An Implementation of Block Conjugate Gradient Algorithm on CPU-GPU Processors

Hao Ji
Department of Computer Science
Old Dominion University
Norfolk, VA 23529
hji@cs.odu.edu

Masha Sosonkina
Department of Modeling, Simulation, and Visualization Engineering
Old Dominion University
Norfolk, VA 23529
msosonki@odu.edu

Yaohang Li
Department of Computer Science
Old Dominion University
Norfolk, VA 23529
yaohang@cs.odu.edu

Abstract *— In this paper, we investigate the implementation of the Block Conjugate Gradient (BCG) algorithm on CPU-GPU processors. By analyzing the performance of various matrix operations in BCG, we identify the main performance bottleneck in constructing new search direction matrices. Replacing the QR decomposition by eigendecomposition of a small matrix remedies the problem by reducing the computational cost of generating orthogonal search directions. Moreover, a hybrid (offload) computing scheme is designed to enables the BCG implementation to handle linear systems with large, sparse coefficient matrices that cannot fit in the GPU memory. The hybrid scheme offloads matrix operations to GPU processors while helps hide the CPU-GPU memory transaction overhead. We compare the performance of our BCG implementation with the one on CPU with Intel Xeon Phi coprocessors using the automatic offload mode. With sufficient number of right hand sides, the CPU-GPU implementation of BCG can reach speedup of 2.61 over the CPU-only implementation, which is significantly higher than that of the CPU-Intel Xeon Phi implementation.*

Keywords—Block Conjugate Gradient; Multi-core CPU; Graphics Processing Unit; Intel Xeon Phi; Performance Evaluation.

I. INTRODUCTION

Modern many-core devices, such as Graphics Processing Units (GPU) and Intel Xeon Phi processors, are capable of delivering higher computing power than multi-core CPUs. This has led to increasing interest of using GPU or Intel Xeon Phi as coprocessors (accelerators) to enable additional accelerations to scientific computations carried out on host system. For instance, once a many-core device is attached to the host system, expensive computational operations can be offloaded to the many-core hardware during execution, which is referred to as the "offload mode" [1-3].

The Conjugate Gradient (CG) method is widely used in a variety of scientific computing applications for solving large, sparse, symmetric positive-definite (SPD) systems of linear equations [4, 9]. As a generalized form of CG with multiple right hand sides, the Block Conjugate Gradient (BCG) method [5] is attractive due to potentially faster convergence, reduced number of visits of the coefficient matrix, and being more suitable for parallel computing architectures [6].

The earlier study of the BCG performance on distributed systems can be found in [7]. Murli [22] later proposed a multi-grained distributed implementation of the parallel BCG over distributed heterogeneous architectures. Freiberger et al. [21] performed block conjugate gradient methods with multigrid preconditioners for high-speed diffuse optical tomography on GPUs.

In this paper, we investigate the performance of BCG implementation when GPU processors are employed as accelerators. First of all, we analyze the performance of various matrix operations in a GPU-only implementation to identify the main performance bottleneck. Then, to handle large linear systems whose coefficient matrices cannot fit in the GPU memory, a hybrid (offload) computing scheme is presented to offload matrix operations to GPU processors and to hide the CPU-GPU memory transaction overhead. Finally, we compare the performance of our BCG implementation on CPU-GPU processors with the one on CPU with Intel Xeon Phi as coprocessor using the automatic offload mode.

The computational experiments described in this paper are carried out on the XSEDE TACC Stampede System, where the compute node has dual Intel Xeon E5-2680 CPUs sharing 32 GB memory, one Intel Xeon Phi SE10P Coprocessor with 8GB memory, and one NVIDIA K20 GPU with 5GB memory. The BCG program is compiled using the Intel icc compiler with "-O3" optimization flag on CPU and Intel Xeon Phi processors while using NVIDIA nvcc compiler with "-O3" flag on GPU.

The rest of the paper is organized as follows. Section II describes the BCG algorithm in general as well as the matrix operations in BCG. Then, in Section III, we analyze the performance of matrix operations in a GPU-only BCG implementation. The BCG implementation on hybrid CPU-GPU processors is presented in Section IV. Section V compares the performance between the CPU-GPU implementation of BCG and the implementation on CPU with an Intel Xeon Phi coprocessor using the automatic offload mode. Finally, Section VI summarizes the paper and proposes future research directions.

978-1-4799-7564-8/14 $31.00 © 2014 IEEE
DOI 10.1109/Co-HPC.2014.10

II. Block Conjugate Gradient Algorithm

A. Overview

The BCG algorithm [5, 6] is an effective iterative method for solving a linear system with multiple right hand sides

$$AX = B$$

where the coefficient matrix A is a large, sparse $n \times n$ SPD matrix, X is an $n \times s$ matrix of unknowns, s is the number of right hand sides, and B is a given $n \times s$ constant matrix. The fundamental idea of BCG is to recursively update the solution matrix X to minimize the underlying block quadratic function, in which its unique global minimizer is the solution of the given linear system. Since breakdown may occur in the original BCG algorithm proposed by O'Leary [5] due to potential rank deficiency during BCG iterations, in this paper, we adopt the Breakdown-Free version of the Block Conjugate Gradient algorithm (BFBCG) [6] instead. BFBCG is equivalent to the original BCG algorithm proposed by O'Leary [5] if no breakdown occurs during BCG iterations.

Algorithm 1: Breakdown-Free Block Conjugate Gradient Method

Input: matrix $A \in R^{n \times n}$, right hand side matrix $B \in R^{n \times s}$, initial guess $X_0 \in R^{n \times s}$, preconditioner $M \in R^{n \times n}$, tolerance $tol \in R$ and maximum number of iterations $\max it \in R$

Output: an approximate solution $X_{sol} \in R^{n \times s}$

$R_0 = B - AX_0$

$Z_0 = MR_0$

$P_0 = orth(Z_0)$

For $i = 0, \cdots, \max it$

$Q_i = AP_i$

$\alpha_i = \left(P_i^T Q_i\right)^{-1} \left(P_i^T R_i\right)$

$X_{i+1} = X_i + P_i \alpha_i$

$R_{i+1} = R_i - Q_i \alpha_i$

If converged, then stop.

$Z_{i+1} = MR_{i+1}$

$\beta_i = -\left(P_i^T Q_i\right)^{-1} \left(Q_i^T Z_{i+1}\right)$

$P_{i+1} = orth\left(Z_{i+1} + P_i \beta_i\right)$

End

$X_{sol} = X_{i+1}$

The BFBCG algorithm is illustrated in Algorithm 1. X_0 is the initial guess of the solution matrix. At the ith iteration, R_i is the $n \times s$ residual matrix while Z_i is the corresponding preconditioned residual matrix under an $n \times n$ sparse, SPD preconditioner M. P_i is an $n \times r$ search matrix providing an orthogonal basis of search space to find the next approximated solution X_{i+1}, where r is the dimension of search space. In addition, parameter matrices α_i and β_i are generated to ensure orthogonality of search directions and minimization properties of BCG method. Function $orth(\cdot)$ is employed for constructing new search direction matrix P_{i+1}, where QR decomposition with column pivoting as rank revealing algorithm is used.

B. Matrix Operations in BCG

Because of the iterative nature of the BCG algorithm, in general, more than $\lceil n/s \rceil$ iterations are required for finding the solutions. At each iteration, a series of matrix operations, such as matrix-matrix multiplications and additions, matrix decompositions, and factorizations, need to be performed. Therefore, the behavior of these matrix operations has significant impacts on the overall performance of BCG implementations.

1) Multiplications between sparse matrix and tall-and-skinny matrix

In BCG, typically, the number of right hand sides s is significantly less than the dimension n of coefficient matrix A. Consequently, the n x s matrices of X_i, P_i, R_i, Q_i, and Z_i are "tall-and-skinny" and dense. Generations of Q_i and Z_{i+1} at each iteration, as shown in Figure 1, requires multiplications between a sparse matrix and a tall-and-skinny matrix, which are implemented using the DCSRMM routines[8].

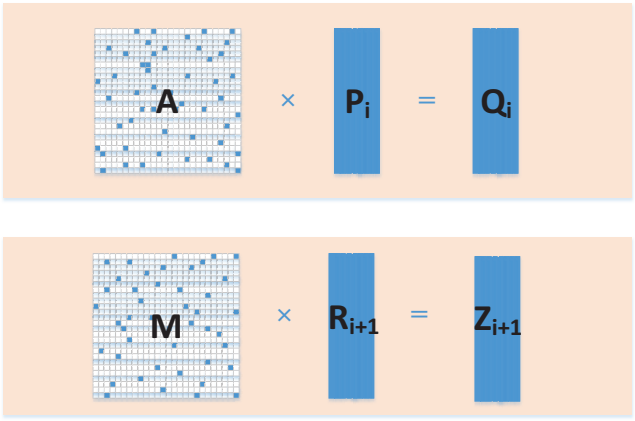

Figure 1: Sparse (compressed sparse row format) matrix-matrix multiplication for $Q_i = AP_i$ and $Z_{i+1} = MR_{i+1}$

2) Operations on tall-and-skinny matrices

An attractive property of BCG is that BCG involves a lot of operations related to dense, tall-and-skinny matrices, which is particularly suitable for processors such as GPU using SIMT (Single Instruction Multiple Threads) architectures. Figure 2 depicts the tall-and-skinny matrix operations in BCG to generate X_{i+1} and R_{i+1}. Routines of operations on dense rectangular matrices, such as DGEMM, are used.

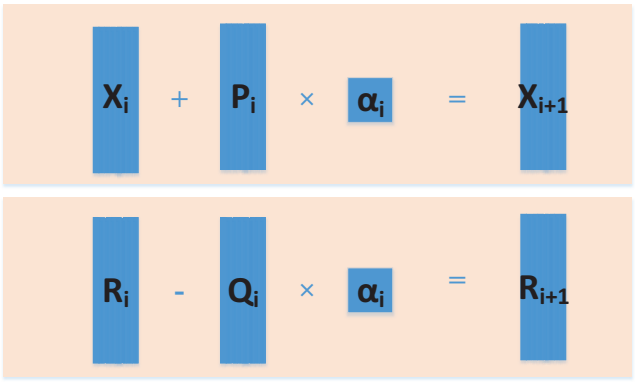

Figure 2: Tall-and-skinny matrix operations in BCG to generate X_{i+1} and R_{i+1}

3) Cholesky factorization

In addition to the rectangular matrices multiplications, generation of the parameter matrices α_i and β_i in BCG (Figure 3) requires calculation of the inverse of a small s x s matrix $P_i^T Q_i$, which is most appropriately done by Cholesky factorizations using linear algebra routines DPOTRF and DPOTRS.

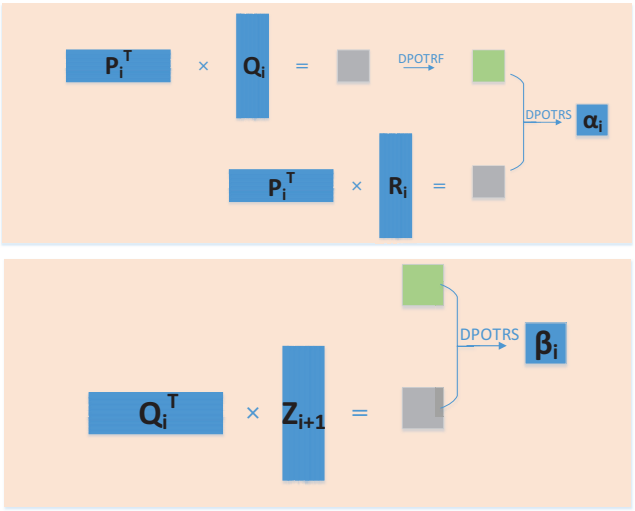

Figure 3: Cholesky decomposition to generate the parameter matrices α_i and β_i

4) QR decompostion

QR decompostion is used to construct new search direction matrix P_{i+1} that is orthogonal to the previous searching spaces, where QR decomposition with column pivoting as rank revealing algorithm is used. QR decomposition with column pivoting yields lower computational cost than Singular Value Decomposition (SVD). Routines DGEQP3 and DORGQR are used for QR decomposition, as shown in Figure 4.

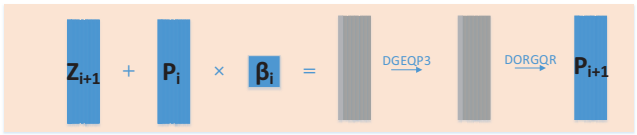

Figure 4: QR decomposition to generate new search direction matrix P_{i+1}

III. BCG ON GPU

In this section, we investigate the native implementation of BCG on GPU processors, where all numerical operations are carried out on GPU and the coefficient matrix also resides in the GPU memory. This implementation uses the matrix functions in the CUDA Basic Linear Algebra Subroutines (CUBLAS) library [11] for dense matrix operations, advanced matrix decompositions functions in the MAGMA library [12] for Cholesky factorizations, and sparse matrix routines in the CUSPARSE library [13] for sparse matrix operations.

Figure 5 compares the average elapsed computational time per iteration for different matrix operations in BCG on CPU and GPU processors. The coefficient matrix is "nd12k" from the UFL sparse matrix collection [14], which is a 36,000x36,000 sparse, SPD matrix with 14,220,946 nonzero entries. The number of right hand sides is set to 2,048. The elements in the right-hand side matrix are random numbers generated uniformly from interval [0, 1). The reported execution times are obtained from an average over 10 runs. The CPU implementation of BCG is built on the multithreaded Intel Math Kernel Library (MKL) [10], which consists of highly optimized linear algebra subroutines.

One can notice that the computational times of all matrix operations per iteration in BCG on GPU are less than those on CPU, where the improvements of tall-and-skinny matrix operations are of most significance. Nevertheless, the dominating operation in both CPU and GPU implementations is constructing the new search direction matrix P_{i+1}, i.e., $P_{i+1} = orth\left(Z_{i+1} + P_i \beta_i\right)$.

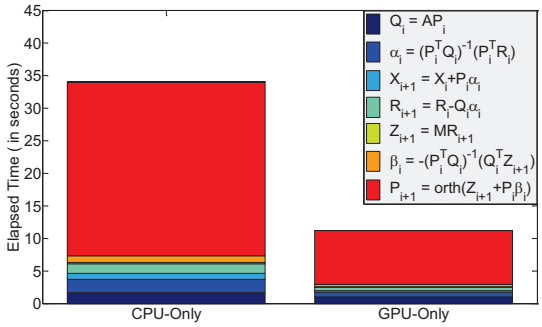

Figure 5: The average elapsed computational time for different steps in BCG on CPU and GPU processors

To reduce the computational cost in the constructing new search direction matrix P_{i+1}, we modify the BCG algorithm by using eigendecomposition on $Z^T Z$, where $Z = Z_{i+1} + P_i \beta_i$,

instead. In this case, Z^TZ is a small *sxs* symmetric matrix. Therefore, although calculation of Z^TZ leads to additional overhead of matrix-matrix multiplications, computing the eigendecomposition on Z^TZ is still significantly less costly than directly applying DGEQP3 to the *nxs* tall-and-skinny matrix Z for QR decomposition. As shown in Figure 6, the eigenvectors V of Z^TZ can be computed by using the DSYEVD routine. Once the eigenvectors V is available, the search matrix P_{i+1}, as an orthogonal basis of the space spanned by Z, can be very efficiently derived by normalizing each column of matrix product ZV using DNRM2 routine.

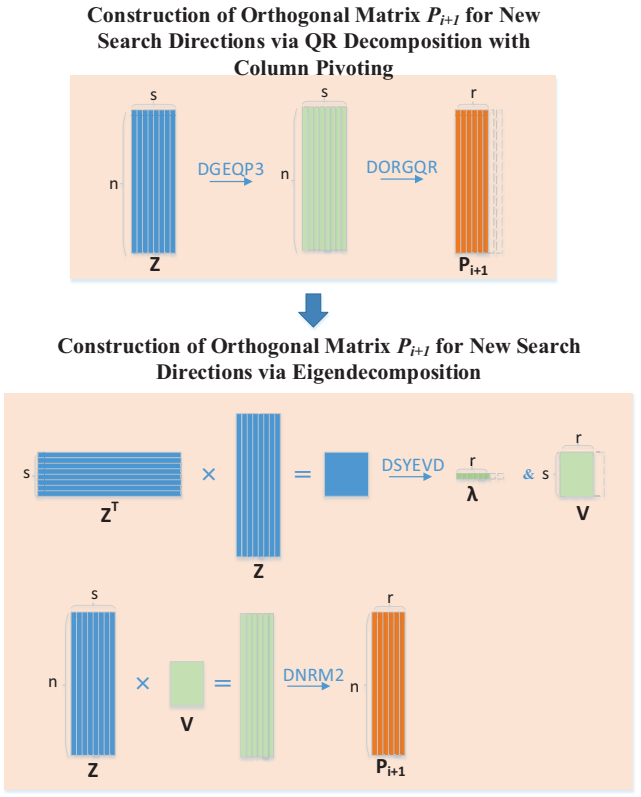

Figure 6: Eigendecomposition on Z^TZ to replace QR decomposition on Z to obtain orthogonal new search direction matrix P_{i+1}

Figure 7 shows the performance of the improved BCG implementation using eigendecomposition on Z^TZ to obtain new search directions. In comparison with Figure 5, one can find that the time spent on constructing new search direction matrix P_{i+1} is significantly reduced by 60.7% and 73.5% on CPU and GPU implementations, respectively. The overall speedup of the GPU-only implementation over the CPU implementation reaches 2.63.

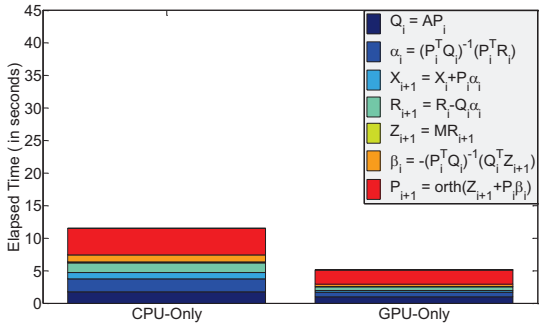

Figure 7: Comparison of the average elapsed computational time per iteration for different steps in BCG on CPU and GPU processors when eigendecomposition on Z^TZ is used to replace QR decomposition on Z to obtain orthogonal new search direction matrix P_{i+1}

IV. BCG ON HYBRID CPU-GPU PROCESSORS

In the case that the coefficient matrix is too big or the number of right hand sides is too many, consequently, the GPU memory is not big enough to fit all the matrices in BCG iterations. In this section, a BCG implementation on hybrid CPU-GPU processors is presented. In our implementation, CPU only coordinates data transfer and computation offload to GPU and does not directly participate in BCG computation. We use the routines in the CUBLAS-XT library [11] to support overlapping data transfers and execution for dense matrix operations. Page-locked memory is employed to increase the bandwidth between host memory and GPU memory.

Based on the sparse matrix routines in CUSPARSE, we implement the tiled multiplication between a sparse matrix and a tall-and-skinny matrix. Similar to the tiling strategy used in the CUBLAS-XT library, rows of sparse matrix is partitioned into tiles that can fit in the GPU memory while the tall-and-skinny matrix is split into tiles by columns. The tile size is selected so that the tiles can fit in the GPU memory. The procedure of tiling is illustrated in Figure 8.

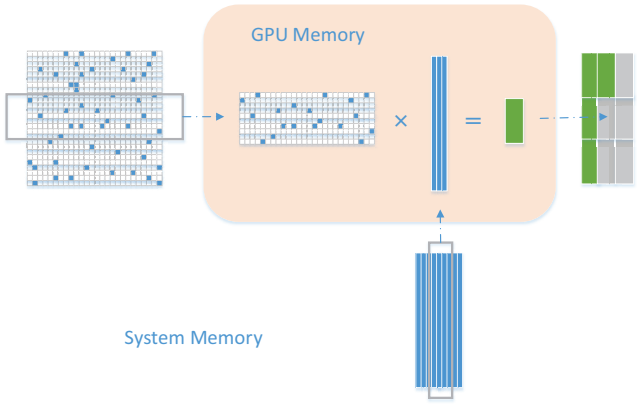

Figure 8: Tiled multiplication between a sparse matrix and a tall-and-skinny matrix

An important feature of the hybrid CPU-GPU BCG implementation is that data transfers and kernel computation for each tile can be performed concurrently so that the memory transaction time can be hidden. We assign each tile with a GPU stream, and asynchronous operations are placed into each stream. Figure 9 shows the timeline of sparse matrix multiplication and data transfer in an instance of calculating the product of the sparse coefficient matrix and the tall-and-skinny solution matrix. One can find that except for initialization, more than half of the data transfer operations occur concurrently with matrix multiplications, which can be hidden efficiently.

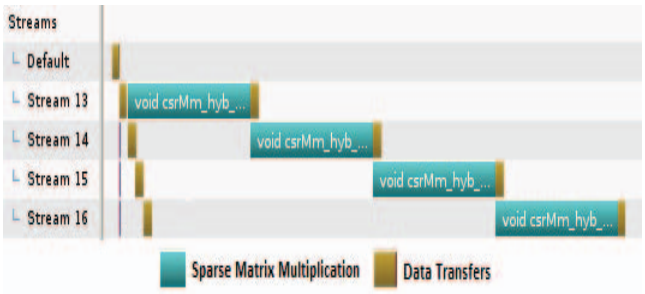

Figure 9: Data transfers and kernel computation for each tile are performed concurrently to hide the memory transaction time between CPU and GPU

Figure 10 shows the elapsed computational time per iteration in hybrid CPU-GPU BCG implementation in comparison with the GPU-only computational time and data transfer time without overlapping. In hybrid CPU-GPU scheme, 50.1% of the data transfer time is hidden due to concurrent execution with matrix operations.

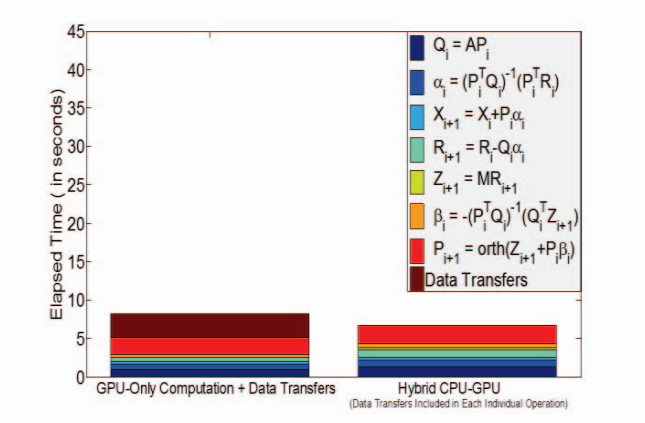

Figure 10: Comparison of the elapsed computational time per iteration in hybrid CPU-GPU BCG implementation with the GPU-only computational time and data transfer time. 50.1% of the data transfer time is hidden in the hybrid CPU-GPU scheme.

V. Comparison with Intel Xeon Phi Processor with Automatic Offload Mode

In this section, we compare the performance of the hybrid CPU-GPU implementation of BCG with the BCG implementation on CPU-Xeon Phi Processor using automatic offload mode against their theoretical performance peak, where MKL library provides the optimal computational work division for matrix operations of BCG over CPU- Xeon Phi Processor. The theoretical peak performance is widely used as upper bound in comparing computational power among parallel computing systems [15]. For a certain parallel computing system, the corresponding theoretical peak double precision performance P is calculated as

$$P = ncores \times clockspeed \times flops/cycle$$

where $ncores$ represents number of cores in a processor, $clockspeed$ is the corresponding clock rate, and $flops/cycle$ denotes the number of double-precision floating point operations per cycle [16-18].

Each Dual Xeon E5 processor has 8 cores clocked at 2.7GHz. Because the Dual Xeon E5 processor supports the Fused Multiply-Add (FMA) operations, in which one multiply and one add can be completed in a single cycle, each core of Dual Xeon E5 can perform up to 8 double-precision floating point operations per clock cycle. As a result, the theoretical peak double precision performance P_{cpu} of CPU can reach

$$P_{cpu} = (8 \times 2) \times 2.7 \times 8 = 345.6 GFLOPS .$$

The NVIDIA K20 GPU [19,20] has 13 Streaming Multiprocessors (SMs) clocked at 0.706GHz while 64 double-precision floating point units on each SM. The theoretical peak double precision performance P_{gpu} of GPU is calculated as

$$P_{gpu} = (64 \times 13) \times 0.706 \times 2 = 1{,}174.784 GFLOPS .$$

For the 61-core coprocessor Xeon Phi SE10P, each core clocked at 1.1GHz has 16 floating-point operations in double precision per clock cycle. As 60 cores are commonly used for computing, the theoretical peak performance P_{mic} of Xeon Phi coprocessor is

$$P_{mic} = 60 \times 1.1 \times 16 = 1{,}056 GFLOPS .$$

Ideally, if the linear algebra routines for those matrix operations in BCG can fully take advantage of the peak performance on hardware while the memory transaction overheads are hidden, executing BCG implementation directly on GPU or Intel Xeon Phi can roughly outperform CPU-only version by three times, according to the theoretical peak performance analysis on these hardware devices.

We use a large linear system with "thermomech_TC" from the UFL sparse matrix collection [14] as the coefficient matrix to test the CPU-GPU and CPU-Xeon Phi implementations of BCG. "Thermomech_TC" is a 102,158×102,158 sparse, SPD matrix with 711,558 nonzero entries. Figure 11 compares the overall speedup factors for the CPU-GPU implementation and

the CPU-MIC implementation of BCG algorithm over the CPU-Only version with different number of right hand sides *s*. The overall speedup of CPU-GPU can reach up to 2.61 when 4,096 right hand sides are used, which is significantly higher than that of CPU-Xeon Phi (1.61) in automatic offload mode.

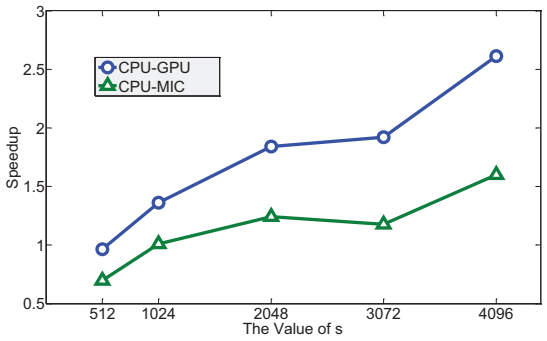

Figure 11: The Overall Speedup of CPU-GPU and CPU-Xeon Phi of BCG Implementations with Different Number of Right Hand Sides.

VI. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

In this paper, we take advantage of the GPU processors to improve the performance of BCG algorithm. Replacing the QR decomposition for orthogonalization by eigendecomposition of a small matrix significantly reduces the computational cost of generating new search direction matrices. A hybrid (offload) computing scheme is designed to offload matrix operations to GPU accelerators and hide the CPU-GPU memory transaction overhead, which enables our BCG implementation to handle linear systems with large, sparse coefficient matrices that cannot fit in the GPU memory. The hybrid CPU-GPU implementation outperforms the CPU-Xeon Phi implementation with automatic offload mode with significantly higher speedup compared to CPU-only implementation.

There is a lot of space to improve the hybrid CPU-GPU implementation. In our current implementation, CPU only coordinates data transfer and computation offload and does not directly participate in BCG computation. Our future implementation will allow GPU to contribute to BCG iterations. Moreover, further improvement of BCG algorithm can be achieved when multiple many-core hardware devices are used. Our future work will be extending our hybrid CPU-GPU implementation of BCG to take advantage of multiple many-core hardware devices.

ACKNOWLEDGMENT

Yaohang Li acknowledges this work is partially supported by NSF grant 1066471. The work of Masha Sosonkina was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476 and by the National Science Foundation grants NSF/OCI-0941434, 0904782, 1047772. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources. URL: http://www.tacc.utexas.edu. Hao Ji acknowledges support from ODU Modeling and Simulation Fellowship.

REFERENCES

[1] Hao Ji, Yaohang Li. GPU Accelerated Randomized Singular Value Decomposition and Its Application in Image Compression. Modeling, Simulation, and Visualization Student Capstone Conference, Suffolk, VA, 2014.

[2] Teodoro, George, et al. "Comparative Performance Analysis of Intel Xeon Phi, GPU, and CPU." arXiv preprint arXiv:1311.0378 (2013).

[3] Ashraf Yaseen, Hao Ji, Yaohang Li. A Load-Balancing Workload Distribution Scheme for Three-Body Interaction Computation on Graphics Processing Units (GPU). Journal of Parallel and Distributed Computing, 2 2014. (Submitted)

[4] J. R. Shewchuk, An Introduction to the Conjugate Gradient Method without the Agonizing Pain, Tech. report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1994

[5] D. P. O'Leary, The block conjugate gradient algorithm and related methods, Linear Algebra Appl., 29 (1980), pp. 293–322.

[6] Hao Ji, Yaohang Li. Breakdown-Free Block Conjugate Gradient Method. SIAM Journal on Numerical Analysis, 2014. (Submitted)

[7] D. P. O'Leary, Parallel implementation of the block conjugate gradient algorithm, Parallel Comput., 5 (1987), pp. 127–139.

[8] Dongarra, Jack J., et al. "A proposal for an extended set of Fortran basic linear algebra subprograms." ACM Signum Newsletter 20.1 (1985): 2-18.

[9] G. H. Golub and C. F. Van Loan, Matrix Computations, 4th ed., The Johns Hopkins University Press, Baltimore, 2012.

[10] Intel, M. K. L. "Intel Math Kernel Library." (2013).

[11] Nvidia, C. U. D. A. "Cublas library." NVIDIA Corporation, Santa Clara, California15 (2014).

[12] Tomov, S., et al. "MAGMA Library." Univ. of Tennessee and Univ. of California, Knoxville, TN, and Berkeley, CA (2014).

[13] NVIDIA, CUDA. "CUSPARSE library." NVIDIA Corporation, Santa Clara, California (2014).

[14] T. A. Davis, University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/.

[15] Hager, Georg, and Gerhard Wellein. Introduction to high performance computing for scientists and engineers. CRC Press, 2010.

[16] Konstantin S. Solnushkin (2013, Feb. 17). Memory Bandwidth for Intel Xeon Phi (And Friends). Retrieved from http://clusterdesign.org/2013/02/memory-bandwidth-for-intel-xeon-phi-and-friends/

[17] FLOPS. In Wikipedia. Retrieved from http://en.wikipedia.org/wiki/FLOPS.

[18] Masci, F. "Benchmarking the Intel Xeon Phi Coprocessor." (2014).

[19] NVIDIA "NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM K110", Retrieved from http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[20] NVIDIA. Tesla K20 GPU Active Accelerator. Retrieved from http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Active-BD-06499-001-v02.pdf.

[21] M. Freiberger, H. Egger, M. Liebmann, and H. Scharfetter, "Towards high speed diffuse optical tomography on graphics hardware," SFB Research Center, Tech. Rep. 2010–018, 2010:.

[22] Murli, Almerico, et al. "A multi‐grained distributed implementation of the parallel Block Conjugate Gradient algorithm." Concurrency and Computation: Practice and Experience 22. 15 (2010),pp. 2053-2072.