CSIRO PUBLISHING www.publish.csiro.au/journals/eg Copublished paper, please cite all three:
Exploration Geophysics, 2011, **42**, 98–104; *Butsuri-Tansa*, 2011, **64**, 98–104; *Jigu-Mulli-wa-Mulli-Tamsa*, 2011, **14**, 98–104

Acceleration of computation speed for elastic wave simulation using a Graphic Processing Unit

Norimitsu Nakata[1,2] Takeshi Tsuji[1] Toshifumi Matsuoka[1]

[1]Department of Urban Management, Faculty of Engineering, Kyoto University, C1-118, Kyotodaigaku-Katsura, Nishikyo-ku, Kyoto 615-8540, Japan.
[2]Corresponding author. Email: n_nakata@earth.kumst.kyoto-u.ac.jp

Abstract. Numerical simulation in exploration geophysics provides important insights into subsurface wave propagation phenomena. Although elastic wave simulations take longer to compute than acoustic simulations, an elastic simulator can construct more realistic wavefields including shear components. Therefore, it is suitable for exploration of the responses of elastic bodies. To overcome the long duration of the calculations, we use a Graphic Processing Unit (GPU) to accelerate the elastic wave simulation. Because a GPU has many processors and a wide memory bandwidth, we can use it in a parallelised computing architecture. The GPU board used in this study is an NVIDIA Tesla C1060, which has 240 processors and a 102 GB/s memory bandwidth. Despite the availability of a parallel computing architecture (CUDA), developed by NVIDIA, we must optimise the usage of the different types of memory on the GPU device, and the sequence of calculations, to obtain a significant speedup of the computation. In this study, we simulate two- (2D) and three-dimensional (3D) elastic wave propagation using the Finite-Difference Time-Domain (FDTD) method on GPUs. In the wave propagation simulation, we adopt the staggered-grid method, which is one of the conventional FD schemes, since this method can achieve sufficient accuracy for use in numerical modelling in geophysics. Our simulator optimises the usage of memory on the GPU device to reduce data access times, and uses faster memory as much as possible. This is a key factor in GPU computing. By using one GPU device and optimising its memory usage, we improved the computation time by more than 14 times in the 2D simulation, and over six times in the 3D simulation, compared with one CPU. Furthermore, by using three GPUs, we succeeded in accelerating the 3D simulation 10 times.

Key words: CUDA, elastic wave propagation, Graphic Processing Unit, seismic modelling, 3D Finite-difference.

Introduction

In seismic exploration, the simulation of wave propagation is a useful tool for understanding wave phenomena in subsurface structures (e.g. diffraction patterns, refractions and reflections from boundaries). We construct a structure model from a priori geological information and simulate wave propagation to understand the seismic wave responses (e.g. Juhlin, 1995). Applications of simulations are evident in various fields such as the design of receiver arrays or source points for optimal acquisition geometry, inversion, migration, confirming the effect of noise, and verifying the suitability of new data processing methods.

When we simulate wave phenomena in three-dimensional (3D) structures, with the construction of many shot records, we usually use a ray tracing method to reduce the computation time. In ray tracing, we assume a high-frequency wave and estimate wave propagation ray paths through the model (Cerveny, 2001). Because the algorithm for this simulation is not expensive in terms of computational resources, it is useful for practical applications. Ray tracing cannot, however, construct all wavefields (e.g. surface waves) together with frequency variations.

Wave theory modelling, such as the Finite-Difference Time-Domain (FDTD) method, was developed to enable the construction of all wavefields including surface waves, diffractions, and multiple scattered waves (Virieux, 1986). Although the algorithm for FDTD is not complicated, it requires more computation power than does ray tracing simulation. Despite the recent rapid improvement in computer performance, it is difficult to complete the simulation of a large 3D subsurface model that might be required in normal geophysical exploration within a practicable computational time whilst also retaining sufficient accuracy.

In this paper, we propose algorithms to accelerate the calculation speed by using a Graphic Processing Unit (GPU) via CUDA, which is the computing architecture in NVIDIA GPUs (Owens et al., 2008; Nickolls and Dally, 2010). The GPU is usually mounted as a peripheral in a personal computer to handle large volumes of graphical data, because graphical data processing requires special algorithms. Recently, however, several scientific researchers have adopted this specific device as a scientific computational tool. The main applications of GPU computing in scientific fields are in computational chemistry (Stone et al., 2007), computational fluid dynamics (Liu et al., 2004), and astrophysics (Nyland et al., 2007). In this paper, we propose a specialised parallelisation algorithm for a GPU, for 2D and 3D elastic wave simulations using the FDTD method for geophysical applications.

A wave simulation usually solves the wave equation at each time step. Acoustic and electromagnetic wave simulations on a GPU have already been proposed by several authors (Takada et al., 2008; Micikevicius, 2009). Although acoustic wave simulation is useful for reverse time migration (Abdelkhalek et al., 2009; Moussa, 2009), elastic wave simulation is necessary to process many types of geophysical data such as P-SV converted waves, or to estimate the exact

© ASEG/SEGJ/KSEG 2011 10.1071/EG10039 0812-3985/11/010098

reflection coefficients. Thus, the aim of this paper is to develop algorithms to decrease the computation time whilst maintaining high accuracy in elastic wave simulation. The characteristics of GPU computing are high performance due to being capable of ~10 times more floating-point operations per second (flops) than a CPU, and having 12 times more memory bandwidth than the system memory on the motherboard, as well as conservation of space and electric power, and low cost. The wide memory bandwidth becomes a key factor in the FDTD program, which needs to read and write large amounts of data.

A FDTD scheme using a staggered grid on a GPU

The hardware implementation of a GPU is quite different from that of a CPU. A GPU has several hundred processors and several different kinds of memory in the form of device memory, shared memory, constant memory, registers, etc., which differ in both size and bandwidth (Figure 1). We can implement parallel computing using the several hundred processors. All processors in a GPU are basically designed to execute the same code, so GPU computing can accelerate simple iterative calculations effectively. Because a GPU cannot directly read the system memory mounted on the CPU motherboard, we must transfer data between the system memory and device memory on the GPU board. It is better to reduce the number of transfers of data for GPU computing. Device memory on the GPU board has a capacity of a few gigabytes, but the speed of a memory access is not very fast. However, the other three kinds of memory on the GPU chip, that is shared memory, constant memory, and registers, are much faster and can be used similarly to the cache memory of a CPU. GPU programming differs from CPU programming in that it is necessary to declare how much on-chip memory will be occupied by the program. Effective usage of the different types of memory, especially shared memory, is the key factor in accelerating processes. Because the on-chip memory is quite fast, yet small, we must evaluate how much data can be stored in the memory. To do this, we must understand the architecture of the GPU and optimise our code for the specific use of the GPU hardware.

The bottleneck in wave propagation simulation is usually the speed of a memory access, not the operation speed of the processors. Thus, optimising memory access time is the key to decreasing computation time. The flow of data in a GPU computation involves first copying subsurface structure data from system memory to device memory and then from device memory to the on-chip memory. These data can be processed in parallel on the GPU device and the computational results are then transferred to system memory.

In CUDA programming, the CPU acts as a host that controls the GPU, whilst the GPU acts as a device. We need to write two different programs, the CPU (host) code and GPU (device) code. The code for these programs is similar to C programming codes. The host code calls kernel functions, in the device code, thus using the GPU. The concept of GPU programming is shown in Figure 2. One kernel function constructs a single GPU-grid that includes several blocks. A single GPU-grid runs on one GPU board, whilst one block corresponds to the calculation range of one multiprocessor, which has one shared memory. A thread is the execution unit in the device code. Therefore, it is the fastest if the number of blocks is a multiple of the number of multiprocessors, and one thread is a multiple of the number of processors in one multiprocessor. We can implement parallelisation by executing the program simultaneously on several hundred processors; the NVIDIA Tesla C1060 board used in this study has 240 processors. It is, however, difficult to speed up all kinds of programs because of the low clock speed in each processor and the additional memory copy time. We adapt our code for CUDA to speed up the computation time.

In this paper, we propose 2D and 3D elastic wave simulators using the FDTD method. Our FDTD scheme solves the stress-strain relation and equation of motion instead of the wave equation (Graves, 1996). We adopt a staggered-grid method with fourth-order accuracy in the space domain and second-order accuracy in the time domain. We are able to calculate the waveform with higher accuracy by taking higher-order approximations, either eighth or sixteenth order, of the differentiation operators, although using these operators involves a trade-off between computation speed and FD time interval. In GPU computing, we have a small memory capacity but high computation power. Therefore, fourth-order accuracy, a small time interval, and many iterations are appropriate for the GPU calculation. We simulate a semi-infinite space and apply a free-surface boundary condition (Levander, 1988) at the upper boundary of the subsurface model and a non-reflecting boundary

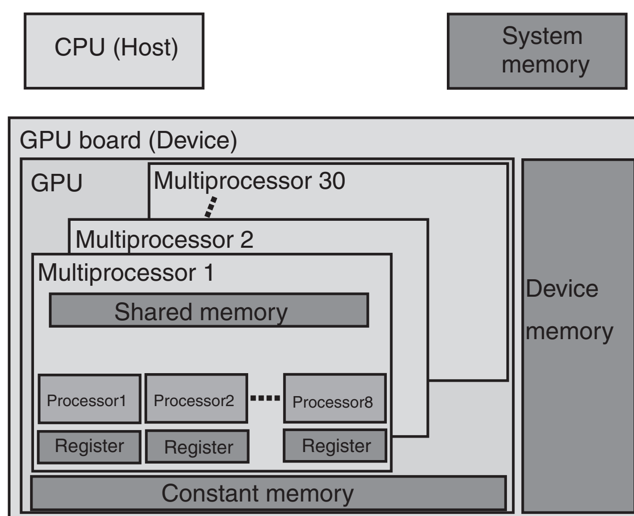

Fig. 1. Schematic diagram of the hardware implementation of a GPU. The GPU is the computation unit on a GPU board and includes processors and memory. Device memory can be seen outside the GPU (but on the GPU board). The numbers of multiprocessors and processors are as for the NVIDIA Tesla C1060.

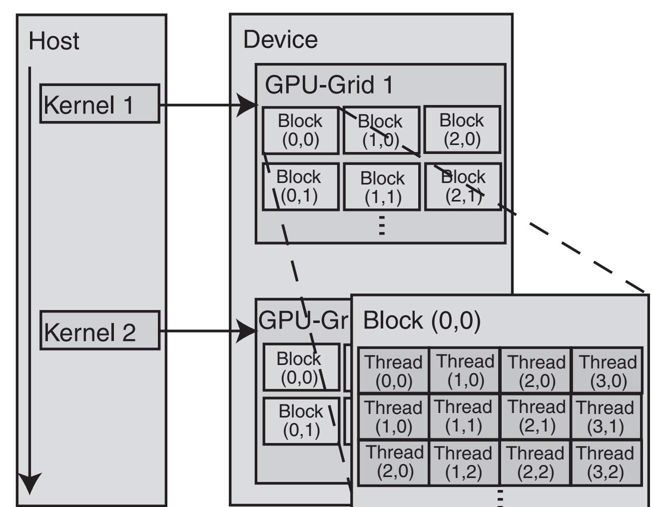

Fig. 2. The concept of GPU programming. One thread is executed on one processor, with one block input into one multiprocessor.

condition (Cerjan et al., 1985) at the remaining five boundaries in the 3D case. In a 3D elastic simulation, the FDTD scheme requires nine variables at each grid point: three particle velocity values (*x*, *y*, and *z* components) and six stress values (*xx*, *xy*, *xz*, *yy*, *yz*, and *zz* components). In addition, the subsurface model property parameters are required: density and two Lame constants.

The NVIDIA Tesla C1060 has 4 GB of device memory, which has 102 GB/s memory bandwidth, and 16 KB each of shared memory; device memory restricts the simulation size of the GPU-grid, and shared memory restricts that of each block, respectively (Figure 2). A processor can use only same multiprocessor's shared memory. Because we use single precision in this study, which is limited by the processors on the GPU, a single GPU has a maximum grid size of ~500 $\times$ 500 $\times$ 300 grids.

To obtain high performance computation, it is beneficial to know the maximum data size in shared memory. Because 12 variables (nine variables and three model parameters) are specified on each grid, we can store only 341 grid data in one shared memory of 16 KB. In our FD scheme, we require values from 13 grids to evaluate one data point (Figure 3*a*). If we calculate the next time step for all values of shared memory, we must use device memory, because the edge of two grids in shared memory requires values from outside the shared memory, which is device memory. Device memory is, however, much slower than shared memory, and therefore it is better to complete the calculation using only the shared memory data. To overcome this performance issue, we input two surplus grids (wing area) in each direction into shared memory; the shape of input data is shown in Figure 3*b*. Based on Figure 3*b*, we can calculate the next time step value in 4 $\times$ 4 $\times$ 4 grids for each block. Because synchronisation between blocks is not possible with the latest release of the CUDA, we close the kernel at every time step and share the data between blocks.

Simulation results

Single-GPU

Computation times are evaluated to compare the effectiveness of shared memory usage in a 2D simulation. Test scenarios include using only device memory (Case A), using both device and shared memory (Case B) without wing area data in the shared memory, and using only shared memory (Case C) with wing area data for calculating each time step. These scenarios also differ with respect to the number of accesses to device memory. In Case A device memory is accessed at each calculation step, in Case B this memory is accessed when we calculate the edge of the grid in shared memory, whilst in Case C it is accessed twice at the beginning and end of each time step. Table 1 gives the computation times for a single CPU, an Intel Core i7 920, which is optimised by Open Multi-Processing (OpenMP; Chandra et al., 2000), whilst Table 2 gives the computation times for a single GPU and the acceleration ratio. We have also compared the effect of using different block sizes, the size of data stored in shared memory. Both Tables 1 and 2 give data for 1000 time steps. One block is calculated in one multiprocessor, and the GPU iterates calculations until calculation of the whole area is finished. We show the computation time of each case based only on the fastest block size (Figure 4). Case C with an 8 $\times$ 8 block size is the fastest and it executes 14 times faster than on the CPU. Thus, we should reduce the number of accesses to device memory and search for the optimal block size.

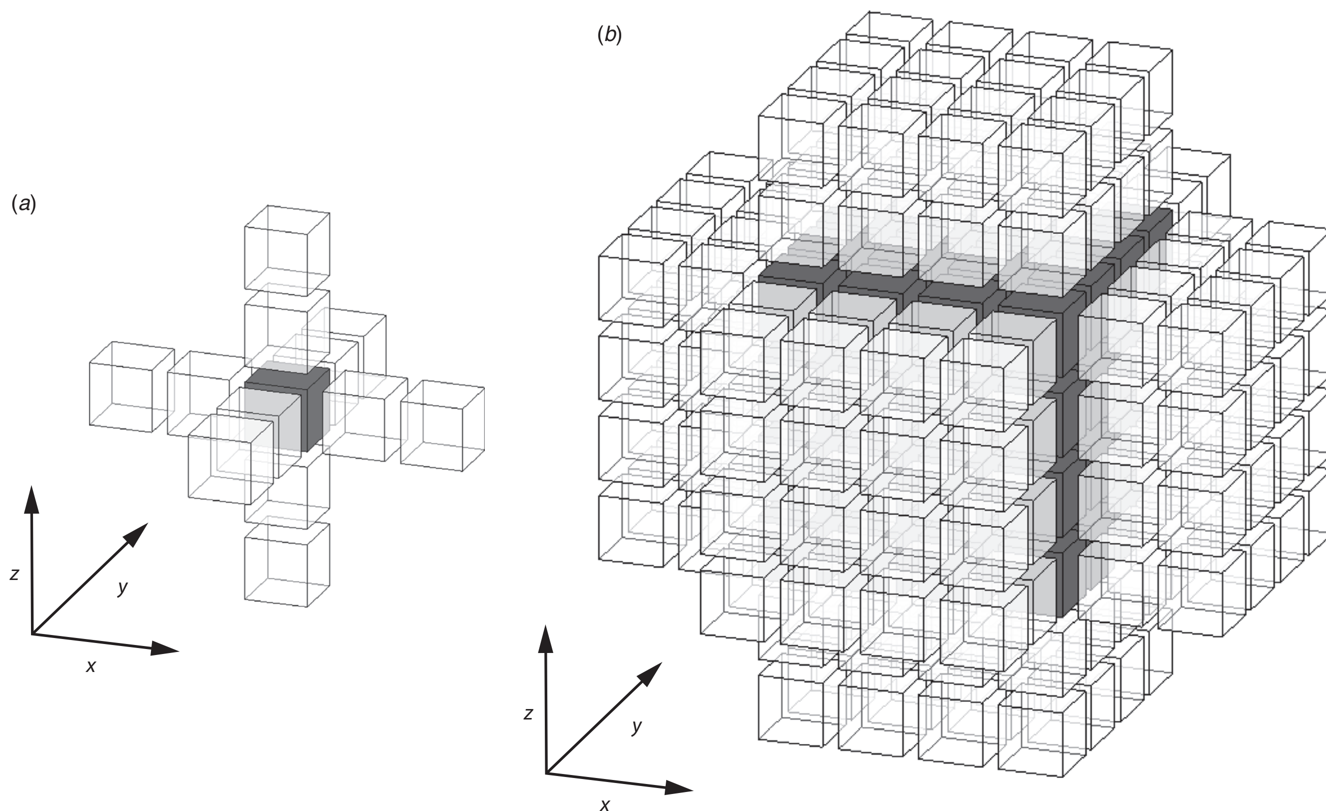

Fig. 3. (*a*) Thirteen grid positions (white and grey cubes) needed for computing the single grid (grey cube) at the centre in the next step. (*b*) Grids copied to shared memory, including the wing area. White cubes depict the wing area whilst grey cubes show the calculation volume for subsequent time steps. Block size is 4 $\times$ 4 $\times$ 4. The number of grids in the wing is three times the number in the calculation volume.

Table 1. Computation time for 2D elastic wave propagation simulation on CPU.

Number of grids	480 × 480	960 × 960	1440 × 1440	1920 × 1920
Computation time (s)	24.08	90.44	189.10	324.90

Table 2. Computation time for 2D elastic wave propagation simulation on GPU.
Case A uses only device memory, Case B uses both device and shared memory, whilst Case C uses only shared memory. The value in parentheses denotes the acceleration ratio between GPU and CPU computation. The value in bold typeface represents the fastest time in each case.

Number of grids	Case	Block size 4 × 4	8 × 8	16 × 16
480 × 480	A	3.15 (7.6×)	4.59 (5.3×)	7.34 (3.3×)
	B	6.59 (3.7×)	3.2 (7.5×)	3.82 (8.5×)
	C	3.13 (7.7×)	2.57 (9.4×)	**2.46 (9.8×)**
960 × 960	A	13.50 (5.4×)	16.13 (5.6×)	32.04 (2.8×)
	B	25.28 (3.6×)	12.56 (7.2×)	10.08 (9.0×)
	C	11.99 (7.5×)	**6.95 (13.0×)**	9.98 (9.1×)
1440 × 1440	A	35.10 (5.4×)	44.26 (4.3×)	88.72 (2.1×)
	B	56.79 (3.3×)	31.20 (6.1×)	22.84 (8.3×)
	C	29.32 (6.4×)	**14.38 (13.2×)**	23.56 (8.0×)
1920 × 1920	A	71.89 (4.5×)	67.05 (4.8×)	195.00 (1.7×)
	B	100.00 (3.3×)	57.52 (5.6×)	40.72 (8.0×)
	C	54.95 (5.9×)	**22.72 (14.3×)**	41.31 (7.9×)

In the same manner, that is, using shared memory as in Case C, the computation time for 3D elastic wave propagation is evaluated. The fastest block size, $4 \times 4 \times 4$, is used (Table 3), and we calculate 1000 time steps. According to these results, we can achieve a six times increase in computation speed over the CPU computation. Snapshots of the 3D elastic wave propagation simulation are shown in Figure 5. The model size comprises $500 \times 500 \times 250$ grids, that is, $2.5 \times 2.5 \times 1.25$ km, and absorbing boundaries, which are 20 grids deep each. The *X*- and *Y*-directions denote the horizontal whilst the *Z*-direction denotes the vertical: with $Z = 0$ corresponding to a free surface. The source is a Ricker Wavelet whose peak frequency is 7.5 Hz, at the centre of the free surface. We use a flat horizontal two-layered model, with the horizontal boundary at $Z = 120$. We can observe the reflected waves from this horizon; reflected P- and S-waves can be seen in Figure 5 in panels (*d*) and (*e*), respectively. We compare this result and the analytic solution of the *Z*-component displacement to confirm the suitability of this simulator (Figure 6). The analytic solution was derived by Saito (1993),

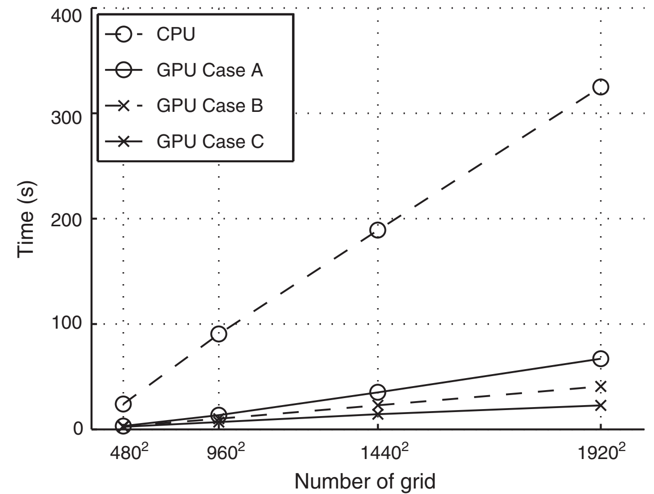

Fig. 4. Computation time for 2D elastic wave propagation simulation. CPU (dashed line and circle); GPU Case A, using only device memory, (solid line and circle); GPU Case B, using both device and shared memory, (dashed line and cross); and GPU Case C, using only shared memory, (solid line and cross). The values in this plot are shown in Table 2. The fastest time for each grid and scenario is chosen without considering block size.

Table 3. Computation time for 3D elastic wave propagation simulation.

Number of grids	96 × 96 × 96	192 × 192 × 192	288 × 288 × 288	384 × 384 × 384
CPU (s)	220.04	1737.70	8603.40	14790.00
CPU (s)	46.11	398.22	1378.70	2628.90
Accelerating ratio	4.8×	4.4×	6.2×	5.6×

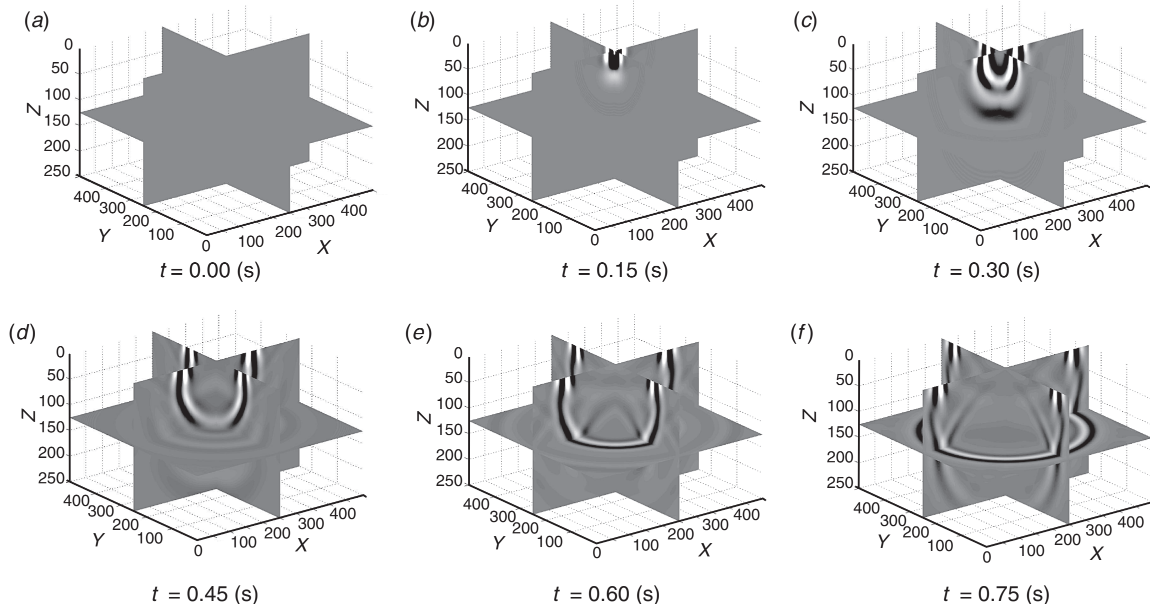

Fig. 5. Snapshots of 3D elastic propagation simulation at 0.15 s intervals. We can see the reflected P-wave at $t = 0.45$ s and S-wave at $t = 0.60$ s from the horizon at $Z = 120$.

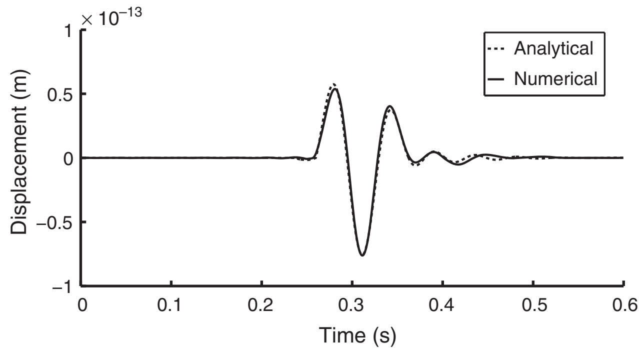

Fig. 6. Comparison of the simulated *Z*-component displacement with the analytic solution. Dashed line depicts the analytical result, whilst solid line shows the simulation value. The distance between source and receiver is 400 m.

who solved Lamb's problem in the frequency domain. The estimated error between the simulation result and the analytical solution gives the accuracy of the simulator. We define the error as follows:

$$E = \frac{\sum_{i=t}(u_i - u_i^a)^2}{\sum_{i=t}(u_i^a)^2}, \qquad (1)$$

where u_i is the displacement calculated by this simulator, and u_i^a is the value of the analytical solution. The error for Figure 6 is 2.2%, which shows that our simulator executing on a GPU has similar accuracy as our CPU simulation and is therefore useful for geophysical modelling.

Multi-GPU

Using one GPU we can speed up the computation time of a 3D simulation six times compared with one CPU. However, the size of device memory in a single GPU limits the grid size. Thus, it is better to deal with a large memory space by using multiple GPUs (Micikevicius, 2009). We have two methods for synchronising data between multiple GPUs: OpenMP or Message Passing Interface (MPI). OpenMP can deal with a single node, whereas MPI can deal with multi-node computing. In this study, we use three GPUs in one node, and adopt the OpenMP method.

Multi-GPU computing is similar to parallel CPU computing. First, we divide the entire data space between the number of GPUs (three regions in this study: depicted in light grey in Figure 7) with each block executed by one GPU. Next, to calculate the edge of each area, surplus data, that is, two layers of grids from adjacent blocks are copied to each block (dark grey in Figure 7). Owing to this data copy, all light grey areas can be calculated in one GPU. To reduce the computation time, we calculate first the edge data and then we synchronise the edge data between the GPUs and do a calculation inside each block, simultaneously. For multi-GPU computing, therefore, additional data copies must take place between GPUs and this process is the main reason why GPU computing does not significantly improve the computing speed. The computation method for each block is the same as in the single GPU implementation. Using three GPUs, we can improve the computation speed 10 times (1452 s with $384 \times 384 \times 384$ grids, 1000 time steps) over a single CPU computation.

Discussion

In this paper, we have demonstrated that the usage of memory, especially shared memory, is the most important factor in accelerating of computation speed when using a GPU (Table 2). When using only device memory, we need to access that memory many times (i.e. the same number of times for every term in the FDTD scheme). The effectiveness of shared memory can clearly be seen in Case B (Table 2). The access ratio of device memory to shared memory decreases as we increase the block size. With a 16×16 block size, as the simulator accesses to device memory the least number of times, computation time for every

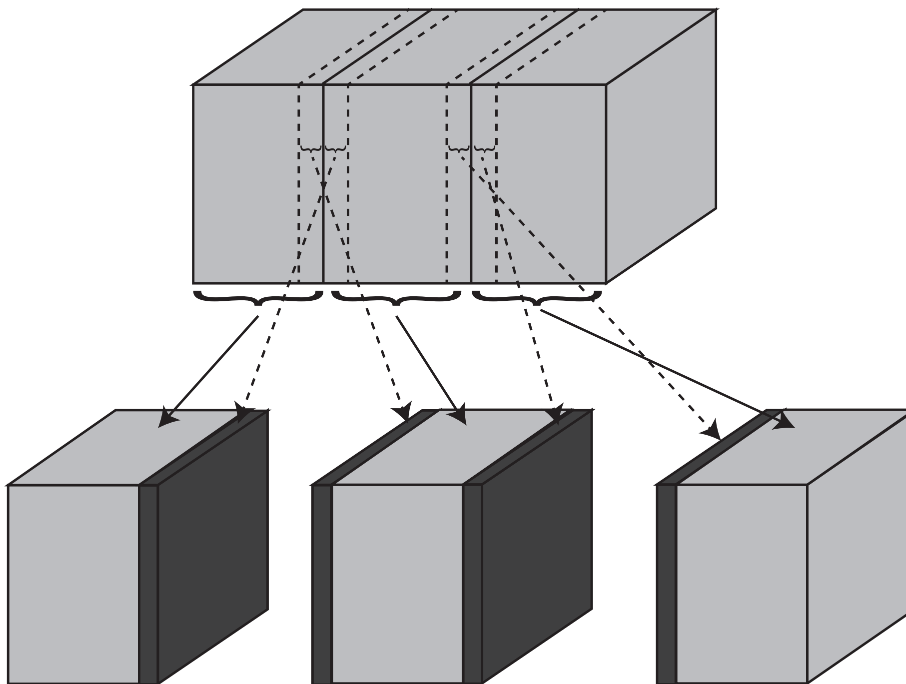

Fig. 7. Volume division method for multi-GPU computing. The upper box is assumed to be the whole data volume, which we divide into three blocks (light grey, solid line and arrow). To calculate the edge of each block, we also copy two layers of adjacent blocks (dark grey, dashed line and arrow).

grid is the fastest. Moreover, in Case C only two accesses to device memory are made, at start up and shut down of the kernel, and therefore, this is the fastest case. Because GPU can run several kernels simultaneously if their shared memory is available, the 8×8 block size is faster than 16×16 block size. The block size also influences the volume of communication traffic for one access.

The device memory bandwidth is used most efficiently with simultaneous memory accesses by 16 processors, as this can be coalesced into a single memory transaction of 32, 64, or 128 bytes (NVIDIA CUDA, 2010). For example, copying 4 bytes of data from device memory is not efficient, since we must copy 32 bytes of data including 28 bytes of useless data. Thus the time taken to copy memory without coalescing is ~10 times slower than with coalescing. Processors must access device memory sequentially for coalescing. The order of loops also influences coalescing. If we can coalesce a memory copy by copying in the X-direction, we may not coalesce when copying in the Y-direction. Data are stored linearly in memory and coalescing happens only in one direction. Thus, we optimise the coalescing by choosing the best loop order. Because we can only use a small block size and must copy large portions of memory, the 3D elastic wave propagation simulation has a smaller acceleration ratio than the 2D simulation. The key points of optimisation for elastic wave simulation are using faster memory, appropriate block size, and coalescing.

Conclusions

A GPU enables us to increase the simulation speed in a 2D elastic wave propagation simulation by more than 14 times and over six times in a 3D elastic wave propagation simulation. We also achieved 10 times acceleration using three GPUs. By using this GPU simulator, we can simulate fully elastic waves for $500 \times 500 \times 300$ grids in less than an hour. When using our CPU simulator on the other hand, it takes ~10 h to calculate elastic waves in the same size grid. This computation time is considered a reasonable computation time for industrial use.

Shared memory should be used as much as possible in GPU computing, since the access speed of shared memory is faster than that of device memory. Increasing the computation speed depends on both block size and coalescing. Ultimately, we should improve the rate of communication between shared and device memory. For large-grid modelling, multi-GPU computing is preferable. Although in this paper the simulation is carried out in single precision, by obtaining GPUs with a large shared memory space and double-precision processors, we could perform double precision simulations and we will then be able to use the GPU for inverse problems such as full-wave inversion.

Acknowledgements

We thank Masafumi Kato of JGI, Inc. for technical support. This work was partly supported by a Grant-in-Aid for Young Scientists (B) from the Japanese Society for the Promotion of Science: 20760568. Norimitsu Nakata is grateful for support from the Japan Society for the Promotion of Science (JSPS: 22–5857).

References

Abdelkhalek, R., Calandra, H., Coulaud, O., Latu, G., and Roman, J., 2009, Fast seismic modeling and reverse time migration on a GPU cluster: *Proceedings of the 2009 High Performance Computing & Simulation – HPCS09*, 36–43.

Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J., 2000, Parallel programming in OpenMP: Morgan Kaufmann.

Cerjan, C., Kosloff, D., Kosloff, R., and Reshef, M., 1985, A nonreflecting boundary condition for discrete acoustic and elastic wave equations: *Geophysics*, **50**, 705–708. doi:10.1190/1.1441945

Cerveny, V., 2001, Seismic ray theory: Cambridge University Press.

Graves, R. W., 1996, Simulating seismic wave propagation in 3D elastic media using staggered-grid finite differences: *Bulletin of the Seismological Society of America*, **86**, 1091–1106.

Juhlin, C., 1995, Imaging of fracture zones in the Finnsjon area, central Sweden, using the seismic reflection method: *Geophysics*, **60**, 66–75. doi:10.1190/1.1443764

Levander, A. R., 1988, Fourth-order Finite-Difference P-SV seismograms: *Geophysics*, **53**, 1425–1436. doi:10.1190/1.1442422

Liu, Y., Liu, X., and Wu, E., 2004, Real-time 3D fluid simulation on GPU with complex obstacles: *Proceedings of the 12th Pacific Conference on Computer Graphics and Applications* – PG04, 247–256.

Micikevicius, P., 2009, 3D Finite Difference computation on GPUs using CUDA: *GPGPU2*.

Moussa, N. W., 2009, *Seismic imaging using GPGPU accelerated reverse time migration CS315A final project report*: Technical Document, Stanford University.

Nickolls, J., and Dally, W. J., 2010, The GPU computing era: *IEEE Micro*, **30**, 56–69. doi:10.1109/MM.2010.41

NVIDIA CUDA, 2010, Programming Guide Version 3.0.

Nyland, L., Harris, M., and Prins, J., 2007, Fast N-body simulation with CUDA in Nguyen, H. (ed.), *GPU Gems 3*: Addison-Wesley, pp. 677–695.

Owens, J. D., Housto, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C., 2008, GPU computing: *Proceedings of the IEEE*, **96**, 879–899. doi:10.1109/JPROC.2008.917757

Saito, M., 1993, Branch line contribution in Lamb's problem: *Butsuri Tansa*, **46**, 372–380.

Stone, J. E., Philips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G., and Schulten, K., 2007, Accelerating molecular modelling applications with graphics processors: *Journal of Computational Chemistry*, **28**, 2618–2640. doi:10.1002/jcc.20829

Takada, N., Takizawa, T., Gong, Z., Masuda, N., Ito, T., and Shimobaba, T., 2008, Fast computation of 2-D Finite-Difference Time-Domain method using Graphics Processing Unit with unified shader: *IEICE*, **J91-D**, 2562–2564.

Virieux, J., 1986, P-SV wave propagation in heterogeneous media: velocity-stress finite difference method: *Geophysics*, **51**, 889–901. doi:10.1190/1.1442147

Manuscript received 16 August 2010; accepted 4 November 2010.

Graphic Processing Unitによる弾性波動シミュレーションの高速化

仲田典弘[1]・辻　健[1]・松岡俊文[1]

1 京都大学大学院工学研究科都市社会工学専攻

要　旨： 物理探査において、波動シミュレーションは、地下を伝播する波動場を把握するための重要なツールとなっている。通常、我々が対象としているのは弾性場であるため、音響波動計算よりも剪断成分も計算することのできる弾性波動計算の方が、より正確なシミュレーションが可能となる。しかし計算時間の制約などから、音響波動計算が用いられることが多い。そこで、我々はGraphic Processing Unit (GPU)を用いて、弾性波動シミュレーションの高速化を提案する。GPUは通常、画像データを扱っている計算ユニットであり、多数のプロセッサと高速なメモリを搭載していることが特徴である。我々の使用したNVIDIA社製Tesla C1060では、240個のプロセッサと102GB/secの帯域速度のメモリが搭載されている。このGPUを科学計算に用いることで、巨大な並列計算環境を容易に構築することができる。我々は、２次元と３次元の弾性波動計算を、スタッガードグリッドを用いた有限差分法によって計算した。我々のシミュレータはGPUボード上のメモリを効率的に用いることにより、通常のCPUを用いた計算と比べて２次元では14倍、３次元では6倍の計算速度を実現した。また、より大きな領域の計算に対応するため、3枚のGPUを用いた計算を行い、３次元で10倍の計算速度を実現した。このシミュレータを用いることで、500×500×250 グリッドの弾性波動計算を、十分な精度を保ったまま、1時間以内で完了することができることが示された。

キーワード：CUDA，弾性波動伝播，Graphic Processing Unit，数値シミュレーション，３次元有限差分法

그래픽 프로세서를 이용한 탄성파 수치모사의 계산속도 향상

Norimitsu Nakata[1], Takeshi Tsuji[1], Toshifumi Matsuoka[1]

1 교토대학교 대학원 도시사회공학 전공

요 약： 탐사 지구물리학에서 수치 모사는 지하매질에서의 탄성파 전파 현상을 이해하는데 중요한 통찰력을 제공한다. 탄성파 모사는 음향파 근사에 의한 수치 모사보다 계산시간이 많이 소요되지만 전단응력 성분을 포함하여 보다 현실적인 파동의 모사를 가능하게 한다. 그러므로 탄성파 모사는 탄성체의 반응을 탐사하는데 적합하다고 할 수 있다. 계산 시간이 길다는 단점을 극복하기 위해 본 논문에서는 그래픽 프로세서(GPU)를 이용하여 탄성파 수치 모사 시간을 단축하고자 하였다. GPU는 많은 수의 프로세서와 광대역 메모리를 갖고 있기 때문에 병렬화된 계산 아키텍처에서 사용할 수 있는 장점이 있다. 본 연구에서 사용한 GPU 하드웨어는 NVIDIA Tesla C1060으로 240개의 프로세서로 구성되어 있으며 102 GB/s의 메모리 대역폭을 갖고 있다. NVIDIA에서 개발된 병렬계산 아키텍처인 CUDA를 사용할 수 있음에도 불구하고 계산효율을 상당히 향상시키기 위해서는 GPU 장치의 여러 가지 다양한 메모리의 사용과 계산 순서를 최적화해야만 한다. 본 연구에서는 GPU 시스템에서 시간영역 유한차분법을 이용하여 2차원과 3차원 탄성파 전파를 수치 모사하였다. 파동 전파 모사에 가장 널리 사용되는 유한차분법 중의 하나인 엇갈린 격자기법을 채택하였다. 엇갈린 격자법은 지구물리학 분야에서 수치 모델링을 위해 사용하기에 충분한 정확도를 갖고 있는 것으로 알려져 있다. 본 논문에서 제안한 모델링 기법은 자료 접근 시간을 단축하기 위해 GPU 장치의 메모리 사용을 최적화하여 가능한 더 빠른 메모리를 사용한다. 이 점이 GPU를 이용한 계산의 핵심 요소이다. 하나의 GPU 장치를 사용하고 메모리 사용을 최적화함으로써 단일 CPU를 이용할 경우보다 2차원 모사에서는 14배 이상, 3차원에서는 6배 이상 계산시간을 단축할 수 있었다. 세 개의 GPU를 사용한 경우에는 3차원 모사에서 계산효율을 10배 향상시킬 수 있었다.

주요어：CUDA, 탄성파 파동 전파, 그래픽 프로세서, 탄성파 모델링, 3 차원 유한차분