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

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

Search Space Characterization

10.1 Introduction

Sparse matrix computations are an important class of compute intensive codes and are extensively used. Not surprisingly, many techniques have been developed to optimize sparse matrix computations. An important technique is the selection of a smart data structure for storing the sparse matrix corresponding to the computation to be carried out. However, wrapping the sparse matrix data in a specific data structure obscures the compiler optimization process and thus forms a major obstacle for further effective optimization and code generation.

As a consequence, many HPC applications rely on the use of sparse algebra run-time libraries to provide efficient, hand-optimized, implementations of common sparse matrix operations. However, given the complexity of today’s CPU and GPU architectures, predetermined implementations of sparse algebra routines cannot get maximum performance out of the architecture. This is also the case for novel implementations based on expression templates to optimize performance, such as Blaze [43].

Besides the continuing advances in the target architectures, the parametrized nature of supplied sparse algebra routines and the fact that these routines are implemented for a limited number of data layouts also inhibit maximum performance from being achieved. For example, sparse matrix times matrix multiplication can be performed for a right-hand matrix with different numbers of columns $k$. Commonly, sparse algebra libraries provide a single routine for this multiplication that is parametrized for $k$. However, there is no single implementation of this computation that is a best fit for all possible parameters. Similarly, no sparse storage format exists that is optimal for different computations, different parameters for a computation or different sparse matrices. In short, predefined implementations based on predefined data layouts can never achieve optimal performance. Furthermore, the way these routines are implemented, by abstracting, or obscuring, the matrix data into a specific data structure, hampers optimizing compilers from producing more efficient codes.
Therefore, instead of maintaining predetermined implementations of sparse algebra routines in run-time libraries, we argue for a transition towards automatic instantiation of sparse algebra routines. In this chapter, we introduce computation-driven reassembly of sparse data structures, which is a key component of this automatic routine instantiation process. By combining sparse data structure reassembly with the code instantiation process, the construction of optimal data structures is made an integral part of the code optimization process. With computation-driven reassembly, an optimal sparse matrix storage format is derived from the actual sparse matrix computation, contrary to selecting a predefined storage format as is done in existing methods. In this approach, code and data layout are optimized hand in hand.

The automatic instantiation of sparse algebra routines and data layout reassembly is implemented as a series of code transformations in the forelem framework [83]. Within the forelem framework, all data is accessed through a tuple space. Data to be processed is specified as (multi)sets of tuples. In this case, the tuples arise from “disassembling” the original sparse matrix structure. The computation is expressed in terms of loops processing the tuples. Different transformations are implemented in the framework, ranging from standard compiler optimizations, such as Loop Interchange [4], Loop Fusion [52], Scalar Expansion and Def-Use analysis [2, 50], to transformations that address the order in which tuples are executed and stored. Among these latter transformations is Materialization, which is an important, enabling, transformation for computation-driven data structure reassembly.

Because the optimization process is made responsible for automatic generation of routines, the search space of this process is significantly enlarged. We demonstrate that more than 130 principal forms\(^1\) of sparse matrix times \(k\) vector(s) and sparse matrix times matrix multiplication can be instantiated, with over 25 different reassemblies of the original sparse matrix data structure. So, essentially, 25 different data structures are being generated. This exemplifies the strength of our approach when compared to sparse algebra libraries, that on average implement 4, or less, pre-defined sparse data storage formats. When combined with parametric compiler optimizations, such as loop unrolling and loop blocking, the search space is enlarged by two to three orders of magnitude. A characterization of the possible instantiation search space shows that there are many different optimal instantiations. As a consequence, it is very hard, if not impossible, to predict which instantiation would be optimal for a given matrix, computation, computation parameters and architecture instance.

In addition to the search through automatically instantiated sparse algebra routines, a search through the parametric optimization search space, set up by transformations such as loop blocking and loop unrolling, can be done. This parametrized search space is complementary to the search space set up by the transformations leading up to differently structured computations and storage formats. We demonstrate that it is important to consider this second search space, as, for a single matrix, an instantiation that is optimal for unroll level \(a\), is not necessarily the optimal instantiation for an optimal unroll level \(b\). We show that

\(^1\)Actually, 200 principal forms were generated, but for the experimentation 70 of these were deleted because they were too inefficient and therefore caused unacceptably long experimentation times.
with an exhaustive search through these combined search spaces, an automatically instantiated routine can be found that mostly outperforms the implementations from existing sparse algebra libraries or at least is equivalent in performance.

This chapter is organized as follows. Section 10.2 describes how different routines are instantiated and how data structure reassembly is carried out. In Section 10.3, an initial exploration is done of the search space of the different instantiations of sparse matrix times \( k \) vector(s) multiplication. Section 10.4 quantifies the irregularity of this search space using rank correlations. Section 10.5 discusses the irregularity observed in two other sparse matrix kernels. Section 10.6 presents the results of the comparison of our approach to a number of existing sparse algebra libraries. Section 10.7 presents our conclusions and plans for future work.

### 10.2 Reassembling Data Structures

This section describes the process and techniques that are used to instantiate efficient sparse algebra routines and reassemble the original sparse matrix data storage automatically. Using these techniques, many different forms of a sparse algebra routine can be generated, along with different reassemblies of the original sparse data storage.

As these techniques are implemented in the forelem framework [83], we first briefly introduce this framework. The principal syntactic construct in the forelem framework is the forelem loop. A forelem loop iterates (a subset of) a multiset of tuples and performs an operation on these tuples. As an example, consider a multiset \( T \) containing tuples with fields \( \text{field1} \) and \( \text{field2} \): 

\[(\text{field1}, \text{field2})\]

Then, the following loop sums the values of \( \text{field1} \) of tuples of which \( \text{field2} \) equals the value 9:

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

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

For the sparse algebra kernels to be expressed as a forelem loop, the sparse data structures are stored as sets of tuples and the dense data structures remain dense. A sparse matrix \( A \) can be represented as a set of tuples of the form \( (\text{row}, \text{column}, \text{value}) \). Based on this, we can express a loop computing the sparse matrix vector product \( c = Ab \) as follows:

```c
forelem (m; m \in pA)
```

The expression of the computation that is performed by a sparse algebra routine in terms of tuples is the first step in the code instantiation process. The tuples arise from “disassembling” the original sparse matrix data structure. All non-zero
matrix elements are extracted from this original structure and are represented as tuples, one tuple per non-zero element.

On this initial specification of the sparse computation, various transformations can be applied that may modify the order in which the tuples are accessed and thus may influence how the tuples are reorganized. Since in `forelem` loops no explicit order is imposed on the iteration, tuples may be visited in any order. However, through a transformation known as Orthogonalization [83], a certain order can be imposed on the iteration based on the value of the fields of the tuples. A possible result of Orthogonalization is the following loop nest, which processes the above computation on a row-by-row basis:

```c
for (i; i ∈ N_n)
  forelem (m; m ∈ pA.row[i])
```

where \( N_n \) is the number of rows in matrix \( A \) and the index set \( pA.row[i] \) makes the `forelem` loop only iterate tuples in \( A \) with the value of field `row` equal to \( i \).

To move towards a concrete implementation of this computation from this point, the Materialization transformation performs an important enabling role. The purpose of Materialization is to materialize the tuples iterated by a `forelem` loop using the accompanying index set to an array in which the data is represented in consecutive order and is accessed with integer subscripts. So, at this point tuples are physically reorganized into a particular order, based on the computation. Although this transformation can be seen as a simple normalization operation, it is an important enabling step that allows the compiler to address and modify the order of data access to these arrays. In fact, by materialization the execution order of an inner loop is fixed.

Note that these two transformations cause the original sparse data structure to be reassembled on a row-by-row basis. A compiler performing these transformations can thus exert control on the order in which data is stored. Up till now, optimizing compilers could not exert this amount of control on data storage order. Techniques have been described in the literature that do modify data structures, such as structure splitting [23], array regrouping [103] and field reordering [23], but these techniques are limited to rearranging data stored in arrays of structures in order to improve cache usage. Through Materialization much more invasive transformations of the data structure are enabled, such as translation from an unordered set of tuples to separate sequences of column indices and values stored in the order of ascending row number. See also the previous chapter.

Finally, after all transformations have been carried out, an implementation in C code is generated from the tuple program and a reassembled copy of the sparse data structure is instantiated based on the organization of tuples selected by the optimization process.

### 10.3 The Transformation Search Space

Using the approach described in the previous section, many different instantiations of the same sparse matrix routine can be generated. Different instantiations
can be distinguished because different transformations were performed before and/or after Materialization or transformations were performed in a different order. This section presents the results of the initial exploration of the search space of instantiations of sparse matrix times $k$ vectors multiplication.

The full transformation tree of sparse matrix times $k$ vector multiplication is shown in Figure 10.1. The starting point of the transformation space is labeled with 1 and shown in the center of the figure. This is the minimal representation of the computation as a forelem loop. From this point, there are several different branches of transformations as shown in the picture, resulting in many different variants, or principal forms. Whenever the label of a node is prefixed with “tmp”, the node represents a stage for which no executable is generated. In all other cases, the executables (variants) are labeled from 1 to 130. Next to these 130 different implementations, also 25 different data structures are generated 2, ranging from simple coordinate storage to compressed row or column schemes, with or without zero-padded rows or columns, and jagged diagonal like schemes wherein the rows of the matrix have been permuted or not. For all these data structures, also corresponding initialization procedures are automatically generated.

An initial characterization of the search space of principal forms can be obtained by looking at execution times3. For a given matrix, we can measure the execution time for different variants and different values of $k$. The results of these experiments are visualized in order of ascending execution time. The visualization is limited to display the 200 best-performing experiments. Two of such plots are shown in Figure 10.2. The dark gray star denotes the fastest experiment and the light gray circle the 200th instance in the order. The ordered sequence of the experiments is shown by the arrows, which fade out with increasing execution time.

In our discussion of these two plots, we use the fact that variants from the same subtrees in Figure 10.1 have numbers that are near each other. So, in the plot a longer arrow is a jump to another subtree, generated from another orthogonalization, etc. Both plots show a very different structure. We do notice, however, that the fastest variant is in both cases from the same subtree, with the dark gray star being located around 115. Subsequent fastest variants are different for both matrices, as the arrows progress in a vastly different manner. Another artifact that becomes clear is that for the Erdos971 matrix, variants from more different subtrees are within the 200 best results. For example, the variants in the region $[0, 10]$ are frequently hit for the Erdos971 matrix, but less so for the OPF.10000 matrix.

Further plots can be created for other matrices, sparse algebra kernels and architectures. However, the two plots that are shown already present a clear difference in best performing variants for two matrices. One of the main questions is whether a ranking of the variants on execution time for one configuration of matrix, kernel and parameters can be used to predict the ranking for another matrix, kernel and parameter configuration. In other words, whether two such rankings

---

2In this chapter, we did not consider loop blocking. If loop blocking would have been taken into account, a multitude of different combinations of these data structures would have been generated.

3The execution times were measured on an Intel Xeon 5150 CPU at 2.66 Ghz, with 16GB RAM, running Ubuntu Linux 10.04.4. To remove fluctuation from the results, the computation performed by each variant is repeated 10 times. The compiler that was used is gcc 4.4. Several matrices have been used for the experiments and these were obtained from the University of Florida Matrix Collection [28].
10.3. The Transformation Search Space

Figure 10.1: The full transformation tree of sparse matrix times $k$ vector multiplication.
Figure 10.2: The plots show the 200 best-performing experiment instances, ordered in execution time from fastest to slowest. The star denotes the fastest variant, the circle the slowest.
bear any resemblance to one another. Instead of evaluating this graphically, we will quantify the differences in ranking in the next section, using the technique of Rank Correlations from the field of statistics.

10.4 Rank Correlations

As could be seen in the previous section, already the initial search space consisting of the principal forms is rather erratic. Although the plots give a good first impression of the transformation search space, these are not sufficient to come to a satisfactory quantification of the irregularity. In this section, we will use the Rank Correlation Coefficient as a measure for the relationship between two rankings.

Let a ranking be a permutation of a subset of \( n \) natural numbers: \( \{ i \mid i \in \mathbb{N} \land i \in [1, n] \} \). Given two rankings with the same \( n \), a Rank Correlation Coefficient indicates the similarity of these two rankings. The coefficient takes on a value in the interval \([-1, 1]\), where 1 indicates the two rankings are equal and -1 indicates the rankings are each other's reverse. Different methods to compute a Rank Correlation Coefficient exist. In this chapter, we make use of Kendall’s \( \tau \) [49], which is easy to compute. Computation of the coefficient is carried out by scoring the ranking. The score is computed by determining whether each pair of numbers in the ranking sequence is in the same order compared to the other ranking. For every in-order pair 1 is added to the score, for every out-of-order pair 1 is deducted from the score. The final score is divided by the maximum possible score (all pairs are in-order) to obtain the rank correlation coefficient.

The goal is to quantify the relationship of the ranking of variants for two experiment instances, for which the matrix, \( k \) and architecture are specified. The quantification is performed for the \( n \) best performing variants. To be able to compute the Rank Correlation Coefficient, a numerical ranking in \([1, n]\) is necessary for every variant in both experiments, so for the union of the variants found in both experiment instances. However, it can be the case that a variant from the \( n \) best performing variants for matrix A is not part of the best performing variants for matrix B. For such variants, a ranking is not specified for matrix B. This problem is resolved by assigning such variants a ranking of \( n + 1 \). In summary, best performing variants for a matrix A, which are not contained in the intersection of best performing variants of matrix A and B, are assigned a ranking of \( n + 1 \) for matrix B.

In Figure 10.3 the correlations of the rankings of the 10 best variants of experiment instances of the same matrix, but different \( k \), are quantified. The line \( y = x \) indicates the symmetric axis in this picture. From this figure it is observed that the correlation coefficient ranges from -0.4 to 0.5. That is, the majority of points are closer to 0, indicating a very weak correspondence between the rankings.

Another perspective is shown in Figure 10.4, where \( n \) best variants is varied and the rank correlation is shown between a number of vectors on the y axis and 50 vectors at the top, 10 vectors at the bottom. The plot quantifies the relationship between a ranking for 50 (or 10) vector multiplications and \( y \) vector multiplications. Note that the results for 50 versus 50 (and 10 vs. 10) have been set to 0 for clarity. For the OPF_10000, a clear relationship is seen between the ranking for 50
Figure 10.3: Quantification of the ranking correlation between a number of vectors $x$ and $y$ multiplied with the Erdos971 matrix, considering the 10 best variants. Note that the values for $x = y$ are set to 0 and the symmetry of the figure is highlighted by the line $y = x$.

vectors and 100 vectors. However, for other values of $k$ this relationship is weak. In the Erdos971 plot, there is a strong correlation with $k = 5$ if the number of best variants considered is increased. A similar, but not as strong, trend is found for $k = 50$. Interesting is the clear reverse correlation area in the top-left corner of the plot. The 10 best variants at $k = 10$ are not included in the best variants for $k = 500$.

The quantification presented in this section shows that the transformation search space is indeed erratic. The lack of relationship between the rankings of different instances of a matrix, and a number $k$ makes it very hard, if not impossible, to predict which variant would be optimal for a given instance. As a consequence, optimization must be carried out by performing an exhaustive search through this transformation search space.

Note that the transformation search space that has been described up till now does not contain any parametric compiler optimizations such as loop blocking and loop unrolling. Rather, the transformations that have been described affect the order in which computation is performed and the manner in which the input data is organized. In fact, application of parametric compiler optimizations is to be done next to the transformations that have been described so far. The exploitation of this parametric optimization search space is the second stage in our approach. We will show in Section 10.6 that by considering the transformation search space combined with a parametrized search space just containing Loop Unrolling, we are able to generate codes that achieve better results than existing library implementations of sparse algebra routines.
Figure 10.4: Quantification of the ranking correlation between a base number of vectors (50 at the top, 10 at the bottom) with increasing number of best variants being considered. Note that the values for \( y \) equal to base are set to 0 for clarity.
10.5 Irregularity Of Other Kernels

In the preceding sections, we have focused on the sparse matrix times $k$ vectors kernel. We have also conducted experiments with two other sparse matrix kernels: sparse matrix matrix multiplication and lower triangular solve for unit matrices. The routines for sparse matrix times $k$ vector multiplication and sparse matrix matrix multiplication have been instantiated with a similar transformation process of the code and data storage format. Because of this, the transformation search space is similar to Figure 10.1. However, for the triangular solve kernel, the transformation space is decreased from 130 to 76 variants, as can be seen in the transformation tree depicted in Figure 10.5.

Figure 10.5: The full transformation tree of triangular solve.

Also in the case of sparse matrix matrix multiplication and triangular solve, the ordering of best-performing variants is highly irregular, as can be seen in Figure 10.6. These plots can be compared with the plot at the top of Figure 10.2,
10.6 Comparison To Existing Sparse Computation Libraries

In this section, a comparison is presented of the performance of sparse matrix codes generated with our framework to existing sparse algebra libraries. A detailed comparison is given for the sparse matrix times vector multiplication kernel with $k = 1$, because this routine is present in every sparse algebra library. The codes that have been generated by our framework follow from an exhaustive search through the described transformation search space, combined with a search through different possible Loop Unroll parameters.

The comparison comprises the following sparse algebra libraries: Blaze 1.2 [43], with the matrix stored in both row-major and column-major order; MTL4 [40], with the matrix stored in both row-major and column-major order; SPARSE1.3 [59]; SparseLib++ 1.7 [32], with the matrix stored in coordinate storage format, compressed row storage format and compressed column storage format.

The experiments have been performed on two architectures. The first architecture has already been introduced in this chapter and consists of an Intel Xeon 5150 CPU at 2.66 Ghz, with 16GB RAM, running Ubuntu Linux 10.04.4. The second architecture is a machine that consists of an Intel Xeon E5-2650 CPU at 2.00 GHz, with 64 GB RAM, running CentOS 5.0. The compiler used in both cases is gcc 4.4. These architectures will be referred to as the Xeon 5150 and Xeon E5 architectures respectively. Twenty matrices have been surveyed, taken from the University of Florida Matrix Collection [28]. To remove fluctuation from the results, the computation performed by each variant or library is repeated 10 times.

The results for the Xeon 5150 architecture are shown in Figure 10.7. In this figure, the result of the smaller matrices and the results of SPARSE1.3 have been omitted for the sake of legibility. The matrices along the x-axis have been ordered in increasing execution time for the code generated with our framework. The execution time is arranged along the y-axis. From the figure can be observed that although the row-major order variant of the different libraries is competitive in performance, the code generated using our approach is always faster.

Figure 10.8 shows the speedup in execution time of the code generated by our approach over the best-performing implementation from the existing sparse algebra libraries. For the majority of instances, our approach realizes at least a factor 1.1 speedup in performance. In some cases, such as G2.circuit and Raj1 for the Xeon 5150, and or2010 for the Xeon E5, the code generated using our framework is significantly faster, achieving speedups up to a factor of 1.9.

10.6.1 Search Space Reduction for Loop Unroll Optimization

In this section, we presented the results attained by our approach that followed from an exhaustive search of the transformation search space and the parametrized
Figure 10.6: The plots show the 200 best-performing experiment instances for the sparse matrix matrix multiplication (above) and triangular solve (below) kernels.
10.6. Comparison To Existing Sparse Computation Libraries

Figure 10.7: Comparison in execution time of automatically instantiated sparse matrix times vector computation and the different sparse algebra libraries on the Xeon 5150 architecture.

Figure 10.8: Speedup in execution time of the automatically instantiated sparse matrix times vector computation compared to the best-performing implementation from the existing sparse algebra libraries.
optimization search space. So far, only Loop Unrolling has been considered in the parametrized search. When other optimizations are included in the search, such as loop blocking, the size of the search space will grow substantially. Therefore, it is important to find methods to reduce the parametrized search space.

An often used technique for finding good parameter settings for parametric optimizations is Iterative Compilation [54, 34]. Note that Iterative Compilation is constrained to what we have described as the parametrized search space. In our case, techniques are needed for finding instances that are optimal with respect to the transformation search space and parametrized search space. For example, a variant that is not optimal within the transformation search space, may become the optimal variant after searching through the parametrized search space. The technique must be able to deal with these “cross-overs”.

Figure 10.9 displays the relationship between the percentage of performance an instance is distanced from the optimum for an unroll level of 1 and the global optimum, considering all unroll levels. The execution time of the instances on the Xeon E5 architecture are shown. The lines represent different matrices. In the figure, it can be seen that variants at an unroll level of 1 must be searched up to about 28% distanced from the optimal performance found for an unroll level of 1, in order to find the global optimum. For the majority of matrices surveyed, however, the global optimum will be found by exploring the variants with a performance within 18% of the optimum for an unroll level of 1. This is a good indicator that it is not necessary to search the full parametrized search space to be able to eliminate the problem of cross-overs.

The extent of the parametrized search space that must be searched in order to find a global optimum is shown in Figure 10.10. Because the variants for an unroll level of 1 must always be searched, the extent of the search space that must be explored is at least 12.5%. For all matrices surveyed, it is only necessary to explore up to 19.0% of the parametrized search space in order to find a global optimum. This is a significant reduction compared to an exhaustive exploration of the search space, indicating the feasibility of pruning the parametrized search space in our approach.

10.6.2 Other Kernels

We also conducted the above described experiments on two other kernels: sparse matrix times matrix multiplication (with a 100-column dense matrix) and lower triangular solve with unit matrices. These experiments have been conducted on the Xeon 5150 architecture. The comparison with sparse matrix matrix multiplication from existing sparse algebra libraries has only been carried out with Blaze and MTL4, because SPARSE1.3 and SparseLib++ did not contain API for this computation. The \( SPMM \) column of Table 10.1, reports the speedups attained by the generated routine over the fastest implementation from an existing library. In all cases, a speedup of at least 1.17 is achieved, which is a decent improvement. For several matrices, the generated routine achieves an impressive speedup beyond a factor of 2 up to a speedup of a factor of 2.37.

An implementation of triangular solve is only found in the MTL4 and SparseLib++ libraries. The results are shown in the \( TrSv \) column of Table 10.1.
Figure 10.9: The relationship is shown between the percentage of performance an instance is distanced from the optimum for an unroll level of 1 and the global optimum, considering all unroll levels. The lines represent different matrices.

Figure 10.10: This figure illustrates the extent of the parametrized search space that must be searched to find a global optimum. The lines represent different matrices.
Table 10.1: Speedup in execution time of the sparse matrix matrix multiplication and triangular solve kernels generated using our approach compared to the best-performing implementation from the existing sparse algebra libraries on the Xeon 5150 architecture.

In the majority of cases, a speedup is realized by the automatically generated routines, as big as a factor of 1.59 to 2.21 for a number of cases. For four cases however, a slight decrease in performance is seen. The decrease is so slight that the performance of the automatically generated routine is on par with the fastest library routine found.

In the previous subsection it was shown that it is only necessary to search up to 19% of the parametrized search space to find a global optimum. Similar results are found for the two kernels discussed in this subsection: up to 17% of the search space needs to be searched for sparse matrix matrix multiplication and up to 16% for triangular solve. So, also for other kernels it is feasible to significantly prune the search space.

### 10.7 Conclusions

In this chapter, we have described our approach for the automatic instantiation of efficient sparse algebra routines. This instantiation is combined with the reassembly of the original sparse matrix data structure into a form that is better aligned with the computation performed by the instantiated code. The transformations that lead to these different instantiations set up a large transformation search space. We have shown this search space to be very erratic and as a consequence no manageable subset of best-performing implementations can be chosen to be collected in a library. Rather, optimization must be carried out by performing an exhaustive search through this transformation search space.

We have shown that in addition to this transformation search space, parametric optimizations, such as loop unrolling and loop blocking, must be considered. These parametric optimizations set up a parametrized search space, further expanding the size of the search space as a whole. The size of the resulting search space is two orders of magnitude larger than search spaces derived in Iterative Compilation. Evidence has been presented that the search through the parametrized search space can be pruned to 20%, while a global optimum is still found.
Finally, we have shown that when this search space is properly exploited this results in variants of the computation that are faster than the implementations found in existing sparse algebra libraries, or at least equivalent in performance. Based on this, we argue that in order to get the maximum performance out of an architecture one cannot count on existing libraries, but an exhaustive search through a series of automatically instantiated routines must be performed.