Sparsity-Aware Storage Format Selection

Kazem Cheshmi, Leila Cheshmi, Maryam Mehri Dehnavi
Rutgers University, Department of Electrical and Computer Engineering, Piscataway NJ, USA
{kazem.ch; maryam.mehri}@rutgers.edu, leila.cheshmi@gmail.com

POSTER PAPER

***Abstract*—The simulations in many scientific applications require the solution to linear systems of equations. Libraries are often used for high-performance executions of direct solvers of linear systems. However, the performance of these libraries is often tied to the underlying hardware platform and the code is not portable. We propose to develop a framework called Axb, which will use information about the architecture and the application. Axb automatically selects a good storage format and implementations that generate efficient code for the sparse direct solver. The results demonstrate that the performance of direct solvers can be accelerated upto 11X compared to using a library approach for the direct solver on multicore architectures. The work is an extension to our previous publication [1].**

***Keywords-component*—Numerical methods, direct solvers, parallelization, sparse computation.**

I. Introduction

The performance of many scientific simulations on high-performance computing platforms depends heavily on the optimizations used to implement sparse direct solvers. Standard approaches of using libraries to obtain highly optimized code for direct solvers suffers from several insufficiencies since the performance of the program is often tied to the underlying architecture, problem instance, and algorithmic features [2]. The libraries generally have to be hand-tuned for the underlying architecture and their performance is typically sacrificed for code maintainability. As a result, libraries might not produce high-performance and scalable code for direct solvers.

Our framework Axb has access to source code, data input, and architecture specifications, thus, it can provide useful information for optimization. The contribution of this work is building a framework named Axb to automatically generate highly optimized code for direct solvers for a target architecture. Following enumerates the internal features of Axb:

- Detection of the input matrix sparsity pattern using runtime information or programmer input;
- Selection of the best direct solver types and optimization techniques for a given linear system;
- Detection of the most optimal sparse storage format;
- The application of necessary code transformations based on the target solver and storage format.

The combination of the above-mentioned features allows Axb to outperform libraries. The remainder of this paper is organized as follows: a brief introduction to sparse storage formats and direct solvers integrated in Axb can be found in Section II; Section III provides a short survey of available libraries for sparse direct solvers and indicates their shortcomings compared to domain-specific compilers; The internals of the Axb framework are presented in the following section; Finally, results and conclusions are presented in sections V and VI respectively.

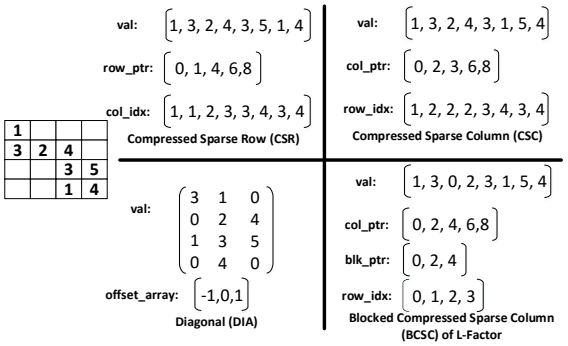


Figure 1: Different sparse storage format representations.

II. Preliminaries

The Axb framework uses input sparsity information to choose the best implementation strategy for solving linear systems. This section introduces some of the preliminaries necessary to understand the internals of the proposed framework.

A. Sparse Storage Formats

Sparse matrices are typically stored in compressed formats for improved locality and memory usage [3]. Various sparse storage formats exist often characterized by their storage structure. Some of the sparse storage formats used in Axb include: *i)* Compressed sparse column (CSC) where the nonzero elements of the matrix are stored in column order in a val vector. The pointer to the first entry of each column and the corresponding row indicates are stored in col_ptr and row_idx vectors respectively; *ii)* Compressed sparse row (CSR) in contrary to CSC, stores rows in orders using vectors val, row_ptr, and col_idx; *iii)* Blocked CSC [4] and Blocked CSR (BCSC & BCSR) both of which are blocked storage formats for CSC and CSR. In the L-factor of matrix factorizations that are stored in BCSC, columns that have a similar sparsity structure are stored in dense blocks and in addition to the three arrays used for CSC, a row_blk is also used to store the starting point of the corresponding row block

978-1-5386-7879-4/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCS.2018.00162

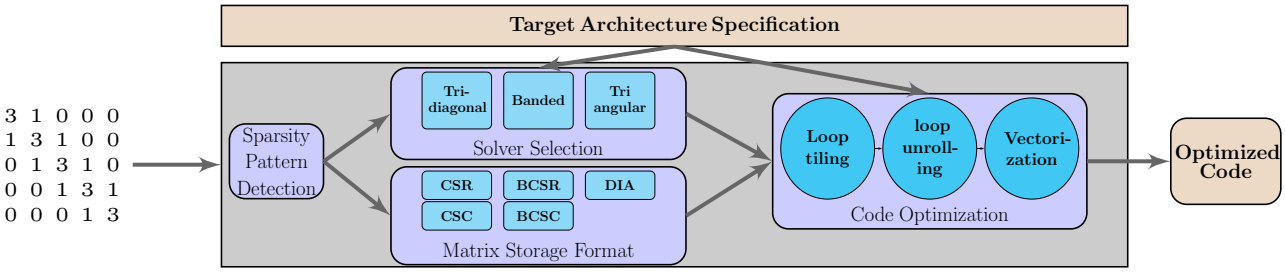


Figure 2: A general overview of the Axb framework.

in the row_idx array; *v)* Diagonal storage format (DIA) where the diagonal and off-diagonal elements are stored in a val vector and the diagonal offsets are kept in an $offset\_array$. Fig. 1 shows some of the illustrated formats used in Axb.

B. Sparse Direct Solvers

Direct solvers are often used to solve sparse linear systems, e.g. $Ax = b$. A is factorized into factors such as lower and/or upper triangular matrices using methods such as the Cholesky algorithm and LU decomposition. The resulting factors are then used in a triangular system solve stage to find the solution to the linear system. Various methods exist for implementing a factorization algorithm as well as solving a triangular system. Also, for matrices with better structured sparsity patterns, e.g. a narrow-banded matrix, more optimized direct solver implementations can be used.

III. Related Work and Challenges

In this section we will briefly overview available libraries for sparse direct solvers. The survey will motivate our approach to building a framework which will be elaborated in subsequent sections. Numerous libraries [2], [4]–[9] have been developed to solve sparse linear systems using direct methods. To perform best for a domain or on a target hardware platform many of these libraries are hand-tuned for specific inputs, domains, and hardware architectures. For example, SPIKE [6] and the library proposed by Zhang *et. al.* [5] are designed to best solve banded and tridiagonal matrices. Cholmod [7] is designed for symmetric positive definite matrices. Other libraries such as SuperLU [4] and UMFPACK [10] are hand-tuned for a larger class of inputs. To tailor multiple hardware platforms, libraries such as SuperLU have three different distributions, each for a different hardware platform (serial architectures, multicore processors, and distributed systems). Each of their library implementations demand separate tuning and tedious efforts for hardware-specific optimizations.

As discussed, libraries for sparse solvers typically need to be hand-tuned for different hardware platforms and inputs. Thus, domain experts often have to test their application on a number of these libraries to find the best performing implementation. Changes in the hardware platform also demand a revisit to the library and the tuned code. Furthermore, in order to apply low-level optimizations, tuned libraries sacrifice code readability for better performance which increases the overhead to maintain such codes. All of these challenges in using libraries, has motivated domain-specific compiler and framework development. Our Axb framework is the first for sparse direct solvers that automatically makes many of the design choices a user has to make if they were to use libraries. Highly optimized code is generated by Axb through automatic analysis of the input data structures, problem characteristics, and the underlying hardware platform.

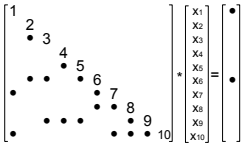
IV. Our approach: The Axb Framework

This section demonstrates the motivation behind building Axb, its structures, and its internals. The objectives and contributions of our work are also presented here.

A. Motivation

The choice of a solver and its implementation strategy depends on the numerical characteristics of the original matrix and its sparsity pattern. Such choices have a direct effect on the performance and execution time of the sparse solver. For example, if the numerics of the input matrix indicate a symmetric positive definite matrix, Cholesky decomposition factorization methods can be used instead of the typically less efficient Gaussian elimination methods. The sparsity structure of the input matrix also plays a vital role on determining the best solver implementation. For example, sparsity patterns such as tridiagonal, banded, triangular, and symmetric allow using a simplified and easier to optimize solver implementation that improves the direct solver's performance. Often matrix reordering methods are used to convert the sparsity structure of a matrix to one of these patterns and facilitate more optimized implementations. Reverse cuthill mckee (RCM) ordering, is commonly used in scientific simulations to create a banded matrix. Other reordering methods such as approximate minimum degree (AMD) ordering are used to restructure the sparsity pattern of the input matrix to increase inherent parallelism in factorization codes. Also, changes in the underlying hardware platform typically demands re-tuning the optimized code. Many of the above-mentioned design choices complicates the process of choosing the best library or optimized code by domain experts.

A domain-specific framework such as the proposed Axb can overcome many of these limitations. It can automatically explore the large optimization space of a high-performance implementation and select the best optimization strategy based



(a) Triangular system

```
for (j=0;j<n;j++){
  x[j]/=val[col_ptr[j]];
  for(p=col_ptr[j]+1;p<col_ptr[j+1];p++)
  {
    x[row_idx[p]]-=val[p]*x[j];
  }
}
```

(c) CSC format

```
for(j=0;j<n;j++){
  sum=0;
  for(p=row_ptr[j]+1;p<row_ptr[j+1];p++)
  {
    sum+=val[p]*x[col_idx[p]];
  }
  x[j]/=val[row_ptr[j+1]-1];
}
```

(d) CSR format

```
for (int i = 0; i < blkNo; ++i) {// for each block
   int curCol = i!=0 ? blk2col[i-1]:0;
   int nxtCol = blk2col[i];
   int blkWdt = nxtCol-curCol;
   int nblkR = row_blk[nxtCol]-row_blk[curCol];
   double *Ltrng = &val[col_ptr[curCol]];//first nnz of current block
   for (int i = 0; i < blkWdt; ++i) {
       x[curCol+i]/=Ltrng[i*nblkR+i];
       for (int j = i+1; j < blkWdt; ++j) {
           x[curCol+j]-=Ltrng[i*nblkR+j]*x[curCol+i];
       }
   }
   Ltrng = &val[col_ptr[curCol]+blkWdt];//first nnz of below diagonal
   for (int i = 0; i < blkWdt; ++i) {
       for (int j = 0; j < nblkR-blkWdt; ++j) {
           tempVec[j] += Ltrng[i*nblkR+j]*x[i+curCol];
       }
   }
   for (int l = row_blk[curCol]+blkWdt,k=0; l < row_blk[nxtCol]; ++l, ++k) {
       x[row_idx[l]]-=tempVec[k];
       tempVec[k]=0;
   }
}
```

(b) Blocked CSC

Figure 3: The generated code by Axb for a triangular solver using different memory storage formats.

on a given sparsity pattern. To the best of our knowledge, Axb is the first domain-specific framework for direct solvers.

B. Axb Internals

Our Axb framework has three layers. The middle-end shown in Fig. 2, is specifically customized to analyze input sparse structures and apply optimizations for direct solvers. The sparsity pattern of the input matrix is first detected. The most efficient solver type is then automatically determined by the framework for the detected sparsity pattern. Axb constructs the communication pattern of the algorithm for the selected solver to identify the best matrix storage format. Finally, code optimization passes to the generated code are applied in Axb using architecture information.

As shown in Fig. 2, using two different classes of code transformations, namely algorithmic transformations and standard optimizations, Axb changes the original unoptimized direct solver code to highly-optimized and automatically tuned code: *i)* in the algorithmic transformation phase the framework first chooses the best performing solver type and storage format. Current solver types supported in Axb are tridiagonal, banded, triangular, and the Gaussian elimination solver. A sparse storage format is then selected based on the input sparsity pattern and the solver type. The CSR, CSC, DIA, and blocked CSC storage formats are currently supported in Axb; *ii)* in the proceeding standard optimization stage various low-level optimization strategies such as tiling, unrolling, factorization, and parallelization are applied to the transformed code. These low-level optimizations tune the generated code to increase memory throughput and data locality on the underlying hardware platform.

To better illustrate the transformations in Axb, Fig. 3a shows the space of transformations for a triangular system solver and demonstrates how our framework generates optimized code for such a setting. In the first step, the sparsity pattern of the matrix is detected as triangular which leads to the selection of the forward substitution algorithm. Then based on the matrix structure, the application requirements, and the target architecture, Axb selects a sparse storage format. For example, if the selected solver type accesses matrix elements in column- or row-major order, the transformations shown in Fig. 3c and Fig. 3d are applied respectively. If the input matrix has many consecutive columns of similar sparsity patterns, the Blocked CSC storage format will be used. Fig. 3b shows the Axb generated code for the forward substitution method using the blocked CSC format. A number of standard code transformations such as loop unrolling will also be applied to the transformed code in Fig 3. In experimental results we show how such transformations can lead to better performance.

V. Experimental Results

This section presents experimental results to demonstrate the performance of our Axb framework. Section V-A shows the effects of selecting the best solver type and sparse storage formats on the performance of direct solvers. In Section V-B, we show how Axb can generate high-performance code that outperforms available libraries by detecting and analyzing the sparsity structure of the input matrix. Our results are obtained on an Intel core i7-5820K processor. All matrices are selected from the University of Florida Matrix repository [11]. Matrix names starting with SYN are synthetically generated.

A. Sparsity and Memory Storage Selection

We first show the effects of solver type selection in improving the performance of the Axb generated code compared to Eigen [12], a highly-optimized library for sparse matrix solvers. The results are tested using three synthetically generated tridiagonal matrices with ranks 16K, 32K, and 64K. For tridiagonal problems, Axb uses the cyclic reduction algorithm from [5]. The CSC storage format is selected in both Eigen and the Axb generated code. Table I shows the Axb generated code is up 5.8X faster than Eigen which uses an LU decomposition method. Since the sparse storage format is the same in both settings, this improvement is mainly the result of detecting a more efficient solver by Axb.

Fig. 4 shows the importance of choosing a well-performing storage format by Axb. We chose a number of triangular matrices as inputs to our framework. These inputs were obtained by factorizing a set of matrices from the UF repository collection. Because of inserted fill-ins during the factorization process, all

of these matrices have a blocked structure. Axb automatically detects that the input matrix is blocked and thus uses the BCSC storage format. As shown in Fig. 4 between the three different sparse storage formats CSR, CSC, and blocked CSC, the BCSC format selected by Axb outperforms others.

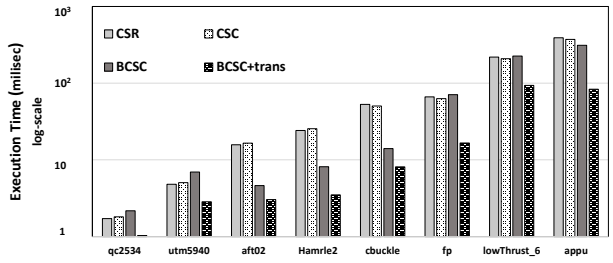


Figure 4: The performance of triangular solve using different sparse storage formats.

Another important observation from Fig. 4 is the effects of applying standard code transformations in Axb after choosing the storage format. The $conv\_BCSC$ and $BCSC$ values in the figure demonstrate the Axb generated code with and without applying standard transformations respectively. As shown, the transformations increase the performance up to 76 percent for matrix fp.

Table I: PERFORMANCE COMPARISON FOR DIFFERENT SOLVER TYPES. AXB CHOOSES A TRIDIAGONAL SOLVER WHILE EIGEN USES LU FACTORIZATION; BOTH TOOLS USE THE CSC STORAGE FORMAT.

Matrix	**Axb** (s)	**Eigen [12] (s)**
SYN16K	0.045	0.261
SYN32K	0.115	0.584
SYN64K	0.194	0.971

B. Axb vs. libraries

Fig. 5 demonstrates that Axb outperforms hand-tuned libraries via selecting a proper solver type and an efficient sparse storage format at the same time. We added some extra matrices to explore all supported sparsity patterns in the current version of Axb. The solver and the storage formats selected by Axb are indicated below the matrix name in Fig. 5. The highest improvement is for matrices $TSPOF\_b39\_c19$ and $qc2534$ which are from electromagnetic and circuit simulations. For both cases the solver type plays an important role in improving the overall performance. In some matrices such as fp, the effects of sparse storage formats are more important because both the framework and the library use an LU solver thus selecting BCSC leads to better gains. Axb outperforms Eigen in other matrices as well.

VI. CONCLUSION AND FUTURE WORK

This work presents a domain-specific framework called Axb for sparse direct solvers. By analyzing the sparsity structure of the input problem and integrating domain-related information, our framework efficiently explores the space of optimizations to generate highly-optimized code for sparse solvers. For

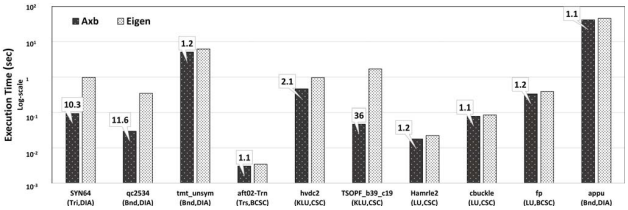


Figure 5: Performance of Axb generated code vs. Eigen [12].

future work we plan to extend Axb to support other solvers and a larger class of storage formats. We also plan to extend our framework to generate code for GPU architectures. Finally, future extensions of our framework will use an LLVM backend to directly generate target architecture binary code.

REFERENCES

[1] K. Cheshmi, G. Xu, S. A. Zonouz, and M. M. Dehnavi, "Axb: A compiler for sparse direct solvers," in *Electromagnetic Field Computation (CEFC), 2016 IEEE Conference on*. IEEE, 2016, pp. 1–1.

[2] T. Davis, *Direct methods for sparse linear systems*. Siam, 2006, vol. 2.

[3] M. M. Dehnavi, D. M. Fernández, and D. Giannacopoulos, "Finite-element sparse matrix vector multiplication on graphic processing units," *IEEE Transactions on Magnetics*, vol. 46, no. 8, pp. 2982–2985, 2010.

[4] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. Liu, "A supernodal approach to sparse partial pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 3, pp. 720–755, 1999.

[5] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the gpu," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 127–136, 2010.

[6] M. Manguoglu, A. H. Sameh, and O. Schenk, "Pspike: A parallel hybrid sparse linear system solver," in *European Conference on Parallel Processing*. Springer, 2009, pp. 797–808.

[7] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 3, p. 22, 2008.

[8] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: Klu, a direct sparse solver for circuit simulation problems," *ACM Transactions on Mathematical Software (TOMS)*, vol. 37, no. 3, p. 36, 2010.

[9] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 3, p. 22, 2008.

[10] T. A. Davis, "Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 196–199, 2004.

[11] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[12] G. Guennebaud, B. Jacob, P. Avery, A. Bachrach, S. Barthelemy *et al.*, "Eigen v3," 2010.