the value of $index$ ranges from 1 to NNZ. Each tuple corresponds to one SpMV statement execution. When code is finally generated, as described in Chapter 7, the coordinate i is used to index into the y array, the coordinate j is used to index into the x array, and the coordinate $index$ is used to index into the A array. We now have a trace of the nonzero elements' coordinates. From this trace, we build polyhedra which capture exactly the points that exist in this input trace. The polyhedra must capture exactly all of the points in the trace and must not contain any extra points that were not in the input trace. This is achieved by matching groups of points from the trace to previously specified patterns defining polyhedra shapes.

5.2 Applying Reconstruction Technique

Now that the input matrix has been parsed into an analyzable form, we are able to apply one of the reconstruction techniques from Chapter 3 a reconstructed trace. Upon applying the micro-codelet mining technique, we now have lists of polyhedra corresponding to each template shape that were input. The polyhedra are each uniquely identified by their origin points and their corresponding template shape. This allows simple code generation to occur by scanning each of these origin points, as described in detail in Chapter 7.

In the specific case of generating code for the SpMV kernel, we mine for two-dimensional hyper-rectangles along the i-dimension and j-dimension. Once a codelet has been identified, we then verify that the third, k-dimension, is a linear combination of the i-dimension and the j-dimension. This is required because the traces which correspond to sparse traces from the SuiteSparse repository have NNZ unique values in the k-dimension, ranging from 0 to NNZ. Thus, there is no possibility for repeated values which are needed to form hyper-rectangle shapes in three dimensions.

Chapter 6

Acceleration Technique for Micro-Codelet Mining

The pattern matching approach for identifying micro-codelets we present suffers from the need to iterate through a large number of points to check for regular sets of points in the trace. To accelerate the process of the identification of groups of points in the trace we present an approach to decrease the total number of points checked when iterating through the input trace to find micro-codelets. The following approach relies on multiple versions of the input trace being stored. These trace versions correspond to the possible lexicographic orderings of the trace with regards to placing different priority on each of the different dimensions. These ordered traces adhere to the following definitions.

Definition 6.1. ($x_1, ..., x_n$-lexicographic order). Let $(a_1, ..., a_n), (b_1, ..., b_n) \in \mathbb{N}^n$. Then $(a_1, \ldots, a_n) \prec_{x_1, \ldots, x_n} (b_1, \ldots, b_n)$ if and only if $a_i < b_i$ for the first i where a_i and b_i differ.

Definition 6.2. ($x_1, ..., x_n$-lexicographic trace). $T = \{c_1, \ldots, c_n\}$ is an $x_1, \ldots, x_n$-lexicographic trace if $i < j \Rightarrow c_i \prec_{x_1, \ldots, x_n} c_j \forall i, j$.

For example, if a two-dimensional trace is being analyzed, we maintain two separate orderings. We order one trace to be an i, j-lexicographic trace and one trace to be a j, i-lexicographic trace. The i, j-lexicographic trace and j, i-lexicographic traces are specific two-dimensional cases of the previously defined $x_1, ..., x_n$-lexicographic trace. We maintain a pointer between each tuple under different orderings. We could have that a given point is located at a vastly different index under one ordering compared to another ordering, so the pointer between its location in each of the lexicographic ordered traces provides a means to check neighboring points in all dimensions without traversing a potentially huge number of indices to locate its neighboring points.

	i	j
1:	0	0
2:	0	3
3:	1	1
4:	0	4
⋮		
500:	0	1
	⋮	

Figure 6.1: A two-dimensional trace that does not adhere to an ordering system, and simply lists the coordinates of each point arbitrarily. This trace of points could be input to the codelet miner.

In contrast to our technique, suppose that we do not utilize any lexicographic ordering of the trace and we seek to find a 1×2 block of points in the input trace of Figure 6.1. We start by checking if the first point in this trace can be an origin point for a block of this size. Enumerating each point in the trace, we identify a match, the point $0, 1$, located at index 500, which is 499 points later in the trace. Thus, 500 points had to be traversed, and individually investigated to determine if they can be added to the origin point to form a block of the given shape.

	i	j
1:	0	0
2:	0	1
3:	0	2
4:	0	3
5:	0	4
6:	1	0
7:	1	1
8:	1	2
9:	1	3
10:	2	0
11:	2	1
12:	2	2
13:	2	3
⋮		
500:	800	0
	⋮	

Figure 6.2: A two-dimensional i, j-lexicographic trace, obtained by lexicographically sorting the trace in Figure 6.1.

Now suppose that we use a single lexicographic ordering of the trace such that the trace is an i, j-lexicographic trace, as shown in Figure 6.2. As before, we check if the first point in the reordered trace can be used as an origin point for a block of size 1×2. We then enumerate points in the trace, verifying if they may be combined with this origin point to form a complete micro-codelet. Once we reach a distance of 2, the block size in the j-dimension, we now know for certain that this codelet shape can be formed using the first two points in the trace. Next, the point $0, 2$ is checked to determine if a block of 1×2 can be formed with it as an origin. We know after checking the next point in the trace that a block of this size can again be formed with $0, 2$ as an origin. Next, the point $0, 4$ is checked to determine if a block of 1×2 can be formed with it as an origin. Again, once we have traversed a distance of 2, the block size in the j-dimension, we know for certain that this codelet shape cannot be formed using $0, 4$ as an origin, since $0, 5$ is the required point and if it exists in the trace, it must be directly after $0, 4$ in this ordering. We can determine this now without checking the 496 points remaining in the trace. Due to the lexicographic ordering, the points past this point in the trace are known to be too distant to form the micro-codelet.

	`i`	`j`
1:	0	0
2:	1	0
3:	2	0
4:	9	0
5:	0	1
6:	1	1
7:	2	1
8:	0	2
9:	1	2
10:	2	2
11:	0	3
12:	1	3
13:	2	3
⋮		
500:	0	900
	⋮	

Figure 6.3: The same trace of points as in Figure 6.1 and Figure 6.2 but reordered to be a j, i-lexicographic trace.

Similarly, the same approach can be applied to template shapes which have an extent greater than one in both the i-dimension and the j-dimension. Now suppose that we utilize two lexicographic orderings of the trace. Also, suppose we are now searching for codelet shapes of size 3×4; that is, shapes with extent of 3 in the i-dimension and extent of 4 in the j-dimension. As before, one trace is an i, j-lexicographic trace so that the non-unit extent along the j-dimension can be explored. Now, we add another trace shown in Figure 6.3, which is a j, i-lexicographic trace so that the non-unit extend along the i-dimension can be checked. Again, the process begins by attempting to identify blocks of this shape with the first point as the origin. We arbitrarily let the i, j-lexicographic trace be the trace defining the overall ordering of potential origin points to iterate through. However, the j, i lexicographic trace could be used for this master ordering, with potential changes in the types of codelet shapes found due to the use of a different ordering when attempting to form a block of a given size with the current observed point as an origin for the codelet.

Now, the exploration process is essentially the same as it was when the single lexicographic ordered trace was maintained, but for each point along a face found in the i-dimension, we must follow its pointer to the next lexicographic ordered trace. This allows us to check neighboring points along the j-dimension as well, without the need to exhaustively search a trace for the required points. For each of the points we follow, we need only to check the validity of points within a range of 3, the extent along the j-dimension, since all points later and before will be out of the required range.

To identify blocks of size 3×4, we start with the origin point $0, 0$. We then enumerate the next 4 points in the i, j-lexicographic trace to check for the existent of the four required points to form the first face of the polyhedron. We see that the required points $0, 1$, $0, 2$, and $0, 3$ all exist. So we now have identified four points that form the first face of the polyhedron. For each of these four points, we follow their pointer to their location in the j, i-lexicographic trace. First, we follow the pointer for $0, 0$ from its location in the i, j-lexicographic trace to its location in the j, i-lexicographic trace. We then check the subsequent 3 points in the j, i-lexicographic trace, which corresponds to the extent of the shape in the i-dimension. The required points $1, 0$ and $2, 0$ are quickly identified and added to the polyhedron. This same process is repeated for the remaining three points $0, 1$, $0, 2$, and $0, 3$ that were identified along the first face, and we find that the entire polyhedron can be formed by checking points which are local to these three points in the j, i-lexicographic trace.

Now that we are equipped with multiple lexicographic orderings, we are able to further accelerate the algorithm by not only checking within a limited window of points, but simply checking the point that is an offset of the extent in the given dimension away from the origin point. Consider the sequence of points in the ij-lexicographic trace in fig:accel:trace-ij. We have an origin point of $0, 3$ being scrutinized. Suppose the codelet extent is 5 along the i-dimension and the codelet stride is unit. This technique only works when polyhedra with unit stride are being searched for, since we require a dense sequence of points. Since we are utilizing the lexicographic order, we only must check the single point 5 points later in the trace to verify if a codelet is feasible with this origin point under this template shape. We now provide a theorem and proof that further explains this claim, as well as the necessary formal definitions.

First, we define our notation for identifying the location of a point within a lexicographic ordered trace.

Definition 6.3. ($id_{ij,(I,J)}$). $id_{ij,(I,J)}$ refers to the index of the tuple (I, J) within the i, j-lexicographic trace.

Definition 6.4. ($id_{ji,(I,J)}$). $id_{ji,(I,J)}$ refers to the index of the tuple (I, J) within the j, i-lexicographic trace.

We now define the $maxblock$ variables, which are used in the pre-processing phase to bound the number of points which are checked for consecutiveness. These variables are dependent upon the template shapes that are input to the program.

Definition 6.5. ($maxblock_j$). $maxblock_j$ is the maximum size of blocks searched for with a fixed i-value and j-values increasing by one for each consecutive point from the block origin.

Definition 6.6. ($maxblock_i$). $maxblock_i$ is the maximum size of blocks searched for with a fixed j-value and i-values increasing by one for each consecutive point from the block origin.

Once the $maxblock$ variables are obtained in each dimension, we are able to construct the values for the $block$ variables. There is a $block$ variable associated with every tuple in the input trace. These indicate how many consecutive points there are from each tuple in each lexicographic ordered trace. We provide definitions for the two-dimensional case.

Definition 6.7. ($block_{ij,(I,J)}$). For the coordinates (I, J), we have that $block_{ij,(I,J)} = k + 1$ if and only if the coordinates at $id_{ij,(I,J)} + k$ are $(I, J + k)$ and the coordinates at $id_{ij,(I,J)} + k + l$ are not $(I, J+k+l)$, for all l such that $k < l < maxblock_j$; that is, the coordinates k positions after (I, J) in the i, j-lexicographic ordered trace are $(I, J + k)$ and the coordinates $k + l$ positions after (I, J) in the i, j-lexicographic ordered trace are not $(I, J + k + l)$, for all l such that $k < l < maxblock_j$.

Definition 6.8. ($block_{ji,(I,J)}$). For the coordinates (I, J), we have that $block_{ji,(I,J)} = k + 1$ if and only if the coordinates at $id_{ji,(I,J)} + k$ are $(I + k, J)$ and the coordinates at $id_{ji,(I,J)} + k + l$ are not $(I+k+l, J)$, for all l such that $k < l < maxblock_i$; that is, the coordinates k positions after (I, J) in the ji-lexicographic ordered trace are $(I + k, J)$ and the coordinates $k + l$ positions after (I, J) in the ji-lexicographic ordered trace are not $(I + k + l, J)$, for all l such that $k < l < maxblock_i$.

From these definitions, we are able to show that consecutiveness of points is guaranteed in the trace under certain conditions.

Theorem 6.1. For the origin point (I, J), $block_{ij,(I,J)} \geq block_j$ implies that all of the $block_j$ points along the first face of the block for the fixed i-value I and j-values increasing by stride one from J exist in the trace.

Proof. Proof by contradiction. Suppose $block_{ij,(I,J)} \geq block_j$. We must show that all of the $block_j$ points along the first face of the block for the fixed i-value I and j-values increasing by stride one from J exist in the trace. Suppose that there are m points, $m > 0$, such that $(I, J + l), 0 < l < block_j$, that do not exist in the trace. Since $block_{ij,(I,J)} \geq block_j$, we have that $block_{ij,(I,J)} = block_j + x$ for some $x \geq 0$. By definition of $block_{ij,(I,J)}$, the coordinates at $id_{ij,(I,J)} + block_j + x - 1$ are $(I, J + block_j + x - 1)$. But m points, $m > 0$, such that $(I, J+l), 0 < l < block_j$, do not exist in the trace. So, there can be at most $block_j + x - 1 - m$ points between (I, J) and $(I, J + block_j + x - 1)$ in the ij-lexicographic trace. But by the definition of $block_{ij,(I,J)}$ there are exactly $block_j + x - 1$ points between (I, J) and $(I, J + block_j + x - 1)$ in the ij-lexicographic trace. Thus, $m = 0$, contradicting that $m > 0$ points are missing from the trace. Therefore, all of the $block_j$ points along the first face of the block for the fixed i-value I and j-values increasing by stride one from J exist in the trace.

From the $block$ variables, we are able to prove that a codelet of a given shape is not feasible from the associated origin point. The following theorem states that a block of points of a given size is not feasible if the $block$ variable is determined to be less than the extent of the template shape along the given dimension.

Theorem 6.2. For the origin point (I, J), $block_{ij,(I,J)} < block_j$ implies that a block of size $block_i \times block_j$ is not feasible.

Proof. Suppose $block_{ij,(I,J)} < block_j$. Then $block_{ij,(I,J)} = block_j - x$ for some $x > 0$. By definition of $block_{ij,(I,J)}$, we have that the coordinates at $id_{ij,(I,J)} + block_j - x - 1$ are $(I, J + block_j - x)$; that is, the coordinates $block_j - x - 1$ positions after (I, J) in the ij-lexicographic ordered trace are $(I, J + block_j - x)$. We also have that the coordinates at $id_{ij,(I,J)} + block_j - x$ are not $(I, J + block_j - l + 1)$; that is, the coordinates $block_j - x$ positions after (I, J) in the ij-lexicographic ordered trace are not $(I, J + block_j - x + 1)$. But to form a block of size $block_i \times block_j$ with origin (I, J), we require the $block_j$ points (i, j) such that $i = I$ and $j = J + l$ such that $0 \leq l < block_j$.

Analogously, in the two-dimensional case, clearly the same theorem applies to the alternate lexicographic ordering.

Theorem 6.3. For the origin point (I, J), and for any k such that $0 \leq k \leq block_j - 1$, $block_{ji,(I,J+k)} < block_i$ implies that a block of size $block_i \times block_j$ is not feasible.

We now show the complete algorithm for applying the described process in the two-dimensional case. We first construct the previously described $block$ parameters, then use these parameters to identify regularity in the two-dimensional trace.

Algorithm 4 Build max j-dimension block sizes

Input: i, j-lexicographic trace T_{ij}, $maxblock_j$

Output: $block_{ij,(I,J)}$ set for each $(I, J) \in T_{ij}$

```
1: procedure BUILD_MAX_J_BLOCK($T_{ij}, maxblock_j$)
2:     for each tuple $(I, J)$ in the $i, j$-lexicographic trace $T_{ij}$ do
3:         for $k = maxblock_j$ downto 0 do
4:             if the coordinates at $id_{ij,(I,J)} + k$ are $(I, J + k)$ then
5:                 $block_{ij,(I,J)} \leftarrow k + 1$
```

Algorithm 5 Build max i-dimension block sizes

Input: j, i-lexicographic trace T_{ji}, $maxblock_i$

Output: $block_{ji,(I,J)}$ set for each $(I, J) \in T_{ji}$

```
1: procedure BUILD_MAX_I_BLOCK($T_{ji}, maxblock_i$)
2:     for each tuple $(I, J)$ in the $j, i$-lexicographic trace $T_{ji}$ do
3:         for $k = maxblock_i$ downto 0 do
4:             if the coordinates at $id_{ji,(I,J)} + k$ are $(I + k, J)$ then
5:                 $block_{ji,(I,J)} \leftarrow k + 1$
```

Algorithm 6 Find blocks two-dimensional

Input: i, j-lexicographic trace T_{ij}, $block_{ij,(I,J)}$ for each $(I, J) \in T_{ij}$, $block_{ji,(I,J)}$ for each $(I, J) \in T_{ij}$, list of template shapes B

Output: list of block origins for each input shape B

```
1: procedure FIND_BLOCKS_2D($i, j$-lexicographic trace $T_{ij}$, $block_{ij,(I,J)}$ for each $(I, J) \in T_{ij}$,
   $block_{ji,(I,J)}$ for each $(I, J) \in T_{ij}$, list of template shapes $B$)
2:     for each block shape $block_i \times block_j \in B$ do
3:         for each tuple $(I, J)$ in the $i, j$-lexicographic trace $T_{ij}$ do
4:             feasible $\leftarrow 1$
5:             if $block_{ij,(I,J)} \geq block_j$ then
6:                 for $k = 0$ to $block_j - 1$ do
7:                     if $block_{ji,(I,J+k)} < block_i$ then
8:                         feasible $\leftarrow 0$
9:                         break
10:            else
11:                feasible $\leftarrow 0$
12:            if feasible = 1 then
13:                list of block origins for $block_i \times block_j \leftarrow (I, J)$
14:                $T_{ij} = T_{ij} -$ points used to form block
```

6.1 N-Dimensional Algorithm for Micro-Codelet Mining

We now define relevant terminology regarding the general case of the algorithm, for reconstructing traces of an arbitrary n dimensions.

Definition 6.9. ($consecutive_{i,(t_j1,...,t_jn),s_{ji}}$). $consecutive_{i,(t_j1,...,t_jn),s_{ji}}$ is the number of consecutive points in trace T_i strided by s_{ji} along the i-dimension starting from point $(t_{j1},\ldots,t_{jn}) \in T_i$

Definition 6.10. ($maxblock_i$). $maxblock_i$ is the largest block size explored in the i-dimension.

The algorithm takes as input an n-dimensional trace T, an ordered list B of block shapes, and an ordered list S of strides. First, the $sort\_trace(T)$ function is called, with the n-dimensional trace T passed as input. The trace T is sorted into a set of n n-dimensional traces $\{T_1, ..., T_n\}$ such that $T_i \in \{T_1, ..., T_n\}$ is ordered with the i^{th} coordinate position as the rightmost lexicographically ordered dimension.

Now that the n lexicographic orderings of the trace T have been obtained, the function to construct the $consecutive$ parameters is called, taking the set of n traces $\{T_1, ..., T_n\}$ as input, as well as the $maxblock$ values, which are defined above. The ordered list of strides S is also passed as input. The Build Consecutive function populates the $consecutive$ parameters with their proper values. For each trace T_i, for each element $(t_{j1},\ldots,t_{jn}) \in T_i$, the value of $consecutive_{i,(t_{j1},...,t_{jn}),s_{ji}}$ is set to the number of consecutive points in trace T_i strided by s_{ji} along the i-dimension starting from point $(t_{j1},\ldots,t_{jn}) \in T_i$. The pseudocode for this function is shown below.

Calling the Build Consecutive function was the final step in the pre-processing required to search for blocks in the trace. We now have the $consecutive$ parameters fully populated. We are now able to call the Find Blocks function, which performs the identification of codelets in the set of traces. Below is the pseudocode for the Find Blocks function, the key driver for the n-dimensional micro-codelet mining algorithm.

Thus, the final n-dimensional micro-codelet mining process consists of three function calls shown below, the first two of which are pre-processing steps and the $find\_blocks$ function being the primary driver.

Algorithm 7 Build Consecutive

Input: $\{T_1, ..., T_n\}$, $maxblock_i$ for $i \in \{1, ..., n\}$, ordered list S of strides

Output: $consecutive_{i,(t_{j1},...,t_{jn}),s_{ji}}$ populated for each lexicographic-ordered trace T_i and for each $(t_{j1}, \ldots, t_{jn}) \in T_i$ and for each stride $s_{ji} \in S$

```
1: procedure BUILD_CONSECUTIVE($\{T_1, ..., T_n\}$, $maxblock_i$ for $i \in \{1, ..., n\}$, ordered list $S$ of strides)
2:     for each trace $T_i \in \{T_1, \ldots, T_n\}$ do
3:         for each $(t_{j1}, \ldots, t_{ji}, \ldots, t_{jn}) \in T_i$ do
4:             for each stride $(s_{j1}, \ldots, s_{jn}) \in S$ do
5:                 for $k = maxblock_i$ downto 0 do
6:                     if the values $k \times s_{ji}$ points after $(t_{j1}, \ldots, t_{ji}, \ldots, t_{jn})$ in $T_i$ are $(t_{j1}, \ldots, t_{ji} + k \times s_{ji}, \ldots, t_{jn})$ then
7:                         $consecutive_{i,(t_{j1},...,t_{jn}),s_{ji}} \leftarrow k + 1$
```

Algorithm 8 Find Blocks

Input: trace T, ordered list of block shapes B, ordered list of strides S, $consecutive_{i,(t_{j1},...,t_{jn}),s_{ji}}$ parameters

Output: list of block origins for each block shape and stride combination populated

```
1: procedure $find\_blocks(T, B, S,$ $consecutive_{i,(t_{j1},...,t_{jn}),s_{ji}}$ for each dimension, trace element, and stride)
2:     for each block shape $(b_{i1}, \ldots, b_{in}) \in B$ do
3:         for each stride $(s_{j1}, \ldots, s_{jn}) \in S$ do
4:             for each element $(t_{k1}, \ldots, t_{kn}) \in T$ do
5:                 $feasible \leftarrow 1$
6:                 for $x = 1$ to $n$ do
7:                     if $x = 1$ then
8:                         if $consecutive_{1,(t_{j1},...,t_{jn}),s_{j1}} < b_{i1}$ then
9:                             $feasible \leftarrow 0$
10:                    else
11:                        for each $(y_1, \ldots, y_{x-1}) \in \{(0, \ldots, 0), \ldots, (b_{i1}, \ldots, b_{ix-1})\}$ do
12:                            $(t'_{j1}, \ldots, t'_{jn}) \leftarrow (t_{j1}, \ldots, t_{jn}) + (0, \ldots, 0, y_1, \ldots, y_{x-1})$
13:                            if $consecutive_{x,(t'_{j1},...,t'_{jn}),s_{jx}} < b_{ix}$ then
14:                                $feasible \leftarrow 0$
15:                                break
16:                        if feasible = 0 then
17:                            break
18:                if feasible = 1 then
19:                    list of block origins for $(b_1, ..., b_n, s_1, ..., s_n) \leftarrow (t_{k1}, .., t_{kn})$
20:                    $T = T-$ points used to form block
```

Algorithm 9 n-dimensional micro-codelet mining

```
1: procedure $codeletminer(T, B, S, \{maxblock_1, ..., maxblock_n\})$
2:     SORT($T$)
3:     BUILD_CONSECUTIVE($\{T_1, ..., T_n\}, \{maxblock_1, ..., maxblock_n\}, S$)
4:     FIND_BLOCKS($T, B, S$)
```

6.2 Complexity Analysis for Micro-Codelet Mining Algorithm

We now present an analysis for the complexity of the micro-codelet mining algorithm. For iterating through a trace of points to identify polyhedra, we let the access at position i in the trace to incur a penalty of $+1$, and an access at position $i+1$ in the trace to also incur a penalty of $i+1$. When accessing points distant from the base point at index i, we let the access incur a penalty of $+\alpha$, for some $\alpha > 1$ when accessing a point at position $i+k$ for large enough k. Now, to formulate our analysis, we let n be the dimensionality of the trace and m be the number of elements in the trace. We let b be the size of B, which is the set of different block shapes. We also let s be the size of S, which is the set of different strides. As before, we have that $maxblock_i$ is the largest extent of points explored to form blocks in the i-dimension, for $i \in 1, ..., n$. Also, we now let $maxblock_{max} = maxmaxblock_1, ..., maxblock_n$.

Thus, we have that construction of the $consecutive$ parameter takes at most $n \times m \times s \times maxblock_{max}$. When operating on a one-dimensional trace, mining for codelets takes $m \times b \times s$. For a two-dimensional trace, this value becomes $m \times b \times s \times maxblock_1$. We include the $maxblock_1$ term here since the $consecutive$ parameter is checked for each point along an edge of the polyhedron. We do not require the α factor in this case since we are checking points along a single edge, which is within the block size distance from the origin point of the identified polyhedron. However, in the three-dimensional case, this term gets modified to become $m \times b \times s \times (maxblock_1 + maxblock_1 \times maxblock_2 \times \alpha)$. The $maxblock_1$ term appears because we are checking the $consecutive$ parameter for each point along an edge of the polyhedron. Similarly, the $maxblock_1 \times maxblock_2 \times \alpha$ term exists because we must check the $consecutive$ parameter for each point of the face of the polyhedron in the three-dimensional case. The α term appears here because points along the face of the polyhedron could be distant in the trace relative to the origin point.

To generalize to the n-dimensional case, we expand the term used in the previously described traces to apply to hyper-rectangles being identified, thus the time to find blocks in the n-dimensional case takes at most $m \times b \times s \times (maxblock_1 + \alpha \times \sum_{j=2}^{n-1} \prod_{i=1}^{j} maxblock_i)$.

Chapter 7

Code Generation

We now describe the process of transforming the polyhedra that are generated by the program after the analysis of the input trace and synthesizing C code. We use the operation of SpMV for our example, but other operations are also able to have code generated in a straightforward way. We generate code with the aim of maximizing SIMD vectorization opportunities. This is enabled by the pre-defined template shapes typically being selected to facilitate SIMD code vectorizability. We then rely on the compiler to generate vectorized code through utilization of specific flags.

We now present a full example of the input to output process regarding the initial reading of the trace and the final production of executable C code. The entire process takes as input a trace of points and a list of candidate template shapes, produces a set of polyhedra which partition the trace into polyhedra with shapes determined by the input template shapes, and finally generates C code by scanning these polyhedra.

i	j	k
0	0	0
0	1	1
0	2	2
0	4	3
0	9	4
1	7	5
1	8	6
1	9	7
5	4	8
6	7	9
6	8	10
6	9	11
8	7	12
8	8	13
8	9	14

Figure 7.1: Input trace, such as that obtained by the previously described process of scanning the nonzero coordinates of the matrix and appending their index in the trace as the third column.

We use the term extent to describe the number of points grouped together along a given dimension. For example, an extent of three along a certain dimension indicates that three distinct values will be modeled for that dimension. The term stride describes the gaps between points along a given dimension. For example, if the extent is three and the stride is two for a dimension, then three distinct values will be captured for that dimension, such that each point is two integer values away from the preceding point.

extent	stride
$\{1 \times 3\}$	$\{1 \times 1\}$
$\{2 \times 3\}$	$\{2 \times 1\}$

Figure 7.2: Input list of candidate template shapes.

extent	stride	origins
$\{1 \times 3\}$	$\{1 \times 1\}$	$\{0, 0, 0\}, \{1, 7, 5\}$
$\{2 \times 3\}$	$\{2 \times 1\}$	$\{6, 7, 9\}$
$\{1 \times 1\}$	$\{1 \times 1\}$	$\{0, 4, 3\}, \{0, 9, 4\}, \{5, 4, 8\}$

Figure 7.3: Result of the previously described process which mines for regularity in the trace and partitions the trace into polyhedra of shapes determined by the input template shapes. Each input template shape is listed along with its associated origin points that were obtained.

We now describe the exact process for code synthesis. As a result of the prior stages of recognizing micro-codelets based on input template shape, we now have a representation such that each of the template shapes input to the program have matched to it a list of origin points. We have in Figure 7.1 an example input trace and in Figure 7.2 a list of candidate template shapes. Both of these figures are inputs to the program by the user and are dealt with in the previously described process of identifying micro-codelet origin points. The result of identifying codelet origin points in this trace under these input template shapes is shown in Figure 7.3. In Figure 7.3, we have that for each candidate template shape, there is a list of zero or more origin points associated with shapes of the given size that were able to be identified, as well as a list of point associated with the points that were not able to be grouped into the predefined shapes. The shapes that were not able to be grouped are listed as the $\{1 \times 1\}$ shape, although these leftover points are identified by default and the user does not input 1×1 as a candidate template shape. The structures shown in Figure 7.3 are the input to the code generation process.

Since the input trace is reorderable by design, we are allowed to generate code in any given ordering. Thus, there is no restriction on the order that we scan the obtained polyhedra. In practice, there are two options regarding the order in which the final code is generated. In one implementation, code is generated in the same order that the user input the template micro-codelet shapes. The points that were not able to be grouped in a regular structure by being represented as a polyhedra of cardinality greater than one under the template shapes that were input; that is, the leftover points, must have their code emitted one-by-one, since the program was not able to represent them in groups due to the template shapes not being of a form to capture these points In this scenario, the leftover points are printed as single-line statements at the end of the code body. However, this leads to poor spatial locality since potentially distant points can be executed in sequence.

The second approach regarding the order in which codelets are executed does a lexicographic ordering of the codelets, specifically the origin points of each codelet are sorted in lexicographic order prior to code generation. The ordering is done in an $i - k - j$ ordering, so that the index for the y array has priority over the index for the A array, which has priority over the index for the x array. It is not always possible to generate the code for each statement that is executed in exactly this ordering since in the case where the template shapes have non-unit strides, there can be overlapping iteration domains for multiple codelets. This approach leads to codelets of varying shapes, including leftover points, to be executed inter-mixed throughout the program. Additionally, this approach improves spatial access locality for the y vector and A array due to distant points now being executed in sequence regardless of their associated template shapes.

In order to allow reordering of the codelet origin shapes regardless of their associated template shapes, we must have for each origin point, an associated template shape. For example, in Figure 7.3, the point $6, 7, 9$ will have an identifier so that it is known that this is the origin point for an extent 2×3 and 2×1 strided micro-codelet shape. From this information, we simply scan through each codelet origin point, and emit a macro that inserts loop code corresponding to its associated template shape.

```
macro_2_3_2_1(6, 7, 9)
```

Figure 7.4: The macro inserted in the body of the program that represents a micro-codelet shape of extent 2 in the i-dimension, extent 3 in the j-dimension, stride 2 in the i-dimension, and stride 1 in the j-dimension corresponding to an origin point of $6, 7, 9$.

```
for (i = 0; i < 2; ++i) {
  for (j = 0; j < 3; ++j) {
    y[6 + 2i] += A[9 + 3i + j] * x[7 + j];
  }
}
```

Figure 7.5: The expanded macro code for the macro of Figure 7.4.

So, for the point $6, 7, 9$ and its associated shape of an extent 2×3 and stride 2×1 micro-codelet, we emit the following macro: macro_2_3_2_1(6,7,9) which is a macro specifically for all origin points with this template shape, with inputs of the origin point coordinates passed as input so that the loop body can be executed relative to the given starting indices. Specifically, the first two identifiers in the title of the macro show the extents along the i and j dimensions, respectively. In this example the extent along the i dimension is two and the extent along the j dimension is three. Additionally, the stride of the template shape is explicitly encoded in the identifying title of the macro, as the next two identifiers of the macro. The stride of the i dimension is listed as the third index in the identifier and the stride of the j dimension is listed as the fourth index in the identifier. In this example, the stride encoded for the i dimension is two and the stride encoded for the j dimension is 1.

As shown in Figure 7.6, The program body now consists of a list of these macro statements, so that when the macros are expanded we have a list of single-statement loops consisting of one or two nesting levels. In our implementation, we rely on the compiler to generated vectorized code for these expanded macros. However, SIMD intrinsics could be used in place of the compiler for hand-written vectorized implementations for specific codelet shapes. In practice, the codelet shapes are selected so that the compiler can easily perform automatic vectorization of the generated code.

```
macro_1_3_1_1(0, 0, 0)
y[0] += A[3] * x[4];
y[0] += A[4] * x[9];
macro_1_3_1_1(1, 7, 5)
y[5] += A[8] * x[4];
macro_2_3_2_1(6, 7, 9)
```

Figure 7.6: The program body once the polyhedra have been scanned in an $i-k-j$ lexicographic order and macros emitted corresponding to each origin point, as well as single-statements emitted for the points that were not able to be grouped in polyhedra of the predefined shapes.

7.1 Efficient Synthesis for Micro-Codelets

The performance of the generated code for the codelet-centric SpMV code is hurt due to L1 instruction cache misses. This is due to the nature of generating code for micro-codelets, whose generated loop-based code may frequently contain only three or fewer points per loop nest when only small micro-codelets can be identified. Thus, we obtain a program that has a large number of instructions. This number of instructions is proportional to the size of the input trace. It was observed by Rodríguez et al. that performance is decreased due to L1 instruction cache misses.

We post-process the generated code to insert explicit instruction cache prefetch instructions using the prefetch1 instruction, which prefetches instructions to the L2 cache. We choose the prefetch distance to be at 4096 bytes, since this was observed to result in the best performance improvement. However, this distance can be modified and was empirically chosen. One prefetch instruction is inserted in the code every 64 bytes. In Algorithm 10, we present the pseudocode for the insertion of the prefetch1 instruction. The impact of inserting the prefetch instruction is show in Figure 7.7.

Algorithm 10 Pseudocode of the prefetch insertion

Input: Assembly code of a function f

Output: Modified assembly code with periodical prefetches

```
1: procedure ADD_PREFETCH($f$)
2:     offset = 0
3:     for each instruction $i$ in $f$ do
4:         if offset >= 64 bytes and not_in_loop($i$) then
5:             insert prefetcht1 before $i$
6:             offset = sizeof_prefetch_instruction
7:         offset += sizeof_instruction($i$)
```

The experiments were conducted on an Intel Core i7 8700K with 64GB of RAM. Additionally, HugePages are utilized to further improve performance for the code generated using the micro-codelet approach. The HugePages are of size 2MB, which is 512 times larger than the standard page size of 4KB. We make the number of HugePages configured on the system to be 512, thus allocating 1GB worth of HugePages. By utilizing HugePages, we reduce the number of overall pages required, thus reducing the number of instruction TLB misses for large programs. The experiments were evaluated on 200 sparse matrices from the SuiteSparse repository, all containing less than 10 million nonzero elements. For each of the 200 matrices, ICC version 18.0.3 with the flags -O2 -xSKYLAKE -vec-threshold0 was used to compile the C code for the SpMV operation. It was observed that compiling with -O3 instead of -O2 both increased compilation time and slightly decreased overall performance of the program. The -vec-threshold flag makes certain that the machine vectorizes the generated code to its full capabilities. These flags were chosen to provide best performance.

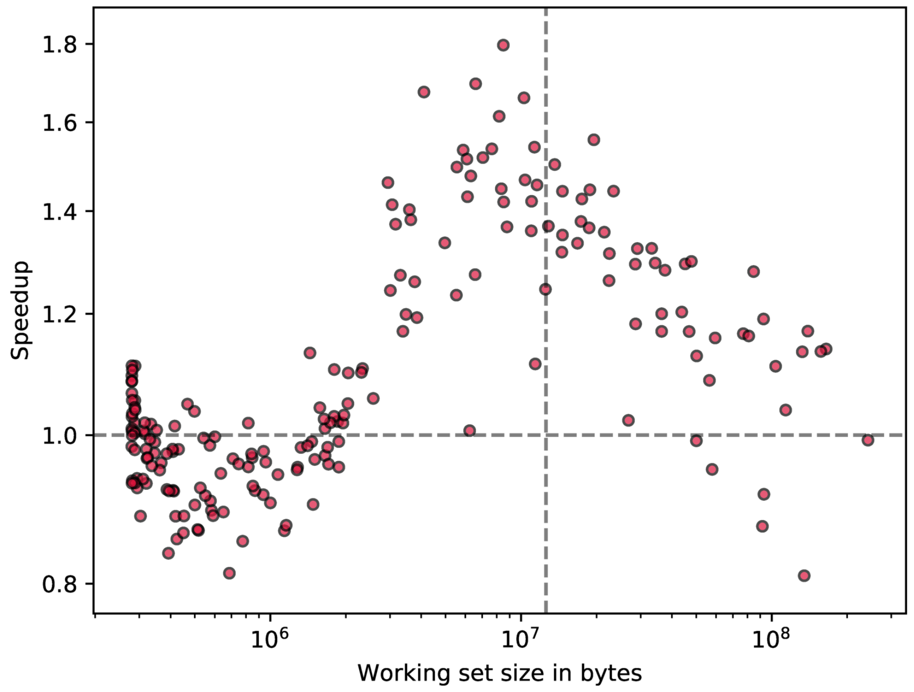

Figure 7.7: Speedup of the program with prefetch instructions inserted compared to the program without prefetch instructions inserted [18]. The vertical dashed line shows the approximate size of the last-level cache.

7.2 Results of Reconstruction Techniques

We now show the performance results Figure 7.11 which display the best performing version of reconstruction with respect to the best performing baseline. The baseline compared to is the best performing version between classical irregular SpMV and Intel MKL; that is, whichever of these two performed faster was the baseline for the given matrix. The overhead for initializing Intel MKL sometimes outweighs the time to perform the computation operating on small matrices, thus we use classical irregular SpMV as an alternative. It is apparent that the reconstructed code is able to achieve a speedup of up to four times the baseline. However, performance does decrease in some cases. The performance was evaluated on the same 200 sparse matrices from the SuiteSparse repository, with the same experimental setup as described in the preceding section.

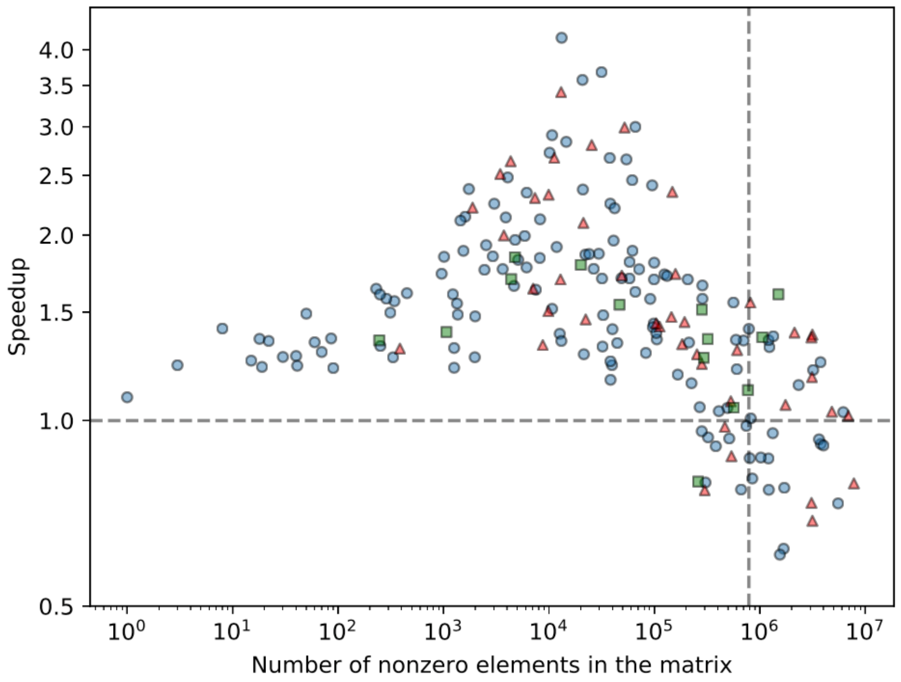

Figure 7.8: Best performing reconstructed code compared to the best performing baseline [18]. The vertical dashed line shows the approximate size of the last-level cache. The different shapes of markers indicate which version of reconstruction performed best for the given matrix. Micro-codelet reconstruction is marked with a circle. Micro-codelets grouped under a one-dimensional loop nest are represented with a triangle. Micro-codelets grouped under a two-dimensional loop nest is represented with a square.

7.3 Scalability of Synthesis

We now present results regarding the time it takes to perform the complete reconstruction process. The timed process includes reading the input trace of tuples, lexicographically reordering the trace, mining for micro-codelet origins, and generating executable C code. Experiments were performed on a 4 core 3.2GHz Intel Xeon E3-1230. We evaluate on 110 matrices from the SuiteSparse repository, with number of nonzero elements ranging from 140 nonzero elements to 9.8 million nonzero elements. We present data comparing the seconds to perform the reconstruction process with the number of nonzero elements in each matrix. We perform separate evaluations for different number of shapes input to the micro-codelet mining process. We evaluate separately on 10, 50, and 100 input template shapes. In the cases where we evaluate on 10 shapes, we input template shapes that check for consecutive points along either dimension individually in the trace of nonzero coordinates. That is, one dimension in the input template shapes is always of value one. The maximum extent along either dimension checked is 6 and unit strides are the only strides allowed. When evaluating on 50 or 100 shapes, the maximum extent checked along either dimension of the trace is 10 and only unit strides are allowed. Also, the shapes consist of combinations of extents with values greater than one in both dimensions. We observe a similar linear trend in each plot corresponding to number of input template shapes. There is a 94% correlation between the number of nonzero elements in the matrix and the time to perform the reconstruction process. Thus, the reconstruction time is primarily a function of the number of nonzero elements in the matrix, but also depends on the sparsity pattern of the matrix. In these plots, out of the 110 matrices operated on, we observe a maximum time to perform the timed reconstruction process is 204 seconds when operating on a matrix with 8.3 million nonzero elements with 100 input template shapes.

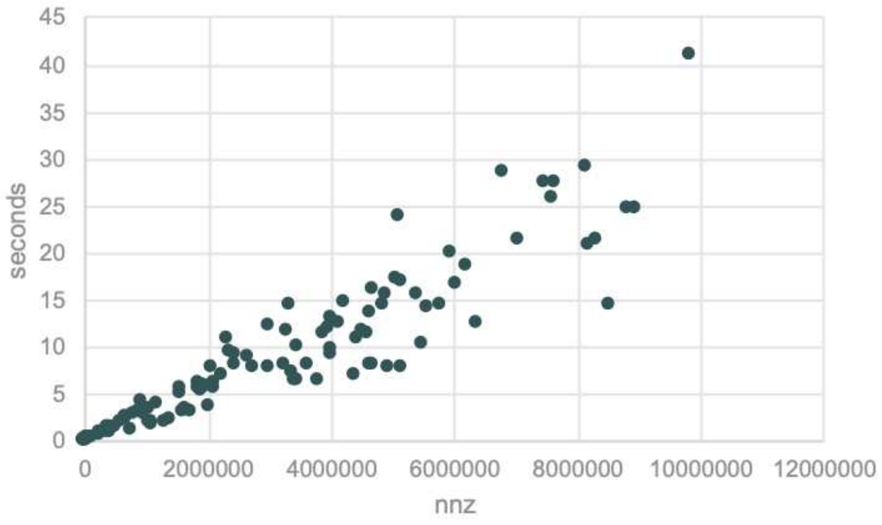

Figure 7.9: Reconstruction time in seconds compared to number of nonzero elements for 110 matrices from the SuiteSparse repository when using 10 template shapes as input.

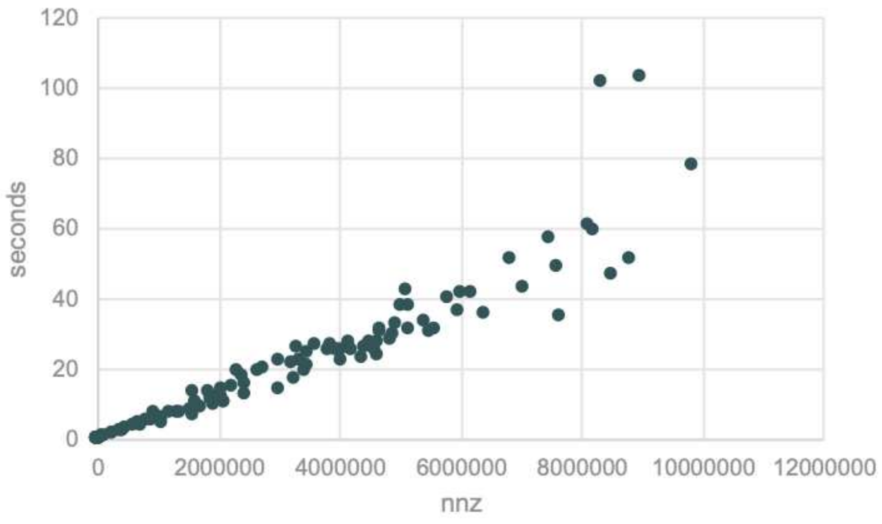

Figure 7.10: Reconstruction time in seconds compared to number of nonzero elements for 110 matrices from the SuiteSparse repository when using 50 template shapes as input.

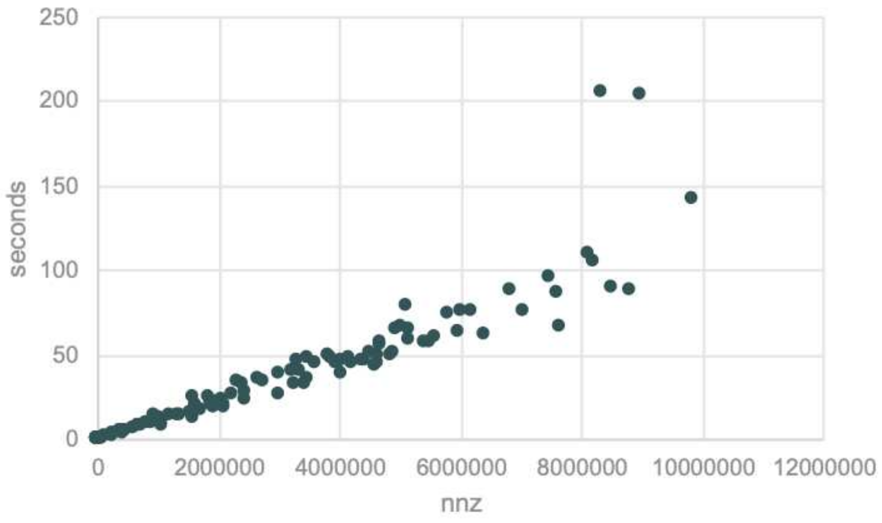

Figure 7.11: Reconstruction time in seconds compared to number of nonzero elements for 110 matrices from the SuiteSparse repository when using 100 template shapes as input.

Chapter 8

Conclusion

In summary, a novel approach was taken to represent sparse matrices. An input trace, representing the nonzero coordinates of a sparse matrix, is automatically partitioned into sets of polyhedra of pre-selected sizes. Code is synthesized that is specific to the sparsity structure of the input matrix. The procedure generates a collection of independent loop-nests. The approach requires no indirection array and does not require storage of zero-valued elements.

The input trace is mined to identify regularity with priority placed on identifying polyhedra with largest cardinality as well as identifying shapes in the trace which are amenable to SIMD vectorization. A method was explored to decrease the time to identify regularity by maintaining multiple lexicographic traces and decreasing the total number of checks required to confirm or deny a regular grouping of points exists. The generated code consists of dense sub-components of the input trace and can be easily generated by scanning the origin points of the identified polyhedra. These dense loop-nests can then be automatically vectorized by the compiler.

After micro-codelet mining has occurred, techniques were presented for applying hierarchical reconstruction to further identify regularity in the sparse structure. Techniques were described for recursively applying the micro-codelet mining algorithm so that the input becomes the traces of micro-codelet origin points and macro-codelets are identified. Applying the geometric power of the Trace Reconstruction Engine to form complex shapes out of micro-codelet origin points was also explored.

Prefetch instruction insertion was utilized to improve performance of the generated code. Experimental results confirmed that the insertion of prefetch instructions was essential to improving the performance of the generated codelet-centric code. Results were presented for synthesis time, showing that the time to perform the entire reconstruction process is mostly correlated with the number of nonzero elements being operated on, with some impact on reconstruction time coming from the sparsity pattern of the input. An increase in performance was observed in some cases when applying our technique to SpMV using 200 matrices from the SuiteSparse repository, achieving a speed-up of up to four times compared to a baseline of either classical irregular SpMV or Intel MKL.

Bibliography

[1] Gabriel Rodríguez, José M. Andión, Mahmut T. Kandemir, and Juan Touriño. Trace-based affine reconstruction of codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 139–149, New York, NY, USA, 2016. ACM.

[2] Gabriel Rodríguez and Louis-Noël Pouchet. Polyhedral modeling of immutable sparse matrices. In *8th International Workshop on Polyhedral Compilation Techniques. Manchester, UK*, 2018.

[3] P. Clauss and B. Kenmei. Polyhedral modeling and analysis of memory access profiles. In *IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 191–198, Sep. 2006.

[4] Philippe Clauss, Bénédicte Kenmei, and Jean Christophe Beyler. The periodic-linear model of program behavior capture. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, pages 325–335, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] Alain Ketterlin and Philippe Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 94–103, New York, NY, USA, 2008. ACM.

[6] Richard Wilson Vuduc and James W Demmel. *Automatic performance tuning of sparse matrix kernels*, volume 1. University of California, Berkeley Berkeley, CA, 2003.

[7] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2007.

[8] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[9] Jee W Choi, Amik Singh, and Richard W Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *ACM sigplan notices*, volume 45, pages 115–126. ACM, 2010.

[10] Xintian Yang, Srinivasan Parthasarathy, and Ponnuswamy Sadayappan. Fast sparse matrix-vector multiplication on gpus: implications for graph mining. *Proceedings of the VLDB Endowment*, 4(4):231–242, 2011.

[11] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Csx: An extended compression format for spmv on shared memory systems. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 247–256, New York, NY, USA, 2011. ACM. Proposes new storage format for sparse matrices targeting SpMV called Compressed Sparse eXtended.

[12] Anand Ekambaram and Eurípides Montagne. An alternative compressed storage format for sparse matrices. In *International Symposium on Computer and Information Sciences*, pages 196–203. Springer, 2003.

[13] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[14] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):123, 2018.

[15] Joel Saltz, Kathleen Crowley, Ravi Michandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, 1990.

[16] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. Parsy: inspection and transformation of sparse matrix computations for parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 62. IEEE Press, 2018.

[17] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 13. ACM, 2017.

[18] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 625–639. ACM, 2019.