Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2012) 1–10
Procedia Computer Science

International Conference on Computational Science, ICCS 2012

An MPI-CUDA Implementation and Optimization for Parallel Sparse Equations and Least Squares (LSQR) [1]

He Huang[a,*], Liqiang Wang[a], En-Jui Lee[b], Po Chen[b]

[a]*Department of Computer Science, University of Wyoming, Laramie, WY 82071, USA.*
[b]*Department of Geology and Geophysics, University of Wyoming, Laramie, WY 82071, USA.*

Abstract

LSQR (Sparse Equations and Least Squares) is a widely used Krylov subspace method to solve large-scale linear systems in seismic tomography. This paper presents a parallel MPI-CUDA implementation for LSQR solver. On CUDA level, our contributions include: (1) utilize CUBLAS and CUSPARSE to compute major steps in LSQR; (2) optimize memory copy between host memory and device memory; (3) develop a CUDA kernel to perform transpose SpMV without transposing the matrix in memory or preserving additional copy. On MPI level, our contributions include: (1) decompose both matrix and vector to increase parallelism; (2) design a static load balancing strategy.

In our experiment, the single GPU code achieves up to 17.6x speedup with 15.7 GFlops in single precision and 15.2x speedup with 12.0 GFlops in double precision compared with the original serial CPU code. The MPI-GPU code achieves up to 3.7x speedup with 268 GFlops in single precision and 3.8x speedup with 223 GFlops in double precision on 135 MPI tasks compared with the corresponding MPI-CPU code. The MPI-GPU code scales on both strong and weak scaling tests. In addition, our parallel implementations have better performance than the LSQR subroutine in PETSc library.

Keywords:
Parallel Scientific Computing, LSQR, MPI, GPU, CUDA, CUSPARSE, CUBLAS, Seismic Tomography, Geoscience

1. Introduction

Sparse Equations and Least Squares (LSQR), proposed by Paige and Sanders [1], is a numerical method for solving linear equation problems in an iterative way. As one of the most widely used inversion methods in seismic tomography, LSQR method is highly efficient on solving different types of linear systems (*e.g.* overdetermined, underdertermined, or both) for large linearized inverse problems [2]. As LSQR method is based on conjugate gradient method, the estimated solution usually converges fast. In general, LSQR algorithm is well suited to tomographic inversion problems that usually involve sparse matrices during inversion.

[1]The work was supported in part by NSF under Grants 0941735 and NSF CAREER 1054834, and by the Graduate Assistantship of the School of Energy Resources at the University of Wyoming. This research used resources of the Keeneland Computing Facility supported by the NSF under Contract OCI-0910735. Thanks to Galen Arnold of NCSA and Dr. John Dennis of NCAR for their insightful suggestions.
[*]Corresponding author. Email address: hhuang1@uwyo.edu

It is very challenging to apply LSQR to real-world problems in seismic tomography. First, the problem size can be huge. The matrix, which represents the coefficients of the equations, is usually very large. In a typical real-world application, the number of rows in the matrix can be hundreds of millions, the number of columns can be tens of millions, and the matrix may be hundreds of gigabytes even in compressed format. Such huge data cannot be computed without parallelism because of both execution time and memory limits. Another issue is that the nonzero elements in the matrix can be very sparse and their distribution is highly uneven. The number of nonzero elements in different rows can range from tens of thousands to several. Therefore, a naive data decomposition strategy will not work well.

To address the above challenges, we design and implement a parallel LSQR implementation using MPI and CUDA. To the best of our knowledge, our MPI-CUDA code is the first implementation of LSQR with multiple GPUs for large-scale dataset in seismology. Our major contributions include:

- To accelerate LSQR, we use CUBLAS to accelerate vector operations, and use CUSPARSE to accelerate sparse matrix vector multiplication (SpMV), which is the most compute-intensive part of LSQR. However, CUSPARSE is efficient only on handling regular SpMV in Compressed Sparse Row (CSR) format, but inefficient on SpMV with matrix transpose. We design two approaches to handle transpose SpMV with trade-off on memory versus performance. The first approach utilizes a different matrix format, *i.e.*, Compressed Sparse Column (CSC) for transpose SpMV. Although its performance is much better than using CUSPARSE directly, it requires storing two copies of the matrix. As an alternative way, we design the second approach to support both regular and transpose SpMV and avoid storing additional matrix transpose. It has the almost same performance as the first approach on NVIDIA C2050 GPU, but is slower on NVIDIA M2070.

- To optimize memory copy between host memory and GPU device memory, we utilize a "register-copy" technique to successfully speed up the performance of copy between host memory and device memory by 20%. In addition, we minimize CPU operations by porting all matrix and vector based operations into GPU. During computation, the intermediate results reside on device memory, and only a small amount of data is copied between host and device memories for MPI communication.

- To increase parallelism, we decompose both matrix and vector. To obtain good load balance, we decompose the matrix in row-wise order and distribute rows according to the number of nonzero elements. We use MPI-IO to allow multiple MPI tasks to load data simultaneously.

2. Related Work

In our implementation, matrix is stored in Compressed Sparse Row (CSR) format, which only preserves values of nonzero elements as well as their positions. CSR is one of the most popular formats used for SpMV due to its space-efficiency and data alignment. The CSR format consists of three arrays: *ptr*, *indices*, and *data*. The integer array *ptr* stores row pointers to the offset of each row. The integer array *indices* stores the column indices of the nonzeros. The array *data* stores the values of nonzeros. CSR is not sensitive to the distribution of nonzeros. It is efficient when being accessed in row-wise order. Our parallel scheme utilizes this feature, so we partition matrix based on row. We also use Compressed Sparse Column (CSC) to store matrix transpose. CSC follows the same principle as CSR except that it stores matrix by column. It also has three arrays: *ptr* storing the offset of each column, *indices* storing row indices, and *data* storing nonzero values.

SpMV is the most compute-intensive part in LSQR. Different GPU-based approaches have been investigated to speed it up. Bolz *et al.* [3] first applied GPU computing to SpMV. Baskaran and Bordawekar [4] implemented a few optimized SpMV kernels in CUDA. Bell and Garland [5] implemented SpMV kernel in CUDA for different sparse matrix formats, including DIA, ELLPACK, CSR, COO, and a hybrid ELL/COO format. Choi *et al.* [6] proposed and implemented a blocked ELLPACK (BELLPACK) storage formats and a performance model. In [7], we designed a novel framework to partition sparse matrix and store in different formats based on the fact that different storage formats of sparse matrix can significantly affect the performance of SpMV.

There are several CPU-based implementations of parallelized LSQR. Baur and Austen [8] presented a parallel implementation of LSQR by means of repeated vector-vector operations. PETSc [9] also contains an optimized

implementation. Liu *et. al* [10] proposed a parallel LSQR algorithm for seismic tomography. They decompose the matrix by row. Their approach computes SpMV and tranpose SpMV in parallel based on distributed storage, and requires reduction on two vectors in each iteration. This approach limits the scalability and involves much more communication if the two vectors are very large.

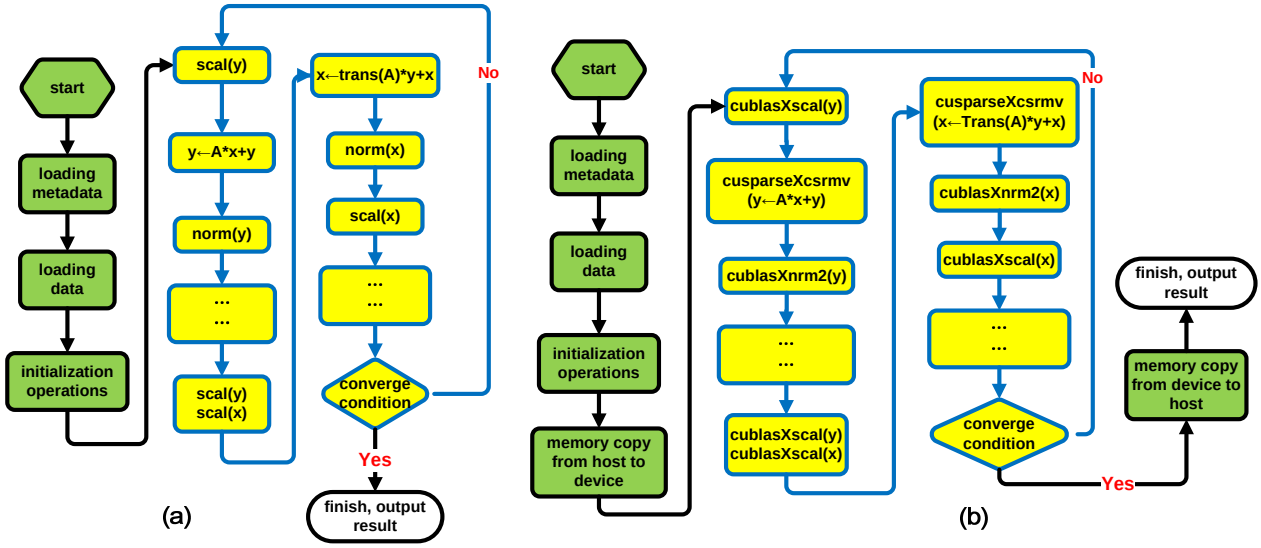

Figure 1: (a) The workflow of LSQR in CPU version; (b) the workflow of LSQR in GPU version, where the green steps are executed on CPU and the yellow steps are executed on GPU.

3. Introduction to LSQR

LSQR algorithm aims to solve linear systems like $A \times x = b$ in an iterative way, where A is a sparse matrix for coefficients of a linear systems, b is a constant vector, and x is the variable vector to be solved.

Figure 1 (a) shows the workflow of LSQR algorithm. Here x and y are intermediate vectors during computation. x's initial value is zeros, y's initial value is derived from vector b. The green procedures denote the algorithm's initialization steps including loading the metadata and real values and initializing variables. The yellow procedures are the major iterative steps including:

1. Scale operation on vector y.
2. An SpMV and a vector addition ($y \leftarrow y + A \times x$).
3. Norm of vector y.
4. Scale operations on vector y and vector x based on y's norm value.
5. Another SpMV and a vector addition ($x \leftarrow x + A^T \times y$). Note that the matrix is in its transpose format.
6. Norm and scale operations on vector x.
7. Check convergence condition. If satisfied, the algorithm terminates, x's derived value is the final solution of the linear systems; otherwise the algorithm continues the next iteration. The whole execution may converge till after hundreds or thousands of iterations, depending on matrix A.

The most compute-intensive portions are $y \leftarrow y + A \times x$ and $x \leftarrow x + A^T \times y$, which usually take up more than 70% of the execution time. Both norm and scale operations totally take up about 20% in the whole execution time. In this paper, we focus on parallelizing and optimizing these four components.

Table 1: Summary of our CPU and four GPU implementations for LSQR. The third and fourth columns indicate "kernel name (input matrix format)". "csrmv" is the subroutine name in CUSPARSE, which means matrix vector multiplication in CSR format.

Implementation	Matrix storage format	$y \leftarrow y + A \times x$ kernel	$x \leftarrow x + A^T \times y$ kernel
CPU	one copy of CSR	our own csrmv (CSR)	our own csrmv trans (CSR)
GPU-I	one copy of CSR	CUSPARSE csrmv (CSR)	CUSPARSE csrmv trans (CSR)
GPU-II	one copy of CSC	CUSPARSE csrmv trans (CSC)	CUSPARSE csrmv (CSC)
GPU-III	one copy of CSR, one copy of CSC	CUSPARSE csrmv (CSR)	CUSPARSE csrmv (CSC)
GPU-IV	one copy of CSR	CUSPARSE csrmv (CSR)	our own csrmv trans (CSR)

4. Single GPU Approach

Figure 1 (b) shows the workflow of LSQR GPU version. All iterative steps in yellow are ported to GPU. The initialization steps are exactly the same as the CPU version. After initialization, both matrix and vectors are copied to device memory from host memory. Compared with Figure 1 (a), the scale and norm operations in Figure 1 (b) are computed using CUBLAS; and SpMVs, *i.e.*, $y \leftarrow y + A \times x$ and $x \leftarrow x + A^T \times y$, are computed using CUSPARSE. CUBLAS also contains subroutines to perform matrix vector multiplication, but CUSPARSE is specially optimized for sparse matrix. In our implementation, the matrix and vector reside in device memory during computation in order to avoid frequent memory copy between host and device. After the algorithm converges, the solution vector is copied back to host memory from device.

4.1. Selecting the Best GPU Kernel for SpMV

SpMV is the most time-consuming and memory-consuming part in LSQR. We investigate several GPU implementations of SpMV for $y \leftarrow y + A \times x$ and $x \leftarrow x + A^T \times y$ and compare their performance in our application. Table 1 summarizes our CPU implementation and four types of GPU implementations of LSQR using different SpMV kernels. Figure 2 compares the performance of these implementations.

4.1.1. GPU Implementation I (GPU-I)

The CUSPARSE library provides subroutines for SpMV. We use `cusparseXcsrmv (cusparseHandle_t handle, cusparseOperation_t transA, ...)` to calculate $y \leftarrow y + A \times x$ and $x \leftarrow x + A^T \times y$, where X could be S for single, D for double, C for complex, or Z for double complex. Here A must be stored in CSR format. `transA` is set to `CUSPARSE_OPERATION_NON_TRANSPOSE` for $A \times x$ and `CUSPARSE_OPERATION_TRANSPOSE` for $A^T \times y$. GPU-I is a straight forward approach. From Figure 2 , we can see that the GPU-I is even slower than the CPU serial implementation on $x \leftarrow x + A^T \times y$, because $A^T \times y$ requires additional time and space to transpose the matrix. However, the performance of $y \leftarrow y + A \times x$ is improved dramatically compared with the CPU serial implementation.

Implementation	$y \leftarrow y + A \times x$	$x \leftarrow x + A^T \times y$
CPU	60.766	75.567
GPU-I	6.008	121.289
GPU-II	553.721	11.285
GPU-III	5.907	10.949
GPU-IV	5.956	10.415

Figure 2: Execution time comparison of the four single-GPU implementations with the CPU serial implementation in single precision on NVIDIA Tesla C2050 (Fermi). The execution is based on a sample dataset and measured in seconds.

4.1.2. GPU Implementation II (GPU-II)

The CUSPARSE library supports matrix format conversion between different storage formats, *e.g.*, CSR to CSC, and CSR to Coordinate Storage (COO). We utilize this feature to convert matrix from CSR to CSC format. Note that, we need to do conversion only once at the first iteration, then the CSC matrix is stored in GPU memory for future iterations because matrix A keeps unchanged during iterations. As we introduced in Section 2, CSR is a row-wise compressed format, while CSC is similar but in column-wise. In the GPU-II code, we first convert the original matrix

in CSR to CSC format. Then a CSC matrix can be treated as a CSR matrix, *i.e.*, refer column pointer of CSC as row pointer of CSR, and the row indices of CSC as column indices of CSR. Therefore, the converted CSC matrix is equivalent to the transpose of original matrix in CSR format. Hence, we can assign `CUSPARSE_OPERATION_NON_TRANSPOSE` to `cusparseXcsrmv()` to perform $A^T \times y$, and assign `CUSPARSE_OPERATION_TRANSPOSE` to perform $A \times x$.

Compared with the serial CPU implementation, the performance of the GPU-II code is slower on $y \leftarrow y + A \times x$, due to transpose inside the function, but faster on $x \leftarrow x + A^T \times y$ because no transpose is required in this case.

4.1.3. GPU Implementation III (GPU-III)

Both GPU-I and GPU-II have drawbacks. GPU-I requires additional time and space to transpose the matrix in CSR format in order to perform $A^T \times y$. GPU-II requires additional time and space to transpose the matrix in CSC format in order to perform $A \times x$.

GPU-III is the combination of GPU-I and GPU-II that avoids the drawbacks we mentioned above. Specifically, we store two copies of matrix A in device memory, one is in CSR format, and the other is in CSC format. We use CUSPARSE `cusparseXcsrmv()` in CSR format to compute $y \leftarrow y+A\times x$, and CSC format to compute $x \leftarrow x+A^T \times y$. As Figure 2 indicates, GPU-III combines the advantages of GPU-I on $y \leftarrow y + A \times x$ and GPU-II on $x \leftarrow x + A^T \times y$. So its overall performance is much better than the serial CPU code. However, this implementation requires to store two copies of A, *i.e.*, one copy in CSR and the other copy in CSC.

4.1.4. GPU Implementation IV (GPU-IV): Avoiding Matrix Transpose (Optimization-1)

Because matrix A could be huge, its memory consumption dramatically limits the problem size while two copies of matrix are stored in memory.

We can calculate $A^T \times y$ in another way without actually transposing the matrix. In the matrix transpose, every element in a column of the transposed matrix multiplies the corresponding element in the vector. For example, elements in the first column of the matrix transpose multiply the first element in the vector and produce an intermediate vector, and so on. Finally we sum up all the intermediate vectors and yield the same result as the traditional way.

```
__global__ void spmvGPUCSRTrans(int numRow, int ptr[], int idx[],
                                 float val[], float y[], float x[]){
  // global thread index
  int thread_id = BLOCK_SIZE * blockIdx.x + threadIdx.x;
  // thread index within the warp
  int thread_lane = threadIdx.x & (WARP_SIZE-1);
  // global warp index
  int warp_id = thread_id / WARP_SIZE;
  // total number of active warps
  int num_warps = (BLOCK_SIZE / WARP_SIZE) * gridDim.x;
  for( row=warp_id; row < numRow;  row+=num_warps){
        int row_start = ptr[row];
        int row_end = ptr[row+1];
        for (i=row_start+thread_lane; i < row_end;i+=WARP_SIZE)
              atomicAdd(x+idx[i], val[i] * y[row]);
  }
}
```

Figure 3: Transpose SpMV GPU implementation that avoids matrix transpose in CSR format.

For such an approach to avoid transpose, its CPU implementation is simple. However, its GPU implementation is challenging. Our CUDA kernel implementation is shown in Figure 3, which is an extension of [5]. Our code uses one warp to calculate the whole row of the matrix, so memory accesses are coalesced. Specifically, a row is processed using the whole warp (32 GPU threads) in parallel. All nonzero elements in a row are multiplied with the same element in the vector (*i.e.*, `y[row]`). Each result is added to the corresponding element in vector x. The memory accesses in such a parallel multiplication are coalesced because consecutive threads access consecutive elements. The addition to x cannot be coalesced because the result elements from the multiplication could be inconsecutive. An important technique in this CUDA kernel is the usage of "`atomicAdd`". Without it, different GPU threads may add values to the same element of x simultaneously, which incurs a race condition. "`atomicAdd`" makes the add operation (including read and write) uninterrupted, thus race conditions can be avoided. CUDA 4.0 inherently supports "`atomicAdd`" in single precision for GPU devices with compute capability 2.x. but not for double precision. We implemented it using "`atomicCAS`" for double precision. Figure 2 indicates that this approach has similar performance with the GPU-III on NVIDIA Fermi C2050. However, on NVIDIA Fermi M2070, GPU-IV is slower than GPU-III because of atomic operations in

$x \leftarrow x + A^T \times y$. In that situation, we have two options: the GPU-IV code is more memory-efficient, the GPU-III code is more computation-efficient.

4.2. Accelerating Memory Copy (Optimization-2)

Memory copy between host and device is very expensive. We investigated several techniques to improve it.

Since CUDA 2.2, CUDA allows host memory to be mapped into device memory, also called pinned memory, using "`cudaHostAlloc()`" and "`cudaHostFree()`" to allocate and free page-locked host memory. It eliminates the need to copy data from host to device memory. CUDA 4.0 introduces "`cudaHostRegister()`" and "`cudaHostUnregister()`" that can register or unregister memory allocated by "`malloc()`" in host memory. Hence GPU kernel can access registered host memory directly. In the above two zero-copy mechanisms, the CUDA kernel operates directly on the host memory, which slows down the kernel. The reason is that the PCI express bus that bridges between host and device memory has lower bandwidth and higher latency than device memory. Since our CUDA kernels frequently access the mapped host memory and cannot tolerate long latencies, these approaches are not suitable for our application.

We use a "register-copy" approach to speed up memory copy. We first allocate the array in host memory, and register the allocated memory using cudaHostRegister(). Then we use cudaMemcpy() to copy between host memory and device memory. This is based on the observation that "`cudaHostRegister()`" speeds up memory copy between host and device. For example, in our experiment, "register-copy" reduces memory copy from 10.878s to 8.391s, which is more than 20% improvement. This is because GPU can directly access registered memory, hence has much higher bandwidth than unregistered memory.

5. Multiple GPUs (MPI-CUDA) Approach

Our MPI-CUDA implementation of LSQR is based on the hybrid MPI and CUDA programming model. It works as follows.

Based on the metadata, a load balancing strategy is used to make data evenly distributed among MPI tasks. Each MPI task loads its own piece of data independently and simultaneously. Then each MPI task copies its data from host memory to GPU's device memory.

During computation, SpMV and vector-based operations are executed on GPUs in parallel. At the end of each iteration, an MPI collective communication `MPI_Allreduce` is called to obtain a complete copy of vector x. In order to perform `MPI_Allreduce` among multiple GPUs, the partial values of x located on each individual GPU are copied back to host memory from device memory. After `MPI_Allreduce` is done, the reduced result x is copied back to device memory. The decomposed matrix A and vector y always stay in device memory.

5.1. Data Layout and Decomposition

Before discussing the decomposition method, we first introduce the characteristic of seismic data. The upper part of sparse matrix A is kernel component and the lower part is damping & smoothing component (called "damping" for short in the rest of the paper). The kernel component of matrix A is stored in row-wise. The vast majority of nonzeros of the matrix are located at kernel component. Every row of kernel is stored in a separated file with its name indicating its row index. Actually, the order of rows of the matrix does not affect computation result as long as the order of elements in vector b is adjusted accordingly. The data file for a row only stores the values of nonzeros and column indices. The damping component usually has much more rows than the kernel and is much sparser with only one to four nonzeros per row. Every damping file stores multiple rows of damping data. When loading the matrix, the damping data is appended below the kernel data.

Our decomposition method is based on MPI programming model and the layout of seismic data. As we mentioned before, LSQR mainly conducts two operations $x \leftarrow x + A^T \times y$ and $y \leftarrow y + A \times x$ in an iterative way. Every MPI task keeps a portion of matrix A, a portion of vector y, and a complete copy of vector x. We decompose only vector y because the size of vector y is much larger than that of vector x.

5.2. Static Load Balancing Strategy (Optimization-3)

To obtain good load balance, we evenly partition and distribute the data according to the number of nonzeros in every MPI task. When LSQR starts, the master MPI task (rank=0) first loads the metadata that contains a list of names of kernel row files and their number of nonzeros sorted in descending order. The master MPI task is responsible for sending the file names of kernel dataset to the appropriate compute MPI tasks. We maintain a counter for every MPI task to keep the current total number of nonzeros already allocated to that MPI task. Then we search all MPI tasks to find the one with the least load according to the number of nonzeros. Thus, at the end of allocating, every compute MPI task has similar amount of nonzero elements. Once receiving kernel data file names, the individual MPI task loads its assigned kernel rows independently and simultaneously using MPI-IO.

Because the damping data is evenly divided into a series of files according to the number of MPI tasks. These files are assigned to the MPI tasks according to MPI rank number. The partitions of kernel and damping data are all stored in CSR format in memory.

As the row order of kernel data has been changed for load balancing, the order of corresponding elements in vector b also needs to be adjusted accordingly. We distribute partitions of vector b according to the allocation of their corresponding kernel data partition.

5.3. Parallel Computation (Optimization-4)

Our parallelized computation is based on data decomposition, where the communication between MPI tasks is just one `MPI_Allreduce` during each iteration. The most time-consuming operations in every iteration are SpMV, *i.e.*, $y \leftarrow y + A \times x$ and $x \leftarrow x + A^T \times y$, which are computed in parallel.

During calculating $y \leftarrow y + A \times x$, every MPI task computes the multiplication of a portion of matrix A and a complete vector x and yields a portion of vector y. Then it adds the new value to the previous portion of vector y. All of these operations are performed on GPU.

Calculating $x \leftarrow x + A^T \times y$ involves multiplying a portion of matrix transpose of A with a portion of vector y in GPU. Each MPI task computes $x \leftarrow x + A^T \times y$ simultaneously. This calculation works as follows.

1. Temporal vector x' is copied to host memory from device memory as shown in Figure 4 (a).
2. `MPI_Allreduce` is performed to sum all vectors x' over different MPI tasks and broadcast the reduced vector x'' to all MPI tasks as shown in Figure 4 (a) and (b).
3. The reduced vector x'' is copied to device memory from host memory, as shown in Figure 4 (b).
4. The reduced vector x'' is added to the previous vector x in GPU.

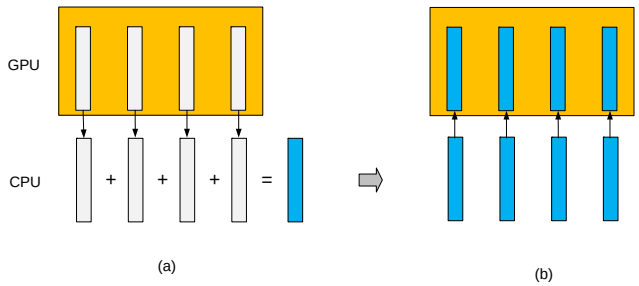

Figure 4: Reduction on the result of $A^T \times y$. A vertical bar represents a vector x copy in an MPI task. (a) Vector x' is copied from device (GPU) memory to host (CPU) memory, and reduction is performed across network. (b) After reduction, the reduced vector x'' is copied from host memory to device memory.

5.3.1. Vector Based Operation

The vector based operations are scale and norm. When performing scale operation on x or y, the scale value is sent to every MPI task for local computation on GPU without synchronization. When performing norm operation on y, every MPI task computes a partial norm value based on its local part of vector y, and the master MPI task is responsible for collecting partial norm value from all MPI tasks and computing the overall norm value. Thus, we eliminate reconstructing a complete vector y in local MPI task and avoid memory copy between host and device.

6. Experimental Result

Our experiments are conducted at Keeneland [11] CPU-GPU cluster at National Institute for Computational Sciences (NICS). Keeneland is equipped with 2.8 GHz Intel Westmere hex-core CPUs, NVIDIA 6GB Fermi M2070 GPUs, and Qlogic QDR InfiniBand switches. The softwares we use are OpenMPI 1.4.3, CUDA 4.0 RC2 and PETSc 3.1.

6.1. Single GPU Experiments

Table 2: The four seismic datasets for experiments.

data name	#row	#column	#nonzero	nonzero ratio (%)
China CN200801ker	2000	444346	90,076,905	10.14
China CN2008ker	9000	444346	165,768,180	4.15
S CA DEC5	1,914,215	302,940	53,909,209	9.2×10^{-3}
S CA DEC3	8,881,087	1,351,680	210,709,885	1.755×10^{-3}

We test our serial CPU code and two single GPU codes (GPU-III and GPU-IV) on four real-world seismic datasets in single precision and double precision. The datasets are listed in Table 2. The first two China datasets are denser than the two Southern California datasets. To simplify the performance comparison, the number of iterations is fixed to 100 for all tests.

Table 3: The execution time (in seconds) for datasets in single precision (SP). CPU's execution time is set as the baseline. "≈" indicates that GPU-III has almost the same performance with GPU-IV. "x" indicates that the memory is not enough to load the dataset. The speedup is calculated by $T_{cpu} \div T_{gpu}$, where T_{cpu} is the execution time on CPU and T_{gpu} is the execution time on GPU.

data	function	# called	CPU(SP)	GPU-IV(SP)	Speedup	GPU-III(SP)	Speedup
CN 200801ker	LSQR main iteration	1	38.617	3.726	10.4	2.369	16.30
	norm	200	0.171	0.022	7.8	≈	≈
	scal	400	0.088	0.002	44.0	≈	≈
	$y \leftarrow y + A \times x$	100	17.54	1.08	16.2	≈	≈
	$x \leftarrow x + A^T \times y$	100	21.037	2.85	7.4	1.533	13.72
CN 2008ker	LSQR main iteration	1	74.964	7.008	10.7	4.27	17.56
	norm	200	0.177	0.022	8.0	≈	≈
	scal	400	0.093	0.002	46.5	≈	≈
	$y \leftarrow y + A \times x$	100	34.541	2.133	16.2	≈	≈
	$x \leftarrow x + A^T \times y$	100	40.567	5.29	7.7	2.546	15.93
S CA DEC5	LSQR main iteration	1	26.786	5.798	4.6	2.795	9.58
	norm	200	1.079	0.032	33.7	≈	≈
	scal	400	0.461	0.002	230.5	≈	≈
	$y \leftarrow y + A \times x$	100	11.863	1.913	6.2	≈	≈
	$x \leftarrow x + A^T \times y$	100	13.496	3.97	3.4	0.94	14.36
S CA DEC3	LSQR main iteration	1	106.461	24.357	4.4	x	x
	norm	200	4.934	0.076	64.9	x	x
	scal	400	2.245	0.002	1122.5	x	x
	$y \leftarrow y + A \times x$	100	46.875	9.079	5.2	x	x
	$x \leftarrow x + A^T \times y$	100	52.788	15.63	3.4	x	x

Tables 3 shows the execution time of major parts of the code in single precision (SP). In single precision, we achieve 4x to 10x speedup with 3.7 to 9.9 GFlops in the transpose-free GPU-IV code, and 9x to 17x speedup with 8.2

to 15.7 GFlops in the GPU-III code on single GPU compared to the serial CPU code. We also do the same experiment on double precision (DP), which is not shown in this paper because of page limit. In double precision, we achieve 2.4x to 3.0x speedup with 1.7 GFlops to 2.4 GFlops in the transpose-free GPU-IV code, and 9.1x to 15.2x speedup with 6.5 to 12.0 GFlops in the GPU-III code on single GPU compared to the serial CPU code. Table 4 summarizes the overall performance in single and double precision.

The GPU-III code has better performance than the GPU-IV on single GPU for both single and double precisions on M2070 GPU because `atomicAdd` is involved in the GPU-IV. Note that they have the almost same performance on C2050 GPU, as shown in Figure 2. The performance degradation of GPU-IV from single precision to double precision is more obvious than that of GPU-III. For example, as shown in the second row of Table 4, GPU-IV degrades from 9.915 GFlops (SP) to 2.36 GFlops (DP) while GPU-III degrades from 15.488 GFlops (SP) to 11.96 GFlops (DP). The reason is that CUDA 4.0 natively supports `int` and `float` but not double version of `atomicAdd`. But GPU-IV uses less memory than GPU-III, *e.g.*, all datasets can be loaded in GPU-IV but not in GPU-III, as indicated by "x" in the table. Therefore, GPU-III is more efficient but uses more memory, whereas GPU-IV is less efficient but uses less memory. There is a trade-off between performance and memory usage.

The two China datasets have higher speedup on both single precision and double precision than two Southern California datasets due to its higher nonzero ratio, because the CUDA kernel in CSR format has higher memory access efficiency when nonzero ratio is higher.

Table 4: Performance (GFlops) of different datasets for CPU and GPU in single precision (SP) and double precision (DP). "x" indicates that the memory is not enough to load the dataset.

data	CPU(SP)	GPU-IV(SP)	GPU-III(SP)	CPU(DP)	GPU-IV(DP)	GPU-III(DP)
CN 200801ker	0.844	9.915	15.488	0.7	2.36	11.96
CN 2008ker	0.796	9.567	15.701	0.719	2.359	x
S CA DEC5	0.765	3.962	8.218	0.642	1.74	6.54
S CA DEC3	0.754	3.721	x	0.617	1.806	x

6.2. Multi GPUs (MPI-CUDA) Experiments

We use one MPI task residing on one CPU core to control one GPU. To keep the MPI communication cost the same, we compare the performance of same number of CPU cores with same number of GPUs.

Strong scalability defines how the execution time varies with the number of cores for a fixed problem size. We tested strong scalability of MPI-CPU code, MPI-GPU code, and PETSc code using CN2008ker dataset with the number of CPU cores or GPUs ranging from 1 to 60. We do not test more CPU cores or GPUs, because the decomposed problem size is too small for each core or GPU, which cannot fully exploit the computing power of CPU or GPU. Figure 5 (a) illustrates the performance of MPI-CPU, MPI-GPU-IV, MPI-GPU-III, and PETSc. The result on one GPU in double precision is missed because the dataset is so large that it cannot be loaded into a single GPU. The results of PETSc on three or less CPU cores are also missed because of the same reason. Both our MPI-CPU and MPI-GPU implementations have better performance than PETSc and scale as the number of CPU cores or GPUs increases from 1 to 60. The best performance of CPU is 29.9 GFlops using 60 CPU cores in single precision, 37x faster than single core; for double precision, the performance of MPI-CPU is 21.1 GFlops using 60 CPU cores, 27x faster than single core. MPI-GPU-III is faster than MPI-GPU-IV in all cases. For single precision, the highest performance of MPI-GPU-III is 55.2 GFlops using 60 GPUs, 1.8x faster than 60 CPU cores; for double precision, it is 43.6 GFlops for 60 GPUs, 2x faster than 60 CPU cores.

Weak scalability shows how the execution time varies with the number of cores for a fixed problem size per core. We tested weak scalability from 15 to 135 CPU cores or GPUs by duplicating China 2008ker dataset to make every core or GPU has roughly the same volume of data. For instance, the dataset used by 135 CPU cores (or GPUs) is 9x as large as the dataset used by 15 CPU cores (or GPUs). Figure 5 (b) shows the experimental result of MPI-CPU, MPI-GPU-III, and PETSc. In this experiment, we use GPU-III as the representative of GPU since it is the fastest GPU version. As Figure5 (b) shows, both performances of MPI-CPU and MPI-GPU-III increase as the number of CPU cores or GPUs increases. MPI-GPU-III is around 3x to 4x faster than the MPI-CPU code with the same number of

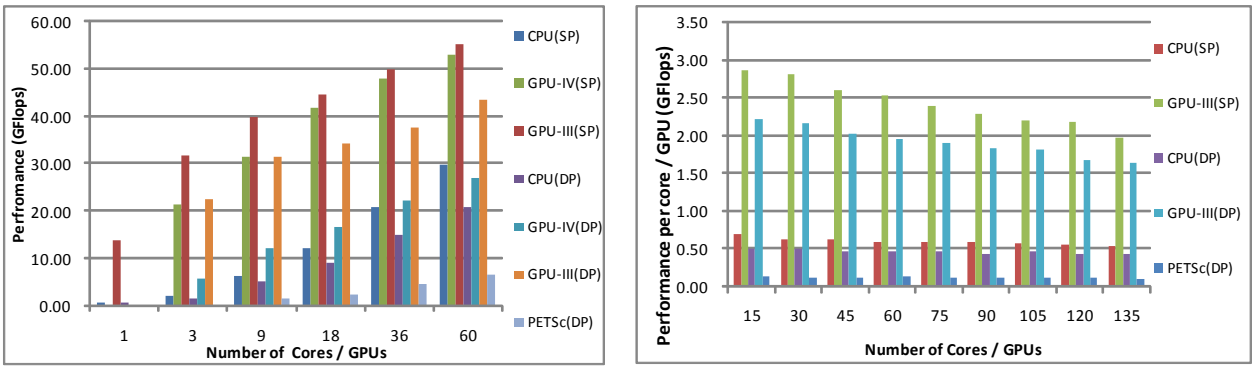

Figure 5: (a) Strong and (b) weak scaling test of MPI-CPU, MPI-GPU and PETSc implementations on CN2008ker dataset.

CPU cores in single precision and double precision. The MPI-GPU-III code achieves 268 GFlops in single precision and 223 GFlops in double precision on 135 MPI tasks. The degradation from single precision to double precision is 20-30% for both MPI-CPU and MPI-GPU-III. Our implementations of MPI-CPU and MPI-GPU-III are faster than PETSc in double precision.

7. Conclusions and Future Work

We design and implement an MPI-CUDA based parallel approach for LSQR solver. Both our single GPU and MPI-GPU codes achieve considerable speedup compared with CPU code. The MPI-GPU code scales on both weak and strong scaling tests, and is faster than PETSc. Our future work will focus on reducing the network communication between MPI tasks to further increase the scalability on very large number of GPUs. We will redesign the parallel computing part of the algorithm based on a new data decomposition approach in order to minimize the communication traffic. Another optimization technique is to utilize GPUDirect technology to speed up data communication between different GPUs when it is fully supported.

References

[1] C. Paige, M. Saunders, LSQR: An algorithm for sparse linear equations and sparse least squares, ACM Transactions on Mathematical Software (TOMS) 8 (1) (1982) 43–71.
[2] G. Nolet, Solving large linearized tomographic problems, Seismic tomography: theory and practice (1993) 248–264.
[3] J. Bolz, I. Farmer, E. Grinspun, P. Schröoder, Sparse matrix solvers on the gpu: conjugate gradients and multigrid, ACM Trans. Graph. 22 (3) (2003) 917–924.
[4] M. M. Baskaran, R. Bordawekar, Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies, Tech. rep., Research Report RC24704, IBM TJ Watson Research Center (2008).
[5] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, NY, USA, 2009, pp. 1–11.
[6] J. W. Choi, A. Singh, R. W. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on gpus, in: PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM, New York, NY, USA, 2010, pp. 115–126.
[7] P. Guo, H. Huang, Q. Chen, L. Wang, E.-J. Lee, P. Chen, A model-driven partitioning and auto-tuning framework for sparse matrix-vector multiplication on gpus, in: the TeraGrid 2011 Conference, Scientific Track, ACM Press, 2011.
[8] O. Baur, G. Austen, A parallel iterative algorithm for large-scale problems of type potential field recovery from satellite data, in: Proceedings Joint CHAMP/GRACE Science Meeting, GeoForschungsZentrum Potsdam, 2005.
[9] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc Web page, http://www.mcs.anl.gov/petsc (2009).
[10] J. Liu, F. Liu, J. Liu, T. Hao, Parallel LSQR algorithms used in seismic tomography, Chinese Journal of Geophysics 49 (2) (2006) 540–545.
[11] J. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, S. Yalamanchili, Keeneland: Bringing heterogeneous gpu computing to the computational science community, Computing in Science Engineering 13 (5) (2011) 90 –95.