Vol.2009-ARC-186 No.15
Vol.2009-HPC-123 No.15
2009/12/1

CG on GPU-enhanced Clusters

Ali Cevahir,[†1] Akira Nukada[†1]
and Satoshi Matsuoka[†1,†2]

Motivated by high computation power and low price per performance ratio of GPUs, GPU accelerated clusters are being built for high performance scientific computing. In this work, we explain implementation of a mixed precision Conjugate Gradient solver for unstructured matrices on a GPU-extended cluster. Basic computations of the solver are held on GPUs and communications are managed by the CPU. For sparse matrix-vector multiplication, which is the most time-consuming operation, solver automatically selects the fastest between several high performance kernels running on GPUs. In a GPU-extended cluster, it is more difficult than traditional CPU clusters to obtain scalability, since GPUs are very fast compared to CPUs. GPU-extended clusters demand faster communication between computation units. We demonstrate performance of the solver and discuss communication bottleneck for the solver using up to 64 GPUs.

1. Introduction

GPUs, which are originally designed for accelerating computer graphics applications, are now being used for wide range of general purpose applications such as physics simulations, bioinformatics, cryptography, database, etc.[8] Modern GPUs provide higher compute capability and memory bandwidth with lower price and power consumption per performance ratios, compared to conventional CPUs. Therefore, nowadays they are considered as a good companion or alternative computing resources to CPUs for applications requiring high computation power and memory bandwidth. Manufacturers like NVIDIA and ATI supports general purpose computing on GPUs and release software APIs (CUDA[18] and CTM[1]) which hide graphics interface, making easy to program GPUs as highly parallel many core coprocessors. Recent GPUs support double precision floating point operations based on IEEE 754 standard, for scientific applications requiring higher accuracy.

Compute Unified Device Architecture (CUDA) is NVIDIAs new generation GPU hardware and software architecture. A CUDA GPU contains number of SIMD multiprocessors. GPU has a device memory that is accessible by all processors. Memory access latency of many core GPU devices is hidden by running high number of threads in parallel. Each multiprocessor contains its own shared memory and read-only constant and texture caches that are accessible by all processors within the multiprocessor. Threads in the same multiprocessor can communicate through fast shared memory. CUDA API supports programming different memory types.

Some systems integrate GPU clusters to be used as visualization resources. Examples include installations by GraphStream with 212-node Eureka system at Argonne National Laboratory and 256-node Gauss system at Lawrence Livermore National Laboratory[13]. Considering GPUs as high-performance low-cost many core co-processors, GPU clusters are being deployed for high performance scientific computing. Worlds largest GPU cluster is TSUBAME system deployed in our university, Tokyo Institute of Technology, which is currently in 41st place of Top500. TSUBAME integrates 680 NVIDIA Tesla GPUs and is the first system that is listed in Top500 as a GPU-enhanced cluster[17]. Some non-graphics applications in the literature running on GPU accelerated clusters are flow simulation using the Lattice Boltzmann model on 32 GPUs[10], biomedical image analysis with 32 GPUs[15] and FEM calculations with 160 GPUs[12].

In this work, we study sparse linear iterative solvers on a GPU-enhanced cluster. Each node of cluster may have several GPUs installed on it, which are controlled by host CPU(s). Particularly, we present our approaches and results on a Conjugate Gradient (CG) solver. We adopt a mixed precision iterative refinement algorithm[6], which is faster than full double precision implementation, without sacrificing solution accuracy. Several BLAS operations are consisted in sparse iterative solvers. Sparse matrix vector multiplication (MxV) is usually the most time-consuming of them. Parallel execution of sparse solvers for unstructured problems on a cluster requires considerable amount of communication, e.g.,

†1 Tokyo Institute of Technology
†2 National Institute of Informatics

for sharing input and/or output vector of MxV. Hence, minimization techniques for inter-process communication should be considered for efficient parallel implementations. For a multi-GPU cluster, where each node has more than one GPU, parallelization is even harder. To achieve an efficient parallel implementation, parallelization inside a GPU between GPU cores, inside a node between GPUs and between nodes should be carefully handled. Kernels running on GPUs require high degree of fine-grained parallelization between cores of a GPU. This imposes careful workload distribution between GPU threads. Optimization techniques for accessing GPUs complex memory architecture for memory-intensive kernels, e.g. MxV, should be carefully thought. GPU adds heterogeneity to the cluster, which should be handled by careful organization of operations running on GPUs and CPUs. Additional communication between GPUs and CPU cores is required. Compared to the very fast GPU computation units, communication bandwidth between nodes remains slow, as a result reduces parallel efficiency.

Other than scalar square root and division operations, all basic vector and matrix operations of our solver are held on GPUs. For MxV, our solver automatically selects the fastest between several kernels proposed by NVIDIA researchers[4] and ourselves[9]. For minimization of communication and load balancing of MxV between nodes and GPUs, we utilize state-of-the-art 1D hypergraph partitioning models[7].

To demonstrate effectiveness of our proposals, we held experiments on a set of well-known matrices. We compare strong scalability results of GPU vs. CPU cluster implementations on the same underlying InfiniBand-based network, providing 20 Gbps per node. On 16 nodes of TSUBAME, each node having 2 GPUs, we obtain up to 152 Gflops CG performance and 13.2 times speedup over single GPU implementation. This is 15 times faster than CPU implementation of same number of CPU cores. We use up to 16 cores per node for CPU experiments, and observe that CG is always faster on GPU cluster than CPU cluster.

The organization of the remaining sections is as follows. In Section 2, we give some background information on the techniques we have used in this work. In Section 3, we explain efficient implementation of basic operations of CG on the GPU. In Section 4, we explain the parallel CG algorithm and techniques for efficient parallelization of CG on multi-GPU clusters. In Section 5, we represent and discuss experimental results. We conclude in Section 6.

2. Background

2.1 Sparse matrix-vector multiplication on GPUs

Based on different compressed sparse matrix storage schemes, different MxV algorithms are proposed on the GPU which effectively utilize its resources. There are many compressed storage formats of sparse matrices. Please refer [19] for explanations of storage formats.

NVIDIA has recently released an MxV library, called SpMV, based on CUDA[4]. Six different MxV implementations are explained in the paper for structured and unstructured matrices. Among those different implementations, two of them, which they call CSR vector and HYB kernels, mostly achieve better performance for unstructured matrices. Naïve CSR algorithm is called CSR scalar, in which every matrix row is assigned to a different GPU thread. Memory access is costly for this simple thread assignment. To reduce memory access time by coalesced memory accesses and better load balancing amongst running threads, in CSR vector kernel one warp of threads are assigned for a matrix row. This kernel is sensitive to matrix row sizes. Another faster kernel HYB processes matrix that is decomposed in structured and unstructured parts. A fixed number of nonzeros per row are stored in ELL format and remaining entries are stored in COO format. ELL format is very suitable for vector architectures and full coalescing for global memory reads can be achieved. Each nonzero element of the matrix is assigned to a different thread in COO kernel, and segmented reduction operation is performed to compute sums for output vector.

We have proposed a JDS-based MxV algorithm in our previous work[9]. JDS is suitable for vector architectures. Unlike ELL, JDS allows varying number of nonzeros per row. The algorithm utilizes coalesced memory accesses, texture and constant caches for performance. Each JDS-stored matrix row is assigned to a different thread.

In this work, we are going to use the three kernels we have explained above: HYB, CSR vector and JDS-based kernels. Besides the ones that are mentioned above, several other works are devised for efficient MxV on GPUs. Blocked CSR (BCSR) is used in [5]. BCSR decreases number of memory fetches from the device

memory to some extent, however number of elements to be multiplied increases. A similar algorithm to CSR vector kernel of NVIDIA is reported in [3].

2.2 Mixed precision iterative refinement for Conjugate Gradients

Conjugate Gradient method is used to solve linear systems $\mathbf{Ax} = \mathbf{b}$, where matrix $\mathbf{A}$ is symmetric and positive definite. Solvers on GPUs that do not support double precision floating point operations suffer from loss of accuracy in the result. Therefore, in this work we adopt a mixed precision iterative refinement algorithm for CG[6] which is based on inner-outer iteration method[11]. The algorithm explained in[6] is tested on conventional processors, but reported to be applicable on GPUs, also. Authors report that the mixed precision algorithm achieves faster solution of the same or even better accuracy compared to the full double precision solver.

Basically, mixed precision algorithm runs the preconditioned CG. However, instead of using a fixed preconditioner, preconditioner is solved using a single precision sparse iterative method, in each iteration. Operations other than the inner solver run in double precision. Single precision inner solver may also use preconditioned CG method if a preconditioner is available or any other iterative method that result in symmetric and positive definite operations. Inner solver runs for a predetermined number of iterations and takes most of the time of the overall solution.

3. Implementation of CG solver on GPU

We implement the mixed precision algorithm summarized in Section 2.2. We implement CG for inner solver, assuming that we have no preconditioner readily available. Inner solver iterates for a fixed number of iterations. Core operations of single precision inner solver and double precision refinement operations run on the GPU. Refinement might be implemented on CPU with the cost of additional transfers between GPU and CPU for $\mathbf{r}$ and $\mathbf{z}$ vectors and slower outer solver performance. These additional costs are insignificant since there are only several outer loop iterations for thousands of inner solver iterations. In fact, outer solver should be implemented on CPU if older GPUs without double precision support are used. Mixed precision refinement requires storing both double and single precision entries for matrix and vectors. Hence, it is also useful to make refinement on CPU for reducing memory requirements of GPU.

MxV dominates running time of CG. Different MxV algorithms are proposed for GPUs, as summarized in Section 2.1. Matrix sparsity patterns, such as nonzero density and variance of number of nonzeros in rows/columns, greatly affect algorithms performance. As a result, different algorithms may be faster for different matrices, according to the matrix processed.

We compared NVIDIAs six MxV algorithms[4], whose implementations are publicly available, and our MxV algorithm[9] for 50 symmetric and positive definite matrices chosen from University of Florida Sparse Matrix Collection[20] on a Tesla GPU. For 28 matrices HYB algorithm is faster, for 16 matrices our JDS-based algorithm is faster, for 4 matrices CSR vector algorithm is faster and for 2 matrices ELL algorithm is faster. Algorithms performance difference is usually too big. The fastest algorithm for a matrix instance may be several times faster than the second fastest algorithm for the same matrix.

In the implementation of CG, we use the three fastest MxV kernels: NVIDIAs HYB and CSR vector, and our JDS-based kernels. Before CG iterations, the fastest within these three kernels for the problem is chosen to be the actual running MxV kernel during iterations. We run kernels several times before CG iterations and select the fastest. This selection cost is negligible since until solution usually there are thousands of inner loop iterations, each consists one MxV.

Not only MxV, but all operations of the inner CG solver other than scalar division and square root operations are efficiently implemented on the GPU. Note that division and square root operations are not standard compliant with IEEE 754 in CUDA. Dot product operation is implemented as in the parallel reduction example of NVIDIAs CUDA SDK[14]. For SAXPY operations $y \leftarrow y + ax$, where x and y are vectors and a is a scalar, each output element y_i is calculated by a different thread. We use L_2 norm for convergence, hence there is no need for an extra norm operation. Instead one square root operation is enough.

4. Parallel CG on Multi-GPU-enhanced cluster

We propose a mixed MPI/CPU-thread/GPU data-parallel algorithm on a GPU cluster, in which each node has multiple GPUs and at least one CPU. CUDA supports multiple GPUs run together for an application in a node. For each GPU

$\mathbf{CG}(\mathbf{b_{kl}}, \mathbf{x_{kl}}, \dots)$ //32 bit inner solver
$\mathbf{r^0_{kl}} \leftarrow \mathbf{b_{kl}}$
$\mathbf{p^0_{kl}} \leftarrow \mathbf{b_{kl}}$
$\mathbf{x^0_{kl}} \leftarrow \mathbf{0}$
$\gamma\_prev_{kl} \leftarrow \mathbf{DotProduct}(\mathbf{r^0_{kl}}, \mathbf{r^0_{kl}})$
for $i \leftarrow 1$ **to** iterLimit **do**
  $GPUSendsToHost(\mathbf{p^{i-1}_{kl}}, \mathbf{h\_p^{i-1}_{kl}})$
  if myCPUThreadId $= 0$ **then**
    $AsyncExchangeBtwNodes(\mathbf{h\_p^{i-1}_{kl}})$
  $\mathbf{q^{i-1}_{kl}} \leftarrow \mathbf{MxV}(\mathbf{A^{local}_{kl}}, \mathbf{p^{i-1}_{kl}})$
  $SynchronizeThreads(\ )$
  $GPUrecvsFromHost(\ \mathbf{p^{i-1}}, \mathbf{h\_p^{i-1}})$
  $\mathbf{q^{i-1}_{kl}} \leftarrow \mathbf{q^{i-1}_{kl}} + \mathbf{MxV}(\mathbf{A^{coupling}_{kl}}, \mathbf{p^{i-1}})$
  $\alpha_{kl} \leftarrow \mathbf{DotProduct}(\mathbf{p^{i-1}_{kl}}, \mathbf{q^{i-1}_{kl}})$
  $\gamma\_prev \leftarrow AllReduceSum(\gamma\_prev_{kl})$
  $\alpha \leftarrow \gamma\_prev / AllReduceSum(\alpha_{kl})$
  $\mathbf{x^i_{kl}} \leftarrow \mathbf{VectorSum}(\mathbf{x^{i-1}_{kl}}, \alpha\mathbf{p^{i-1}_{kl}})$
  $\mathbf{r^i_{kl}} \leftarrow \mathbf{VectorSum}(\mathbf{r^{i-1}_{kl}}, -\alpha\mathbf{q^{i-1}_{kl}})$
  $\gamma_{kl} \leftarrow \mathbf{DotProduct}(\mathbf{r^i_{kl}}, \mathbf{r^i_{kl}})$
  $\gamma \leftarrow AllReduceSum(\gamma_{kl})$
  $\beta \leftarrow \gamma / \gamma\_prev$
  $\gamma\_prev \leftarrow \gamma$
  $\mathbf{p^i_{kl}} \leftarrow \mathbf{VectorSum}(\mathbf{r^i_{kl}}, \beta\mathbf{p^{i-1}_{kl}})$
  $\sigma \leftarrow$ sqrt(γ) $//L_2$ norm of $\mathbf{r^i}$
  if $\sigma < \varepsilon$ **then**
    break
done

Fig. 1 Parallel CG algorithm for CPU thread l of node k. Outer loop is implemented in similar way and calls this method as the single precision solver.

in a node, CPU core(s) hold one thread and organizes communication required by that GPU. MPI calls between nodes are held by one thread from each node. We use non-blocking MPI calls, so that inter-node communications with GPU computations of communicating threads can be overlapped.

The parallel CG algorithm for node k and CPU thread l, which controls l^{th} GPU of that node (G_{kl}), is given in Fig. 1. It runs as the single precision inner solver of the mixed precision algorithm explained in Section 2.2. In Fig. 1, scalars are written in Greek letters, vectors in Latin letters and capital **A** denotes the iteration matrix. For scalars, subscript kl denotes the partial result computed in CPU thread l of node k. For matrix **A** and vector variables, subscript kl denotes the portion of the matrix or vector stored by GPU G_{kl}. Row-wise decomposition of **A** and vector partitionings are conformable, i.e., if row i of **A** is assigned to G_{kl}, then i^{th} entries of all vectors are also assigned to G_{kl}. Superscripts *local* and *coupling* over matrix **A** respectively denote submatrices consisting only columns that do not and do incur communication during MxV, i.e., columns corresponding to local input vector entries ($\mathbf{p_{kl}}$) and communicating input vector entries. Superscripts over vectors are iteration numbers. Function calls written with **bold** fonts are executed on GPU G_{kl}. Function calls with *italic* fonts denote communication. MxV, DotProduct and VectorSum respectively correspond to sparse matrix-vector multiplication, vector dot product and SAXPY operations, whose algorithmic details are explained in the previous section. Vector dot products are computed on GPUs and the scalar result is copied to the nodes host memory. *AllReduceSum* computes the global sum of the partial scalar results. Global sums are computed first by thread synchronization within each node, then by global MPI communications between nodes. Solver iterates for *iterLimit* number of iterations, unless residual converges. The optimal value of *iterLimit* is not known. As in[6], we choose it to be the number of iterations it took to do a fixed relative reduction for the residual in the first iteration.

Automatic selection of MxV kernel is done in a similar way with the sequential algorithm. Before execution of the CG solver, each CPU thread executes three MxV algorithms for its local and coupling submatrices -without communication- several times on its assigned GPU and chooses the fastest one as the MxV kernel during iterations. Hence, GPUs within a node may execute different MxV kernels,

according to the sparsity patterns of their assigned submatrices. Note that, selected MxV kernel does not affect the total volume of communication; since the submatrix and vector to be multiplied does not change, but the sparse storage format and multiplication algorithm change.

The communication on a GPU cluster is three-fold: communication between GPU cores within each GPU, between GPUs and host CPU(s) in a node, and between nodes. First type of communication is handled in kernels running on GPUs, whose details are given in previous section. Between GPUs and host CPUs, communication occurs for copying scalar results of dot products computed by GPUs to host memory. Also, GPUs communicate host CPUs for exchanging input vector ($\mathbf{p}$) entries of MxV. Host CPUs of each node hold an array bf h_p in main memory to coordinate the communication between GPUs. This array is also used for inter-node communication. Each GPU copies its $\mathbf{p}$ vector entries that are required for MxV by other GPUs of the cluster to the corresponding indices of its host $\mathbf{h\_p}$ vector. This operation is called $GPUsendsToHost$ in the figure. Once each GPU sends vector entries that are required by others, MPI communication between nodes are held. Threads with id 0 of each node are responsible for exchanging $\mathbf{p}$ vector entries among nodes. Threads with id 0 overlap computation of their local MxV on their assigned GPU with asynchronous inter-node communication. Before each GPU in a node receive $\mathbf{p}$ vector entries that they require for coupling MxV, threads in that node synchronizes to be sure that thread 0 is done with inter-node communication. After GPUs receive required vector entries ($GPUrecvsFromHost$) MxV can be executed for coupling submatrix of the sparse matrix bf A. The result is added to the output vector computed beforehand by MxV for local submatrix.

The algorithm explained in the figure runs for any conformable distribution of the matrix and vector variables to processors. For reduction of communication incurred by MxV and load balancing, we use hypergraph-partitioning-based decomposition[7].

5. Experimental results

We evaluate performance of proposed CG algorithm for multi-GPU clusters on TSUBAME supercomputer[17][16]. Each node of TSUBAME has 8 AMD 2.4 GHz Opteron dual core processors, 2 NVIDIA Tesla GPUs and 32 GB of main memory. We use 1 CPU (2 cores) from each node in our multi-GPU cluster experiments, where each core runs one thread and is responsible for controlling one GPU. Each Tesla GPU contains 30 streaming multiprocessors, each with 8 cores (240 cores in total) and 4 GB of device memory. Nodes are connected with high speed 4x SDR InfiniBand dual rail network, providing 20 Gbps bandwidth per node. Linux version 2.6.16 OS is installed on nodes. C++ programming language is used for coding. CUDA version 2.2 is used for programming GPUs, pthread library is used for CPU threading and Voltaire MPI is used for MPI communication between nodes. For partitioning hypergraphs, recursive partitioning tool PaToH[7] is used, which runs sequentially on the CPU. 11 unstructured sparse matrices that are symmetric and positive definite with real value entries from Sparse Matrix Collection of University of Florida[20] are used for performance evaluation.

Hypergraph partitioning is executed before CG solver as a preprocessing for efficient parallel multi-GPU solver. The preprocessing cost for hypergraph partitioning is amortized during CG iterations. The preprocessing cost for 8-way hypergraph partitioning is worth 60 to 310 sequential single precision CG iterations on the CPU for the matrices in our dataset.

Several refinement iterations in double precision on GPUs are executed for thousands of inner solver iterations. Hence CG solver performance is almost equal to the single precision inner solver performance. Solver performance in Gflops using hierarchical hypergraph-partitioning-based model is given in Fig. 2. In the figure, we compare sequential and parallel solver performances. "1 GPU" denotes sequential performance using single GPU and "$n \times 2GPUs$" denotes performance on n nodes, each having 2 GPUs. Matrices are sorted according to the number of nonzeros they contain on x axis. For each loop iteration, executing 1 MxV and 5 vector operations, $2 \times nnz + 10 \times n$ flops are counted, where nnz is the number of nonzeros and n is the dimension of the matrix. Gflops is the count of billions of flops per second.

As seen in the figure, up to 152 Gflops of CG performance is achieved. Bigger matrices better scale with the increase in number of nodes, because workloads become insufficient to utilize GPU resources for smaller matrices and communi-

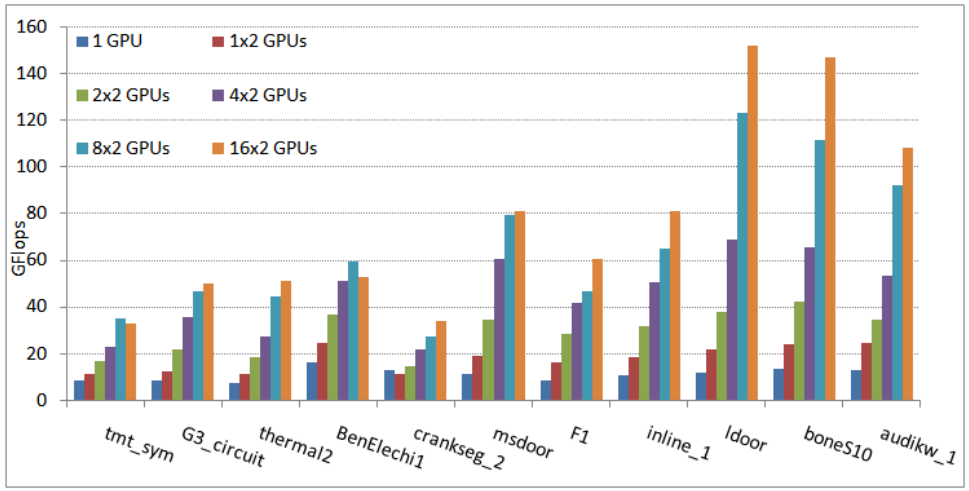

Fig. 2 CG Performance for different matrices and different number of nodes. Matrices are sorted in increasing number of nonzeros they contain.

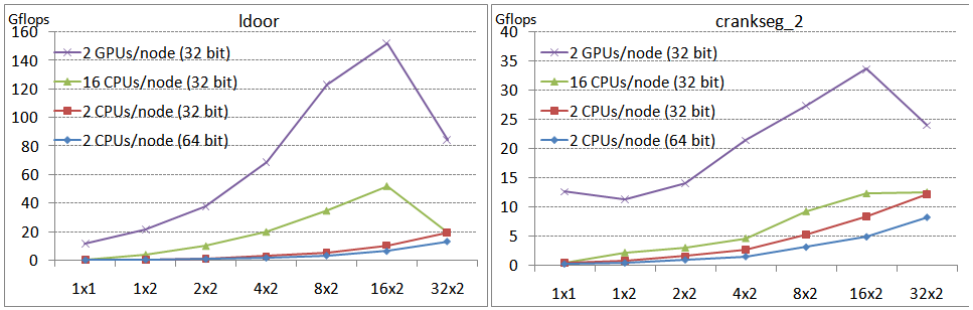

Fig. 3 CG performance on SMP and GPU clusters using hypergraph-partitioning-based decomposition for the fastest and slowest matrix instances.

cation time dominates. Reduction of communication time is the most important factor for performance. Without using hypergraph-partitioning-based models, we obtain almost no speedup. For ldoor matrix, which we obtain best CG performance; on 16 nodes only 26,859 words are communicated between nodes for input vector of MxV, where the dimension of this matrix is 952,203. Local and coupling MxV computation times (excluding communication time) cost 35% of total solution time for this instance. For crankseg_2, which reveals the worst performance, 117,127 words are communicated between 16 nodes, where the dimension of this matrix is 63,838. For this instance, total MxV computation time takes 25% of total solution time. Note that, crankseg_2 is our densest matrix and variation of number of nonzeros within rows for this matrix is very high, which are two main reasons for relatively higher communication volume incurred. For the smallest matrix in the dataset, tmt_sym, the ratio of MxV computation time to total time is 15% and for the biggest matrix, audikw_1, the ratio is 38%, on 16 nodes. These numbers confirm that as the matrix gets smaller with increasing number of GPUs, communication time dominates.

Chosen MxV kernel is another factor that affects the performance of the solver. We observed that for all 11 matrices, HYB is the fastest sequential MxV kernel. However, on 16 nodes, it is interesting to observe that none of the GPUs select HYB as its MxV kernel, for all matrices. For 8 matrices all 32 GPUs select JDS-based kernel, for 2 matrices all GPUs select CSR vector kernel and for inline_1 matrix 25 GPUs select JDS-based kernel and 7 GPUs select CSR vector algorithm. Hence, as demonstrated, HYB kernel is effective sequentially, but it is not desirable for row-wise parallelization. The reason for that is not the partitioning techniques that we have explained, but decomposition itself. During our experiments with a set of matrices that we do not report all of them in this section, we have found out that HYB performance is worse for smaller matrices.

Strong scaling comparisons on GPUs and CPUs shows that GPUs are superior to CPUs at their performance saturation points. We used largest symmetric and positive definite matrices from the University of Florida Sparse Matrix Collection; however sizes of the matrices are still insufficient to obtain speedups for 32 nodes over 16 nodes. Performance of the solver is bounded by the network communication between nodes for number of nodes beyond 16. We demonstrate this by comparing CG performance on multi-GPU cluster with SMP cluster on TSUBAME using exactly same network between nodes, while theoretical peak flops per node is completely different. Single precision peak performance of 16 CPU cores in a node is 153.6 Gflops ($2.4 \times 4 \times 2 \times 8$) and 2 GPUs in a node is 2073.6 Gflops ($1.44 \times 3 \times 240 \times 2$). Comparison results are given in Fig. 3.

For CPU implementation, pthread library is used for parallelization among cores within a node. As in the GPU case, only one thread from each node is responsible for non-blocking MPI communication. Communication is minimized using hierarchical partitioning as explained for multi-GPU clusters. Hence, using two cores per node, exactly same submatrices are assigned to cores, instead of GPUs, and total communication volume between cores is same with the multi-GPU cluster implementation. Regardless of the number of cores per node, inter-node communication volume is same, using hierarchical partitioning. We experiment double and single precision CG with 2 CPU cores per node. We also experiment single precision CG with 16 cores per node. Hence, we use up to 512 CPU cores from TSUBAME.

In Fig. 3, we only draw comparisons for two matrices in the border, but characters of the performance ratios for other matrices are almost same. We observe fall-offs for 16 cores per node on 32 nodes, just like the GPU case. Performance scales for 2 CPU cores per node, because the computation can be still considered as slow compared to network communication time. SMP cluster performance is always below of the multi-GPU cluster performance with the same underlying network. For all matrices in our dataset, average single precision CG performance is 21 Gflops for 512 cores and 46 Gflops for 64 GPUs.

For 16 nodes of GPU cluster, peak single precision performance is around 32 Tflops, where we achieve up to 152 Gflops, only 0.5% of the theoretical peak. For single GPU, we achieve up to 15.6 Gflops performance over 1Tflops theoretical peak, which is around 1.5% of peak. This is because performance of the sparse solver is bounded by the peak memory bandwidth, instead of peak flops. Actually, we achieve up to 87 GB/s effective memory bandwidth using texture cache memory on single GPU, whose theoretical device memory bandwidth is 102.4 GB/s. Assuming cache hit rate of 100%, we achieve device memory bandwidth up to 60 GB/s on single GPU. For 16 nodes and 32 GPUs, our peak effective memory bandwidth utilization is 838 GB/s using texture cache. Assuming 100% cache hit rate, we achieve up to 586 GB/s device memory bandwidth utilization, which is around 18% for theoretical bound of 3.3 TB/s.

3.8 Gflops of CG performance is reported using NAS Parallel Benchmark (NPB) with class B matrix[2] for a Myrinet interconnected 128 node Pentium 4 1.7 GHz cluster[21]. Using same number of cores (8 nodes $\times$ 16 cores), MPI-based NPB with class B matrix achieves 4.8 Gflops in TSUBAME. Former utilizes 0.87% of the peak flops rate and TSUBAME utilizes 0.78%. Using 32 nodes, each having 2 CPU cores, TSUBAME achieves 5.5 Gflops NPB CG performance, utilizing 1.8% of peak flop performance. The reason for TSUBAME to achieve less performance with 128 cores is the large communication volume incurred by randomly generated class B matrix, although the computation density of class B matrix is very high per row compared to the ones in our dataset. On the other hand, on 32×2 cores our CPU implementation with reduction of communication by hypergraph-partitioning-based models achieves average of 11.6 Gflops double precision performance for all 11 matrices in our dataset, utilizing 3.8% of peak flops.

6. Conclusion

We have explained a scalable implementation of a mixed precision Conjugate Gradient solver for unstructured problems on a cluster, in which each node of the cluster contains multiple GPUs with double precision support. Although we have explained proposed techniques on a CG solver, they can be easily adopted to other sparse iterative solvers. We have integrated several fast kernels for basic operations of CG. CPU-GPU organization and intra- and inter-node communications are successfully handled in the parallel algorithm. Network communication is bottleneck in performance of parallel sparse solvers on traditional clusters. Interconnection between compute nodes of a GPU cluster -compared to fast GPU computation units- is even much slower for obtaining scalability. However, we have shown that scalability can be still obtained by application of techniques that are traditionally used for distributed memory systems. We have adopted state-of-the-art 1D matrix decomposition models based on hypergraph partitioning for communication reduction and load balancing among GPUs. We have shown strong scalability up to 16 nodes, each node having 2 Tesla GPUs, using well-known unstructured matrices in our dataset. We have also shown that SMP cluster implementations could not catch the performance of the multi-GPU cluster performance, with exactly same communication network.

Hypergraph partitioning is executed as a preprocessing to CG solver. There

are publicly available, effective and efficient partitioning tools for hypergraphs, but they are implemented on CPUs. Peak performance of GPUs is much higher than that of CPUs. Hence, preprocessing incurred by partitioning might be more severe for applications running on GPU - although in our case, preprocessing cost is amortized since there are many CG iterations. There are publicly available parallel partitioning tools which might be an alternative to reduce preprocessing time. Actually, it is required to use a parallel partitioning tool for bigger matrices, in case of insufficient single-node memory. Still, there is a gap in performance scales of GPU computing and parallel partitioning tools that are developed for CPUs.

References

1) Advanced Micro Devices, Inc.: *ATI CTM Guide Technical Reference Manual*, (2006).
2) Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V. and Weerantunga, S.: The NAS Parallel Benchmarks, *RNR Technical Report RNR-94-007* (1994).
3) Baskaran, M.M. and Bordawekar, R.: Optimizing Sparse Matrix-Vector Multiplication on GPUs, *IBM Research Report, RC24704* (2008).
4) Bell, N. and Garland, M.: Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors, *Proc. SC '09: ACM/IEEE Conference on Supercomputing*, Portland, OR, USA (2009).
5) Buatois, L., Caumon, G. and Lévy, B.: Concurrent Number Cruncher: An Efficient Linear Solver on the GPU, *Proc. HPCC 2007, LNCS 4782*, pp. 358–371 (2007).
6) Buttari, A., Dongarra, J., Kurzak, J., Luszczek, P. and Tomov, S.: Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy, *ACM Transactions on Mathematical Software*, Vol.34, No.4 (2008).
7) Catalyurek, U.V. and Aykanat, C.: Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication, *IEEE Transactions on Parallel and Distributed Systems*, Vol.10, No.7, pp. 673–693 (1999).
8) Che, S., Li, J., Sheaffer, J.W., Skadron, K. and Lach J.: Accelerating Compute Intensive Applications with GPUs and FPGAs, *Proc. IEEE Symposium on Application Specific Processors (SASP)* (2008).
9) Cevahir, A., Nukada, A. and Matsuoka, S.: Fast Conjugate Gradients with Multiple GPUs, *Lecture Notes in Computer Scienc*, Vol.5544, Springer, pp. 898–903 (2009).
10) Fan, Z., Qiu, F., Kaufman, A. and Stover, S.Y.: GPU Cluster for High Performance Computing, *Proc. SC04: ACM/IEEE Conference on Supercomputing* (2004).
11) Golub, G.H. and Ye, Q.: Inexact Preconditioned Conjugate Gradient Method with Inner-Outer Iterations, *SIAM Journal on Scientific Computing*, Vol.21, No.4, pp.1305–310 (2000).
12) Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S.H.M., Grajewski, M. and Turek, S.: Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, *Parallel Computing*, Vol.33, No.10–11, pp. 685–699 (2007).
13) GraphStream Inc.: *GraphStream Scalable Computing Platforms*, `http://www.graphstream.com` (*accessed* 2009).
14) Harris, M.: Optimizing Parallel Reduction in CUDA, NVIDIA Developer Technology (2007).
15) Hartley, T.D.R., Catalyurek, U.V., Ruiz, A., Ujaldon, M., Igual, F. amd Mayo, R.: Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores, *Proc. 22nd ACM International Conference on Supercomputing*, pp.15–25 (2008).
16) Matsuoka, S.: The Road to TSUBAME and Beyond, *Petascale Computing: Algorithms and Applications*, Chapman & Hall Crc Computational Science Series, pp. 289–310 (2008).
17) Matsuoka, S., Aoki, T., Endo, T., Nukada, A., Kato, T. and Hasegawa, A.: GPU-accelerated Computing – From Hype to Mainstream, the Rebirth of Vector Computing, *J. Physics: Conference Series 180*, (2009).
18) NVIDIA Corporation: *NVIDIA CUDA Compute Unified Device Architecture Programming Guide* (2007).
19) Saad, Y.: SPARSKIT: A Basic Tool Kit for Sparse Matrix Computation, *Tech. Rep. CSRD TR 1029*, University of Illionis, Urbana, IL (1990).
20) University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices/`.
21) Yi, H., Hong, J., Park, H. and Lee, S.: Scalability of a Tera-Scale Linux-Based Clusters for Parallel ab initio Molecular Dynamics Applications, *Proc. Third LCI Conference* (2002).