Parallel Sparse Linear Solver GMRES for GPU Clusters with Compression of Exchanged Data*

Jacques M. Bahi, Raphaël Couturier, and Lilia Ziane Khodja

University of Franche-Comte, LIFC laboratory,
Rue Engel-Gros, BP 527, 90016 Belfort Cedex, France
{jacques.bahi,raphael.couturier,lilia.ziane_khoja}@univ-fcomte.fr

Abstract. GPU clusters have become attractive parallel platforms for high performance computing due to their ability to compute faster than the CPU clusters. We use this architecture to accelerate the mathematical operations of the GMRES method for solving large sparse linear systems. However the parallel sparse matrix-vector product of GMRES causes overheads in CPU/CPU and GPU/CPU communications when exchanging large shared vectors of unknowns between GPUs of the cluster. Since a sparse matrix-vector product does not often need all the unknowns of the vector, we propose to use data compression and decompression operations on the shared vectors, in order to exchange only the needed unknowns. In this paper we present a new parallel GMRES algorithm for GPU clusters, using compression vectors. Our experimental results show that the GMRES solver is more efficient when using the data compression technique on large shared vectors.

Keywords: GMRES, GPU cluster, CUDA, MPI, data compression.

1 Introduction

Iterative linear solvers are often more suited than direct ones for solving large sparse linear systems. In fact, an iterative method computes a sequence of approximate solutions converging to the exact solution. In contrast, a direct method determines the exact solution after a finite number of operations which may lead to an expensive consumption in both computation time and memory space, and thus, it is not very well suited for large linear systems. GMRES (Generalized Minimal REsidual method) is one of the most widly used iterative solvers chosen to deal with the sparsity and the large order of linear systems. It was initially developed by Saad and al. [1] to deal with nonsymmetric and non-Hermitian problems, and indefinite symmetric problems too. The convergence of the restarted GMRES with preconditioning is faster and more stable than those of some other iterative solvers. Furthermore, the GMRES algorithm is mainly based on mathematical matrix/vector operations that are easily parallelizable, and therefore, they allow us to exploit the computing power of parallel platforms.

* This work was supported by Région de Franche-Comté.

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 471–480, 2012.
© Springer-Verlag Berlin Heidelberg 2012

For the past few years, GPUs (Graphic Processing Units) have proved their ability to provide better performance than CPUs for many parallel applications, including solving linear systems [2]. They have become high performance accelerators for data-parallel tasks and intensive arithmetic computations. Therefore, several works have proposed the efficient GMRES algorithms using the computing power of GPUs [3][4].

Nowadays, the parallel platforms exploiting the high performances of this architecture are GPU clusters. They are very attractive for high performance computing, given their low cost compared to their computational power and their abilities to compute faster and to consume less energy than their pure CPU counterparts [5].

We have already used GPU clusters to accelerate numerical computations of GMRES method for solving large sparse linear systems [6]. We have tested our parallel GMRES solver on linear systems with banded sparse matrices. We have noticed that GPU clusters are less efficient in case of large matrix bandwidths. Indeed a matrix bandwidth defines the size of the shared vectors that must be exchanged between GPUs in order to perform the full matrix-vector products of the GMRES method. So a large matrix bandwidth leads to the transfer of large vectors between a CPU core and its GPU, whereas a GPU/CPU data transfer is the slowest communication in GPU cluster and it affects greatly the performances of sparse linear system solutions.

In this paper, we propose a new parallel GMRES solver with some improvements to reduce the communication overheads. We use the *compression* and the *decompression* operations on the shared vectors. This technique has already used to speed-up the data transfers between the computing nodes of a cluster. In [7], the author has proposed a dynamic algorithm with the compression/communication overlap which is suited for any data transfer whatever the speed of the network. In our case, this technique allows a GPU to communicate only the shared unknowns required by other GPUs to its CPU core. This paper is organized as follows. In section 2 a general overview of the GPU architecture is given. In section 3 the main key points of our improved parallel GMRES solver for GPU clusters are presented. Section 4 is devoted to the performance evaluation of our solver. Section 5 concludes this paper.

2 GPU Architecture

A GPU architecture is composed of hundreds of processors organized in several streaming multiprocessors. It is also equipped with a memory hierarchy. It has a private read-write *local memory* per processor, a fast *shared memory* and read-only *constant* and *texture* caches per multiprocessor, and a read-write *global memory* shared by all its processors. To exploit the computing power of this architecture, Nvidia has released the CUDA programming language (Compute Unified Device Architecture) [8] allowing us to program GPUs for general purpose computations of graphic and non-graphic applications. In CUDA programming environment, the GPU is viewed as a co-processor to the CPU. All data-parallel

operations of a CUDA application running on the CPU are off-loaded onto the GPU. CUDA is C programming language with a minimal set of extensions to define the parallel functions to be executed by the GPU as *kernels*.

At the GPU level, the same kernel is executed by a high number of parallel CUDA threads grouped together as a grid of thread blocks. Each multiprocessor of the GPU executes one or more thread blocks in SIMD fashion (Single Instruction, Multiple Data) and in turn each processor of a GPU multiprocessor runs one or more threads within a block in SIMT fashion (Single Instruction, Multiple threads). In order to avoid the execution dependencies between thread blocks, the number of CUDA threads involved in a kernel execution is computed according to the size of the problem to be solved. In contrast, the block size is restricted by the limited memory resources of a processor. On current GPUs, a thread block may contain up to 1024 concurrent threads.

GPUs only work on data filled in their global memories and the final results of their kernel executions must be communicated to their hosts (CPUs). Hence, the data must be transferred *in* and *out* of the GPU. However, the speed of memory copy between the GPU and the CPU is slower than the memory copy speed of the GPUs. Accordingly, it is necessary to limit data transfers between the GPU and its host during the computations.

3 GMRES Implementation on GPU Clusters

3.1 Parallel GMRES Algorithm for GPU Clusters

Algorithm 1 shows the key points of the parallel GMRES algorithm for GPU clusters that we developed in our previous work [6]. It must be executed in parallel by each pair (CPU core, GPU) of the cluster, such that each CPU core holds one MPI process managing one GPU. GMRES is mainly based on matrix/vector operations: sparse matrix-vector products denoted in Algorithm 1 by $SpMV()$, dot products (line 15), scalar-vector products (lines 8 and 19), Euclidean norms (lines 7, 18 and 27) and AXPI operations (line 16). All parallel and mathematical functions inside the main loop of GMRES are executed as kernels by the GPU. The superscripts *local* and *shared* over solution vector x and vector v respectively denote the local vector and the shared vectors with neighbor processes, required to perform full sparse matrix-vector products.

Besides these local computations, synchronizations between GPUs must be performed to ensure the solving of the complete sparse linear system. Before computing an *SpMV* product, it is mandatory to construct the global vector x (or v_j) required for the full product. First each GPU copies the entries of vector x^{local} (resp. v_j^{local}) to its host vector h_tmp^{local} (lines 3, 10 and 24), then all MPI processes in the cluster exchange their shared entries of vector h_tmp^{local} (lines 4, 11 and 25) using an $MPI_Alltoallv()$ function and, finally each MPI process copies entries of the computed shared vector h_tmp^{shared} to its GPU vector x^{shared} (resp. v_j^{shared}) (lines 5, 12 and 26). After each vector operation, as Euclidean norms and dot products, the MPI processes must perform a reduction operation on local scalars computed by their GPUs, by using

Algorithm 1. Left-Preconditioned GMRES with Restarts for GPU Clusters

1: Set ε the tolerance for the residual norm r, $convergence = false$ and x_0
2: **while** $!convergence$ **do**
3: $gpu_to_cpu(x_0^{local}, h_tmp^{local})$
4: $\boldsymbol{DataExchange}(h_tmp^{local}, h_tmp^{shared})$
5: $cpu_to_gpu(h_tmp^{shared}, x_0^{shared})$
6: $r_0 \leftarrow M^{-1} \times (b - SpMV(A, x_0^{local}, x_0^{shared}))$
7: $\beta \leftarrow \|r_0\|_2$
8: $v_1 \leftarrow r_0 / \boldsymbol{Sqrt}(\boldsymbol{AllReduceSum}(\beta^2))$
9: **for** $j = 1$ to m **do**
10: $gpu_to_cpu(v_j^{local}, h_tmp^{local})$
11: $\boldsymbol{DataExchange}(h_tmp^{local}, h_tmp^{shared})$
12: $cpu_to_gpu(h_tmp^{shared}, v_j^{shared})$
13: $w_j \leftarrow M^{-1} \times SpMV(A, v_j^{local}, v_j^{shared}))$
14: **for** $i = 1$ to j **do**
15: $h_{i,j} \leftarrow (w_j, v_i^{local})$
16: $w_j \leftarrow w_j - \boldsymbol{AllReduceSum}(h_{i,j}) \cdot v_i^{local}$
17: **end for**
18: $h_{j+1,j} \leftarrow \|w_j\|_2$
19: $v_{j+1}^{local} \leftarrow w_j / \boldsymbol{Sqrt}(\boldsymbol{AllReduceSum}(h_{j+1,j}^2))$
20: **end for**
21: Set $V_m = \left[v_1^{local}, \ldots, v_m^{local}\right]$ and $\bar{H}_m = (h_{i,j})$
22: Solve: $min_{y \in \mathbb{R}^m} \left\|\beta e_1 - \bar{H}_m y\right\|_2$
23: $x_m^{local} \leftarrow x_0^{local} + V_m y_m$
24: $gpu_to_cpu(x_m^{local}, h_tmp^{local})$
25: $\boldsymbol{DataExchange}(h_tmp^{local}, h_tmp^{shared})$
26: $cpu_to_gpu(h_tmp^{shared}, x_m^{shared})$
27: $\delta \leftarrow \left\|M^{-1} \times (b - SpMV(A, x_m^{local}, x_m^{shared}))\right\|_2$
28: **if** $\boldsymbol{Sqrt}(\boldsymbol{AllReduceSum}(\delta^2)) < \varepsilon$ **then**
29: $convergence \leftarrow true$
30: **end if**
31: $x_0^{local} \leftarrow x_m^{local}$
32: **end while**

MPI_Allreduce() function. In Algorithm 1, the function calls written with bold fonts denote the functions to be executed by the MPI process. ***DataExchange()*** denotes the *MPI_Alltoallv*() function to build the global vectors, ***AllReduceSum()*** denotes the *MPI_Allreduce*() function using the summation operation and, ***Sqrt()*** denotes the square root operation. For more details about our parallel GMRES algorithm, please refer to [6].

3.2 Minimizing Communication Overheads

As we can see from Algorithm 1, our parallel GMRES solver requires data transfers between the different components of the GPU cluster. Indeed before any computing of the *SpMV* product, we must construct the global vector of unknowns x required for this operation by using the following data transfers:

(1) a memory copy of the local vector from the GPU memory to the host memory, (2) a data exchange of the shared vectors between all MPI processes and, (3) a memory copy of the shared vector from the host memory to the GPU memory.

However, as we mentioned in Section 2, data transfers *from* or *to* the GPU memory are the slowest communications in a GPU cluster. Hence, the GPU/CPU data transfers of large local and/or shared vectors can dramatically reduce the performances of solving sparse linear systems. Nevertheless the sparse matrix-vector products do not often need all the values of the shared vector. So in order to reduce the GPU/CPU and CPU/CPU communication overheads, we propose to perform *compression* and *decompression* operations on the shared vectors of unknowns.

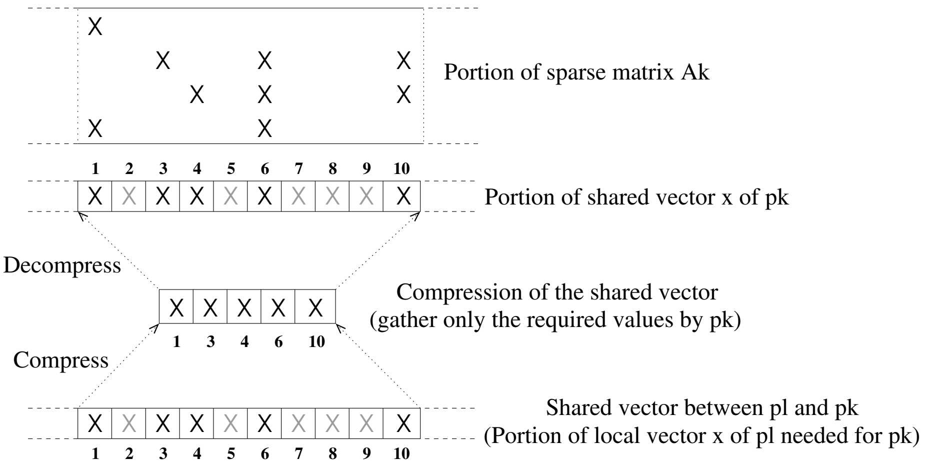

Fig. 1. Compression/decompression of shared vector x between processes p_k and p_l

After the data partitioning, each process p_k sends to its neighbors the indices of global vector of unknowns x_k needed for its full *SpMV* products. As shown in Figure 1, process p_k needs unknowns corresponding to indices 1, 3, 4, 6 and 10 in the local vector of process p_l. So before the GPU→CPU data transfer of local vector x_l^{local}, neighbor process p_l uses these indices (1, 3, 4, 6 and 10) to *compress* x_l^{local}. The *compression* operation allows process p_l to build a small shared vector x^{comp} from its local vector x_l^{local}, consisting of only the shared unknowns needed by process p_k (vector elements drawn with bold fonts X). The CPU↔CPU data exchanges of these shared compressed vectors must be performed between all processes on the cluster. Once the GPU←CPU data transfer of vector x^{comp} is held, process p_k *decompresses* shared vector x^{comp} received from its neighbor p_l such that each value of x^{comp} is copied to the corresponding index of its shared vector x_k^{shared}.

In order to accelerate the computations, the compression/decompression operations must be performed by GPUs. We developed in CUDA two kernels to compress and decompress the exchanged shared vectors of unknowns before

gpu_to_cpu() and after *cpu_to_gpu*() communications. These operations allow the transfer of small vectors between the MPI processes and between an MPI process and its GPU. Hence they minimize the GPU/CPU and CPU/CPU communication overheads.

4 Performance Evaluation

4.1 Our GPU Cluster

Our GPU cluster is an Infiniband cluster having six Xeon E5530 CPUs. Each CPU is a Quad-Core processor running at 2.4GHz. It provides a RAM memory of 12GB with a memory bandwidth of 25.6GB/s, and it is equipped with two Nvidia Tesla C1060 GPUs. In turn, each GPU contains in total 240 processors running at 1.3GHz. It provides 4GB of global memory with a memory bandwidth of 102GB/s, accessible by all its processors and also by the CPU through the PCI-Express 16x Gen 2.0 interface with a throughput of 8GB/s. Hence, the memory copy operations between the GPU and the CPU is about 12 times slower than those of the Tesla GPU memory.

Linux cluster version 2.6.18 OS is installed on CPUs. C programming language is used for coding the GMRES algorithm on both GPU cluster and CPU cluster. CUDA version 3.1.1 [8] is used for programming GPUs, using CUBLAS 3.1 [9] to deal with vector operations and CUSP library [10] to perform a HYB SpMV product in GPUs, and finally MPI functions of OpenMPI 1.3.3 are used to carry out communications between CPU cores.

4.2 Sparse Matrices of Tests

We chose to work on linear systems having banded sparse matrices, since they arise in many numerical computations, and large sizes exceeding 10 million of unknowns. For that, we developed in C a generator of large sparse matrices which takes one real matrix of the Davis collection [11] as an initial matrix to build large banded matrices. This generator must be executed in parallel by all MPI processes before starting the resolution of the linear system.

In addition to the matrix generation, the generator performs the data partitioning of the generated matrix among all pairs of (MPI process, GPU). According to the desired matrix size n of the sparse linear system and the number of the pairs (MPI process, GPU) p in the cluster, each MPI process k computes the size of its sub-matrix $sizeloc_k$ and its $offset_k$ in the global generated matrix, such that:

$$sizeloc_k = \frac{n}{p}. \tag{1}$$

$$offset_k = \begin{cases} 0 & \text{if } k = 0 \\ offset_{k-1} + sizeloc_{k-1} & \text{otherwise} \end{cases} \tag{2}$$

The offsets and the sizes of the sub-matrices on the cluster allow a process to determine which processes own the needed unknown values.

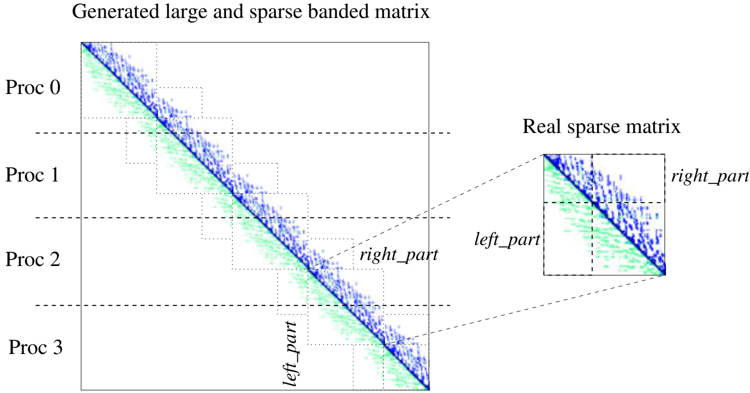

Fig. 2. A large sparse banded matrix generated by four processes from a real matrix of the Davis collection

After that, each MPI process k builds its sub-matrix of size $sizeloc_k$ by performing several copies of the same real matrix of the Davis collection. And all generated sub-matrices in the cluster construct the global sparse matrix of the linear system. In order to generate banded matrices, each MPI process places its copies on its part of the main diagonal of the global matrix as shown in Figure 2. Furthermore, the empty spaces between two consecutive copies on the main diagonal are fulfilled by sub-copies *right_part* and *left_part* of the same initial real matrix.

4.3 Experimental Results

The performance evaluation of our GMRES solver is made in double precision data. All experimental results obtained from our tests are for a residual tolerance threshold $\varepsilon = 10^{-10}$, a restart limit of GMRES method $m = 16$, a right-hand side b filled with 1 and an initial guess x_0 filled with 12. For the sake of simplicity, we took the preconditioning matrix M as the main diagonal of the sparse matrix A of the linear system. Indeed it allows us to easily compute the required inverse matrix M^{-1} and it provides a relatively good preconditioning in most cases.

Table 1 shows the main characteristics of the banded sparse matrices on which we performed our tests. First column gives the type of test matrices: *symmetric* or *unsymmetric*. In the second column, we have the set of real sparse matrices chosen in the Davis collection and from which we generated our sparse matrices of tests. All sparse linear systems solved in our tests are of size 90 million of unknowns. The fourth and fifth columns show respectively the number of nonzero values and the bandwidth of the generated matrices of tests.

In our tests, we compared the performances of the parallel GMRES solver implemented on a cluster of 12 GPUs with those obtained on cluster of 12 CPU cores and those obtained on a cluster of 24 CPU cores. Tables 2 and 3 report respectively the performances of the GMRES solver *without* and *with* the data

Table 1. The main characteristics of the generated sparse banded matrices of tests

Matrix type	Real matrix	Nb. rows	Nb. nonzeros	Bandwidth
Symmetric	ecology2	$90 \cdot 10^6$	449,729,174	1,002
	finan512	$90 \cdot 10^6$	915,824,547	106,017
	G3_circuit	$90 \cdot 10^6$	443,429,071	525,429
	shallow_water2	$90 \cdot 10^6$	360,751,026	23,212
	thermal2	$90 \cdot 10^6$	643,458,527	1,928,223
Unsymmetric	cage14	$90 \cdot 10^6$	1,674,718,790	1,266,626
	language	$90 \cdot 10^6$	276,894,366	398,626
	stomach	$90 \cdot 10^6$	1,277,498,438	22,868
	swang2	$90 \cdot 10^6$	600,518,274	5,801
	torso3	$90 \cdot 10^6$	1,561,856,844	327,737

compression technique. We took into account the speedups of both GMRES solvers implemented on the GPU cluster compared to those implemented on CPU clusters.

The fourth and sixth columns of these tables show respectively the ratio of execution times T_{12cpus} and T_{gpu} and the ratio of execution times T_{24cpus} and T_{gpu}, where T_{gpu} is the execution time obtained on the cluster of 12 GPUs, T_{12cpus} is that obtained on the cluster of 12 CPUs and T_{24CPUs} is that obtained on the cluster of 24 CPUs. The ratios define the relative gains of the GMRES solvers implemented on the GPU cluster compared to those implemented on CPUs clusters, such that:

$$ratio = \frac{T_{cpu}}{T_{gpu}}. \tag{3}$$

Table 2. Performances of GMRES solver *without* data compression on a GPU cluster and CPU clusters

Matrix	T_{gpu}	T_{12cpus}	$ratio_{12cpus}$	T_{24cpus}	$ratio_{24cpus}$	#iter	Prec.	Δ
ecology2	1.68s	14.22s	8.46	9.85s	5.86	22	1.86e-10	2.32e-10
finan512	5.35s	43.03s	8.04	28.97s	5.42	52	1.03e-09	2.66e-15
G3_circuit	2.05s	16.44s	8.02	11.30s	5.50	25	1.44e-09	9.66e-13
shallow_water2	2.45s	22.98s	9.83	15.88s	6.49	33	4.27e-15	1.35e-18
thermal2	3.95s	25.91s	6.56	17.12s	4.34	31	3.15e-09	5.88e-15
cage14	2.95s	20.44s	6.93	15.62s	5.30	21	1.39e-08	1.20e-11
language	9.60s	84.58s	8.81	56.57s	5.89	112	2.61e-08	3.78e-10
stomach	12.66s	107.97s	8.53	74.40s	5.88	125	1.10e-08	2.13e-14
swang2	3.84s	32.42s	8.44	22.10s	5.76	45	5.75e-08	3.41e-13
torso3	15.20s	125.45s	8.25	86.40s	5.68	134	1.93e-08	3.13e-13

From both ratios shown in Table 2 and Table 3, we can see that the GMRES solvers implemented on the GPU cluster are faster than those implemented on the CPU clusters. Moreover, the GMRES solver *with* the data compression technique is faster than that *without* the data compression. Indeed, the GMRES

Table 3. Performances of GMRES solver *with* data compression on a GPU cluster and CPU clusters

Matrix	T_{gpu}	T_{12cpus}	$ratio_{12cpus}$	T_{24cpus}	$ratio_{24cpus}$	#iter	Prec.	Δ
ecology2	1.06s	14.13s	13.33	9.75s	9.18	22	1.86e-10	2.32e-10
finan512	4.27s	42.95s	10.06	29.02s	6.80	52	1.03e-09	2.66e-15
G3_circuit	1.32s	16.33s	12.37	11.24s	8.53	25	1.44e-09	9.66e-13
shallow_water2	1.78s	22.95s	12.89	15.85s	8.92	33	4.27e-15	1.35e-18
thermal2	2.36s	25.26s	10.70	16.64s	7.05	31	3.15e-09	5.88e-15
cage14	2.25s	20.50s	9.11	13.99s	6.21	21	1.39e-08	1.20e-11
language	7.04s	84.02s	11.93	56.63s	8.05	112	2.61e-08	3.78e-10
stomach	10.11s	107.73s	10.66	74.30s	7.35	125	1.10e-08	2.13e-14
swang2	2.95s	32.37s	10.97	22.14s	7.51	45	5.75e-08	3.41e-13
torso3	11.61s	124.70s	10.74	86.24s	7.43	134	1.93e-08	3.13e-13

solver using the data compression on the GPU cluster is about 11 times faster than on the cluster of 12 CPUs and about 7.7 times faster than on the cluster of 24 CPUs. In contrast, the GMRES solver without the data compression on the GPU cluster is about 8 times faster than on the cluster of 12 CPUs and about 5.6 times faster than on the cluster of 24 CPUs. Hence, it is interesting to use the *compression/decompression* operations on the shared vectors of unknowns *before/after* the GPU/CPU data transfers in the GMRES solver on GPU clusters. Indeed it allows us to exchange small shared vectors between the different components of the GPU cluster, and thus, to minimize the communication overheads in solving large sparse linear systems using GMRES solver.

The seventh, eighth and ninth columns of Table 2 and Table 3 give respectively the number of iterations for solving the linear system, the solution precision $Prec$ computed on the GPU cluster and the difference Δ between solutions computed on the CPU clusters and the GPU cluster, such that:

$$Prec = max(M^{-1} \cdot (b - AX^{GPU})). \tag{4}$$

$$\Delta = max|X^{CPU} - X^{GPU}|. \tag{5}$$

where X^{CPU} and X^{GPU} are respectively the solutions computed on the CPU cluster and the GPU cluster. We can see that the precisions $Prec$ of the solutions computed on the GPU cluster are sufficient, varying from 5.75e-8 to 4.27e-15, and the two versions of GMRES solver compute almost the same solutions in both CPU cluster and GPU cluster, with Δ varying from 3.78e-10 to 1.35e-18.

5 Conclusion

In this paper we have presented an efficient parallel GMRES algorithm for solving large sparse linear systems on GPU clusters. We have shown that it is interesting to use compression/decompression techniques on the sub-vectors of unknowns

shared between GPUs of the cluster. In fact, the parallelization of GMRES method on a GPU cluster requires CPU/CPU and GPU/CPU data transfers in order to perform full matrix-vector products. And since the parallel sparse matrix-vector product does not often need all values of the vector, the use of the data compression and decompression techniques on the shared vectors before and after the GPU/CPU data transfers allows us to minimize the communication overheads of the parallel GMRES solver on GPU clusters. The experimental results show that the GMRES solver is more efficient when it uses the data compression/decompression techniques on GPU clusters.

Obviously, even if the compression/decompression techniques reduce globally the communication overheads, they do not provide enough gains to the GMRES solver on GPU clusters. In fact, a large matrix bandwidth produces many data dependencies between GPUs of the cluster. It means that each GPU will have many neighbors with which it will share data. Therefore, a large matrix bandwidth increases the number of communications, and thus, it reduces dramatically the performances of solving large sparse linear systems on GPU clusters. In future work, we will study the data partitioning methods required to minimize the data dependencies between the computing nodes of a GPU cluster and to improve the performance of the GMRES solver for GPU clusters.

References

1. Saad, Y., Schultz, M.: GMRES: a Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. SIAM J. Sci. Stat. Comput. 7(3), 856–869 (1986)
2. Jost, T., Contassot-Vivier, S., Vialle, S.: An Efficient Multi-algorithms Sparse Linear Solver for GPUs. In: EuroGPU Mini-Symposium of ParCo 2009, Lyon, pp. 546–553 (2009)
3. Wang, M., Klie, H., Parashar, M., Sudan, H.: Solving Sparse Linear Systems on NVIDIA Tesla GPUs. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009. LNCS, vol. 5544, pp. 864–873. Springer, Heidelberg (2009)
4. Ghaemian, N., Abdollahzadeh, A., Heinemann, Z., Harrer, A., Sharifi, M., Heinemann, G.: Accelerating the GMRES Iterative Linear Solver of an Oil Reservoir Simulator using the Multi-Processing Power of Compute Unified Device Architecture of Graphics Cards (2010)
5. Abbas-Turki, L., Vialle, S., Lapeyre, B., Mercier, P.: High Dimensional Pricing of Exotic European Contracts on a GPU Cluster, and Comparison to a CPU Custer. In: IPDPS 2009, pp. 1–8. IEEE Computer Society (2009)
6. Bahi, J., Couturier, R., Ziane Khodja, L.: Parallel GMRES Implementation for Solving Sparse Linear Systems on GPU Clusters. In: HPC Symposium, pp. 23–30. ACM/SIGSIM, Boston (2011)
7. Jeannot, E.: Improving Middleware Performance with AdOC: An Adaptive Online Compression Library for Data Transfer. In: IPDPS, vol. 1, p. 70. IEEE, USA (2005)
8. Nvidia: NVIDIA CUDA C Programming Guide, Version 3.1.1 (2010)
9. Nvidia: Cuda Cublas Library, Version 3.1 (2010)
10. CUSP library, `http://code.google.com/p/cusp-library/`
11. Davis, T., Hu, Y.: The University of Florida Sparse Matrix Collection (1997), `http://www.cise.ufl.edu/research/sparse/matrices/`