Optimizing LOBPCG: Sparse Matrix Loop and Data Transformations in Action

Khalid Ahmad, Anand Venkat, and Mary Hall

University of Utah, Salt Lake City UT 84112, USA
{khalid,anandv,mhall}@cs.utah.edu,
http://ctop.cs.utah.edu/ctop/

Abstract. Sparse matrix computations are widely used in iterative solvers; they are notoriously memory bound and typically yield poor performance on modern architectures. A common optimization strategy for such computations is to rely on specialized representations that exploit the nonzero structure of the sparse matrix in an application-specific way. Recent research has developed loop and data transformations for sparse matrix computations in a polyhedral compilation framework. In this paper, we apply these and additional loop transformations to a real application code, the LOBPCG solver, which performs a Sparse Matrix Multi-Vector (SpMM) computation at each iteration. The paper presents the transformation derivation for this application code and resulting performance. The compiler-generated code attains a speedup of up to 8.26x on 8 threads on an Intel Haswell and 30 GFlops; it outperforms a state-of-the-art manually-written Fortran implementation by 3%.

1 Introduction

Sparse matrix computations arise in numerous engineering and science applications. Sparse matrices are represented by data structures that store only nonzero elements, with additional auxiliary structures to identify the corresponding row and column of each element [1, 2]. Consider the representative sparse matrix-vector multiplication (SpMV), a performance bottleneck in solving sparse linear systems and eigenvalue problems because it is performed hundreds or thousands of times during a single execution of an application [3]. Frequent indirection through auxiliary arrays and a lack of data reuse lead to low computational intensity, i.e. number of arithmetic operations per memory reference [4]. There is extensive prior work dealing with the development, optimization, and improving the performance of parallel SpMV kernels for both multi-core and many-core architectures, e.g. [1, 3, 5–9]. One common strategy is to specialize a sparse matrix representation to exploit the nonzero structure of the sparse matrix and thus reduce memory accesses and simplify the generated code. This approach usually involves using an optimized library that converts to the desired representation from a standard format such as Compressed Sparse Row (CSR) or Coordinate (COO).

While ideally a compiler can be used to perform these optimizations and data transformations, compilers have been severely limited in their ability to optimize sparse matrix computations due to the indirection that arises in indexing and looping over just the nonzero elements. This indirection gives rise to *non-affine* subscript expressions and loop bounds; i.e., array subscripts and loop bounds are no longer linear expressions of loop indices. A common way of expressing such indirection is through *index arrays* such as, for example, array `B` in the expression `A[B[i]]`. Code generators based on polyhedra scanning are particularly restricted in the presence of non-affine loop bounds or subscripts [10–14]. As a consequence, most parallelizing compilers either give up on optimizing such computations, or apply optimizations very conservatively.

Recent work has developed non-affine support and loop and data transformations in a polyhedral transformation and code generation framework and shown to be effective in optimizing SpMV for multicores and GPUs [15]. In this paper, we demonstrate that such compiler technology can be extended so that it is suitable for the far more complex support required by real applications. We apply our compiler transformations to optimize the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) solver [16]. Specifically, an important kernel within LOBPCG is the sparse matrix multi-vector multiplication (SpMM), which is a generalization of the SpMV kernel in which a sparse m-by-n matrix A is multiplied by a tall and narrow dense n-by-k matrix B (k<<n). SpMM is used in a variety of sparse matrix computations such as those using block Krylov subspace methods for solving several linear systems simultaneously as well as obtaining several eigen pairs of eigenvalue problems, e.g. [17–24]. Other applications that require SpMM operations include: (i) aerodynamic design optimization [25], (i) the search engine PageRank algorithm, and (iii) atmospheric modeling [24]. A characteristic of SpMM is that arithmetic intensity is significantly higher than SpMV if access to the sparse matrix can be reused by the vectors, as clarified in Table 1. In the table, *nnz* refers to the number of nonzero elements and n is the number of columns in the sparse matrix; k is the number of dense vectors.

Table 1. Arithmetic intensity of SpMV and SpMM.

	SpMV	k independent SpMV	SpMM
Flops	2 * nnz	2k * nnz	2k * nnz
Words moved	nnz + 2n	k * nnz +2k * n	nnz + 2k * n

The remainder of the paper will demonstrate the applicability of prior loop and data transformations and the new challenges that arise in optimizing LOBPCG for very large matrices that characterize the application in which it is used [16]. The novel contributions of the paper are as follows: (1) we apply these transformations to automatically generate an inspector that produces a new matrix representation, *compressed sparse block (CSB)*, starting from a standard *compressed sparse row (CSR)*; (2) we generate an optimized SpMM, implemented

for a symmetric matrix by computing both SpMV and SpMV^T (transposed SpMV) [1]; (3) we identify additional optimizations to reduce the data movement for indexing expressions and optimize AVX SIMD execution; and, (4) we demonstrate the collection of optimizations that lead to a 3% performance gain over the manually-written state-of-the-art Fortran implementation [16].

The remainder of the paper is organized as follows. The next two sections provide background on the CSR, COO and CSB storage formats, inspector/executor, the compiler approach and the LOBPCG solver. Section 4 provides the compiler derivation of the optimized inspector and executor. We then discuss the experimental setup and provide a performance comparison of the compiler-generated code and the manual code. Section 6 discusses related work. Finally, we conclude this work with a summary of contributions and ideas for possible future work.

Fig. 1. A 6*6 example sparse matrix.

11	12	13	14	0	0
0	22	23	0	0	0
0	0	33	34	35	36
0	0	0	44	45	0
0	0	0	0	0	56
0	0	0	0	0	66

Fig. 2. The COO representation.

data	11	12	13	14	22	23	33	34	35	36	44	45	56	66

row index	0	0	0	0	1	1	2	2	2	2	3	3	4	5

column index	0	1	2	3	1	2	2	3	4	5	3	4	5	5

Fig. 3. The CSR representation.

data	11	12	13	14	22	23	33	34	35	36
							44	45	56	66

row pointer	0	4	6	10	12	13	14

column index	0	1	2	3	1	2	2	3	4	5	3	4	5	5

Fig. 4. The CSB representation.

data	11	12	22	13	14	23	33	34	44	35	36	45	56	66

blkptr	0,0	0,1	1,1	1,2	2,2	

row index	0	0	1	0	0	1	0	0	1	0	0	1	0	1
column index	0	1	1	0	1	0	0	1	1	0	1	0	1	1

2 Background

The remainder of the paper relies on understanding sparse matrix storage formats, inspector/executor paradigm and an overview of the compiler approach, all briefly described in this section.

2.1 Storage Formats

Coordinate Storage Format (COO). COO is often used as the entry format in sparse matrix packages [1, 26]. In COO, a *data* vector stores the nonzero elements of the matrix and two integer vectors, *row* and *column*, store the row

and column indices of the corresponding nonzero elements in the *data* vector. Although nonzero elements and their corresponding indices can be stored in any order, they are usually stored by ascending row order.The amount of required storage is proportional to the number of nonzero elements. Figure 2 shows an example of storing a matrix using COO.

Compressed Sparse Row (CSR). Like COO, CSR (see Figure 3) stores the nonzero elements of the matrix in a *data* array and column indices in an integer array. The third array stores pointers to the beginning of each row of the matrix in the *data* and *columns* arrays. The *rowpointers* array is of size N+1, where N is the number of matrix rows. The last element in the *rowpointer* array contains the total number of nonzero elements in the matrix. CSR requires less storage for row indices. In addition, the *rowpointer* array allows for easy computations of some quantities of interest for a matrix such as the number of nonzero elements in a row $i = ptr[i+1] - ptr[i]$ and the total number of nonzero elements $ptr[N+1]$.

Compressed Sparse Block (CSB). In the CSB format, matrix A is partitioned into small blocks and each block is treated as a COO matrix. CSB consists of three arrays *blkptr*, *indices*, and *data*. Array *blkptr* is a two-dimensional array storing the offset of the first nonzero of each block. The *indexarray* stores the concatenated row and column indices of nonzeros in a block; in Figure 4, *row* and *columnindices* are shown separately. Array *data* stores nonzeros. In CSB, a row (column) of blocks is designated as a blockrow (blockcolumn).

2.2 Inspector/Executor

A general technique to analyze data accesses through index arrays and consequently reschedule or reorder data at run time employs an *inspector/executor* paradigm whereby the compiler generates *inspector* code to be executed at runtime that can collect the index expressions and then an *executor* employs specific optimizations that incorporate the run-time information [27–30]. These inspector/executor optimizations have targeted parallelization and communication [28, 31] and data reorganization [32–36].

2.3 Overview of Approach

In this paper, we employ an inspector in conjunction with data transformations to convert a symmetric matrix from CSR to CSB format, and generate an optimized, parallel executor for the CSB representation. The generation of both optimized inspector and executor is performed by the CHiLL polyhedral transformation and code generation framework. CHiLL's operations are driven by a transformation recipe which specifies the functions and loops to optimize and the transformations to apply.

Recent work has extended CHiLL to support non-affine computations that incorporate indirection through index arrays [37, 15, 38]. CHiLL is able to tolerate and maninputate non-affine loop bounds and array access expressions using the abstraction of *uninterpreted function symbols*, expanding on their use in Omega [11]. Data transformations are composed with standard and non-affine transformations in [15] to convert between matrix formats and realize optimized executors by introducing transformations used in this paper, described as follows:

- *make-dense* takes as input a set of non-affine array index expressions and introduces a guard condition and as many dense loops as necessary to replace the non-affine index expressions with affine accesses. The *make-dense* transformation enables further affine loop transforamtions such as tiling.
- *compact* and *compact-and-pad* are *inspector-executor* transformations; an automatically generated inspector gathers the iterations of a dense loop that are actually executed and the optimized executor only visits those iterations. The executor represents the transformed code that uses the compacted loop, which can then be further optimized.
- Using *compact-and-pad*, the inspector also performs a data transformation, inserting explicit zeros when necessary to correspond with the optimized executor. In this paper, *compact-and-pad* is used to reorder the data, but does not add zeros.

In the remainder of the paper, we will describe LOBPCG and then present how these transformations and others are used to derive an optimized implementation.

3 LOBPCG

LOBPCG is a subspace iteration method which starts with an initial guess about the eigenvectors and refines the guess at each iteration of the solver [16]. It is used in the Many-body Fermion Dynamics for nuclei (MFDn) application to study the structure of light nuclei. At the heart of LOBPCG lies SpMM, which multiplies a sparse matrix with multiple dense eigenvectors. Due to the very large size of the input matrix used, the symmetry of the matrix is exploited to store only half of the matrix entries to optimize for memory footprint. Since the matrix is symmetric, performing SpMM using the entire matrix is accomplished by SpMM

```
for(i=0; i<n; i++)
  for(j=index[i]; j<index[i+1]; j++)
    for(k=0; k<m ; k++)
      y[i][k]+= A[j]*x[col[j]][k];
```

(a) SpMM code using CSR format.

```
for(i=0; i<n; i++)
  for(j=index[i]; j<index[i+1]; j++)
    for(k=0; k<m; k++)
      y[col[j]][k]+= A[j]*x[i][k];
```

(b) SpMM^T code using CSR format.

Fig. 5. SpMM for symmetric matrix requires a matrix representation suitable for two separate computations.

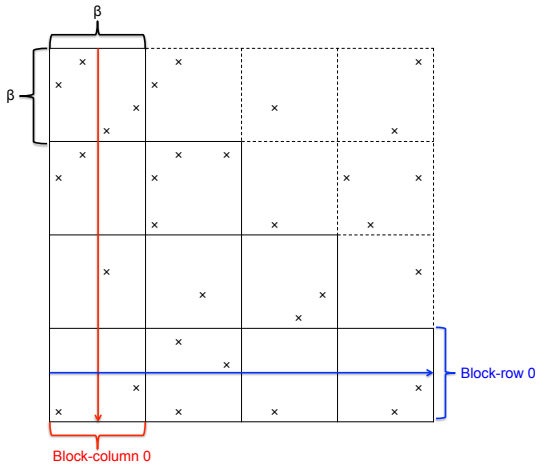

Fig. 6. Parallelization strategy using CSB format. Nonzeros are represented by crosses. Input matrix is blocked into $\beta \times \beta$ blocks. Blocks with dotted boundaries represent symmetric portion of matrix which is not stored. SpMM is parallelized by block rows, while transposed SpMM is parallelized by block columns.

over half of the symmetric matrix, followed by an additional transposed SpMM (SpMM^T) over the same half.

SpMM can be trivially parallelized using CSR format by computing each row computation in parallel. However computing the SpMM^T in parallel using the CSR format is difficult due to write conflicts on the output vector when the row computations are parallelized. The Compressed Sparse Column (CSC) format might be ideal for parallelization of SpMM^T, but then a similar problem would arise for computing SpMM using CSC.

The CSB format solves this problem by blocking the actual matrix dimensions into square blocks of $\beta \times \beta$. It then determines the nonzeros falling in each block and stores them in the COO format in addition to storing the start and end offsets of each block. Now, SpMM can be parallelized by block rows since they do not have any write conflicts and SpMM^T can be parallelized by block column without conflicts. This strategy is illustrated in Figure 6. Block-column-wise parallelization for SpMM^T is indicated by a vertical line while block-row-wise parallelization is indicated by a horizontal line. The tiles with dotted boundaries are actually not stored but serve to illustrate that the actual matrix is symmetric.

4 Compiler Approach

This section describes how make-dense and compact-and-pad are used to derive the parallel SpMM implementation for symmetric matrices using CSB format,

```
source: csb_v2.c # SpMM
procedure: csb
format : rose
loop: 0

original()
remove_dep(0,1)
fuse([0,1], 2)
split_with_alignment(0,1,4096)
split_with_alignment(1,1,4096)

make_dense(0,2,k)
known(lb == 0)
known(ub == 2412565)
known(n == 2412469)

#tile outer row and col loops by 4096
tile(0,2,4096,1,counted)
tile(0,2,4096,1,counted)

#normalize tiled loops
shift_to(0,4,0)
shift_to(0,3,0)
```

```
compact(0,[3,4],[A_prime], 0, [A])

distribute([0,1,2,3], 1)
permute(1,1,[2,1])

#OpenMP code generation
mark_omp_threads(0,[0])
mark_omp_threads(1,[0])
mark_omp_threads(2,[0])
mark_omp_threads(3,[0])

# simd code generation
mark_pragma(0,4, simd)
mark_pragma(1,4, simd)
mark_pragma(2,3, simd)
mark_pragma(3,3, simd)

#set number of OpenMP threads
omp_par_for(1,1,8)

known(index_ < index__)
known(m > 1)
```

Fig. 7. CHiLL script for SpMM based on the CSB format.

which will be integrated into LOBPCG. The complete CHiLL transformation recipe is shown in Figure 7. In this section, we focus on the effects of *make-dense* and *compact-and-pad*[1], and describe additional optimizations needed to further reduce the memory footprint and exploit SIMD execution.

4.1 Compiler-Generated Inspector to Derive CSB Representation

To expose the dense loops that correspond to the actual dimensions of the matrix, the *make-dense* transformation is firstly called on the SpMM code yielding the intermediate code shown in Figure 8(a). Next, tiling is applied to the two outermost loops to yield the $\beta \times \beta$ blocks in CSB. Here β is the tiling factor in Figure 8(b).

Finally *compact-and-pad* is applied to the consecutive third and fourth loop levels(i, l), which are treated as a single logical loop level. The input sparse matrix is also reorganized by compact-and-pad into a new layout reflecting the updated traversal order of the nonzeros. Additionally the *offset_index*, *expl_index_1* and *expl_index_2* arrays are populated.

[1] Both compact and compact-and-pad use variations of the CHiLL compact command; a matrix is provided as an argument for compact-and-pad.

```
for(i=0; i < n; i++)
  for(l=0; l < n; l++)
    for(j=index[i]; j < index[i+1]; j++)
      for(k=0; k < m ; k++)
        if(l == col[j])
          y[i][k]+= A[j]*x[l][k];
```

(a) SpMM after make-dense.

```
for(ii=0; ii < n/beta; ii++)
  for(ll=0; ll < n/beta; ll++)
    for(i=0; i < beta; i++)
      for(l=0; l < beta; l++)
        for(j=index[ii*beta + i]; j < index[ii*beta+i+1]; j++)
          for(k=0; k < m ; k++)
            if(ll*beta + l == col[j])
              y[ii*beta + i][k]+= A[j]*x[ll*beta + l][k];
```

(b) SpMM after tiling.

```
for (ii = 0; ii <= 587; ii += 1)
    for (ll = 0; ll <= 589; ll += 1) {
      _P1[590 * ii + ll] = 0;
      _P_DATA1[590 * ii + ll + 1] = 0;
    }
  for (ii = 0; ii <= 587; ii += 1)
    for (i = 0; i <= 4095; i += 1)
      for (j = index_(4096 * ii + i); j <= index__(4096 * ii + i) - 1; j += 1) {
        ll = (col[j] - 0) / 4096;
        l = (col[j] - 0) % 4096;
        _P_DATA5 = ((struct a_list *)(malloc(sizeof(struct a_list ) * 1)));
        _P_DATA5 -> next = _P1[590 * ii + ll];
        _P1[590 * ii + ll] = _P_DATA5;
        _P1[590 * ii + ll] -> A = 0;
        _P1[590 * ii + ll] -> col_[0] = i;
        _P1[590 * ii + ll] -> col_[1] = l;
        chill_count_1 += 1;
        _P_DATA1[590 * ii + ll + 1] += 1;
        _P1[590 * ii + ll] -> A = A[j];
      }
  for (ii = 0; ii <= 587; ii += 1) {
    if (ii <= 0) {
      _P_DATA2 = ((unsigned short *)(malloc(sizeof(unsigned short ) * chill_count_1)));
      _P_DATA3 = ((unsigned short *)(malloc(sizeof(unsigned short ) * chill_count_1)));
      A_prime = ((float *)(malloc(sizeof(float ) * chill_count_1)));
    }
    for (ll = 0; ll <= 589; ll += 1) {
      _P_DATA5 = _P1[590 * ii + ll];
      for (newVar0 = 1 - _P_DATA1[590 * ii + ll + 1]; newVar0 <= 0; newVar0 += 1) {
        _P_DATA2[_P_DATA1[590 * ii + ll] - newVar0] = _P_DATA5 -> col_[0];
        _P_DATA3[_P_DATA1[590 * ii + ll] - newVar0] = _P_DATA5 -> col_[1];
        A_prime[(_P_DATA1[590 * ii + ll] - newVar0) * 1] = _P_DATA5 -> A;
        _P_DATA5 = _P_DATA5 -> next;
      }
      _P_DATA1[590 * ii + ll + 1] += _P_DATA1[590 * ii + ll];
    }
  }
```

(c) SpMM generated inspector code.

Fig. 8. Steps of generating the inspector.

The generated inspector is shown in Figure 8(c). The offset of each $\beta \times \beta$ block into the array of nonzeros is stored in `_P_DATA1`. Each entry of the array `_P1` corresponds to a single block, and the block's nonzeros are stored as a linked list because the size of the matrix is unknown. For each nonzero, its block is identified using the indices *ii* and *ll*. These indices specify the entry of `_P1`, whose linked list is appended with the nonzero. The row and column offsets within the block correspond to indices `i` and `l` and are stored in the linked list fields `col_[0]` and `col_[1]` respectively. The total count of nonzeros is stored in `chill_count_1` and the individual nonzero count of each block is stored in the corresponding entry in `_P1`. Once all nonzeros have been gathered, the offset and explicit index arrays are allocated within the memory for the right size. The data is then copied from the linked list to the arrays and, the offset of each block is updated using `_P_DATA1`.

4.2 Optimized Executor

The effect of compact-and-pad additionally results in an optimized executor. The generated CSB code was parallelized using OpenMP directives across block rows for SpMM and block columns for SpMM^T. For transposed SpMM, the two outermost loops were permuted so that the resulting code would be traversed by block columns. A further optimization that reduced the memory footprint of index arrays was declaring the row and column index arrays, or *expl_index_1* and *expl_index_2* within a $\beta \times \beta$ block to be short data type. To detect that the size of the index array used did not exceed the maximum allocatable size with 16 bits, the loop bounds and array access expressions were queried during *compact-and-pad* to verify the maximum possible value of the array index expression.

Also, the innermost loop of SpMM does not carry a dependence, and is data parallel, and hence is parallelized with the SIMD pragma annotation for further performance benefits. The pragma annotation is supplied via the transformation interface with the loop level for the annotation, and the code generator inserts the pragma at this loop level. The final parallelized codes for SpMM and SpMM^T, containing SIMD pragmas are shown in Figure 9.

5 Experimental Evaluation

In this section, we measure performance of the generated combined SpMM and SpMM^T executor code, and compare its performance to the manual FORTRAN implementation in [16].

5.1 Methodology

The experiments were performed on an an Intel i7-4770 (Haswell) CPU with 256KB L1 cache, 1 MB L2 cache, 8MB L3 cache, and 32GB of memory. The clock rate is 3.40GHz frequency, with 4 physical cpu cores and 8 threads. The

```
#pragma omp parallel  private(ii,ll,i,k)
{
  #pragma omp for  schedule(dynamic,1)
  for(ii=0; ii < n/beta; ii++)
    for(ll=0; ll < n/beta; ll++)
      for(i=offset_index[ii][ll]; i < offset_index[ii][ll+1]; i++)
        #pragma simd
        for(k=0; k < m ; k++)
          y[ii*beta + expl_index_1[i]][k]+= A[i]*x[ll*beta + expl_index_2[i]][k];
}
```

(a) Final SpMM parallelized code.

```
#pragma omp parallel  private(ii,ll,i,k)
{
  #pragma omp for  schedule(dynamic,1)
  for(ll=0; ll < n/beta; ll++)
    for(ii=0; ii < n/beta; ii++)
      for(i=offset_index[ii][ll]; i < offset_index[ii][ll+1]; i++)
        #pragma simd
        for(k=0; k < m ; k++)
          y[ii*beta + expl_index_2[i]][k]+= A[i]*x[ll*beta + expl_index_1[i]][k];
}
```

(b) Final SpMM^T parallelized code.

Fig. 9. Optimized parallel executors.

Intel version 15.0.0 compilers were used: the Fortran compiler for the manual code and the C compiler for the generated code.

Because our goal is to examine performance in the context of realistic engineering and scientific problems that are used by MFDn, we consider large matrices arising from the finite element discretization. Half of our application specific test sparse matrix is generated by the original Fortan code in [16]. The matrix has 2412469 rows, 2412566 columns and contains 429895762 nonzero elements stored in single precision to reduce the total size of the file, which makes it easier to handle and less time-consuming to read during program execution.

To obtain reliable timing measurements, each computation is run 100 times, and the median value is recorded. The initialization of the codes are not included in the timings as in a real world situation the environment can be set up once and then reused for a large number of calculations. Performance in GFLOPs can be calculated using the following equation, where nnz is number of nonzeros, nvd is the number of dense vectors, and t is execution time in seconds.

$$GFLOPs = (nnz * 4 * nvd)/(t * 10^9)$$

5.2 Performance Measurements

We show the performance in two ways. In Figure 10, we examine performance as a function of the number of dense vectors and threads using a beta value of 4096. We used a number of different dense vectors = {1, 4, 8, 12, 16, 24, 32, 48, 64, 80, 96} and a number of threads = {1,2,4,6,8,10}. As k increases, we see a significant performance improvement which benefits from parallelization up until about $k = 16$, and then the reuse becomes difficult to fully exploit. For larger values of k, the best performance is achieved on fewer threads.

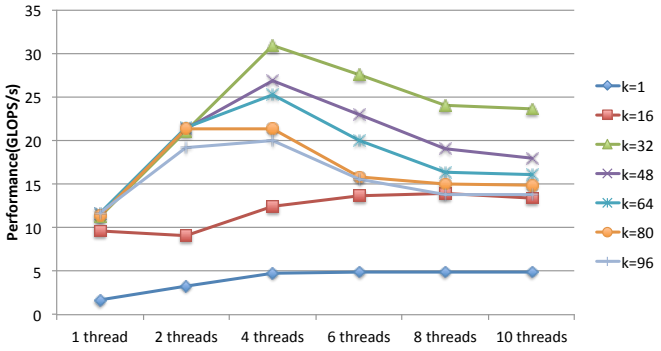

Fig. 10. Performance in GFLOPs of generated implementations for varying numbers of dense vectors and threads.

Figure 11 shows the increase in speedup due to using each optimization from the previous section. We used a fixed blocking factor 4096 in this graph and compute an average speedup from the results of the previous section. The baseline generated code achieves only 0.77x of the performance of the manual Fortran code. By using 16-bit indices for the row and column index arrays (short vector), and inserting the SIMD pragma, the compiler is able to achieve even better performance than the original Fortran code, a speedup of 1.03x. We attribute this difference to simplifications in the indexing that arise in the compiler implementation.

6 Related Work

The majority of literature describes optimized SpMV implementations and strategies targeting different architectures. However, work on SpMM and SpMM^T is not as prevalent.

6.1 Application-specific Approaches

Applications such as biconjugate and quasi-minimal residual iterative linear solvers require computing both SpMV and SpMV^T (transposed SpMV) [1]. Gen-

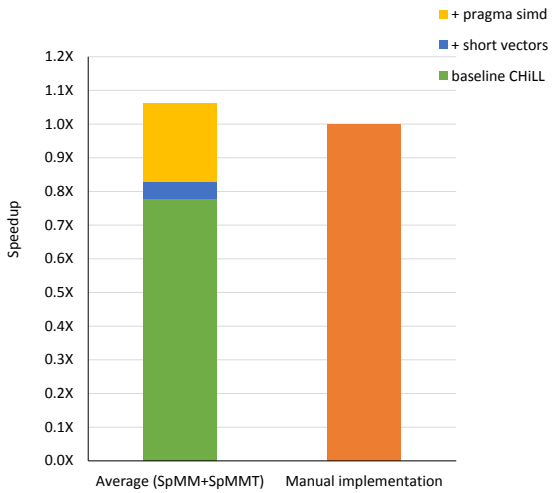

Fig. 11. Speedup attributed to different optimizations in the generated code as compared to the manually-written Fortran code.

erally this problem is solved by transposing the sparse matrix and then performing regular SpMV, and, intuitively this is expensive in space and data movement. Buluc et al. developed the CSB storage format to compute SpMV and SpMV^T at the same time and requires similar storage to CSR or CSC [39]. This representation was then used as part of parallel SpMM and SpMM^T by Aktulga et al. [16].

6.2 Compiler Approaches

Some compiler approaches begin with a dense abstraction of a sparse matrix computation; these compilers then generate sparse data representations during code generation, placing a burden on the compiler to optimize away the sometimes orders of magnitude difference in performance between dense and sparse implementations [40–42]. To our knowledge, the only prior compiler approach that starts with a sparse computation and derives new sparse matrix representations is that of Wijshoff et al. [43]. They convert code with indirect array accesses and loop bounds into dense loops that can then be converted into sparse matrix code using the MT1 compiler [44, 45].

7 Conclusion and Future Work

This paper demonstrated the effectiveness of compiler-generated code for SpMM, when used in the context of the LOBPCG solver on a real-world scientific application at the scale of a problem that fits on a single socket. The key finding is that compiler-generated C code can outperform manual code written in Fortran.

We discovered the importance of 16-bit index arrays and AVX SIMD execution to match the manual code's performance. We found out that the performance benefits when using multiple vectors trails off when the vector becomes too large.

As a continuation of this work, we are exploring the generation of CUDA code, which was not attempted by the application developers. Our future work also includes comparing against the extended CSB implementation for GPUs described in [46] to our implementation.

References

1. Y. Saad, *Iterative methods for sparse linear systems*. Siam, 2003.
2. E. Montagne and A. Ekambaram, "An optimal storage format for sparse matrices," *Information Processing Letters*, vol. 90, no. 2, pp. 87–92, 2004.
3. N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 18.
4. E.-J. Im and K. A. Yelick, *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.
5. H. Anzt, S. Tomov, and J. Dongarra, "Implementing a sparse matrix vector product for the sell-c/sell-c-σ formats on nvidia gpus."
6. M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for modern processors with wide simd units," *arXiv preprint arXiv:1307.6209*, 2013.
7. D. Lowell, J. Godwin, J. Holewinski, D. Karthik, C. Choudary, A. Mametjanov, B. Norris, G. Sabin, P. Sadayappan, and J. Sarich, "Stencil-aware gpu optimization of iterative solvers," *SIAM Journal on Scientific Computing*, vol. 35, no. 5, pp. S209–S228, 2013.
8. J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 115–126.
9. S. Williams, N. Bell, J. Choi, M. Garland, L. Oliker, and R. Vuduc, "Sparse matrix-vector multiplication on multicore and accelerators," *Scientific Computing with Multicore and Accelerators*, pp. 83–109, 2010.
10. C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Apr. 1991.
11. W. A. Kelly, "Optimization within a unified transformation framework," Ph.D. dissertation, University of Maryland, Dec. 1996.
12. F. Quilleré and S. Rajopadhye, "Generation of efficient nested loops from polyhedra," *International Journal of Parallel Programming*, vol. 28, no. 5, pp. 469–498, Oct. 2000.
13. N. Vasilache, C. Bastoul, and A. Cohen, "Polyhedral code generation in the real world," in *Proceedings of the 15th International Conference on Compiler Construction*, Mar. 2006.
14. C. Chen, "Polyhedra scanning revisited," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12, Jun. 2012, pp. 499–508.
15. A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

16. H. M. Aktulga, A. Buluc, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1213–1222.
17. I. Yamazaki, T. Dong, R. Solcà, S. Tomov, J. Dongarra, and T. Schulthess, "Tridiagonalization of a dense symmetric matrix on multiple gpus and its application to symmetric eigenvalue problems," *Concurrency and Computation: Practice and Experience*, 2013.
18. I. Yamazaki, H. Tadano, T. Sakurai, and T. Ikegami, "Performance comparison of parallel eigensolvers based on a contour integral method and a lanczos method," *Parallel Computing*, vol. 39, no. 6, pp. 280–290, 2013.
19. C. Campos and J. E. Roman, "Strategies for spectrum slicing based on restarted lanczos methods," *Numerical Algorithms*, vol. 60, no. 2, pp. 279–295, 2012.
20. K. Meerbergen and R. Vandebril, "A reflection on the implicitly restarted arnoldi method for computing eigenvalues near a vertical line," *Linear Algebra and its Applications*, vol. 436, no. 8, pp. 2828–2844, 2012.
21. R. B. Morgan and D. A. Nicely, "Restarting the nonsymmetric lanczos algorithm for eigenvalues and linear equations including multiple right-hand sides," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 3037–3056, 2011.
22. W. Jiang and G. Wu, "A thick-restarted block arnoldi algorithm with modified ritz vectors for large eigenproblems," *Computers & Mathematics with Applications*, vol. 60, no. 3, pp. 873–889, 2010.
23. A. H. Baker, J. M. Dennis, and E. R. Jessup, "On improving linear solver performance: A block variant of gmres," *SIAM Journal on Scientific Computing*, vol. 27, no. 5, pp. 1608–1626, 2006.
24. Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the solution of algebraic eigenvalue problems: a practical guide*. Siam, 2000, vol. 11.
25. X. Pinel and M. Montagnac, "Block krylov methods to solve adjoint problems in aerodynamic design optimization," *AIAA journal*, vol. 51, no. 9, pp. 2183–2191, 2013.
26. N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.
27. R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley, "Principles of runtime support for parallel processors," in *Proceedings of the 2nd International Conference on Supercomputing*, 1988, pp. 140–152.
28. L. Rauchwerger and D. Padua, "The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, ser. PLDI '95, 1995.
29. M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems," in *Proceedings of SC'12*, November 2012.
30. A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2006.
31. J. Saltz, C. Chang, G. Edjlali, Y.-S. Hwang, B. Moon, R. Ponnusamy, S. Sharma, A. Sussman, M. Uysal, G. Agrawal, R. Das, and P. Havlak, "Programming irregular applications: Runtime support, compilation and tools," *Advances in Computers*, vol. 45, pp. 105–153, 1997.

32. C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, May 1999, pp. 229–241.
33. N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999, pp. 192–202.
34. J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings," *International Journal of Parallel Programming*, vol. 29, no. 3, pp. 217–247, 2001.
35. H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 606–618, 2006.
36. B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '13, 2013.
37. A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14, 2014.
38. R. Kaleem, A. Venkat, S. Pai, M. Hall, and K. Pingali, "Synchronization trade-offs in gpu implementations of graph algorithms," in *30th IEEE International Parallel & Distributed Processing Symposium*, 2016.
39. A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 2009, pp. 233–244.
40. A. Bik and H. A. Wijshoff, "Advanced compiler optimizations for sparse computations," in *Supercomputing '93 Proceedings*, Nov 1993, pp. 430–439.
41. W. Pugh and T. Shpeisman, "Sipr: A new framework for generating efficient code for sparse matrix computations," in *Proceedings of the Eleventh International Workshop on Languages and Compilers for Parallel Computing*, August 1998.
42. N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar, "Next-generation generic programming and its application to sparse matrix computations," in *Proceedings of the 14th International Conference on Supercomputing*, Santa Fe, New Mexico, USA, May 2000, pp. 88–99.
43. H. van der Spek and H. Wijshoff, "Sublimation: Expanding data structures to enable data instance specific optimizations," in *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 106–120.
44. A. Bik and H. Wijshoff, "On automatic data structure selection and code generation for sparse computations," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Springer Berlin Heidelberg, 1994, vol. 768, pp. 57–75.
45. A. J. C. Bik and H. A. G. Wijshoff, "Automatic data structure selection and transformation for sparse matrix computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 2, pp. 109–126, 1996.
46. Y. Tao, Y. Deng, S. Mu, Z. Zhang, M. Zhu, L. Xiao, and L. Ruan, "Gpu accelerated sparse matrix-vector multiplication and sparse matrix-transpose vector multiplication," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 14, pp. 3771–3789, 2015.