Advanced Materials Research Vol. 709 (2013) pp 555-562
Online available since 2013/Jun/27 at www.scientific.net

doi:10.4028/www.scientific.net/AMR.709.555

Accelerating PQMRCGSTAB Algorithm on Xeon Phi

Cheng Chen[1, a], Canqun Yang[1, b], Wenke Yao[1, c], Jin Qi[1, d], Qiang Wu[1, e]

[1]School of Computer Science
National University of Defense Technology
Changsha, Hunan 410073, China

[a]gaoye23@126.com, [b]canqun@nudt.edu.cn, [c]yaowenke2008@gmail.com,
[d]qijin2012@yeat.net, [e]qiangwu.cs.nudt@gmail.com

Keywords: MIC; Xeon Phi; SIMD; PQMRCGSTAB algorithm

Abstract. Utilizing iterative method to solve the large sparse linear systems is the key to many practical mathematical and physical problems. Recently, Intel released Xeon Phi, a many-core processor of Intel's Many Integrated Core (MIC) architecture, comprises 60 cores and supports 512-bit SIMD operation. In this work, we aim at accelerating an iterative algorithm for large spare linear system, named PQMRCGSTAB, by using both Xeon Phi's 8-way vector operation and dense threads. Then, we propose three optimizations to improve the performance: data prefetching to hide the data latency, vector register reusing, and SIMD-friendly reduction. Our experimental evaluation on Xeon Phi delivers a speedup of close to a factor 6 compared to the Intel Xeon E5-2670 octal-core CPU running the same problem.

Introduction

Heterogeneous computing with multiple levels of exposed parallelism is a leading aspect of consideration for the design of future HPC [1]. A coprocessor (such as GPU/FPGA) connecting to host (such as multi-core CPU) is the overall architecture of current heterogeneous. However, coprocessors (e.g. the GPUs) commonly require special programming constructs (e.g. NVIDIA's CUDA language) which brings about difficulties for programmers. To address this problem, Intel has announced its Many Integrated Core (MIC) Processor [2]. The MIC is based on Intel Architecture (IA), thus compatible with X86 cores written in C/C++ and Fortran [5]. An attractive feature of this architecture is its support for standard threading models like OpenMP [3] and MPI [4], which have been widely used in many scientific applications. MIC provides higher compute density than the current multi-core processors by packing up to 60 smaller cores that are equipped with hardware threads into a single MIC co-processor [2]. Moreover, MIC supports 512-bit SIMD operation, that is, 8-way double precision floating-point vector operation. In this paper, we study the programming and tuning of MIC by porting an iterative algorithm, i.e. the PQMRCGSTAB algorithm, on to Xeon Phi, a commercial release of Intel MIC.

Iterative algorithm is an effective method for solving large sparse linear equations. The Quasi-Minimal Residual Bi-Conjugate Gradient Stable (abbr. for QMRCGSTAB) algorithm [6] is a class of Krylov subspace methods. It is an improvement of BiConjugate Gradient STABle (Bi-CGSTAB) [7] algorithm. QMRCGSTAB takes the advantages of both the Quasi-Minimal Residual (QMR) method and the Bi-CGSTAB method [8], and has short iterative steps and stable convergence. Although many numerical computing researchers have made much effort to optimize iterative algorithms, there is no previous work focusing on accelerating QMRCGSTAB algorithm through SIMD extension. When porting PQMRCGSTAB algorithm to Xeon Phi, we should fully use its 8-way SIMD operation in one thread and spawn to multi threads at the same time.

In this paper, our goal is to accelerate PQMRCGSTAB algorithm on Xeon Phi. Our contributions are as follows:

1) Exploit data level parallelism of PQMRCGSTAB on each core of Xeon Phi via 8-way SIMD extension.

2) Employ multiple threads across to exploit threads level parallelism of PQMRCGSTAB algorithm on Xeon Phi.

3) Propose several optimizations: (1) data prefetching to hide the data latency; (2) vector register reusing, and (3) SIMD-friendly reduction.

Our experimental results demonstrate that our PQMRCGSTAB algorithm on one Xeon Phi achieves up to 6 speedups comparing to the non-accelerated implementation on one Intel Xeon E5 octal-core CPU.

The remainder of this paper is organized as follows. Section 2 introduces the MIC Architecture and PQMRCGSTAB algorithm. Section 3 reviews the related work. Section 4 presents our implementation of PQMRCGSTAB algorithm on Xeon Phi. Section 5 describes our optimization method. Section 6 gives the experimental results. Conclusion is drawn in Section 7.

Intel MIC. Intel MIC is a multi-processor computer architecture developed by Intel incorporating earlier work on the Larrabee many core architecture [9]. As a prototype product, Knights Ferry was announced and released in 2010 to developers. Xeon Phi, using Intel's Tri-gate technology, is a commercial release [10]. A Xeon Phi consists of 60 cores, running at up to 1.1 GHz. Each core has 4 hardware threads, 32KB of level one instruction and data caches, and 512KB of coherent level two cache [2]. A block diagram showing the placement of cores and the memory hierarchy is depicted in Fig. 1.

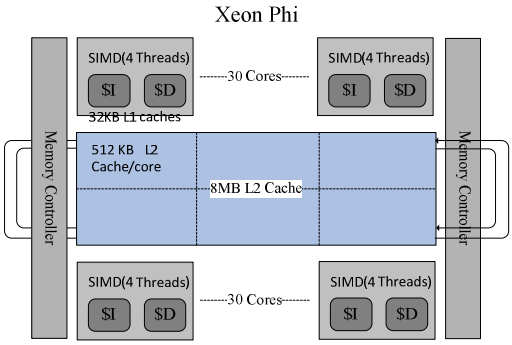

Figure 1. A block diagram of core placement and memory hierarchy on the Xeon Phi

In addition, each core is augmented with a Vector processor (VPU) and Vector register (Zmm register) with 8 64-bit (double precision floating-point) vector lanes. Accordingly, hardware instruction set supports an 8-way double precision floating-point SIMD ("Knights Corner Instructions [10]") vector operation. Developers can use libs and Intel intrinsic functions to implement vectorization.

MIC provides three Programming Models [2], namely, Offload Model, Co-processor-only Model and Symmetric Model, which we briefly introduce in the following.

Offload Model: MIC acts as a co-processor, Xeon hosted with MIC co-processing, where the processes run on the host Xeon CPUs, while the offload is directed towards the MIC devices.

Co-processor-only Model: In this mode (aka "MIC native"), the processes reside solely inside the co-processor. Libraries, the application, and other needed libraries are uploaded to the co-processors. Then an application can be launched from the host or the co-processor.

Symmetric Model: In this mode, the host Xeon CPUs and MIC devices are considered as peer nodes. Either the MIC devices or the host CPUs are considered as co-processors.

PQMRCGSTAB algorithm. The input to PQMRCGSTAB algorithm contains a coefficient matrix A and a vector b, the initial solution value x, the residual r and some other parameters. There are two main processes [11] in solve sparse linear equations: Jacobi pre-conditioner which chooses the reciprocal of main diagonal elements as the pre-conditioner and iterative solving liner equation. The iteration includes three technologic processes: twice of QMR iteration corrections and once of intermediate result computation. The iteration repeats until the residual r satisfies some prescribed controlled condition. Fig. 2 gives the pipeline of PQMRCGSTAB algorithm.

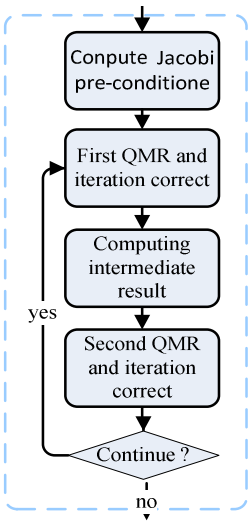

Figure 2. Serial iterative algorithm on CPU

Related work. A plenty of researchers have been working on accelerating sparse linear equations, leading to significant performance boosting on heterogeneous computing. [12] presents general approaches to accelerate iterative methods on GPUs with high computability and memory bandwidth. [13] uses several optimizations to improve the performance of the GPU-accelerated PQMRCGSTAB algorithm.Although SIMD vectorization has been shown as an efficient way to achieve good performance, it restricts how the operations should be performed. [14] proposes a compilation scheme to exploit the partial reuse of the vector register data and obtained a speedup of 1.46 through SSE. [15] presents an efficient implementation and detailed analysis of Merge Sort on current CPU architectures. [16] combines polyhedral tiling and parallelization tools with an aggressive vectorized tile code generator and achieve an improvement of 14.56 on a Knights Ferry. [17] presents parallel sorted set intersection algorithms that are based on Intel SSE 4.2. However there is no previous work on accelerating QMRCGSTAB algorithm via Xeon Phi's 8-way SIMD extension and its dense multi threads.

Porting PQMRCGSTAB Algorithm to Xeon Phi. As this paper focuses on single-node performance, we use Co-processor-only Model when porting PQMRCGSTAB algorithm to Xeon Phi. To fully utilize the 8-way vector intrinsic as well as the high thread density of parallel computing technology on Xeon Phi, as shown in Fig.3, we introduce a two-level parallelism method: First, porting PAMRCGSTAB to Xeon Phi with Intel intrinsic SIMD instructions; Second, spawning a new team of OpenMP threads to run on all other Xeon Phi cores.

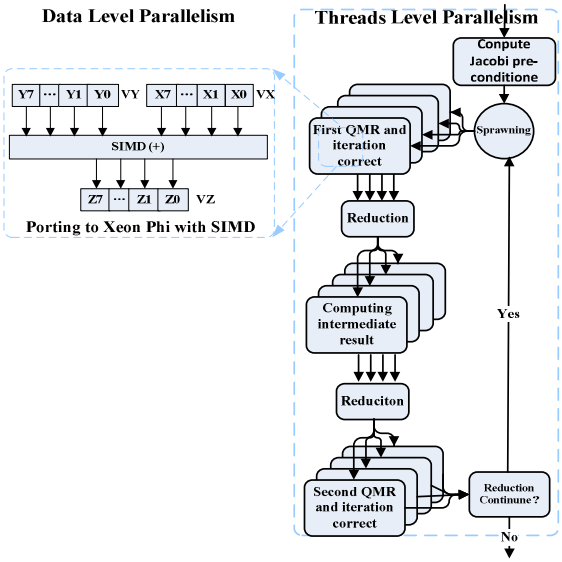

Figure 3. Parallel iteration algorithm on Xeon Phi

Data Level Parallelism. For data level parallelization, we employ SIMD techniques based on loop unrolling. We describe our solution by focusing on unrolling the computation of updating residual error r. After each loop, r is updated by the values of s. This loop represents the most frequent iterative code segment. Other loops in the algorithm have a similar process.

The CPU serial implementation is as follows:

```
1: for(i = 0; i < N; i ++)
2: {
3:   r[i]=s[i]- ω*r[i];
4: }
```

To utilize the 512-bit SIMD instruction, we unroll the loop by a factor of 8 which will consider broadcast ω in the outer loop, and pack 8 groups of r and s to fill the vector registers in the inner loop. After the 8 pairs of coordinates are ready, we can fulfill the formula in one instruction with the help of vector Fused–Multiply–Add (FMA) [18] instruction supported by Xeon Phi. Then write back data from register to memory.

The vectorized implementation is given as follows:

```
1: __m512d m1, m2, m3, m4; //Zmm register declare
2: m1 = _mm512_set1_pd(omega); //broadcast omega
3: for(i = 1; i < N; i = i + 8) //after each iteration i plus 8
4: {
5:   m2 = _mm512_load_pd(&r[i]); //data pack
6:   m3 = _mm512_load_pd(&s[i]); //data pack
7:   m4 = _mm512_fnadd_pd(m1, m3, m2); //FMA instruction
8:   _mm512_store_pd(&r[i], m4); //data unpack and write back
9: }
```

Threads Level Parallelism. We employ multithreading technology to exploit the threads level parallelism of PQMRCGSTAB algorithm across multiple cores, within each Xeon Phi, there are 60 cores. When the thread is about to perform the iterative task, it spawns several worker threads to accomplish the task.

As mentioned in Section 1, Xeon Phi supports both MPI and OpenMP. Iteration is compute-intensive, at the same time Xeon Phi has dense multi shared memory threads. For these reasons, we choose OpenMP to spawning a team of threads to run on all Xeon Phi cores.

Before spawning to multi threads, we should reorganize data to fully utilize shared memory programming model. [19] proposes to reduce the memory space of data and simplify parallel computation. The large-scale band sparse matrix is stored as AIJ format, which includes:

1) Building five vectors C, X1, X2, X3, X4 to store the nonzero elements, where the size of the vector is equal to the matrix dimension;

2) Establishing the relationship between the vector and the nonzero elements in matrix where we have:

C[i]=A(i,i), X1[i]=A(i,i+1), X2[i]=A(i,i-1), X3[i]=A(i,i+n), X4[i]=A(i,i-n).(n represents square root of dimension of A)

Thus, a large-scale band sparse matrix is transformed into five vectors, i.e., x=A×b is transformed to:

$$x[i]=C[i]*b[i]+X1[i]*b[I+1]+X2[i]*b[i-1]+X3[i]* b[i+n]+X4[i]*b[i-n]. \quad (1)$$

As shown in Fig. 4, multi threads cooperate with each other to complete computing task of:

[CT, X1T, X2T, X3T, X3T] × (bi, bi+1, bi-1, bi+n, bi-n) = x together. As shown in Fig.3, the algorithm consists of structured blocks when QMR and iteration correct block start, Xeon Phi forks multi threads, and partitions matrix into blocks. Block size is N/k (k means threads number), and the block addresses begin with N/k*i and end with (N/k+1)*i (i means thread id). Then each thread does its own work. At the end of parallel region, master thread performs synchronization.

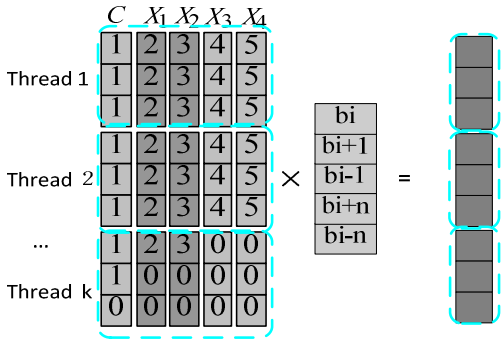

Figure 4. Parallel A×b=x

Further Optimization

Data Prefetching. Wider vector undoubtedly renders more computing capacity and higher peak performance. However, it should be noticed that wider vector registers also put more pressures on the memory hierarchy. We adopt compiling guided prefetching instruction to hide the latency of vector pack.

In order to use prefetching instructions with high efficiency, we should deliberate prefetching distance. Mowry [20] presents prefetching distance formulation: d = [l/s] + 1, where l indicates mean memory access latency, s is time clock cycle of nearest executive way in one loop. Latency of vector-add operation in Xeon Phi VPU is 3 time clock cycles, and multi-sub is 5. L2 Cache access is 11 time clock cycle, and L1 Cache is 3. Consequently, the distance of prefetching form L2 Cache to L1 cache in the loop we discussed in Section 4.1 should be [11/(3+3+5)] + 1= 2.

The loop uses 4 out of the 32 512-bit vector registers that one Xeon Phi core owns. To improve the usage factor further, we unroll the inner-most loop by a factor of 2. After that, we stream the load and computing instructions. That is, during the process i times iteration, the coordinates of i+1 are packed to Zmm registers before the FMA instruction. By adopting this stream method, we can overlap the read latency with the execution of the VPU engine.

Vector Register Reuse. In formula 1, if we directly pack data to vector register, a plenty of superfluous work will be done. For example, when i is 1, pack operation puts b[0], b[1],…b[7] from memory to vector register v1, b[1], b[2],…, b[8] to v2, and b[2], b[3],…, b[9] to v3. Obviously v1, v2 and v3 are the same except head and tail.

Qian et al [14] puts forward a kind of partial reuse of vector register. However, “Knights Corner Instruction” does not support the displacement of register on float point. In our implementation, to address the reuse of partial vector register, we adopt swizzle instruction. As shown in Fig.5, there are two steps: dividing each lane in v1 and v2 into 4 groups; and then choosing the first lane in each group of v1 as well as the second lane in each group of v3, and putting them into the corresponding group of v2 as second and first lane. Then we get v2 through swizzling v1 and v3.

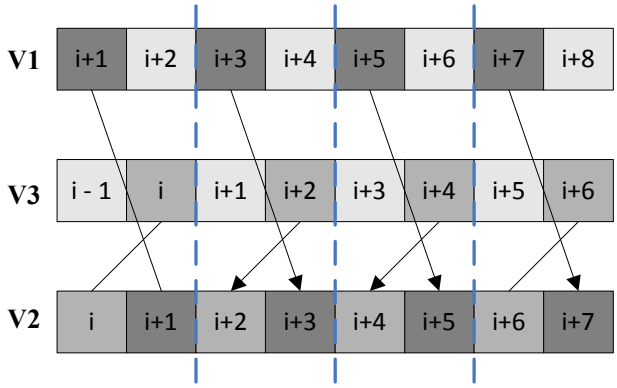

Figure 5. Zmm register swizzle operation

SIMD-friendly reduction. Data pack and unpack consume plenty of clock time cycles [2]. If VPU issues data pack and unpack instruction repeatedly, it will be waiting for data and idle. Increasing lingering time of packed data in register is a kind of optimization method. There is a lot of reduction like computing inner product in iteration. In serial code, we conduct a reduction when each loop iteration over. These instructions can significantly affect the effective instruction throughput by causing a large amount of data pack and unpack which will also increase memory access latency.

In reality, some intermediate results do not need to be-stored. To improve the performance of VPU, we consider a kind of SIMD-friendly reduction: achieving reduction operation in Zmm register. As shown in Fig. 6, it does not write results to memory until one loop completed. Instead it only writes partial result to a global register when each loop iteration over. After one loop completed, it reads every 8 lanes of the global register and accumulates them to memory.

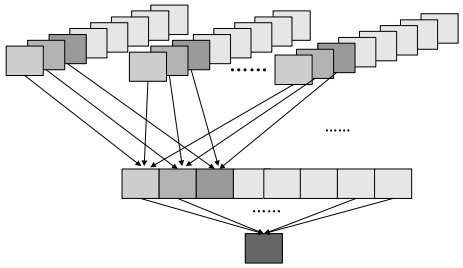

Figure 6. Reduction in Zmm vector register

Performance Evaluation

Experimental Setup. In this section we give the experimental results of our implementation on a Service node containing 2 Intel Xeon E5 octal-core running at 2.6GHz, with 128GB of memory and a Xeon Phi co-processor connected via PICe. The Xeon Phi is a 1.10GHz card with 60 cores running the Intel MIC software stack, with an additional pre-alpha patch.

The host processors are running Linux Red Hat 4.4.5-6, with kernel version 2.6.32-220.el6.x86_64. The version of OpenMP is 3.1. The compilers we use are Intel icc version 13.0 with optimization option "-O3".

Main Insights. To evaluate the performance of our work, we have performed experiments of various problem scales expressed as the dimension of sparse linear equations: 600*600, 800*800, 1000*100, and 1200*1200. Recorded running time starts when entering the iteration algorithm, and stops when the residual error is less than 1×e-7.

Fig. 7 compares the speedup factor obtained by using compiler auto vectorized and manual vectorized with different scale. Fig. 8 depicts the speedup when runs were made at various thread counts.

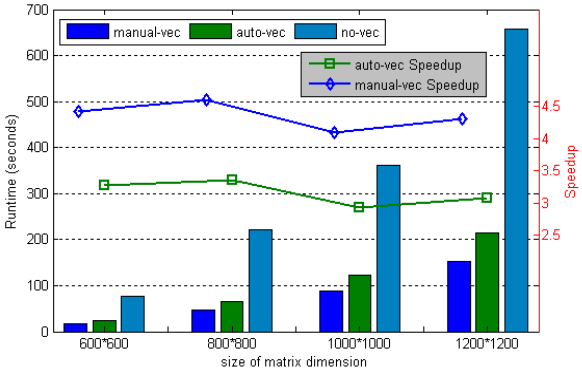

Figure 7. Speedup of vectorization

We define the auto-vec speedup as the Time of compiler with non-auto vectorized code divided by Time of compiler with auto vectorized. And also we define the manual-vec speedup as the Time of compiler with non-auto vectorize divide by Time of 8-way SIMD vectorized code. As shown in

Fig. 8, compiler auto-vec speedups are 3.3, 3.4, 2.9 and 3.0 with different matrix scales. After using the SIMD intrinsic as well as our optimizations we improve the manual-vec speedups to 4.4, 4.6, 4.0 and 4.3 respectively. The average speedup of auto-vec against manual-vec is 1.38, suggesting that Intel Xeon Phi intrinsic vector instructions and optimisms enable more vectorization. Noted that, all these experiments run with one thread.

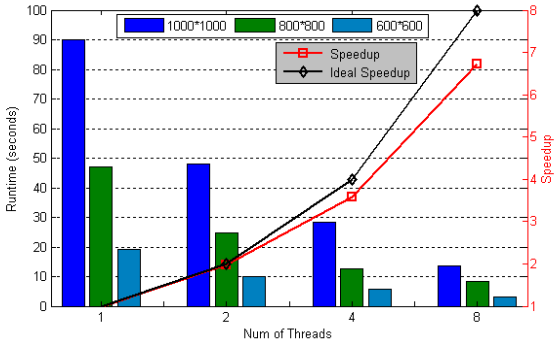

Figure 8. The speed up of multi threads

The speedups of multi-thread vectorized code are depicted in Fig. 8. The x-axis of the graph is the number of threads, increasing from 1 to 8 and the right y-axis indicates the time consumption of different matrix scales, left y-axis is the achieved speedup. When the number of threads is small, the performance may be greatly influenced by threads, so the performance improves scaling in Figure 6 as the number of threads increases. However, when the threads are large enough, the performance cannot be improved as the number of threads increasing (not shown in this figure).

After all, our goal is to indicate Xeon Phi's computing power. In the following table we compare running time (seconds) of Xeon Phi's best performance with Intel E5-2670 octal-core CPU on different matrix scales.

Table 1 Comparison of running times on CPU and Xeon Phi:

Scale	3200*3200	3000*3000	2800*2800	2600*2600
CPU	352.4s	292.6s	258.4s	220.4s
MIC	62.8s	55.5s	48.7s	42.1s

Conclusion

This paper utilizes Xeon Phi to accelerate PQMRCGSTAB algorithm, a linear equation solver. We implement PQMRCGSTAB algorithm on Xeon Phi using a two level parallel method, SIMD vectorization and multi thread parallelism. Then, we carefully design optimizations to match the capabilities of the architecture including data prefetching, vector register reusing and SIMD-friendly reduction. Our experimental results demonstrate that our PQMRCGSTAB algorithm achieves up to 6 speedups on one Xeon Phi comparing to that on one Intel Xeon E5 octal-core CPU. Because MIC not only supports MIC-on Model, but also can run with Offload Model and Symmetric Model. There will be more challenges to accelerate our algorithm on these programming models.

Acknowledgment

This work is supported by the National High Technology Research and Development Program of China (863 Program) No. 2012AA01A301, 2012AA010903, 2012AA01A309, and the National Natural Science Foundation of China (NSFC) NO.61170049.

References

[1] Karl W. Schulz, Rhys Ulerich, Nicholas Malaya, Paul T. Bauman, "Early Experiences Porting Scientic Applications to the Many Integrated Core (MIC) Platform"

[2] "Knights Corner Software Developers Guide", revision 1.03, April 27, 2012

[3] OpenMP: The OpenMP API Specication for Parallel Programming. http://openmp.org/wp/openmp-specications/

[4] MPI-2: Extensions to the Message-Passing Interface, Message Passing Interface Forum, Jul 1997.

[5] Sreeram Potluri, Karen Tomko, Devendar Bureddy, and Dhabaleswar K. Panda "Intra-MIC MPI Communication using MVAPICH2: Early Experience"

[6] Z. H. Yao and M. W. Yuan. "Computational Methods in Engineering & Science", Proceedings of "Enhancement and Promotion of Computational Methods in Engineering and Science X", Sanya, China, Aug, 2006, pp. 21–23.

[7] H.A. van der Vorst, "Bi-CGST AB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems", SIAM J. Sci. Comp. 13(1992), pp. 631-644.

[8] R.W. Freund and N.M. Nachtigal , "QMR: A quasi-minimal residual method for non-Hermitian linear systems", Numerische Mathematik 60(1991), pp. 315-339.

[9] http://en.wikipedia.org/wiki/Intel_MIC

[10] T. Elgar, "Intel Many Integrated Core (MIC) Architecture," in 2nd UK GPU Computing Conference, December 2010. [Online]. Available: http://www.many-core.group.cam.ac.uk/ukgpucc2/programme.shtml

[11] Chany, T.F., Gallopoulos, E., Simoncini, V., Szetoy, T., Tongz, C.H. "A Quasi-Minimal Residual Variant of the Bi-CGStab Algorithm for Nonsymmetric Systems", Scientific Computing, 1993.

[12] GE Zhen，YANG Can-qun，WU Qiang，and CHEN Juan, "Accelerating Iterative Methods in Solving Linear Systems on GPUs",

[13] Canqun Yang, Zhen Ge, Juan Chen, Feng Wang, and Qiang Wu, "Accelerating PQMRCGSTAB Algorithm on GPU",

[14] QIAN Xinglong, ZANG Bimyu, ZHU Chnamqi, "Partial Reuse of the Vector Registers in SIMD Optimization", COMPUTER ENGINEERING&SCIENCE, Vol.29,No.5,2007

[15] Franz Franchetti, StefanKral, JuergenLorenz, and Christoph W. Ueberhube, "Efficient Utilization of SIMD Extensions", IEEE proceedings special issue on program generation, optimization, and platform adaptation,

[16] Kevin Stock, Louis-Noel Pouchet, P. Sadayappan, "Automatic Transformations for Eective Parallel Execution on Intel Many Integrated Core"

[17] Benjamin Schlegel, Thomas Willhalm, Wolfgang Lehner, "Fast Sorted-Set Intersection using SIMD Instructions"

[18] Chris Lomont, "Introduction to Intel® Advanced Vector Extensions"

[19] Canqun Yang, Zhen Ge, Juan Chen, Feng Wang, and Yunfei Du, "Solving 2D Nonlinear Unsteady Convectio n-Diffusion Equations on heterogenous platforms with multiple GPUs",

[20] Todd.C.Mowry, Tolerating Latency through Software-controlled Data Pre-fetching, PhD thesis, University of Standford, March 1994,

Advances in Applied Science, Engineering and Technology
10.4028/www.scientific.net/AMR.709

Accelerating PQMRCGSTAB Algorithm on Xeon Phi
10.4028/www.scientific.net/AMR.709.555