MULTI-GPU BASED TWO-LEVEL ACCELERATION OF FULL WAVEFORM INVERSION

JINGRUI LUO, JINGHUAI GAO and BAOLI WANG

School of Electronic and Information Engineering, Xi'an Jiaotong University, 710049 Xi'an, P.R. China. bluebirdjr@126.com

(Received March 12, 2012; revised version accepted October 15, 2012)

ABSTRACT

Luo, J., Gao, J. and Wang, B., 2012. Multi-GPU based two-level acceleration of full waveform inversion. *Journal of Seismic Exploration*, 21: 377-394.

Full waveform inversion (FWI) of seismic data is very computationally expensive. In this paper, we have developed a two-level parallel scheme to speed up FWI with multiple graphics processing units (multiple GPUs). The first level parallelism is the coarse-grained parallelism among multiple GPUs, which is used to reduce the number of shots; the second level parallelism is the fine-grained parallelism within each individual GPU grid, which is used to speed up the wavefield propagation procedures. The PML boundary condition is used, and the efficient boundary storage strategy is used to avoid the tremendous storage requirement needed on the disk and the data transfer between the disk and memory. We tested the scheme on the INSPUR TS10000 system with 10 Tesla C2050 GPUs by reconstructing the Marmousi velocity model using FWI in the time domain and compared the computation time with that on CPUs and on single GPU, the result showed that the two-level based FWI is about 500 times faster than the CPU-based implementation, and the speedup of the two-level scheme is a product of those of the two levels individually. With this scheme, the turnaround time of FWI has been reduced significantly.

KEY WORDS: full waveform inversion, GPU, acceleration.

INTRODUCTION

Seismic full waveform inversion is a powerful tool for retrieving information of the subsurface, and was introduced by Lailly (1983) and Tarantola (1984,1987). In their work, the gradient of the objective function was calculated using the back-propagation algorithm, and the difficult and time-consuming directly calculating of the gradient was avoided, however, FWI

0963-0651/12/$5.00 © 2012 Geophysical Press Ltd.

is still very computationally expensive. Later, the FWI and the back-propagation method were extended to the frequency domain (Pratt et al., 1998; Pratt, 1999; Pratt and Shipp, 1999), and the implementations can be solved for many shots using LU decomposition of a large sparse matrix, besides, only a few select frequencies are needed for the implementation. However, the implementation of FWI in the frequency domain requires large memory requirements which makes the method unsuitable for large 3D problems (Operto et al., 2007), and because most subsalt targets are deep and many current acquisitions have a some what small finite offset, so often dozens of frequencies are required, these render the time domain implementation more attractive.

FWI in the time domain is very computational expensive, because a large number of shots are needed to be simulated and several propagation procedures are needed in each iteration step. This always causes inconvenience for the research work, so many researchers have hammered away at reducing the computational cost of FWI.

Contracting the number of shots seems to be an intuitional way to reduce the computational cost. Vigh and Starr (2008) implemented FWI using plane-wave gathers other than shot gathers. Mora (1987) generated some super-shots to speed up FWI. Wang and Gao (2010) suggested to randomly regenerate the super-shots between iterations. In recent years, many researchers have proposed to use the encoded source sums technology, for example, Krebs (2009) and Krebs et al. (2009) presented the phase encoding algorithm for two dimensional FWI in the time domain, and Ben-Hadj-Ali et al. (2009) extended the phase encoding technology to three dimensional FWI in the frequency domain.

This super-shots method and encoded source sums technology reduce the number of shots, and can get a speedup which is proportional to the number of shots, however, the thousands upon thousands propagation procedures in the implementation still make FWI very computationally expensive. Owing to the hardware advance such as the graphics processing unit (GPU) (Sanders and Kandrot, 2010; Kirk and Hwu, 2010), the propagation procedures can also be speeded up.

The graphics processing unit (GPU) is the most pervasive parallel processor to date with multicore and multithread, and was first invented by NVIDIA in 1999. Fueled by the insatiable desire for life-life real-time graphics, the GPU has evolved into a processor with unprecedented floating-point performance and programmability, as far as the single-precision floating-point is concerned, the computing capacity of GPUs can reach greater than 1T flops. GPUs greatly outpace CPUs in the arithmetic throughput and memory bandwidth, which make them the ideal processor to accelerate a variety of data parallel applications. Nowadays, the fermi CUDA architecture introduced by

NVIDIA enables programmers to use a variety of high level programming languages, and makes the GPU have wider applications.

In recent years, there have been some researchers using GPU in geophysics applications (Li et al., 2009; Zhang et al., 2009; Kadlec and Dorn, 2010). Wang and Gao (2011) presented the scheme for the acceleration of FWI on single GPU, which reduces the computational cost of the propagation procedures in each inversion step. This scheme, however, only speed up the propagation procedures and put the multi-shot problem unresolved.

In recent years, the computing capacity of GPU has been improved enormously, and multi-GPU systems have been developed, which can be used to solve the multi-shot problem in FWI. In this paper, we present a two-level multi-GPU based parallel scheme to accelerate the 2D FWI in the time domain. The first level parallelism is the coarse-grained parallelism among multiple GPUs via MPI, which is used to reduce the number of shots; the second level parallelism is the fine-grained parallelism within each individual GPU grid via fermi CUDA, which is used to speed up the wave field propagation procedures. By this, the computational cost of FWI is reduced further more.

REVIEW OF FWI IN THE TIME DOMAIN

FWI can retrieve information of the subsurface by measuring the difference between the simulated data and the recorded data, the classical least squares misfit function is given by:

$$S(\mathbf{m}) = \sum_{s=1}^{N_s} \int_0^T dt \sum_1^{N_r} [d_{obs}(\mathbf{x}_r,t;\mathbf{x}_s) - d_{cal}(\mathbf{x}_r,t;\mathbf{x}_s)]^2 \quad , \tag{1}$$

where $d_{obs}(\mathbf{x}_r,t;\mathbf{x}_s)$ is the observed wavefield at receiver $\mathbf{x}_r$ from the source $\mathbf{x}_s$, $d_{cal}(\mathbf{x}_r,t;\mathbf{x}_s)$ is the synthetic data at receiver $\mathbf{x}_r$ from the source $\mathbf{x}_s$, N_s is the number of sources, N_r is the number of receivers, and T is the recording time, $\mathbf{m}$ is the model parameter, $S(\mathbf{m})$ is the misfit between the observed data and the synthetic data.

Our goal is to obtain the model $\mathbf{m}$ for which the misfit $S(\mathbf{m})$ is a minimum. As we all know, this problem is highly nonlinear, and a gradient method, such as the conjugate-gradient method (Moro, 1987, 1988; Freudenreich and Shipp, 2000) can be used, and the model is updated as follows:

$$\mathbf{m}_{n+1} = \mathbf{m}_n + \alpha_n \gamma_n \quad , \tag{2}$$

where α_n is the step length in the n-th iteration, γ_n corresponds to the updating direction which can be obtained from the gradient of the misfit function, so the gradient of the misfit function with respect to the model **m** must be calculated first.

We consider a finite difference acoustic simulator in 2D media with constant density. The space coordinate vector **x** can de specified as (x,z), and the wavefields satisfy the acoustic wave equation:

$$[1/v^2(\mathbf{x})][\partial^2 d(\mathbf{x},t;\mathbf{x}_s)/\partial t^2]$$

$$= [\partial^2 d(\mathbf{x},t;\mathbf{x}_s)/\partial x^2] + [\partial^2 d(\mathbf{x},t;\mathbf{x}_s)/\partial z^2] + f(\mathbf{x}_s,t) \quad , \tag{3}$$

where $v(\mathbf{x})$ is the velocity and $f(\mathbf{x}_s,t)$ is the source function. The wavefield $d(\mathbf{x},t;\mathbf{x}_s)$ must also satisfy the initial condition.

The gradient of the misfit function with respect to the velocity can be calculated by zero-lag correlation of the forward propagated source wavefield and the backward propagated residual wavefields:

$$\partial S/\partial v(\mathbf{x}) = [2/\partial v(\mathbf{x})] \sum_{s=1}^{N_s} \int_0^T dt[\partial^2 d(\mathbf{x},t;\mathbf{x}_s)/\partial t^2]\lambda(\mathbf{x},t;\mathbf{x}_s) \quad , \tag{4}$$

where $d(\mathbf{x},t;\mathbf{x}_s)$ is the forward propagated source wavefields defined in eq. (3), and $\lambda(\mathbf{x},t;\mathbf{x}_s)$ is the backward propagated wavefields, which satisfies the wave equation with the residual wavefields as the source:

$$[1/v^2(\mathbf{x})][\partial^2 \lambda(\mathbf{x},t;\mathbf{x}_s)/\partial t^2]$$

$$= [\partial^2 \lambda(\mathbf{x},t;\mathbf{x}_s)/\partial x^2] + [\partial^2 \lambda(\mathbf{x},t;\mathbf{x}_s)/\partial z^2] + (d_{obs} - d_{cal})(\mathbf{x},t;\mathbf{x}_s) \quad . \tag{5}$$

The wavefield $\lambda(\mathbf{x},t;\mathbf{x}_s)$ must also satisfy the final condition.

FERMI CUDA PROGRAMMING MODEL

GPU is the ideal processor to accelerate a variety of data parallel applications. In recent years, the computing capacity of GPU has been improved enormously and systems containing multiple GPUs have become more and more common, which can serve us better.

CUDA (an acronym for Compute Unified Device Architecture) is a bran-new parallel computing architecture developed by NVIDIA, which makes

GPUs available for other computations besides image processing. It contains the instruction set architecture (ISA) and the parallel computing engine for GPUs, and provides the access interfaces for GPUs, so developers can interact with GPUs without knowing any graphic language such as DirectX, OpenGL and Cg, and can develop their own applications with C/C++ Program Language or with Fortran Language.

In CUDA programming model, CPU is the host, while GPUs are co-processors or devices, one host can have several devices. Serial sections and more logical sections are dealt with on CPU, while highly threaded parallel tasks are coded as kernels running on GPUs (Fig. 1). Both CPU and GPUs maintain their own memory, which allows running these two separate sections of codes simultaneously without the concern for memory collisions.

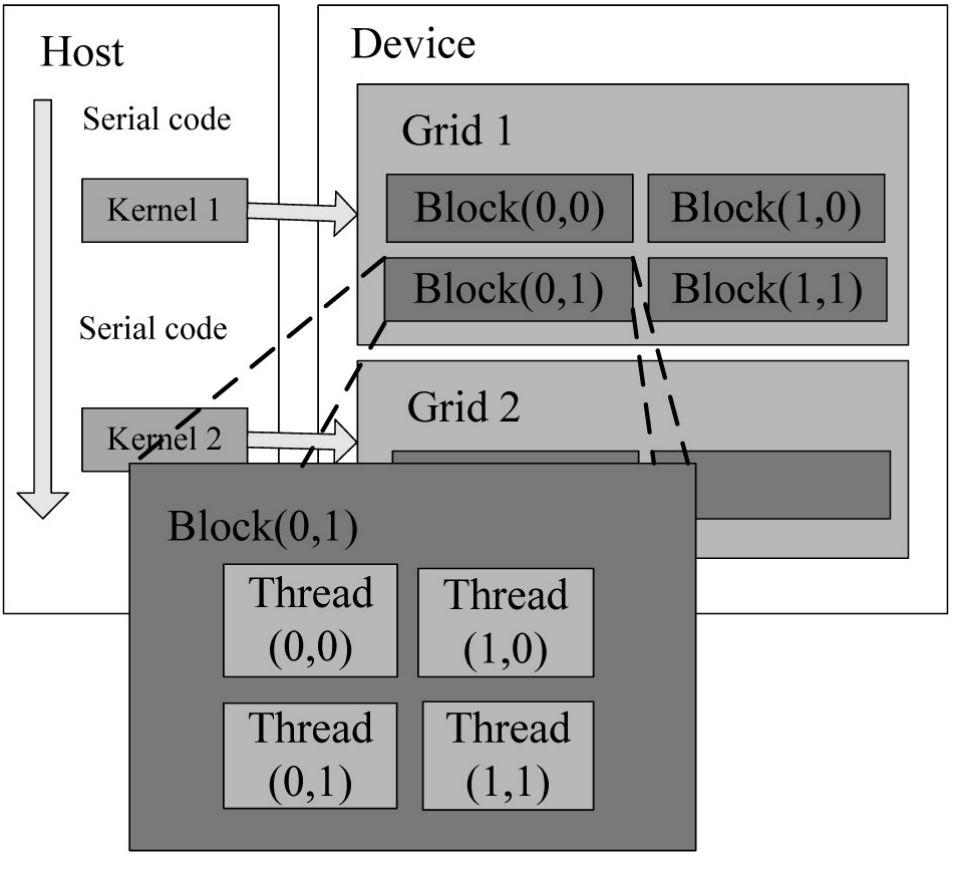

Fig. 1. Schematic of the GPU architecture. The left is the CPU host, which is used to deal with serial sections and more logical sections; the right is the GPU devices, which are used to deal with highly parallel tasks coded as kernels. The three-level parallel hierarchy in GPU architecture are thread, block and grid.Several threads are organized into a one dimension or two dimension block, and several blocks are organized into a one dimension or two dimension grid.

Fermi architecture is the newest and the most advanced CUDA architecture. In Fermi architecture, CUDA programming model has three-level parallel hierarchy: thread, block and grid, which is also shown in Fig. 1. Thread is the smallest unit, several threads are organized into a one dimension or two dimension block, and several blocks are organized into a one dimension or two dimension grid.

CUDA program calls parallel kernels, a kernel executes in parallel across a set of parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks. The GPU instantiates a kernel program on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread local memory, inputs, and outputs.

A thread block is set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid.

A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory and synchronize between dependent kernel call. In the CUDA parallel programming model, each thread has a per-thread local memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in global memory space after kernel-wide global synchronization. Besides, all the thread blocks share constant memory and texture memory. Fig. 2 illustrates these different kinds of memory and their subordination. Each kind of memory has its own characteristic, and can be used for program optimization.

There are some limits to the number of threads per block and the number of blocks per grid. To most of GPUs, threads within a block should be less than 1024, and blocks within a grid should be less than 65535.

ACCELERATING WITH MULTI-GPU

FWI in the time domain is very computationally expensive because of two aspects reasons. The first is the large number of shots needed to be simulated. In FWI, we always need to simulate tens of, even hundreds of shots to get a better inversion result. The second reason is the thousands upon thousands propagating procedures needed to be done. For each shot gather, in each iteration step, at least three propagation simulations must be done: one for the calculation of the forward wavefields, one for the backward wavefields, and at least one for the calculation of the step length. Our goal is to accelerate the

implementation of FWI in the time domain with multiple GPUs, and we will consider the two aspects at the same time, that is, we will accelerate the forward modeling procedures and, meanwhile, reduce the number of shots.

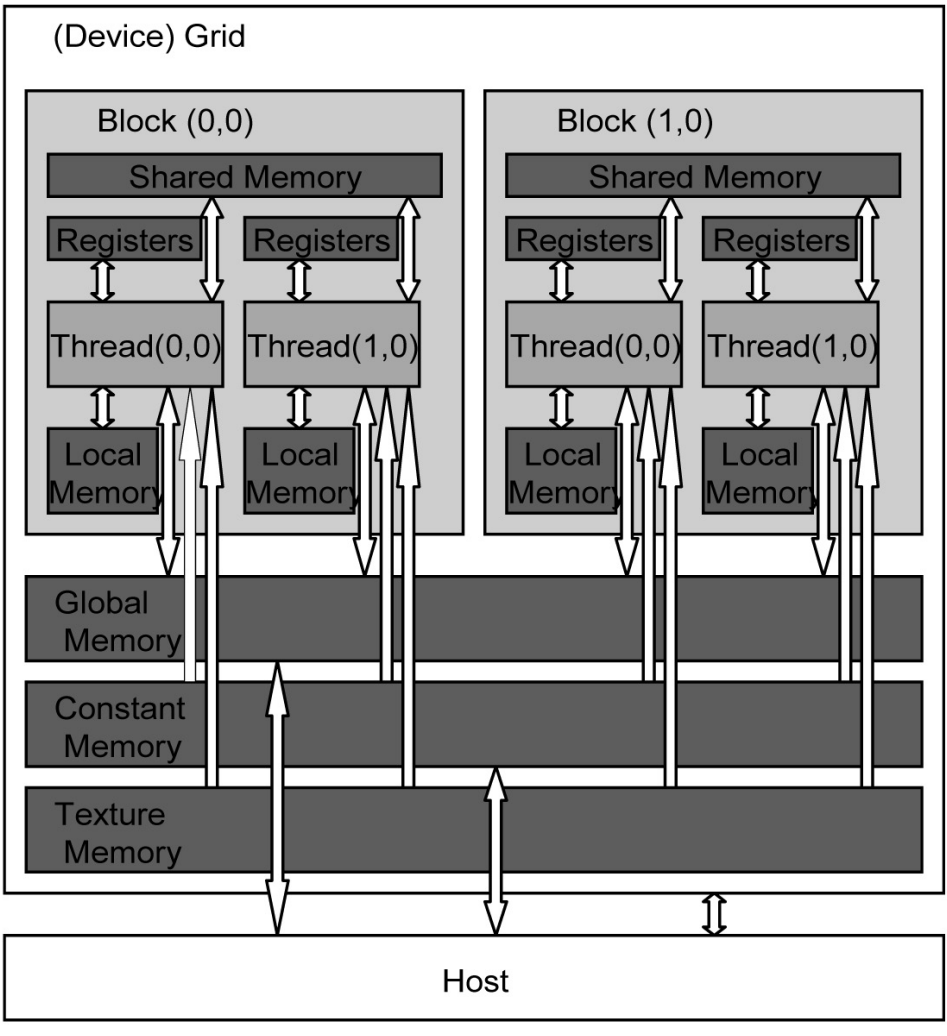

Fig. 2. Different kinds of memory in a GPU grid. Each thread has its own register memory and local memory, threads within a block can cooperate by sharing data through shared memory, and all the blocks share global memory, constant memory and texture memory.

Accelerating the forward modeling procedure

Forward modeling implementation

We implement eq. (3) using staggered-grid finite difference, and the wave equation in formula (3) can be organized as:

$$\begin{cases} [\partial d(\mathbf{x},t)/\partial t] = v^2(\mathbf{x})\{[\partial p(\mathbf{x},t)/\partial x] + [\partial q(\mathbf{x},t)/\partial z]\} \\ [\partial p(\mathbf{x},t)/\partial t] = [\partial d(\mathbf{x},t)/\partial x] \\ [\partial q(\mathbf{x},t)/\partial t] = [\partial d(\mathbf{x},t)/\partial z] \end{cases}, \tag{6}$$

where $p(\mathbf{x},t)$ and $q(\mathbf{x},t)$ are introduced for the implementation of the staggered-grid finite difference. The formula above can be discretized as:

$$\begin{cases} d_{i,j}^{t+1} = d_{i,j}^{t} + \Delta t \cdot v_{i,j}^2 \{[(p_{i+1,j}^{t+1} - p_{i,j}^{t+1})/\Delta x] + [(q_{i+1,j}^{t+1} - q_{i,j}^{t+1})/\Delta z] \\ p_{i,j}^{t+1} = p_{i,j}^{t} + \Delta t \cdot [(d_{i,j}^{t} - d_{i-1,j}^{t})/\Delta x] \\ q_{i,j}^{t+1} = q_{i,j}^{t} + \Delta t \cdot [(d_{i,j}^{t} - d_{i-1,j}^{t})/\Delta z] \end{cases}, \tag{7}$$

where Δx and Δz represent the space sampling intervals, Δt represents the time sampling interval, the subscripts i and j and represent the location of the grid points in space, and the superscript t represent the time step.

Accelerating the forward modeling using GPU

From eq. (7) we can see that the calculations of the wavefield cells at each time step are actually independent, the calculation of the wavefield for one cell at a time step needs only the wavefields of the last time step of its neighboring cells. So all these cells can be calculated simultaneously as long as we have enough concurrent execution units. From previous sections we know that the threads within several blocks can execute parallelly, so we can assign these calculations to the 2D threads of several 2D blocks, and the wavefield propagation can be implemented parallelly.

Fig. 3 shows the schematic of the forward modeling implementation in a GPU grid. The 2D threads of many 2D blocks in a GPU grid form a grid of 2D threads, which correspond to the 2D space in the earth, and each thread corresponds to a wavefield cell in the 2D space. When implementing, every thread does the same operation as shown in the right side of Fig. 3 which is the same as shown in eq. (7). Because these threads can execute parallelly and correspond to different wavefield cells, so the whole wavefields can be

calculated parallelly, while using the CPU, these wavefield cells must be calculated one by one in sequence. Owning to the parallelism of these threads, the calculational cost of the forward modeling can be reduced.

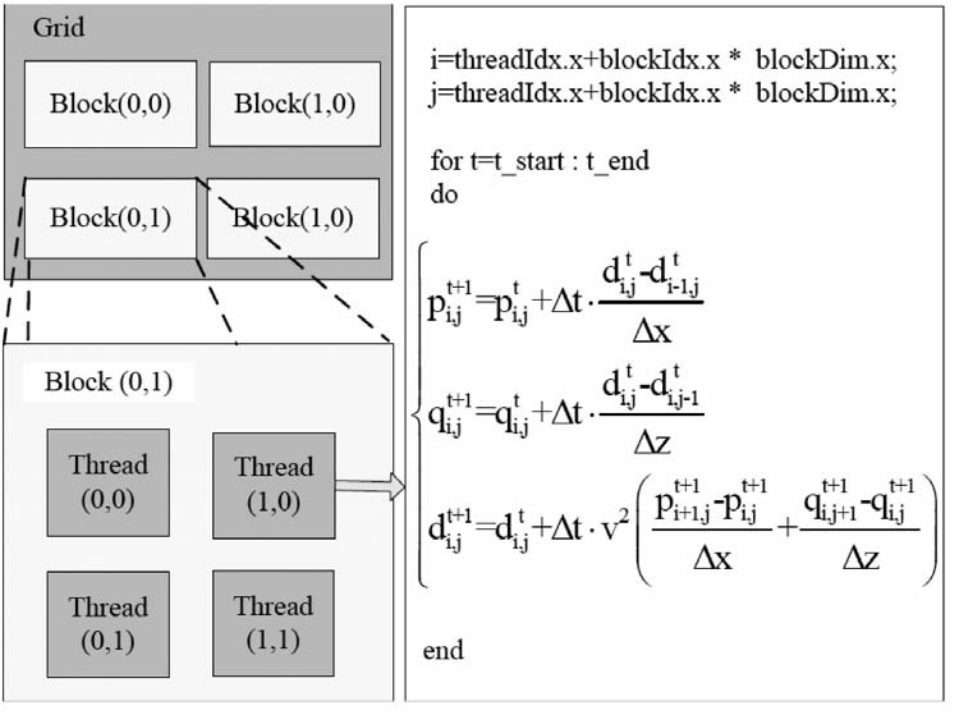

Fig. 3. Schematic of the forward modeling implemented in a GPU grid. The 2D threads correspond to the 2D space in the earth, and each thread corresponds to a wavefield cell in the 2D space. When implementing, every thread does the same operation as shown in the right, and all the wavefields can be calculated parallelly.

Boundary conditions and the storage strategy

In FWI, The gradient of the misfit function with respect to the velocity can be calculated by zero-lag correlation of the forward propagated source wavefields and the backward propagated residual wavefields. To do this correlation, we need the forward propagated wavefields in each time step while calculating the backpropagated wavefields. These forward propagated wavefields are always beyond the memory capacity so we can only first store them on the disk and then transfer them to the memory, which always takes some extra time. From eq. (7) we can see that the wavefield of each time step can be re-calculated by its next time step:

$$\begin{cases} d_{i,j}^{t-1} = d_{i,j}^{t} - \Delta t \cdot v_{i,j}^{2}\{[(p_{i+1,j}^{t} - p_{i,j}^{t})/\Delta x] + [(q_{i,j+1}^{t} - q_{i,j}^{t})/\Delta z] \\ p_{i,j}^{t-1} = p_{i,j}^{t} - \Delta t \cdot [(d_{i,j}^{t-1} - d_{i-1,j}^{t-1})/\Delta x] \\ q_{i,j}^{t-1} = q_{i,j}^{t} - \Delta t \cdot [(d_{i,j}^{t-1} - d_{i,j-1}^{t-1})/\Delta z] \end{cases}, \quad (8)$$

and we see that if we know the wavefield of the last time step and the boundary values of each time step, we can re-calculate the wavefields of all the other time steps. This re-calculation can be accelerated using GPU just as the forward modeling.

In this test, we used the PML boundary conditions (Komatitsch and Martin, 2007), and to avoid the tremendous storage requirement and the data transfer, we used the efficient boundary storage strategy proposed by Wang et al. (2012).

Reducing the number of shots: The two-level scheme

In FWI, we always need to simulate tens of, even hundreds of shot gathers, fortunately, the simulations of these different shot gathers are independent. When we implement the FWI on single-GPU system, these different shot gathers must be derived sequentially. But based on multi-GPU system, we can assign these many shots to multiple GPUs which can also parallelly executed, so the number of shots is reduced and the cost of FWI can be reduced further more.

So we present this two-level parallel scheme to speed up FWI with multiple GPUs. The first level parallelism is among multiple GPUs via MPI, this is the coarse-grained parallelism of the shot gathers, and the speedup is proportion to the number of shots; the second level parallelism is the parallelism of threads within each GPU grid via fermi CUDA, this is the fine-grained parallelism of the forward modeling procedures, the speedup is related to the GPU specifications and the design of the program, and can usually up to a magnitude of several tens. The flow chart of this scheme is shown in Fig. 4. From this two-level parallelism, we can get a speedup which is the product of those of the two levels individually, and the calculational cost can be reduced significantly.

Program optimization

In the implementation of this scheme, we optimize our program in all directions and try to get higher performance.

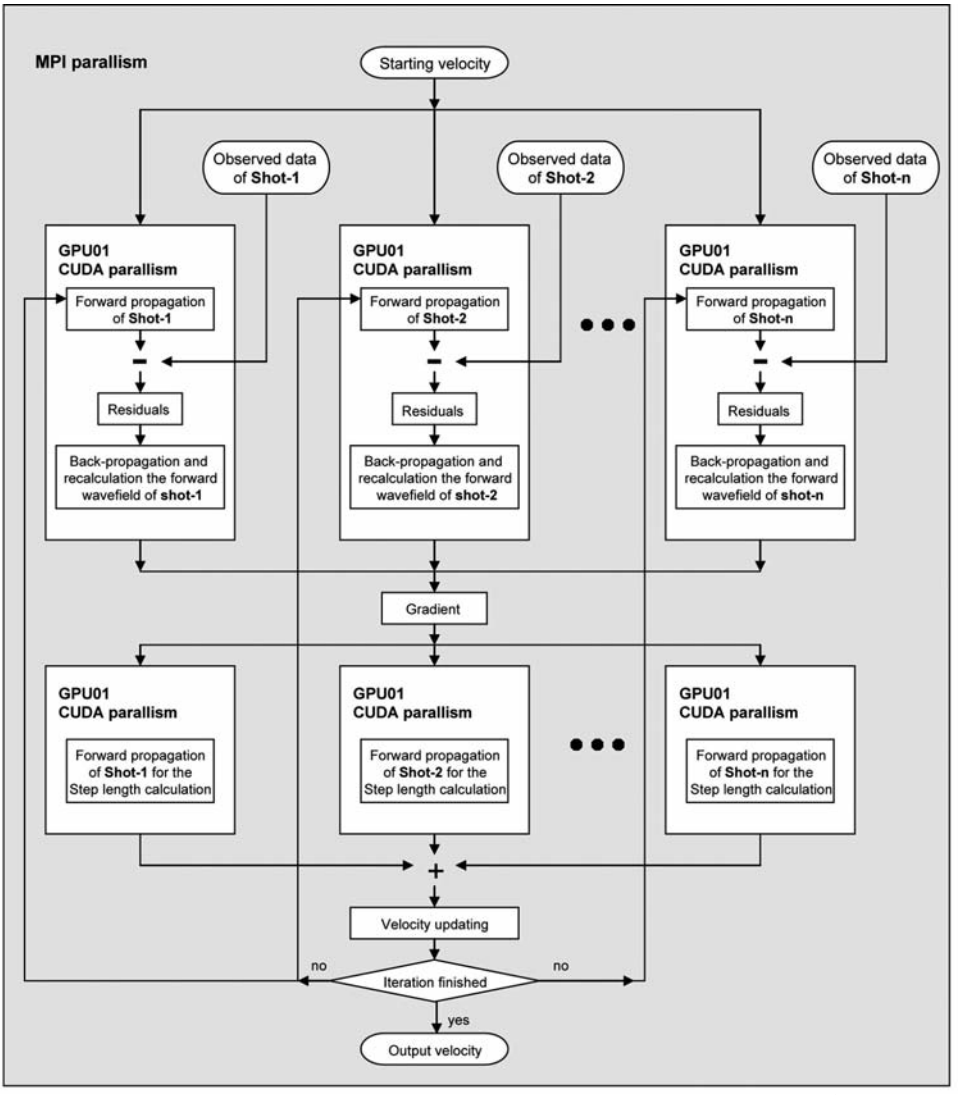

Fig. 4. Schematic of the two-level scheme. The first level parallelism is among multiple GPUs via MPI, this is the coarse-grained parallelism of the shot gathers; the second level parallelism is the parallelism of threads within each GPU grid via fermi CUDA, this is the fine-grained parallelism of the forward modeling procedures.

Using the characteristic of different kinds of memory

There are many different kinds of memory in a GPU gird, and each has its own characteristic, we can appropriately use them to optimize our program.

The bandwidth between device memory and device memory is much higher than that between device memory and host memory, so we should minimize the data transfer between host and device to save time. In our test, all the computations, including propagator and back-propagator, and the storage of PML boundary values, are coded as CUDA kernels executing on GPUs.

Shared memory has higher bandwidth than global memory, it buffers reside physically on the GPU as opposed to residing in off-chip DRAM, so the latency to access shared memory tends to be lower than global memory. We loaded the wavefield and the velocity model into shared memory to avoid the low-efficiency caused by accesses to global memory.

NVIDIA hardware provides 64 KB of constant memory which can be used for data that will not change over the course of a kernel execution. Using constant memory rather than global memory will reduce the required memory bandwidth. We loaded totally 15 constant variables, such as the size of the velocity model, the space sampling intervals and the time sampling interval into the constant memory.

Texture memory is another variety of read-only memory, it is cached on chip, so it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM. In this processing, we first calculated the vectors used for getting the PML boundary values, and then loaded them into the texture memory to avoid reading them from the global memory every time and to get higher performance.

Warps and occupancy

Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads, these threads are assigned to execution resources. In the current generation of hardware, the execution resources are organized into streaming multiprocessors (SMs). Once a block is assigned to a SM, it is further divided into 32-thread units called warps. Warps are not part of the CUDA specification, but is the unit of thread scheduling in SMs. So the number of threads per block should be a multiple of 32 to get higher performance. In our test, we used a two dimension block, the number of threads per block is 32 $\times$ 16.

Occupancy is a very important parameter in the use of GPUs, we can get a higher performance if the occupancy is high. GPU occupancy is related to some other parameters such as the threads used per block, the registers used per thread and the shared memory used per block, and we can calculate the occupancy value using CUDA Occupancy Calculator provided by NVIDIA. The CPU and GPU model we used in this study are listed in Table 1 and Table 2. The compute capability of our GPU model is 2.0, and the shared memory size is 48 Kb, that is 49152 bytes. We used 32×16, that is 512 threads per block, and 6 registers per thread, and the shared memory we used per block is 2048 bytes. With this, we can get a occupancy of 100 percent.

Table 1. CPU specifications.

CPU	Intel (R) Core (TM)2 Quad CPU Q9500
	2.83 GHz
	Memory 4 GB

Table 2. GPU specifications.

System	inspur TS10000
GPU Model	NVIDIA Tesla C2050
GPU Device Count	10
Global Memory	3GB GDDR5
Multiprocessors	14
Threads Per Block	1024
Shared Memory Per mp	48 Kb
Constant Memory	65 Kb
Registers Per mp	32768
Single-precision Float	1.03 Tflops
Double-precision Float	515 Gflops
Compute Capability	2.0

NUMERICAL EXAMPLE

We tested our parallel program implemented on GPUs, and compared the computational cost with that on CPU. We also compared the computational cost of the two-level scheme with schemes containing only one level (containing only the MPI parallelism or only the CUDA parallelism). The specifications of the CPU and GPUs used in this study are listed in Table 1 and Table 2.

We employ the Marmousi model as illustrated in Fig. 5 (a). This model has a grid size of 751 $\times$ 2301, the space sampling intervals are 4 m for Δx and Δz. In the following tests, we use a 10 Hz Ricker wavelet as the source signature and generate 3 s records with time sampling interval of 0.4 ms, that is 7500 time steps.

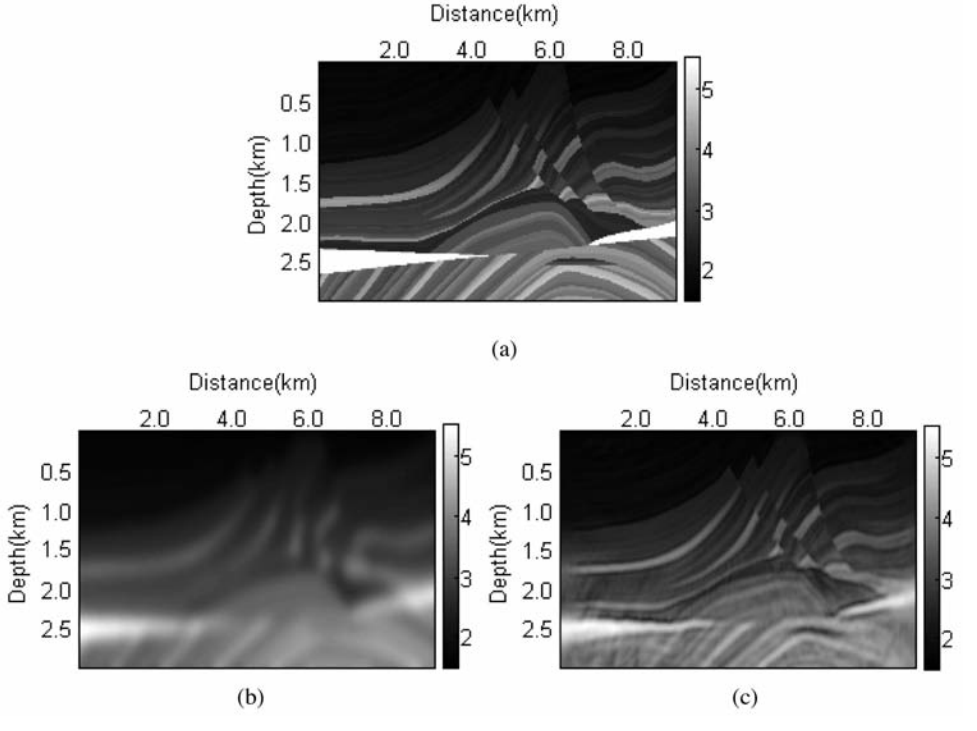

Fig. 5. The Marmousi velocity model. (a) The true velocity model; (b) the starting velocity model; (c) the inverted velocity model after 200 iterations.

Table 3. Comparison of computational cost of forward modeling (the speedup in this table are in respect to the cost on single CPU).

Model	Marmousi
Grid size	751 × 2301
Number of shots	10
Number of time steps	7500
Cost on single	16261.7 (s)
CPU Cost on 10 CPUs (MPI parallel)	1626.3 (s)
Cost on single GPU (CUDA parallel)	325.8 (s)
Cost on 10 GPUs (MPI+CUDA parallel)	32.6 (s)
Speedup of MPI parallel	9.9
Speedup of CUDA parallel	49.9
Speedup of two-level parallel	498.9

Propagating procedure testing

We first test the computational cost for the propagating procedure. We simulated a forward modeling with 10 shots using the Marmousi velocity model, we implement the simulating on CPU (no parallelism), multi-CPUs (MPI parallelism only), single-GPU (CUDA parallelism only) and multi-GPUs (two-level parallelism). The processing time and the speedup with respect to single CPU are listed in Table 3. From the result we can see that, the MPI parallelism only scheme can get a speedup of 9.9, and the CUDA parallelism only scheme can get a speedup of 49.9, while using the two-level scheme, we can get a speedup of 498.9, which is almost the product of those of the two levels individually.

FWI implementation testing

To examine the multi-GPU based FWI method, we put 20 shots in space and 380 receivers in space located on the surface of the Marmousi model. Fig. 5(b) shows the initial velocity model which is derived by smoothing the Marmousi velocity model using a Gaussian filter.

We implement the FWI on the Inspur TS10000 system which has 10 GPUs, so each GPU grid need to simulate 2 shot gathers, that is 1/10 of the initial number of shot gathers. We also implement the FWI using 5 GPU grids and 2 GPU grids, respectively, in these cases, each GPU grid is used to simulate 4 shot gathers and 10 shot gathers respectively (because the computation time on CPU is extremely large, so we did not implement the FWI on CPU platform). The computation time is listed in Table 4.

Table 4. Computational cost of FWI using the two-level scheme.

Model	Marmousi
Grid size	751 × 2301
Number of shots	20
Number of time steps	7500
Cost of one iteration on 2 GPUs	1404.1 (s)
Cost of one iteration on 5 GPUs	561.2 (s)
Cost of one iteration on 10 GPUs	280.8 (s)
Total cost on 2 GPUs	78.0 (hours)
Total cost on 5 GPUs	31.2 (hours)
Total cost on 10 GPUs	15.5 (hours)

The computation time for one iteration on 2 GPUs is 1404.1 s, on 5 GPUs is 561.2 s, and on 10 GPUs is 280.8 s. We get the final result shown in Fig. 5(c) after 200 iterations, and it costs 78 hours for 2 GPUs, 31.2 hours for 5 GPUs, and 15.5 hours for 10 GPUs. We can imagine that if we have more GPU grids, then the computation time can be further reduced.

CONCLUSION

In this study, we have presented a fast two-stage parallel scheme based on multi-GPU system to speed up FWI in the time domain. In this scheme, the PML boundary condition and the efficient boundary storage strategy are used to avoid the tremendous storage requirement needed on the disk and the data

transfer between disk and memory. In our program, all the calculations, including the forward propagating, backward propagating and boundary storage are coded as kernels executed on GPUs to avoid the low-bandwidth data transfer between host and devices, we also utilized the higher bandwidth of the shared memory, constant memory and texture memory.

The proposed two-level multi-GPU based scheme can speed up FWI over the CPU-based implementation by 498.9 times, which is a product of the CUDA-based only scheme and the MPI-based only scheme. So the turnaround time for inverting the velocity model is reduced significantly and researchers can do their work with a much shortened research cycle.

ACKNOWLEDGEMENTS

This work was supported by the National Engineering Laboratory On Offshore Exploration, the National Natural Science Foundation of China (40730424), and the National Science & Technology Major Project (2011ZX05023-005).

REFERENCES

Ben-Hadj-Ali, H., Operto, S. and Virieux, J., 2009. Three-dimensional frequency-domain full waveform inversion with phase encoding. Expanded Abstr., 79th Ann. Internat. SEG Mtg., Houston: 2288-2290.

Freudenreich, Y. and Shipp, T., 2000. Full waveform inversion of seismic data. frequency versus time domain. Lithos Science Rep., 2: 25-30.

Kadlec, B.J. and Dorn, G.A., 2010. Leveraging graphics processing units (GPUs) for real-time seismic interpretation. High-performance computing. The Leading Edge, 29: 60-66.

Kirk, D.B. and Hwu, W.W., 2010. Programming Massively Parallel Processors: a Hands-on Approach. Elsevier Science Publishers, Amsterdam.

Komatitsch, D. and Martin, T., 2007. An unsplit convolutional perfectly matched layer improved at grazing incidence for the seismic wave equation. Geophysics, 72: SM155-SM167.

Krebs, J.R., Anderson, J.E., Hinkley, D., Neelamani, R., Lee, S., Baumstein, A. and Lacasse, M.-D., 2009. Fast full-wavefield seismic inversion using encoded sources. Geophysics, 74: WCC177-WCC188.

Lailly, P., 1983. The seismic inverse problem as a sequence of before stack migrations. In: Bednar, J., Robinson, E. and Weglein, A., (Eds.), Inverse Scattering Theory and Application. Soc. Industr. Appl. Mathemat. (SIAM), Philadelphia: 206-220.

Li, B., Liu, G.F. and Liu, H., 2009. A method of using GPU to accelerate seismic pre-stack time migration. Chin. J. Geophys. [in Chinese], 52: 245-252.

Mora, P., 1987. Nonlinear two-dimensional elastic inversion of multi-offset seismic data. Geophysics, 52: 1211-1228.

Mora, P., 1998. Elastic wave-field inversion of reflection and transmission data. Geophysics, 53: 750-759.

Operto, S., Virieux, A.J., l'Excellent, J., Giraud, L. and Ali, H.B.H., 2007. 3D finite-difference frequency-domain modeling of viscoacoustic wave propagation using a massively parallel direct solver: A feasibility study. Geophysics, 72: SM195-SM211.

Pratt, R.G., 1999. Seismic waveform inversion in the frequency domain, Part 1: Theory and verification in a physical scale model. Geophysics, 64: 888-901. Pratt, R.G., Shin, C.S. and Hicks, G.J., 1998. Gauss-Newton and full Newton methods in frequency- space seismic waveform inversion. Geophys. J. Internat., 133: 341-362.

Pratt, R.G. and Shipp, R.M., 1999. Seismic waveform inversion in the frequency domain, Part 2: Fault delineation in sediments using crosshole data. Geophysics, 64: 902-914.

Sanders, J. and Kandrot, E., 2010. Cuda by example: an introduction to general-purpose GPU program- ng. Addison-Wesley, Boston, MA.

Tarantola, A., 1984. Inversion of seismic reflection data in the acoustic approximation. Geophysics, 49: 1259-1266.

Tarantola, A., 1987. Inverse Problem Theory: Methods for data fitting and model parameter estimation. Elsevier Science Publishers, Inc., New York

Vigh, D. and Starr, E.W., 2008. 3D prestack plane-wave, full-waveform inversion. Geophysics, 73: VE135-VE144.

Wang, B. and Gao, J., 2010. Fast full waveform inversion of multi-shot seismic data. Expanded Abstr., 80th Ann. Internat. SEG Mtg., Denver: 1055-1058.

Wang, B. and Gao, J., 2011. Cuda-based acceleration of full waveform inversion on GPU. Expanded Abstr., 81st Ann. Internat. SEG Mtg., San Antonio: 2528-2533.

Wang, B., Gao, J., Chen, W. and Zhang, H., 2012. Efficient boundary storage strategies for seismic reverse time migration. Submitted to Chin. J. Geophys. [in Chinese].

Zhang, J.-H., Wang, S.-Q. and Yao, Z.-X., 2009. Accelerating 3D Fourier migration with graphics processing units. Geophysics, 74: WCA129-WCA139.