Multicore Acceleration of CG Algorithms Using Blocked-Pipeline-Matching Techniques

David M. Fernández, Dennis Giannacopoulos, and Warren J. Gross

Department of Electrical and Computer Engineering, McGill University, Montreal, QC H3A 2A7, Canada

To realize the acceleration potential of multicore computing environments computational electromagnetics researchers must address parallel programming paradigms early in application development. We present a new blocked-pipeline-matched sparse representation and show speedup results for the conjugate gradient method by parallelizing the sparse matrix-vector multiplication kernel on multicore systems for a set of finite element matrices to demonstrate the potential of this approach. Performance of up to 8.2 GFLOPS was obtained for the proposed vectorized format using four Intel-cores, 17× more than the nonvectorized version.

Index Terms—**Acceleration, blocked formats, conjugate gradient, multicore, sparse matrices, vector processor.**

I. Introduction

MULTICORE processors represent one of the newest mainstream computing trends for enhancing the performance of scientific kernels, bringing about new opportunities and challenges to electromagnetic (EM) practitioners. Parallel programming challenges must now be confronted earlier in application development in order to exploit this new trend. Some EM computations are highly parallelizable, offering different opportunities for multicore acceleration (e.g., parallel mesh builders, parallel iterative solvers, etc.). Of particular interest to this work is the solution of the large-sparse linear systems that arise when solving EM problems.

This work focuses on accelerating the *conjugate gradient* (CG) solver by parallelizing its dominant computing kernel, the sparse matrix-vector multiplication (SMVM) where a sparse matrix A multiplies a dense vector x. A new block-partitioned sparse format is presented to accelerate the SMVM kernel using both high-level parallelism, e.g., scheduling tasks across cores in multicore or clustered processors, and low-level parallelism within processor cores, such as vectorization, loop transformations, and time skewing. First, we identify the key challenges confronted in parallelizing general sparse kernels and then describe the new sparse format and the algorithmic approach, showing performance results to validate it.

II. Previous Work

Parallelizing and optimizing dense linear algebra kernels is a well understood task that has given rise to a variety of Basic Linear Algebra Subprograms (BLAS) libraries (e.g., LAPACK, ScaLAPAC) [1], [2]. Sparse kernels, on the other hand, represent a greater challenge. They enable larger problems, using less memory resources and computing on nonzero matrix entries only; however, drawbacks such as increased instructions overhead and irregular and indirect access to data significantly limit the use of hardware optimization techniques and BLAS operations. Support for parallel sparse kernels on clustered systems is available in some newer packages (e.g., SPARSEKIT, NIST's Sparse BLAS, and PETSc) which operate on general sparse formats (e.g., compressed sparse row-CSR [3]), but are generally only optimized for dense kernels. Efficient multicore implementations and low-level parallelism on sparse kernels have yet to be thoroughly addressed.

Specialized formats have been used as means to regularize the computations and data layout of general sparse formats at the expense of processing some extra zeros, unveiling better opportunities for high- and low-level parallelism. The Block-CSR (BCSR), Ellpack-ITpack and Jagged-diagonals formats are the three classic formats used for this purpose, however, they each demand certain matrix properties to be efficient [3]. Recent work on specialized sparse matrix formats [4]–[6] has shown that further performance improvements are possible without such restrictions. In [6], we report up to 14 times speedup (SU) compared to a single core CSR SMVM implementation obtained by designing a specialized sparse format (called pipeline-matched sparse format-PMS).

III. New Sparse Format for Parallel Processing

Based on the experience gained in [6], we introduce a blocked adaptation of PMS, that exploits both locality and vector units in modern processors, as opposed to other blocked formats that only aim at exploiting locality. PMS offers a fast SMVM but it requires to integrate the multiplying vector into the format in a *vector-spreading* operation. Vector-spreading is embarrassingly parallel and can be overlapped with other instructions in general. However, in some iterative solvers (such as CG) this overlap cannot be efficiently implemented.

A. Blocked Pipeline-Matched Sparse (BPMS) Representation

The new format called *blocked pipeline-matched sparse* (BPMS) representation defines clear data boundaries for partitions as PMS [6], nonetheless it also offers better opportunities to exploit fine grained parallelism and it does not require the vector-spreading operation. In BPMS, the matrix is stored in small dense matrix-blocks, which are enforced to be a multiple of the vector-registers size (128 bit register that can store four single precision floating point values or two double precision in modern Intel Core2 CPUs) on the target architecture as in PMS; thus allowing to easily exploit vector or single-instruction multiple-data (SIMD) parallelism in multicore processors.

Manuscript received December 18, 2009; accepted February 13, 2010. Current version published July 21, 2010. Corresponding author: D. M. Fernández (e-mail: david.fernandezbecerra@mail.mcgill.ca).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TMAG.2010.2044023

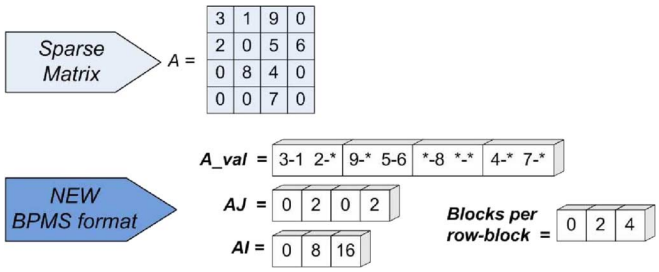

Fig. 1. Sparse matrix represented in BPMS format using 2-by-2 blocks.

Furthermore, when the size is a multiple greater than one, loop transformations can be implemented to enhance performance (not possible on PMS). BPMS stores the matrix data in four linear arrays, as shown in Fig. 1, in the following way: (i) A_VAL stores the nonzero elements of the matrix in dense square/rectangular blocks by rows (elements in blocks are stored row-wise), with zero padding to match the pipeline width of the target processor (as in PMS); (ii) AJ contains the column indices of the first element in each block (as in BCSR); (iii) AI has the index of the first matrix element that starts a new row-block (as in BCSR); and (iv) *BLOCKS per row-block* the number of dense blocks preceding the block pointed to by each of the row indices. The row indices can be used to determine high-level boundaries for data-partitions to spread across processing cores. The newly introduced fourth vector aids in load balancing by providing information on the amount of data to compute on for each coarse data-partition defined.

The main advantage of BPMS over other blocked formats such as BCSR, is that it exploits the vector units in modern processors. The pipeline-matching in BPMS blocks creates vector-data sets aligned to natural vector boundaries in memory, which ease implementing vector operations. Traditional blocked formats only exploit data-locality by creating small blocks that match the size of the register file or the cache lines (also achieved by BPMS), but do not align data to vector boundaries nor do they assure data sizes to fit within vector-registers. In addition to this, BPMS includes a fourth vector that provides important information to load balance the matrix data when partitioning it across processors and provides useful information for low level loop optimizations. To further enhance locality and efficiently support building dense blocks minimizing padded-zeros, we apply a reordering technique (reverse Cuthill-McKee [7]-RCMK) that reduces the matrix bandwidth before creating the BPMS representation. The insight provided by the reordering process can also be used to define coarse data-partition boundaries within the matrix (see Section IV-C).

IV. Conjugate Gradient (CG) Implementation, Vectorization and Matrix Partitioning

This section explains the Conjugate Gradient (CG) alg orithm implemented, the vectorization to accelerate the SMVM operations in CG and the matrix partitioning scheme used for parallel processing.

A. CG Implementation

A parallel nonprecondition CG algorithm was implemented adapted from the sequential version in [8], which is a well known and efficient versions of CG. The algorithm in [8] requires one matrix-vector multiplication per iteration since it approximates the residual using the Krylov subspace, as opposed to the traditional algorithm which requires a second SMVM. Four different CG versions where implemented for the Intel multicore processor, one for each sparse matrix format used (CSR, PMS, BCSR and BPMS). In each case, the SMVM operations were parallelized creating and synchronizing the desired number of POSIX Threads; this allowed to compute the SMVM operation on different number of CPU-cores, thus exploiting high-level parallelism. In the PMS-CG implementation both the SMVM and the *vector-spread* operation were parallelized.

B. SMVM Vectorization and Other Optimizations

Several *low-level* algorithmic optimizations were implemented in the SMVM kernels for the PMS, BPMS, and BCSR formats. The standard CSR format does not allow for any of the proposed optimizations thus none were done. The data structures for PMS, BCSR and BPMS were configured to generate similar zero padding for a fair comparison, and to reduce their memory footprint. The PMS format was configured to generate four-vectors of single precision floating point (SPFP) elements while the blocks in the BPMS and BCSR formats were configured to hold 2-by-2 SPFP to match the 4-SPFP vector pipeline in the Intel CPU targeted. The RCMK algorithm explained in Section III-A was used to further reduce the zero padding. Since RCMK aims at reducing the matrix bandwidth, it increases data locality, reducing cache misses and hiding better memory latency for all sparse formats including the CSR format which requires no zero padding.

In addition, the innermost *for-loop* in the SMVM kernel was unrolled by four, which is the number of SPFP elements used to configure the PMS, BPMS, and BCSR formats. This reduces the number of iterations and enables vectorization. The BPMS and PMS SMVM kernels were vectorized using SIMD intrinsics for the Intel CPU. A thread pool was used to further enhance the compute time for BPMS, the best performing format. In addition we implement a function to align matrix data within natural vector boundaries in memory as they are created. This simple memory layout reduces memory access time since it assures the minimum number of memory reads when data is accessed in vector-sets.

C. Matrix Partitioning and Load Balancing

Multicore (high-level) parallelism was implemented in this work as described in this section. Matrix data is statically partitioned assigning consecutive matrix rows or *row-blocks* to each of the processing cores. Each core then computes the SMVM kernel on their partition and results are synchronized/gathered. The number of rows grouped into such *row-block* sets may vary to statically balance the nonzeros assigned to each core. In particular, for BPMS the third vector (row-index vector) was used to point to the first nonzero element of a *row-block* partition, while the fourth vector (number of blocks per rows) was used to balance the load. The fourth vector was also useful in the inner loop of the SMVM unrolled kernel for BPMS, since it allowed to efficiently know the number of blocks to process for each row block in BPMS. Although partitioning and load balancing

TABLE I
VECTORIZED VERSUS NONVECTORIZED SMVM SPEEDUP

Matrix(#)	fidapm37(1)	bcsstk32(2)	s3dkt3m2(3)	s3dkq4m2(4)	sp10(5)
Non-zeros	765944	2014701	3753461	4820891	10M
added-zeros	(21.63%)	(30.36%)	(19.02%)	(11.32%)	(0.28%)
1C-BCSR	2.53	2.42	2.42	2.44	2.62
1C-PMS	3.33	2.71	2.68	2.70	2.78
1C-BPMS	5.49	4.42	4.35	4.45	4.80
4C-BCSR	3.28	2.11	1.88	1.97	1.81
4C-PMS	2.87	2.19	2.03	1.96	1.73
4C-BPMS	17.14	4.58	3.55	3.29	2.85

Note: 1C and 4C refers to the number of Intel-cores, and (M)illion.

BPMS matrix data is a fast operation of $O(n/block - rank)$ complexity (where n is the number of matrix rows and $block - rank$ is the number of rows in each small dense block defined in the matrix), an even cheaper approach to obtain a similar *row-block* partitioning is to use the *set-vector* generated during the RCMK reordering. This vector contains row indices which define bounds for the matrix data clustered along the main diagonal. Such bounds can be directly used as the partition indices. Nonetheless, the success of such an approach depends on the compression of the matrix bandwidth (BW). If the BW is not evenly compressed, the load will be unbalanced and a more sophisticated load balancing approach must be used. In this work, we only use the first approach explained.

V. RESULTS

This section describes the hardware used to test the algorithms developed, the compiler-based optimizations and finally the discussion of the results. Matrices (1) through (4) were taken from a set of FEM matrices found in [10] and additional matrices were generated to explore bigger problem sizes.

A. Testbed Description

The SMVM and CG kernels were implemented for all the sparse matrix formats in an Intel Core2 Quad 2.40 GHz CPU, with 4 CPU cores, 4 MB of L2 cache per core-pair, 4 GB of global DDR2 DRAM, and running a 64-bit Fedora Core 7 Linux operating system (OS). This Intel CPU contains a vector unit capable of processing four-SPFP values or two double precision values at the same time. For this work, the data structures where configured to exploit the four-SPFP pipeline configuration.

B. Testbed Compilation

Compilation for the Intel processor was done using GCC 4.1.2 with different optimization flags (e.g., -O2, -O3) reporting the best results only. Vectorization results were obtained for the Intel processor using the Intel Compiler Collection (ICC) 11.0, which gave us access to low level SIMD intrinsics also called SSE instructions.

C. Results Discussion

SU results for the SMVM kernel using the PMS, BCSR, and the new BPMS formats with respect to the CSR format are shown in Fig. 2 for increasing matrix sizes on a single Intel core. CSR provides the base computing time since it contains no zero padding, whereas the other formats have extra computational overhead. The SU curves increase as the matrices grow and stabilize around 2.5× for BCSR, 2.9× for PMS, and 4.4× for BPMS using the optimized kernels, as described in Section IV-B. These SU results clearly show that BPMS outperforms the other formats, a trend that stabilizes for the bigger matrices as the cache misses become regular. BPMS also demonstrates good scaling for increasing matrix sizes, requiring less padded-zeros (see Table I). This is true, in general, but zero padding may slightly increase for very irregularly structured matrices or regularly structured matrices with numerous cavities between its entries, e.g., the three matrices in the valley of Fig. 2.

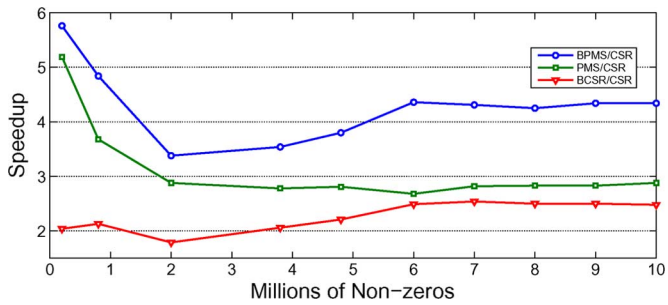

Fig. 2. SMVM SU of PMS, BPMS, and BCSR versus CSR for one-Intel-core.

A high-level parallelized version of the SMVM kernel was implemented across different numbers of cores using PThreads for all matrix formats with and without optimizations. Table I shows SU results obtained for the optimized/vectorized SMVM kernels (BCSR, PMS, and BPMS) with respect to the nonoptimized versions for one and four cores. These results show that the optimizations increase considerably the performance for all specialized formats, in particular, the BPMS format with up to 17.14× performance for matrix # (1). The high SU obtained for matrix # (1) is mainly due to the fact that it fits in the Intel-CPU cache. On the other hand, for the larger matrices that do not fit in the cache the performance increased with the optimizations and the number of cores but was limited by the achievable memory BW for each test matrix. This table also shows, that in general, as the nonzero entries of the matrix grow the added zeros due to padding drop, demonstrating the good scalability behavior of the new format. Table II shows performance results for the vectorized kernels in terms of GFLOPS for four Intel-cores. Overall, a sustained performance of up to 8.24 GFLOPS was observed for the Intel CPU with an average of 3.4 GFLOPS.

The nonvectorized parallel SMVM kernel was used in a parallel CG algorithm for the new BPMS format as *proof-of-con-*

TABLE II
VECTORIZED SMVM PERFORMANCE FOR FOUR-INTEL CORES

Matrix #	(1)	(2)	(3)	(4)	(5)
BCSR-GFLOPS	1.57	1.14	1.12	1.27	1.48
PMS-GFLOPS	1.53	1.20	1.26	1.30	1.38
BPMS-GFLOPS	8.24	2.47	2.13	2.11	2.21

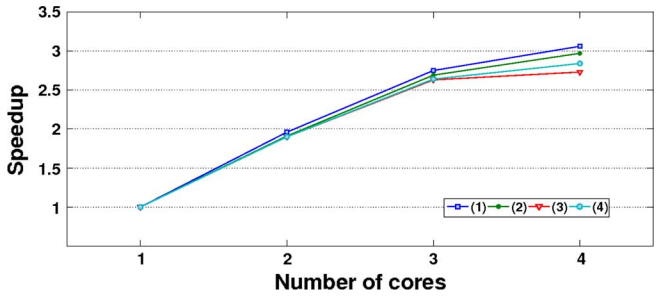

Fig. 3. SU scaling for CG-BPMS using one to four cores.

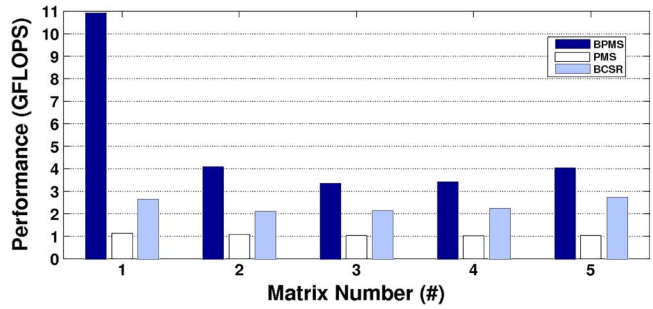

Fig. 4. CG performance results for four Intel-cores.

TABLE III
SPEEDUP OF THE BPMS-CG OVER THE PMS-CG AND BCSR-CG

Matrix (#)	(1)	(2)	(3)	(4)	(5)
1C-BPMS/BCSR	2.3	1.67	1.61	1.66	1.81
1C-BPMS/PMS	3.4	2.64	2.71	3.03	3.88
4C-BPMS/BCSR	4.1	1.90	1.57	1.51	1.47
4C-BPMS/PMS	8.1	3.02	2.89	3.14	3.92

cept. The scaling performance with respect to one-core BPMS-CG is shown in Fig. 3 for four of the biggest test matrices. Near three-times increase in performance (measured in GFLOPS) was obtained when running the nonvectorized BPMS-CG from one to four cores confirming good scaling performance expected from the BPMS accelerated CG. We then used the optimized parallel SMVM kernels to accelerate the CG solver for the three optimized formats. Fig. 4 shows performance results using four Intel-cores for the matrices described in Table I. The average GFLOPS performances are as follows: 2.4 for BCSR, 1.1 for PMS, and 3.7 for BPMS, verifying that BPMS has the best performance. As mentioned in Section I, PMS is not the best suited format for CG given that there are insufficient instructions to hide the *vector-spreading* computation time. The CG SU of the BPMS format versus the BCSR and PMS kernels are presented in Table III for one and four Intel-cores. BPMS achieves 2.1× and 4.2× average SU over BCSR and PMS formats respectively using four cores, and the vectorized kernels. Overall, the low-level optimizations for the BPMS-CG kernel had an average performance SU of up to 4.3×.

VI. CONCLUSION

This work presents a new sparse matrix format called BPMS and its performance advantages with respect to three special and standard sparse formats in modern multicore computing platforms. It demonstrates that by using special sparse formats one can regularize these kernels and significantly increase their performance even when considerable zeros are padded. In addition, we clearly define the key optimizations required to realize the speedups presented and efficiently exploit high/low parallelism in these new multicore systems. As an added benefit, the regularization process provides insight into the coarse partition boundaries that can be used for distributing data among processing cores. The vector results show an important speedup in performance of up to 4.3× over the non-vectorized BPMS kernel which exhibited the best performance results for both the SMVM and CG operations. Using BMPS the CG results show a maximum SU of 12.5× over CSR, 8.1× over PMS and 2× over BCSR, showing a clear advantage of the new BPMS over the other formats. BPMS is better suited for use in iterative solvers than our previous PMS format, since it does not require to copy the vector elements within the format. Nevertheless, the PMS format can be used in other applications where sufficient instructions exist to hide the cost of the *vector-spreading* operations. This work represents an important step to exploit the new trend in computer architecture represented by modern multicore processors. We demonstrate how an important computing kernel in electromagnetics computations (CG algorithm) can be accelerated by exploiting both high-level parallelism (using many cores) and low-level parallelism (using vectorization which offered the best performance improvement), efficiently exploiting the available computing resources.

ACKNOWLEDGMENT

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] C. L. Lawson *et al.*, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, 1979.
[2] J. J. Dongarra *et al.*, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.
[3] R. Barrett *et al., Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed. Philadelphia, PA: SIAM, 1994.
[4] Y. El-Kurdi *et al.*, "Hardware acceleration for finite element electromagnetics: Efficient sparse matrix floating-point computations with FPGAs," *IEEE Trans. Magn.*, vol. 43, no. 4, pp. 1525–1528, Apr. 2007.
[5] P. T. Stathis, "Sparse matrix vector processing formats," Ph.D. dissertation, Delft Univ. Technol., Delft, The Netherlands, Nov. 2004.
[6] D. M. Fernandez, D. Giannacopoulos, and W. J. Gross, "Efficient multicore sparse matrix-vector multiplication for FE electromagnetics," *IEEE Trans. Magn.*, vol. 45, no. 3, pp. 1392–1395, Mar. 2009.
[7] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proc. 1969 24th Nat. Conf. ACM*, 1969, pp. 157–172.
[8] S. J. Richard, "An introduction to the conjugate gradient method without the agonizing pain," School of Computer Science, Carnegie Mellon Univ. Pittsburgh, PA, Aug. 1994, p. 58, Edition 1 1/4.
[9] *"Cell Broadband Engine Programming Handbook,"* ver. 1.1, IBM, New York, Apr. 2007, p. 877.
[10] Matrix Market Nov. 2009. [Online]. Available: http://math.nist.gov/MatrixMarket/