Bulletin of the Seismological Society of America, Vol. 104, No. 2, pp. 953–961, April 2014, doi: 10.1785/0120130236

A Graphics Processing Unit Implementation and Optimization for Parallel Double-Difference Seismic Tomography

by Pei-Cheng Liao, Cheng-Chi Lii, Yu-Chi Lai, Ping-Yu Chang, Haijiang Zhang, and Clifford Thurber

Abstract Double-difference seismic tomography can estimate velocity structure and event locations with high precision, but its high-computation cost along with large memory usage prevents the use of a personal computer to process very large datasets and requires a long-computation time. This work proposes graphics-processing-unit-(GPU)-based acceleration schemes to run the algorithm on a personal computer for very large datasets more efficiently. Generally, the algorithm can be divided into five major steps: input, ray tracing, matrix construction, inversion, and output. This work focuses on accelerating the ray-tracing and inversion steps, which take almost two-thirds of the computation time. Before ray tracing, our algorithm preprocesses the data by sorting all recorded event–station paths according to their lengths. Therefore, those path estimation jobs assigned to GPU cores are suitable for the GPU architecture. Furthermore, our work also minimizes the usage of global and local memory to reduce the GPU computing time needed to handle a very large dataset. In addition to parallelizing the inversion computation, our work proposes a GPU-based elimination method to reduce redundant computation in inversion for further acceleration. In our test, the proposed acceleration schemes can gain maximum speed-up factors of 31.17 and 35.46 for ray tracing and inversion, respectively, in our test. Overall, the GPU-based implementation can reach a maximum of 5.98 times faster than the central processing unit-based implementation.

Introduction

Understanding the velocity structure of a region is important for resource exploration and development as well as for better understanding the tectonic process. There are several ways to estimate the subsurface velocity structure, and double-difference (DD) tomography by Zhang and Thurber (2003) is one of them. DD tomography uses observed arrival times at different observation points for simultaneous estimation of velocity structure and the locations of the earthquakes. The algorithm at a local scale assumes a flat earth model and uses waveform cross-correlation (WCC) techniques (VanDecar and Crosson, 1990) to calculate the differential arrival times for pairs of events at common stations. The main advantage of this method is that a high resolution of subsurface velocity structure can be estimated in a region of high-earthquake density. However, this method generally requires a large number of iterations to construct the set of estimated seismic ray paths and invert for the structure and location perturbations and thus, it usually takes a long time to compute and requires a large amount of memory usage to store all intermediate data. Therefore, this study aims at accelerating the computation process to save time and making it possible to run very large datasets on a personal computer.

Because the computations in ray tracing and matrix operations in least-squares inversion are independent, they can be parallelized to accelerate the computations. The previous studies used a cluster of computers to accelerate the algorithm. However, Grunberg *et al.* (2004) have shown that the large amount of data transfer decreases the amount of acceleration due to the bandwidth limitation of a local area network. Because a cluster of computers is expensive, the recent advanced graphics processing units (GPUs) with highly parallel processing abilities provide an alternative method for accelerating the DD tomography algorithm. The many cores and onboard memory can provide the necessary abilities to simultaneously handle independent computation in parallel.

This study focuses on designing acceleration schemes for the DD tomography algorithm with a parallel processing many-core GPU. Generally, DD tomography can be divided into five main steps: input, ray tracing, matrix construction, least-squares inversion, and output which will be given in more details in the Background section. Currently, the present study focuses on two of the three most time-consuming steps: ray tracing and least-squares inversion. Ray tracing determines the path of minimum travel time from the

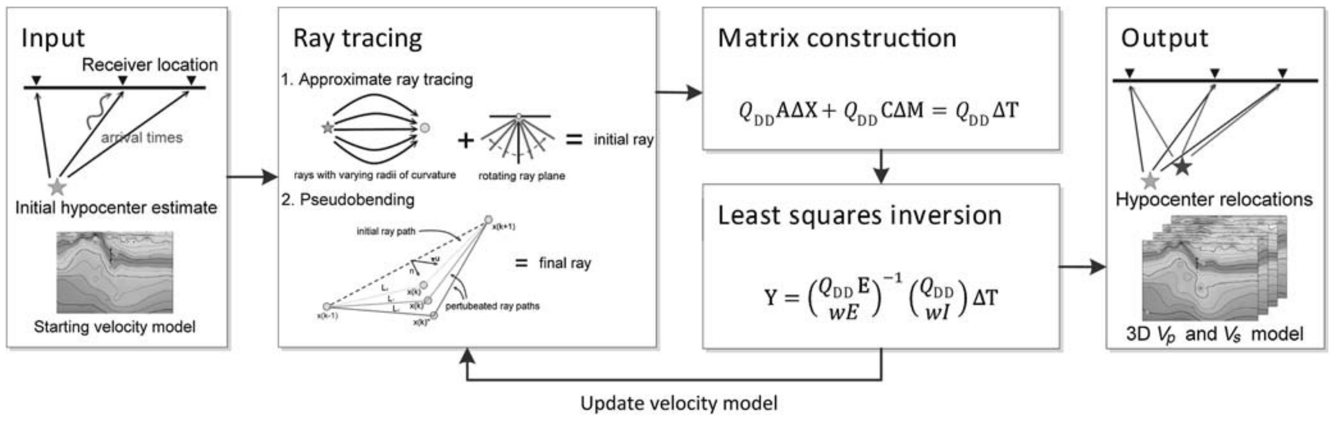

Figure 1. The workflow of DD tomography. It generally has five major steps: input, ray tracing, matrix construction, least-squares inversion, and output. The input and output are only done once, and the other three are done consecutively and iteratively.

earthquake to the observing station. A central-processing-unit- (CPU)-based implementation does not need to consider the length of a path because all path computations are done in a sequential order, and it does not matter which one comes first. However, the GPU-based algorithm carries out path estimation in a parallel structure, and when a set of paths of different lengths are sent simultaneously to the GPU cores for processing, their estimation requires very different computation times, and this reduces the efficiency of utilization of the GPU cores. As a result, to make those computations assigned to a set of cores at the same time slot have a similar computation time, we first sort the estimated paths based on their straight distance between the station and the event location, and the paths are grouped based on a similar distance. Each path then can be estimated independently using a GPU core. Next, we consider the matrix for least-squares inversion. The matrix consists of redundant elements produced by the system. We introduced a GPU-based elimination method to remove these redundant elements and then parallelize the inversion process with a GPU-based matrix inversion algorithm.

For the entire algorithm, in this study, we minimize the amount of global and local memory usage to reduce the number of passes. (When handling a very large dataset, the memory requirement will be over the limit of a GPU. Thus, the computation must be divided into several groups that are sent to a GPU consecutively. Each sending event is defined as a pass.) The empirical results show that the GPU-based acceleration can significantly improve the efficiency of DD tomography. Overall, the GPU-based method shows an acceleration of the ray-tracing process by a rate of 10.88 and 31.17 times and the least-squares inversion process by a rate of 2.28 and 35.46 times on two test datasets.

This study shows the following: (1) our GPU-based implementation accelerates the DD tomography algorithm by a rate of 2.98 and 5.87 for the two datasets, (2) we introduced a sorting process to make the loading of the data more balanced to reduce the waiting time, (3) elimination of redundant elements reduces the size of the sparse array and the computation time for matrix inversion, and (4) the present approach also reduces memory usage and finds the best execution order to optimize the performance.

Background

GPU and CUDA

A GPU (Nvidia, 2009) consists of a cluster of cores that are independent computing units and are optimized to simultaneously perform the same operation on a large set of data. Compute unified device architecture (CUDA; Nvidia, 2009) is designed to develop parallel algorithms on many cores. Several examples including Jeong *et al.* (2006), Kadlec and Dorn (2010), and Wang *et al.* (2010) have used GPU to accelerate the seismic data processing operations. Similarly, the computations of ray-tracing and inversion processes are highly independent. Therefore, this study demonstrates the possibility for GPU-based implementation of DD tomography.

DD Tomography

DD tomography (Fig. 1) uses two separate but close events to simultaneously estimate the locations of earthquakes and the velocity structure accurately. The double difference can be defined as

$$\begin{aligned} dr_k^{ij} &= (T_k^i - T_k^j)^{\text{obs}} - (T_k^i - T_k^j)^{\text{cal}} \\ &= (T_k^{i^{\text{obs}}} - T_k^{i^{\text{cal}}}) - (T_k^{j^{\text{obs}}} - T_k^{j^{\text{cal}}}), \qquad (1) \end{aligned}$$

in which i and j are indexes of earthquake events, k is the index of an observing station, T_k^i is the arrival time of earthquake i at the observing station k, $(T_k^i - T_k^j)^{\text{obs}}$ denotes the difference of the observed arrival time for event i and j at the observing station k, and $(T_k^i - T_k^j)^{\text{cal}}$ denotes the difference of the calculated arrival time for event i and j at the observing station k. Note that DD tomography also uses absolute arrival times for estimation of subsurface velocity structure and earthquake locations as in conventional tomography (Zhang, 2003).

The input to the algorithm is a set of values including the locations of events and observing stations, arrival times of the *P*- and *S*-waves at the observing stations and the cross-correlated time lags for these waves at the observing stations using WCC (VanDecar and Crosson, 1990). A pseudobending ray-tracing algorithm (Zhang and Thurber, 2003) traces the event tracks through the subsurface velocity structure to estimate $(T_k^i - T_k^j)^{\mathrm{cal}}$. Because the approximate paths with the shortest travel times are independent of each other, these path estimations can be processed in parallel, and the entire ray-tracing process can be accelerated by a GPU. The parallel acceleration scheme will be given in detail in the Accelerate Ray Tracing with a GPU section.

After constructing the double differences, the information can be used to update the original event locations and velocity structure. According to Zhang and Thurber (2006), equation (1) can be rewritten as

$$\mathbf{Q}_{\mathbf{DD}}\mathbf{\Delta T} = \mathbf{Q}_{\mathbf{DD}}\mathbf{A}\mathbf{\Delta X} + \mathbf{Q}_{\mathbf{DD}}\mathbf{C}\mathbf{\Delta M}, \tag{2}$$

in which $\mathbf{Q}_{\mathbf{DD}}$ is the DD operator, $\mathbf{\Delta T}$ is the vector of the arrival time residuals, $\mathbf{A}$ is the partial derivative matrix with respect to the event locations and origin times, $\mathbf{\Delta X}$ is the perturbation vector of event locations and origin times, $\mathbf{C}$ is the model derivative matrix with respect to the slowness model, and $\mathbf{\Delta M}$ is the vector of slowness perturbations. $\mathbf{\Delta X}$ and $\mathbf{\Delta M}$ can be used to determine the convergence of the estimation of locations and velocity structure. In equation (2), $\mathbf{Q}_{\mathbf{DD}}$ is fixed and $\mathbf{A}$ and $\mathbf{C}$ must be calculated every time when the event locations and velocity structure change. In this study, we term the process of updating $\mathbf{A}$ and $\mathbf{C}$ the matrix construction process.

To solve $\mathbf{\Delta X}$ and $\mathbf{\Delta M}$, equation (2) can be further transformed to the following form:

$$\mathbf{Y} = \begin{pmatrix} \mathbf{Q}_{\mathbf{DD}}\mathbf{E} \\ w\mathbf{E} \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{Q}_{\mathbf{DD}} \\ w\mathbf{I} \end{pmatrix} \mathbf{\Delta T}, \tag{3}$$

in which $\mathbf{E} = [\mathbf{A} \quad \mathbf{C}]$, $\mathbf{Y} = \begin{bmatrix} \mathbf{\Delta X} \\ \mathbf{\Delta M} \end{bmatrix}$, w is the relative weighting between absolute and relative arrival times as defined in Zhang and Thurber (2003), which is varied in a progressive manner, and $\mathbf{I}$ is the identity matrix. Equation (3) can be solved with the LSQR algorithm (Paige and Saunders, 1982). The main operations for the least-squares matrix process are scaling vectors and matrix multiplications, which can be naturally accelerated by a GPU. Thus, our work will also adapt a GPU-based LSQR scheme proposed in Huang *et al.* (2012) to parallelize the least-squares inversion process. Before solving equation (3), both $\mathbf{A}$ and $\mathbf{C}$ are sparse matrixes and the number of nonzero elements affects the inversion time. We found that there are redundant elements which are produced by the system. Those elements do not affect the computation results and the CPU computation speed, but are not suitable for the GPU computation. Therefore, this work also tries to eliminate those elements to enhance the computational efficiency on GPU (details will be shown in the LSQR with a GPU section). After solving equation (3), $\mathbf{\Delta X}$ and $\mathbf{\Delta M}$ are used to update the event locations and velocity structure until they converge.

GPU Implementation

As discussed in the DD Tomography section, the input and output processes are fixed and can only be accelerated by improving the read/write hardware. The iterative ray tracing, matrix construction, and least-squares inversion steps are the factors for the GPU-based acceleration. To accelerate the algorithm, in this study, we first analyze the performance of each step to identify key factors in these steps and then focus on the schemes to parallelize them with a GPU.

The proper programming model for a GPU is single instruction, multiple threads (SIMT), that is, an instruction can run on a set of different data using multiple threads. CUDA organizes threads into block-and-grid structures, and each block is assigned to a set of cores to execute in a single warp (a group of cores executing at the same time on different data) at the same time. Next, we used the term job to indicate a set of computations sent to a core. To fulfill this programming model, the jobs run in a warp should have a similar number of instructions to make all jobs finish at roughly the same time. In our implementation, every path-estimation job is split into parts as small and similar as possible to follow the SIMT model and parallelize the similar path estimation.

Accelerate Ray Tracing with a GPU

Because the original DD tomography algorithm (Zhang, 2003) has many intermediate variables that exceed the limit of a GPU, in this study, we decompose the computation of a path into several jobs for parallel acceleration in the GPU architecture. However, the result of a path estimation job does not affect the result of other path estimation jobs, that is, all path computations are independent from each other. Therefore, in this study, we optimize the usage of memory and the placement of memory to reduce the number of GPU passes in each iteration and achieve good memory access performance inside a GPU.

Data including the locations of stations and events, slowness perturbations, and inversion parameters are needed and allocated in the GPU memory. However, when estimating a path, the thread needs to temporarily record intermediate information, and the memory used by a thread can be up to 178.4 kb for our larger dataset. Because approximate pseudobending ray tracing is designed to process each event–station path in a sequential order, and each path computation only needs to record one set of temporary intermediate data of 178.4 kb, the memory usage is well under the system limit on CPU. However, when running the computation jobs in

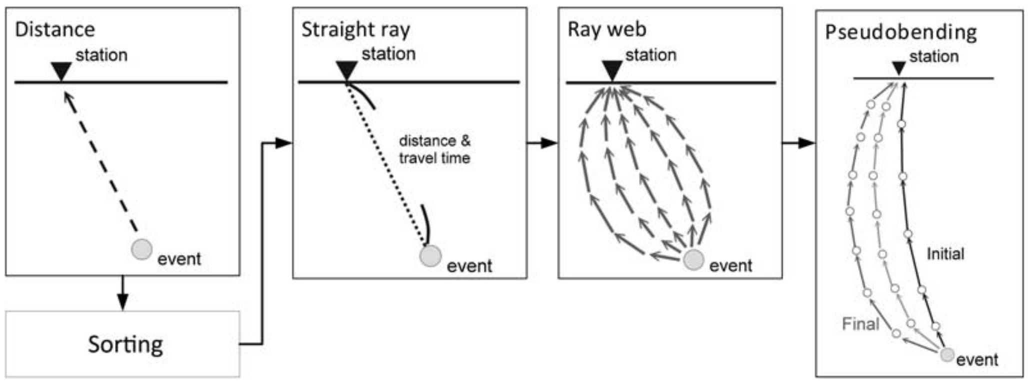

Figure 2. The workflow of the approximate pseudobending ray-tracing algorithm on graphics processing unit (GPU). For details of the substeps, refer to the work by Thurber (1983) and Um and Thurber (1987).

many cores of a GPU, the memory needed for all threads can exceed the limitation of a GPU. Without optimization, the memory required to store all path information is more than 11.9 GB for the largest dataset in the test. It is too large to allocate such a large memory on a GPU. Therefore, the present method reexamines the memory usage to create a set of memory space for those intermediate variables that only appear once in the computation and eliminates 79% of the required information for a path job. This can reduce the number of passes required when the number of paths is large.

Memory placement optimization is also important for performance. The goal of placement is to maximize data transfer bandwidth, that is, minimize data access time by using as much fast memory and as little slow-access memory as possible. CUDA has several types of memory: (1) local memory that is fast, but has limited amount of space, and can be used to save the information of a path for a thread because the information is accessed frequently and should be cached for fast access, (2) global memory is another type, which is slow, but has a very large volume of space, and can be used to save the hypocenters and slowness model, and (3) the third type is texture memory, which is also slow and read-only, but has a caching mechanism for a warp of cores, which provide faster reading access than global memory. It can be used to save the velocity structure, because it is queried when estimating the travel time of a wave passing through.

Even after reducing the size of information kept for each path, the memory required for processing all paths at the same time is still too high to fulfill, and therefore, our work decomposes the ray-tracing process into four small substeps: distance, straight ray, ray web, and pseudobending. Only a few interstep data need to be kept in the memory, and all the intermediate data in each substep is released. For example, the straight distance of each path is computed in the distance substep and only used in the straight substep. It should be released after the straight substep. This can minimize the amount of data required in any moment of the ray-tracing process to allow the system to reduce the number of passes. Additionally, the decomposition also can fulfill the programming concepts, SIMT discussed in previous paragraphs, to gain a better acceleration. Figure 2 presents the simplified workflow of ray tracing.

The distance substep estimates the event–station straight distance and travel time of each path which determines several parameters for the following steps. All distance jobs are collected into a pool and parallelized to GPU cores on a path-based assignment scheme. The straight-ray substep constructs a straight path from the event location to the station, estimates the travel time of the straight path, decides the number of segments and sets up the required memory space for the following computation. The main determining factor in the ray tracing is the event–station distance. When putting jobs having a different number of segments into the same warp, some cores must wait for the others to finish the computation due to the SIMT computation of a GPU. Therefore, it is important to let jobs of a warp have a similar number of segments, and thus, in this study, we sort the path jobs based on their straight distance between the event location and observing station. Then, the jobs with a similar size are assigned to GPU cores at a roughly similar time. Therefore, sorting the distance is set up to run with the straight ray substep in the CPU because when the CPU sorts the jobs, the GPU can allocate memory for later usage at the same time. After sorting paths, all estimation jobs are parallelized based on the path-based assignment scheme. The ray web substep constructs a set of paths using a fixed set of radius of curvature values for a path to identify a path with approximate minimal travel time. All jobs are parallelized to GPU cores on a path-based assignment scheme, but this substep uses a large amount of memory because of the need to store a large set of paths. The memory scheme mentioned at the beginning of this section is applied to reduce the number of passes for a large dataset to save time. The pseudobending substep iteratively perturbs the path until the travel-time difference is less than a threshold. Paths are estimated independently from each other, and thus the path-based parallel scheme can be applied, too. Although the original pseudobending algorithm is executed with two steps, in this study, we merge these two steps into one to save the memory transfer time.

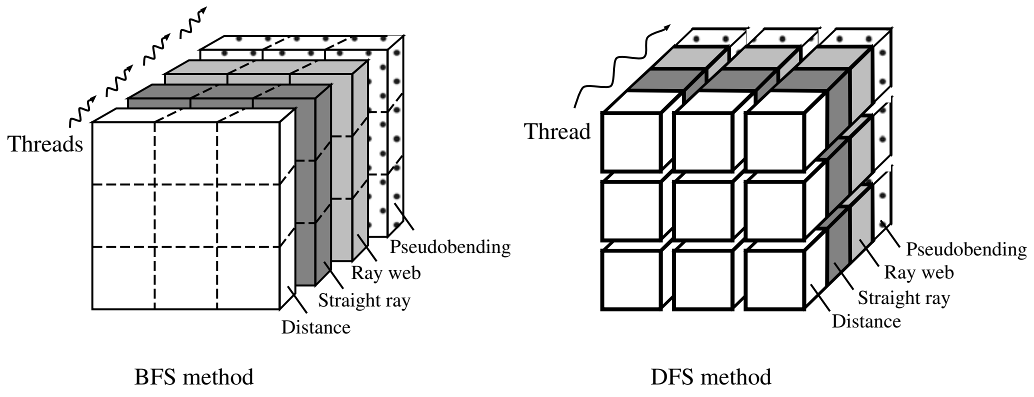

Figure 3. The schematic concept of breadth first search (BFS) and depth first search (DFS). The thread axis shows the assigned sequential steps in a GPU. The distance cubes represent those distance computation jobs, the straight ray cubes represent those straight ray computation jobs, the ray web cubes represent ray web computation jobs, and the pseudobending cubes represent those pseudobending computation jobs. A group of cubes surrounded by black lines are a set of computation jobs assigned in the same warp into the GPU.

After dividing ray tracing into four substeps, because not all of the jobs can simultaneously be computed at one time, we separate them into several groups and send them in a designed order to the GPU. As shown in Figure 3, GPU programming yields two choices to submit these jobs into a GPU: The first one is called breadth first search (BFS) which processes all jobs belonging to the same substep at the same time and then processes all jobs belonging to the next substep after finishing. The second one is called depth first search (DFS), which processes all computation jobs belonging to a path estimation at the same time and then processes other path-estimation jobs after finishing the current one. In Figure 3, the set of cubes represents the computations of the ray-tracing process. A cube drawn with dotted lines represents a set of jobs sent in a warp. A group of cubes surrounded by black solid lines are a set of jobs submitted to a GPU at the same time. The main difference between these two methods is how to split the cubes. BFS slices the jobs based on the substep order, and DFS slices the jobs based on the path structure. Then, each sliced piece is sent to a GPU in a sequential order with the streaming mechanism of a GPU to reduce job launching time and data transfer time. Conceptually, when running these two methods in a CPU, the overall time should be the same but as will be shown in Result and Discussion section, assigning jobs based on BFS performs better than assigning jobs based on DFS.

LSQR with a GPU

After matrix construction, we want to solve equation (3) in a GPU. There are several algorithms to solve this equation, and LSQR is a common choice in the seismic field (Muffels *et al.*, 2006). However, the original LSQR is designed to operate sequentially. We adapt the approach of Huang *et al.* (2012) to accelerate the LSQR process with a GPU. Figure 4 shows the workflow of the GPU-based LSQR in this work. The GPU main computation uses CUDA-provided libraries including CUSPARSE and CUBLAS for basic linear algebra

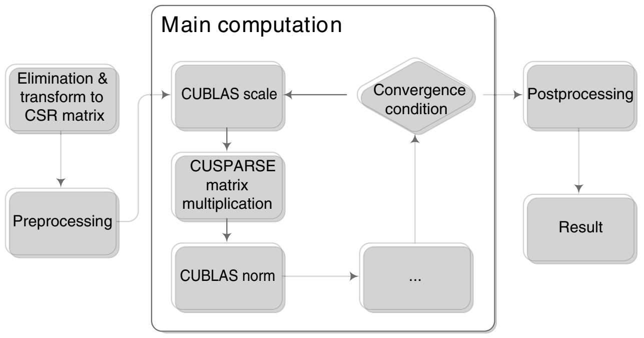

Figure 4. The workflow of the LSQR algorithm on a GPU.

operations to iterate to convergence. The CPU implementation generally uses an array to store the elements and do the sparse operations. When carefully examining the inversion process, because the system eliminates elements the values for which are smaller than a user-defined threshold in a similar manner to Zhang and Thurber (2003), we found that the CPU implementation eliminates these elements by swapping them to the end of the array. The swapping saves the extra cost of removing the element from the array, but those redundant elements still occupy the memory space of the array. The inversion process can use a branch statement to avoid the computation of these redundant elements with almost zero cost for a CPU, but the branch operation is costly in the GPU computation. Additionally, transforming the sparse matrix into the compressed sparse row (CSR) format (Nvidia, 2012) is required for CUDA, because CSR can make sparse matrix multiplication operations faster and save memory usage. However, most sparse matrixes are recorded in coordinate format (COO) (Nvidia, 2012) which is convenient for access and construction. We transform the format before using the CUDA libraries. Therefore, the algorithm used in this study merges the elimination of redundant elements and the transformation of the CSR matrix into the first step. The second step transfers the data from the CPU to the GPU and

Table 1
The Computation Time with the CPU- and GPU-Based Implementation and the Speed-Up Ratio of the GPU-Based Implementation

	Ray Tracing		Matrix Construction		Least-Squares Inversion		Entire Process	
	PF1*	PF2[†]	PF1	PF2	PF1	PF2	PF1	PF2
CPU time[‡] (s)	2.132	33.708	0.643	31.022	0.527	26.277	40.697	1124.56
GPU time[§] (s)	0.196	1.063	0.755	18.730	0.231	0.741	13.651	191.562
Speed-up[‖]	10.88	31.71	0.85	1.66	2.28	35.46	2.98	5.87

*PF1 is Parkfield1 dataset.
[†]PF2 is Parkfield2 dataset.
[‡]CPU time is the time for the CPU-based implementation.
[§]GPU time is the time for the GPU-based implementation.
[‖]Speed-up is the speed-up ratio of the GPU-based implementation over the CPU-based implementation.

allocates memory for computation. The third step is the iterative inversion step including using CUBLAS to calculate scale and norm operations and CUSPARSE to do the matrix multiplication. The last step transfers the result back to the CPU and cleans up the allocated memory.

Result and Discussion

This section will present the results of using GPU-based implementation for seismic tomography. These results are then compared with the traditional CPU-based calculation to analyze the efficiency of the GPU implementation. These two sets of data were collected by Parkfield Area Seismic Observatory provided in Zhang *et al.* (2009). The two datasets are listed as: Parkfield1 which has 67 stations and 50 events and Parkfield2 which has 590 stations and 1562 events. The listed observing stations may not be able to detect all seismic events and thus, the actually observed event–station paths are 2799 in Parkfield1 and 67,038 in Parkfield2. All statistics provided in this section are done with a personal workstation for which the CPU is Intel Xeon E5506 2.13 GHz with a 6 GB DDR3 memory and the graphics card is Nvidia GeForce GTX TITAN (Nvidia, 2013) which has 2688 CUDA cores and a 6 GB GDDR5 onboard memory. The computer runs under Windows 7 SP1 (64-bits). The GPU-based DD tomography algorithm is implemented with CUDA version 5.0, CUBLAS, and CUSPARSE and the CPU-based implementation is provided in Zhang and Thurber (2003).

This study mainly focuses on accelerating the ray tracing and least-squares inversion processes with a GPU. Therefore, we will first give detailed analysis of the proposed acceleration schemes and then present the overall performance later. Table 1 lists the performance of the proposed acceleration schemes for two datasets in the first iteration of the computation. The GPU-based implementation completed all path estimation jobs much faster than the CPU-based implementation. The acceleration rate is 10.88 times for Parkfield1 and 31.71 times for Parkfield2 in the ray-tracing process, and 2.28 times for Parkfield1 and 35.46 times for Parkfield2 in the least-squares inversion process. Note that for these two datasets with very different sizes, the GPU-based implementation becomes more efficient when the data size becomes larger. To know why the GPU-based implementation performs better in a larger dataset, we further analyzed the time spent in the substeps of the ray tracing and least-squares inversion processes.

As shown in Figure 5, the sorting process takes 0.60% for Parkfield1 and 1.66% for Parkfield2, but the acceleration rate can reach 1.03 times for Parkfield1 and 3.28 times for Parkfield2 when assigning jobs based on the sorting results, as shown in Figure 6. This demonstrates that sorting can make the utilization of GPU cores better especially for paths with significantly varied lengths. Obviously, the acceleration rate of each substep (Fig. 7) increases with the size of the dataset. The Parkfield1 dataset in each substep has a similar acceleration rate, but the Parkfield2 dataset in each substep has a very different acceleration rate. The distance, straight ray, and pseudobending substeps are accelerated significantly when the size of the data is increased. In addition, the pseudobending substep has been accelerated the most, with an acceleration rate of 78.33 times, because the data access is sequential and sequential memory access is highly optimized in the GPU architecture. The ray web substep gains the least-acceleration rate because the process requires a huge amount of memory writing operations to record all possible tracks derived from different radius of curvatures for a path estimation job and highly random memory access. We also compare the performance difference when implementing each substep with the BFS and DFS methods (Fig. 6). The one implemented with BFS is 1.03 times faster for Parkfield1 and 1.71 times faster for Parkfield2 than DFS. This is because the BFS job assignment follows the SIMT programming model better by making jobs in the same warp with similar instructions. The DFS job assignment has a larger and more varied number of instructions to prolong the computation time. Therefore, the execution order of substeps impacts the entire acceleration rate.

Figure 6 shows the acceleration rates when separately applying different acceleration schemes, including sort, BFS, and redundant element elimination schemes. The acceleration rate with the element elimination scheme is 1.04 times

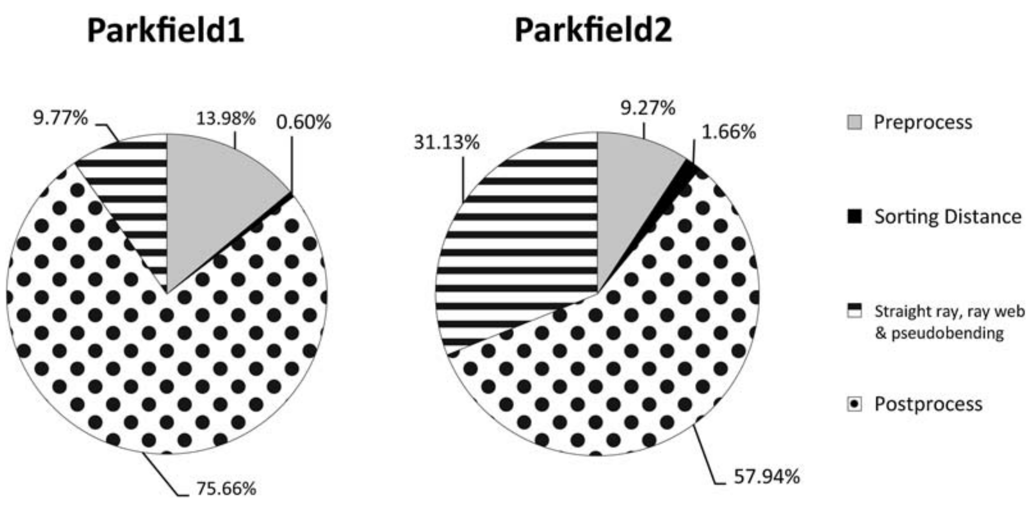

Figure 5. The percentage of time used in each substep in the first iteration for the ray-tracing process. The percentage is the ratio of the execution time in that substep over the total execution time of the ray-tracing process in that iteration. The preprocess is allocating memory and transferring time from central processing unit (CPU) to GPU. The substeps are divided into the distance, straight ray, ray web, and pseudobending substep as shown in the Accelerate Ray Tracing with a GPU section. The postprocess is transferring from GPU to CPU and releasing the memory space-time.

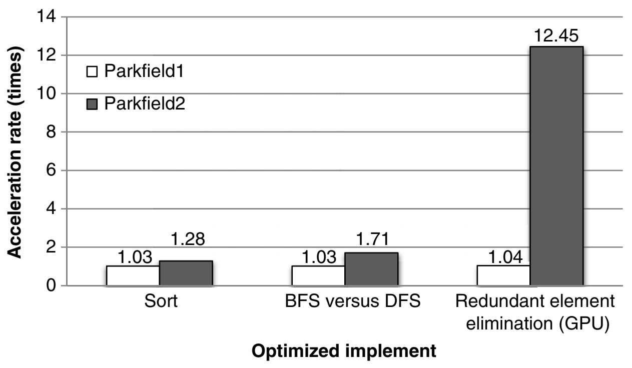

Figure 6. The acceleration rates with different acceleration schemes: sort, BFS, and redundant element elimination. The horizontal axis shows three acceleration schemes and the vertical axis denotes the speed-up ratio. Sort denotes the jobs delivered in an order based on the length of paths. BFS versus DFS denotes the application of BFS. Redundant element elimination is the preprocessing in LSQR algorithm.

for Parkfield1 and 12.45 times for Parkfield2 on GPU. The time considered includes the redundant element elimination. Obviously, our scheme can get a better acceleration rate in Parkfield2 because more elements are eliminated: 83.62% of elements are eliminated in Parkfield2. However, because only 3.8% are eliminated in Parkfield1, the acceleration rate is not significant for elimination. When further analyzing the computation time of each substep (Fig. 8), we found that the elimination and transformation scheme takes 48.63% in Parkfield1 and 61.33% in Parkfield2, but they can gain important acceleration for both datasets. This shows that elimination is more important for GPU operations. Furthermore, the pure acceleration rate of GPU-based LSQR can reach 5.83 for Parkfield1 and 15.10 for Parkfield2 when comparing

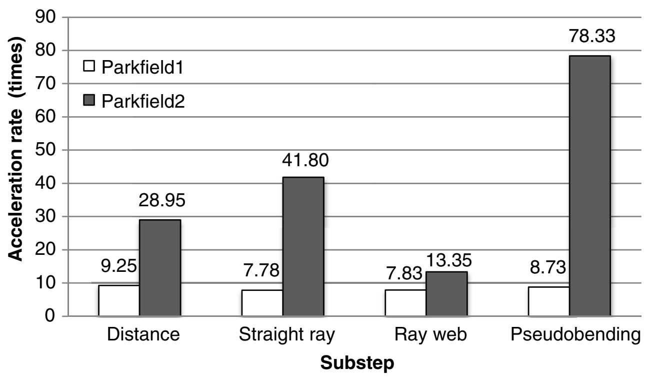

Figure 7. The acceleration of each substep in the ray-tracing process. The horizontal axis shows the four substeps as shown in the Accelerate Ray Tracing with a GPU section, and the vertical axis shows the rate of acceleration.

the inversion performance without considering data transformation which includes elimination, preprocess, and postprocess. We can see that the data transform drags the acceleration down and is the bottleneck for this step.

Table 1 shows the sum-up for overall accelerations with all acceleration schemes. The complete computation time includes the input, output, and 14 iterations of the ray tracing, matrix construction, and least-squares inversion processes for both datasets. The GPU-based implementation is 2.98 times faster for Parkfield1 and 5.87 times faster for Parkfield2 than the CPU-based implementation, because the matrix construction step drags down the overall acceleration. The matrix construction process can only be accelerated by 0.85 times in Parkfield1 and 1.66 times in Parkfield2, and it occupies about one-third of the computation time. In Parkfield1, the matrix construction process consumes 1/6 of the

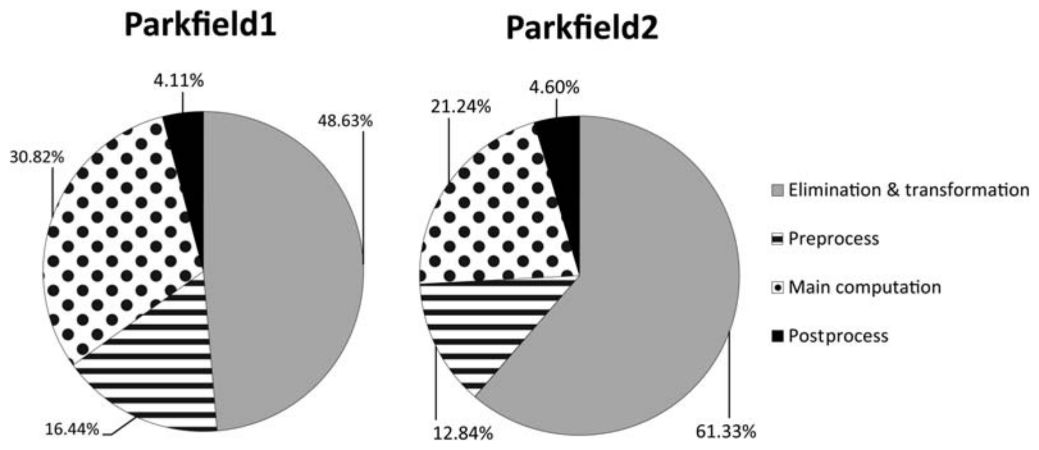

Figure 8. The pie chart that shows the percentage of time needed in each execution step in the least-squares inversion process. The elimination and transformation are eliminating redundant elements and transforming to CSR matrix. The preprocess is transferring time from CPU to GPU. The postprocess is transferring time from GPU to CPU. The details are given in the LSQR with a GPU section.

original computation time and the other two processes use only 1/30 and 1/6 of the original computation time. Therefore, overall the entire process is accelerated 2.98 times faster than the CPU-based implementation. In Parkfield2, the matrix construction process spends only 1/6 of the original computation time, and the other two processes use only 1/60 of the original computation time. Overall, the entire process requires around 11/60 of the original computation time and is accelerated 5.87 times faster than the CPU-based implementation.

However, the matrix construction is a complex calculation and a large number of branch instructions, which have different options in sequential execution, lead to parallelization difficulty. Hence, we need to improve the versatile instructions for GPU-based computing to solve the current acceleration bottleneck in the future. In addition, when handling a large dataset the memory requirement for which is over the available GPU memory, the current implementation would require multiple passes to finish the estimation, thereby reducing the efficiency. Therefore, our future work will extend this single-graphics-board algorithm to a multiple-graphics-board case to further accelerate the process and reduce the need of a cluster of computers.

Conclusion

This study has shown the acceleration of the DD tomography algorithm with a set of GPU-based schemes. They accelerate the two most time-consuming steps in the algorithm: ray tracing and least-squares inversion. All path estimations are parallelized into available GPU cores, and the computation order depends on the length of the path to reduce the latency of synchronization. We also achieve a reduction in the amount of the computation required in the least-squares inversion to further accelerate the parallelized LSQR algorithm. Moreover, the usage of memory is reduced to increase the handling size and decrease the number of passes required to finish the estimation. At the end, the acceleration schemes can speed up the ray-tracing process by a factor of 10.88 and 31.17 and least-squares inversion process by a factor of 2.28 and 35.46 on two datasets. When applying these acceleration schemes, the overall acceleration is a factor of 2.98 and 5.87. As a result, the GPU-based implementation makes the inversion of a large dataset with a personal computer efficient. It may be noted that the tested datasets are from a local scale study but for regional or even global scale studies in which the ray paths are much longer, the acceleration schemes shown in this study will perform even better.

Data and Resources

All data used in this paper came from published sources listed in the references.

Acknowledgments

This material is based upon work supported in part by Grants Numbers NSC 101-2815-C-011-026-E, NSC 100-2221-E-011-151, NSC 101-2221-E-011-153, NSC 101-2116-M-019-003, and NSC 102-2116-M-019-003 from National Science Council of Taiwan, EAR-1141983 and EAR-1114228 from the United States National Science Foundation (C. T.) and Number 41274055 from Natural Science Foundation of China (H. Z.). We thank two anonymous reviewers for their very careful reading of the manuscript and suggestions for its improvement.

References

Grunberg, M., S. Genaud, and C. Mongenet (2004). Seismic ray-tracing and Earth mesh modeling on various parallel architectures, *J. Supercomput.* **29,** no. 1, 27–44.

Huang, H., L. Wang, E. Lee, and P. Chen (2012). An MPI-CUDA implementation and optimization for parallel sparse equations and least squares (LSQR), *Proc. Comput. Sci.* **9,** 76–85.

Jeong, W., R. Whitaker, and M. Dobin (2006). Interactive 3D seismic fault detection on the graphics hardware, Volume Graphics 2006: Eurographics, 111.

Kadlec, B. J., and G. A. Dorn (2010). Leveraging graphics processing units (GPUs) for real-time seismic interpretation, *TLE* **29,** no. 1, 60–66.

Muffels, C., M. Tonkin, H. Zhang, M. Anderson, and T. Clemo (2006). Application of LSQR to calibration of a MODFLOW model: A synthetic study, in *MODFLOW and More: Managing Ground Water Systems*, The Colorado School of Mines, Golden, Colorado, 21–24 May 2006, 283–287.

Nvidia (2009). CUDA C Programming Guide, Nvidia, http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (last accessed February 2014).

Nvidia (2012). CUSPARSE Library User Guide, Nvidia, http://docs.nvidia.com/cuda/cusparse/ (last accessed February 2014).

Nvidia (2013). GeForce GTX TITAN Specification. Retrieved from http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications (last accessed February 2014).

Paige, C. C., and M. A. Saunders (1982). LSQR: An algorithm for sparse linear equations and sparse least squares, *ACM Trans. Math. Software* **8,** no. 1, 43–71.

Thurber, C. H. (1983). Earthquake locations and three-dimensional crustal structure in the Coyote Lake area, central California, *J. Geophys. Res.* **88,** 8226–8236.

Um, J., and C. H. Thurber (1987). A fast algorithm for two-point seismic ray tracing, *Bull. Seismol. Soc. Am.* **77,** 972–986.

VanDecar, J. C., and R. S. Crosson (1990). Determination of teleseismic relative phase arrival times using multi-channel cross-correlation and least squares, *Bull. Seismol. Soc. Am.* **80,** no. 1, 150–169.

Wang, S., X. Gao, and Z. Yao (2010). Accelerating POCS interpolation of 3D irregular seismic data with Graphics Processing Units, *Comput. Geosci.* **36,** no. 10, 1292–1300.

Zhang, H. (2003). Double-difference seismic tomography method and its applications, University of Wisconsin–Madison.

Zhang, H., and C. H. Thurber (2003). Double-difference tomography: The method and its application to the Hayward fault, California, *Bull. Seismol. Soc. Am.* **93,** no. 5, 1875–1889.

Zhang, H., and C. Thurber (2006). Development and applications of double-difference tomography, *Pure Appl. Geophys.* **163,** 373–403.

Zhang, H., C. H. Thurber, and P. Bedrosian (2009). Joint inversion for V_P, V_S, and V_P/V_S at SAFOD, Parkfield, California, *Geochem. Geophys. Geosys.* **10,** Q11002, doi: 10.1029/2009GC002709.

Department of Computer Science and Information Engineering
National Taiwan University of Science and Technology
No. 43, Sec. 4 Keelung Road
Taipei City, 106 Taiwan, ROC
tom86046@gmail.com
nucleargod823543@gmail.com
yu-chi@mail.ntust.edu.tw
(P.-C.L., C.-C.L., Y.-C.L.)

Institute of Applied Geoscience
National Taiwan Ocean University
No. 2, Beining Road
Keelung, 20224 Taiwan, ROC
pingyuc@mail.ntou.edu.tw
(P.-Y.C.)

Laboratory of Seismology and Physics of Earth's Interior
School of Earth and Space Sciences
University of Science and Technology of China
96 Jinzhai Road
Hefei, Anhui 230026, P.R. China
zhang11@ustc.edu.cn
(H.Z.)

Department of Geoscience
University of Wisconsin–Madison
1215 W. Dayton Street
Madison, Wisconsin 53706
thurber@geology.wisc.edu
(C.T.)

Manuscript received 4 September 2013;
Published Online 25 March 2014