Vectorized Sparse Matrix Multiply for Compressed Row Storage Format*

Eduardo F. D'Azevedo[1], Mark R. Fahey[2], and Richard T. Mills[2]

[1] Computer Science and Mathematics Division
[2] Center for Computational Sciences,
Oak Ridge National Laboratory,
Oak Ridge, TN 37831, USA
{dazevedoef, faheymr, rmills}@ornl.gov

Abstract. The innovation of this work is a simple vectorizable algorithm for performing sparse matrix vector multiply in compressed sparse row (CSR) storage format. Unlike the vectorizable jagged diagonal format (JAD), this algorithm requires no data rearrangement and can be easily adapted to a sophisticated library framework such as PETSc. Numerical experiments on the Cray X1 show an order of magnitude improvement over the non-vectorized algorithm.

1 Introduction

There is a revival of vector architecture in high end computing systems. The Earth Simulator[1] consists of 640 NEC SX-6 vector processors and is capable of sustaining over 35 Tflops/s on LINPACK benchmark. It was the fastest machine in the TOP500[2] list in 2002 and 2003. The Cray X1[3] series of vector processors are also serious contenders for a 100 Tflops/s machine in the National Leadership Computing Facility (NLCF) to be built at the Oak Ridge National Laboratory (ORNL). Vector machines have the characteristic that long regular vector operations are required to achieve high performance. The performance gap between vectorized and non-vectorized scalar code may be an order of magnitude or more.

The solution of sparse linear systems using a preconditioned iterative method forms the core computational kernel for many applications. This work of developing a vectorized matrix-vector multiply algorithm was motivated by issues arising

* This Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL). This research used resources of the Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

[1] See http://www.es.jamstec.go.jp/ for details.
[2] See http://www.top500.org for details.
[3] See http://www.cray.com/products/x1/ for details.

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3514, pp. 99–106, 2005.
© Springer-Verlag Berlin Heidelberg 2005

from porting finite element codes that make extensive use of the PETSc [1] library framework for solving linear systems on the Cray X1 vector supercomputer.

Matrix-vector multiply, triangular solves and incomplete LU factorization are common computational kernels of the linear solver. The compressed sparse row storage (CSR) format is used in PETSc and achieves good performance on scalar architectures such as the IBM Power 4. However, it is difficult to achieve high performance on vector architectures using the straight-forward implementation of matrix-vector multiply with CSR. A vectorized iterative linear solver was developed for the Earth Simulator for solving sparse linear equations from finite elements modeling [2]. The algorithm used jagged diagonal storage (JAD) format with multi-coloring of nodes (or rows) to expose independent operations and parallelism. Another approach for sparse matrix multiply on the Cray C90 vector machines was based on fast prefix sum and segment scan [3]. The algorithm was fairly complicated and may require coding in assembly language for best efficiency. Extra storage was also needed to hold the partial prefix sums. The SIAM books by Dongarra [4, 5] are useful references on solving linear equations on vector processors.

The main contribution of this work is the development of a simple vectorized algorithm to perform sparse matrix vector multiply in CSR format *without* data rearrangement. This makes it attractive to implement such a vectorized algorithm in a sophisticated library framework such as PETSc. Numerical experiments on the Cray X1 show an order of magnitude improvement in sparse matrix multiply over the non-vectorized scalar implementation.

The background of vectorizing sparse matrix multiply is contained in Section 2. The new algorithm, compressed sparse row storage with permutation (CSRP), is described in Section 3. Results of numerical experiments on the Cray X1 are described in Section 4.

2 Background

There are many ways to store a general sparse matrix [6, 7]. The commonly used sparse matrix storage format for general nonsymmetric sparse matrices include the compressed row storage (CSR), ELLPACK-ITPACK [8] (ELL) and jagged diagonal (JAD) format.

In CSR, the matrix multiply operation, $y = A * x$, is described in Fig. 1. Here `JA` contains the column indices, `A` contains the nonzero entries, and `IA` points to the beginning of each row. The algorithm is efficient on scalar processors since it has unit stride access for `A` and `JA`. Moreover, the variable `YI` can be held in fast registers. Each visit through the inner loop performs one addition and one multiplication but requires memory fetches for `A(J)`, `JA(J)`, and `X(JA(J))`. The computational efficiency is usually limited by memory bandwidth and the actual attainable performance is only a small fraction of peak performance for the processor. On a vector machine, the vectorization across the row index `J` is limited by the number of nonzeros `(IEND-ISTART+1)` per row, which may be only 7 for a regular finite difference stencil in three dimensions.

```
1   DO I=1,N
2     ISTART = IA(I);IEND = IA(I+1)-1
3     YI = 0.0
4     DO J=ISTART,IEND
5        YI = YI + A(J) * X( JA(J) )
6     ENDDO
7     Y(I) = YI
8   ENDDO
```

Fig. 1. Matrix multiply in CSR format

```
1   Y(1:N) = 0.0
2   DO J=1,NZ
3     Y(1:N) = Y(1:N) +  A(1:N,J)*X( JA(1:N,J) )
4   ENDDO
```

Fig. 2. Matrix multiply in ELLPACK format

If every row of the matrix has approximately equal number of nonzeros, then the ELL format is more efficient. The nonzero entries and column indices are stored in rectangular $\texttt{N} \times \texttt{NZ}$ arrays, where `N` is the number of rows and `NZ` is the maximum number of nonzeros per row. The computation has more regularity and is easily optimized by vectorizing compilers. One algorithm (see Fig. 2) vectorizes along the rows to give long vectors. However, it will incur more traffic to memory since the vector `Y` will be repeatedly read in and written out again. Another variant (see Fig. 3) uses a "strip-mining" approach to hold a short array `YP` in vector registers. This will avoid repeated reading and writing of the `Y` vector. However, the ELL format is not suitable for matrices with widely different number of nonzeros per row. This would lead to wasted storage and unnecessary computations.

The jagged diagonal format may be consider a more flexible version of ELL (see Fig. 4). The rows are permuted or sorted in increasing number of nonzeros (see Fig. 5) and the data rearranged to form long vectors. Conceptually the vectorization is down along the rows but a column-oriented variant might also be efficient. If a matrix is already available in CSR format, extra storage is required to copy and convert the matrix into JAD format. This may be a significant drawback if the application is already trying to solve the largest problem possible.

3 CSR with Permutation (CSRP)

The vectorizable algorithm proposed here performs the matrix vector multiply operation using the CSR format but with a permutation vector such that rows with the same number of nonzeros are grouped together. Conceptually, this may be considered a variant of the JAD format. The algorithm is also similar to the ELL algorithm where "strip-mining" is employed to reuse vector registers. The

```
1   DO I=1,N,NB
2      IEND = MIN(N,I+NB-1)
3      M = IEND-I+1
4      YP(1:M) = 0.0
5   !  ---------------------------------------------
6   !  Consider YP(1:M) as vector registers
7   !  NB is multiple of the size of vector registers
8   !  ---------------------------------------------
9      DO J=1,NZ
10        YP(1:M) = YP(1:M) + A(I:IEND,J) * X( JA(I:IEND,J) )
11     ENDDO
12     Y(I:IEND) = YP(1:M)
13  ENDDO
```

Fig. 3. Variant of matrix multiply in ELLPACK format

```
1   Y(1:N) = 0.0
2   IP = 1
3   DO J=1,NZ
4       I = ISTART(J)
5       M = N - I + 1
6       Y(I:N) = Y(I:N) + A(IP:(IP+M-1)) * X( JA(IP:(IP+M-1)) )
7       IP = IP + M
8   ENDDO
```

Fig. 4. Matrix multiply in JAD format

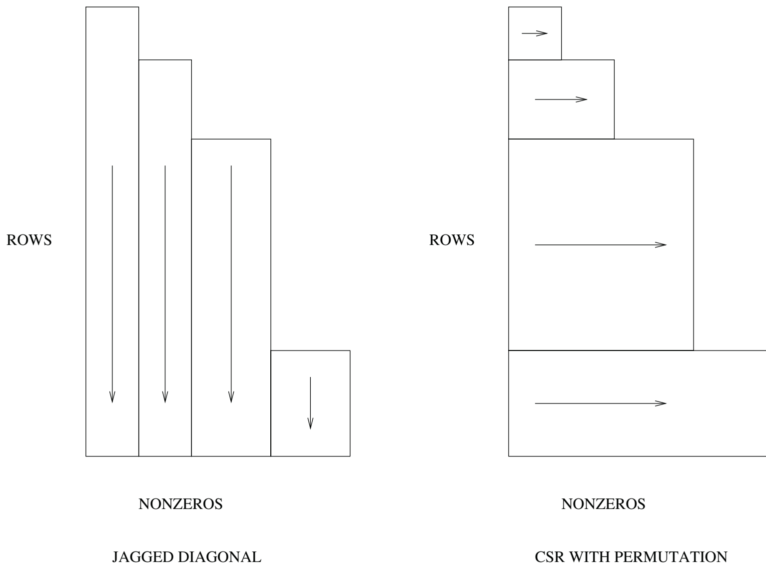
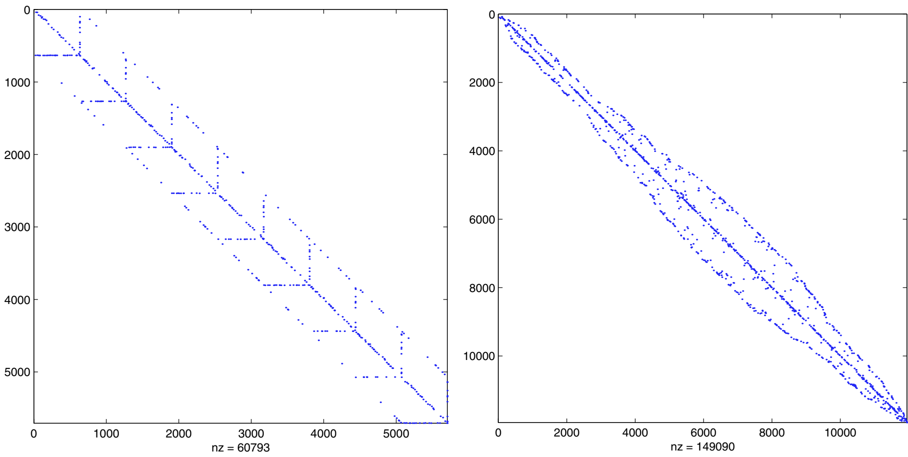


Fig. 5. Matrix profile after rows are permuted in increasing number of nonzeros

key difference is the data remain in place and are accessed indirectly using the permutation vector. The major drawback for leaving the data in place is the irregular access to arrays `A` and `JA`. The algorithm is described in Fig. 6. Here

```
 1   DO IGROUP=1,NGROUP
 2     JSTART = XGROUP(IGROUP)
 3     JEND = XGROUP(IGROUP+1)-1
 4     NZ = NZGROUP(IGROUP)
 5   ! ------------------------------------------------------------
 6   ! Rows( IPERM(JSTART:JEND) ) all have same NZ nonzeros per row
 7   ! ------------------------------------------------------------
 8     DO I=JSTART,JEND,NB
 9       IEND = MIN(JEND,I+NB-1)
10       M = IEND - I + 1
11       IP(1:M) = IA( IPERM(I:IEND) )
12       YP(1:M) = 0.0
13   !   ----------------------------------------
14   !   Consider YP(:), IP(:) as vector registers
15   !   ----------------------------------------
16       DO J=1,NZ
17          YP(1:M) = YP(1:M) + A( IP(1:M) ) * X( JA(IP(1:M)) )
18          IP(1:M) = IP(1:M) + 1
19       ENDDO
20     ENDDO
21     Y( IPERM(I:IEND) ) = YP(1:M)
22   ENDDO
```

Fig. 6. Matrix multiply for CSR with permutation



(a) astro (b) bcsstk18

Fig. 7. Sparsity patterns for test matrices

IPERM is the permutation vector, XGROUP points to beginning indices of groups in IPERM. If extra storage is available, it is feasible to copy and convert each group of rows into ELLPACK format (CSRPELL) to get better performance since access to A and JA would be unit-stride. This would be equivalent to the ITPACK permuted blocks (ITPER) format investigated by Peters [9] for the IBM 3090VF vector processor. For CSRP to be effective, there is an implicit assumption that there are many rows with the same number of nonzeros. This is often the case for sparse matrices arising from mesh based finite element or finite difference codes.

The permutation vector can be easily constructed using bucket sort and two passes over the IA vector in $O(\texttt{N})$ work (N is the number of rows). The algorithm only requires knowing the number of nonzeros per row and does not need to examine the larger JA array.

4 Numerical Experiments on the Cray X1

The CSRP algorithm for sparse matrix multiply has been implemented in the Mat_SeqAIJ "class"[4] in version 2.2.1 of PETSc. The parallel distributed memory sparse matrix class locally uses Mat_SeqAIJ on each processor. Therefore testing with just the sequential matrix class yields important insights in the effectiveness of vectorization with CSRP. The permutation vector is generated in MatAssemblyEnd_SeqAIJ and destroyed in MatDestroy_SeqAIJ. The procedure MatMult_SeqAIJ is modified to use the CSRP and CSRPELL algorithms. The implementation uses C (with Fortran kernels) to minimize the changes to PETSc in only the files aij.h and aij.c.

This implementation has been tested in sequential mode on the Cray X1 at the Center for Computational Sciences at the Oak Ridge National Laboratory. The processing units on the X1 consist of Multi-Streaming Processors (MSPs). Each MSP consists of 4 Single-Streaming Processors (SSPs). Each SSP has 32 vector registers, and each register can hold 64 elements of 64-bit data. The vector clock on an SSP runs at 800MHz and each SSP can perform 4 floating point operations per cycle to yield a theoretical peak performance of 3.2 Gflops/s, or 12.8 Gflops/s per MSP. The 4 SSPs in an MSP share a common 2 MBytes write-back L2 cache. Memory bandwidth is 34.1 GBytes/s from memory to cache and 76.8 GBytes/s from cache to CPU. Job execution on the X1 can be configured for SSP or MSP mode. In SSP mode, each SSP is considered a separate MPI task; whereas in MSP mode, the compiler handles the automatic creation and synchronization of threads to use vector resources of all 4 coupled SSPs as a single MPI task.

The new algorithm has been tested on a number of sparse matrices with regular and irregular patterns (see Table 1). The "astro" matrix is related to nuclear modeling in an astrophysics application obtained from Professor Bradley Meyer at Clemson University. Note that there are many rows with several hundred nonzeros. The "bcsstk18" matrix is a stiffness matrix obtained from the

[4] PETSc is written in C in a disciplined object-oriented manner.

Table 1. Description of matrices

Name	N	Nonzeros	Description
astro	5706	60793	Nuclear Astrophysics problem from Bradley Meyer
bcsstk18	11948	149090	Stiffness matrix from Harwell Boeing Collection
7pt	110592	760320	7 point stencil in $48 \times 48 \times 48$ grid
7ptb	256000	7014400	4×4 blocks 7-pt stencil in $40 \times 40 \times 40$ grid

Table 2. Performace (in MFlops/s) of sparse matrix multiply using CSR, CSRP and CSRPELL in PETSc

	SSP			MSP		
Problem	CSR	CSRP	CSRPELL	CSR	CSRP	CSRPELL
astro	26	163	311	14	214	655
bcsstk18	28	315	340	15	535	785
7pt	12	259	295	8	528	800
7ptb	66	331	345	63	918	1085

Harwell-Boeing collection of matrices[5]. The "7pt" matrix is constructed from a 7 point stencil on a $48 \times 48 \times 48$ rectangular mesh. The "7ptb" matrix is similarly constructed using 4×4 blocks from a 7 point stencil on a $40 \times 40 \times 40$ grid. Table 2 shows the performance (in Mflops/s) for the original CSR algorithm versus the new vectorizable CSRP and CSRPELL algorithms in SSP and MSP modes. The Megaflop rate is computed by timing over 100 calls to PETSc `MatMult`. The data suggest the CSRP algorithm is an order of magnitude faster than the original non-vectorizable algorithm. The CSRP with ELLPACK format (CSRPELL) algorithm achieves even better performance with unit stride access to data, but requires rearranging the data into another copy of the matrix. However, even with this rearrangement, the CSRPELL algorithm achieves less than 11% of theoretical peak performance of the Cray X1 in SSP and MSP modes.

5 Summary

We have presented a simple vectorizable algorithm for computing sparse matrix vector multiply in CSR storage format. Although the method uses non-unit stride access to the data, it is still an order of magnitude faster than the original scalar algorithm. The method requires no data rearrangement and can be easily incorporated into a sophisticated library framework such as PETSc. Further work is still required in vectorizing the generation of sparse incomplete LU factorization and the forward and backward triangular solves.

[5] Available at `http://math.nist.gov/MatrixMarket/data/`.

References

1. Balay, S., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc users manual. Technical Report ANL-95/11 - Revision 2.2.0, Argonne National Laboratory (2004) See also http://www.mcs.anl.gov/petsc.
2. Nakajima, K., Okuda, H.: Parallel iterative solvers for finite-element methods using a hybrid programming model on SMP cluster architectures. Technical Report GeoFEM 2003-003, RIST, Tokyo (2003) See also http://geofem.tokyo.rist.or.jp/members/nakajima.
3. Blelloch, G.E., Heroux, M.A., Zagha, M.: Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, Department of Computer Science, Carnegie Mellon University (1993) See also http://www-2.cs.cmu.edu/∼guyb/publications.html.
4. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Solving Linear Systems on Vector and Shared Memory Computers. SIAM, Philadelphia, PA (1991)
5. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.: Numerical Linear Algebra for High-Performance Computers. SIAM, Philadelphia, PA (1998)
6. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Clarendon Press, Oxford (1986)
7. Saad, Y.: Iterative Methods for Sparse Linear Systems. Second edn. SIAM, Philadelphia, PA (2003) See also http://www-users.cs.umn.edu/∼saad/books.html.
8. Kincaid, D.R., Young, D.M.: The ITPACK project: Past, present and future. In Birkhoff, G., Schoernstadt, A., eds.: Elliptic Problem Solvers II Proc. (1983) 53–64
9. Peters, A.: Sparse matrix vector multiplication techniques on the IBM 3090 VF. Parallel Computing **17** (1991) 1409–1424