GPGPU-Aided 3D Staggered-grid Finite-difference Seismic Wave Modeling

Chang Cai[1], Haiqing Chen[2], Ze Deng[1], Dan Chen[1,*], Samee U. Khan[3], Ke Zeng[1], Minxiao Wu[1]
[1]School of Computer Science
China University of Geosciences
Wuhan, China, Postcode: 430074

[2]School of Optoelectronics
Hankou University
Wuhan, China, Postcode: 430212

[3]Department of Electrical and Computer Engineering
North Dakota State University
North Dakota, US
* Corresponding author's e-mail: chendan@pmail.ntu.edu.sg

Abstract—**Finite difference is a simple, fast and effective numerical method for seismic wave modeling, and has been widely used in forward waveform inversion and reverse time migration. However, intensive calculation of three-dimensional seismic forward modeling has been restricting the industrial application of 3D pre-stack reverse time migration and inversion. Aiming at this problem, in this paper, a parallelized 3D Staggered-grid Finite-difference has been developed using General-purpose computing on the graphics processing unit (GPGPU), namely G-3DFD, since the emergence of graphic processing units (GPU) as an effective alternative to traditional general purpose processors has become increasingly capable in accelerating large-scale scientific computing. We analyze three-dimensional staggered grid finite difference method for the implementation on GPU, making possible the industrial application of 3D pre-stack reverse time migration and inversion. Experiments show that G-3DFD has dramatically improved the runtime performance 88 times on modern GPGPU platforms comparing to the original CPU implementation methods.**
Keywords: Seismic Wave Modeling; 3D Stencil Computation; GPGPU; Parallel Computing

I. INTRODUCTION

Finite-difference (FD) techniques in the time domain (FDTD) are widely used to solve wave equations such as Maxwell's equations [1] or the seismic wave equation [2] and have been used to solve Navier-Stokes' equations [3] as well. A thorough review on FD applied to the seismic wave equation is described in [4]. When more geometrical flexibility is needed for instance to handle geometrically complex models other techniques such as a pseudo-spectral technique [5][6], a boundary-element method [7][8], a spectral-element method [11] or a discontinuous Galerkin technique [12][13] is needed because of its simplicity.

FD is often used in conjunction with a perfectly matched layer (PML) to absorb waves on the artificial edges of the numerical grid to mimic an infinite or semi-infinite medium [14] or a Convolution PML (CPML) can be used to improve the behavior of the discrete PML for waves impinging the artificial edges of the grid at grazing incidence [16][17] for unspotted CPMLs for seismic waves. More recently a formulation of the unspotted CPML that can easily be extended to higher-order time schemes, called the auxiliary differential equation PML (ADE-PML), has been introduced by [18] for Maxwell's equations and by [19] for the seismic wave equation. An improved sponge layer, improperly called the split Multiaxial PML (M-PML), has been suggested by [20], but the perfectly matched character of Bérenger is lost because of the coupling introduced between derivatives along multiple grid axes.

In recent years, graphics processing unit (GPU) has been widely used to accelerate general-purpose applications. A number of physical computing problems have been solved using GPU, e.g. molecular dynamics simulations, fluid dynamics simulations and astrophysical calculations [21], etc. Regarding FD, several applications have been ported to GPUs. Some spectral-element methods are proposed on a single GPU platforms in [6] while some methods are based on multiple GPUs in [22][23].

GPU programming on NVIDIA graphics cards has become significantly easy with the introduction with CUDA programming language, which is relatively easy to learn because its syntax is similar to C. Regarding FD for seismic reverse time migration in the case of an acoustic medium with constant density, Abdelkhalek and Micikevicius (from NVIDIA corporation, the developers of GPU hardware and of CUDA) have recently introduced optimized implementations. Abdelkhalek extended it to the acoustic case with heterogeneous density [24].

In this study, we use CUDA to solve the 3D Staggered-grid Finite-difference Seismic Wave Modeling in a more complex fully heterogeneous environment in parallel. We propose some new techniques to adapt to the programming model of CUDA, including kernels, thread hierarchy and memory hierarchy. Since the number of points is very huge, we use one thread to process points in a row, this method not only solves the problem of limited resource on GPU, but also easily achieves

the optimization of Memory Coalesced to decrease the access times of global memory and fully exploit limited shared memory in GPU hardware. GPU has much more cores and greater compute capability than CPU, so in our program, all calculation works are run on GPU while the memory management and result output are executed on CPU. Then we finish the 3D Staggered-grid Finite-difference Seismic Wave Modeling respectively using C running on CPU and CUDA program running on GPU. Performance comparisons between them show that GPU is greatly suitable for parallelism computation especially for highly compute-intensive issue and the speedup we get ranges from 57× to 88×, which reduces the work time from one hour to half a minute and makes the industrial application of 3D pre-stack reverse time migration and inversion possible.

II. METHODS

A. 3D Staggered-grid Finide-difference

This section describes the formulation of the AWP-ODC numerical model and the analysis of the computation steps. As stated in paper [25], AWP-ODC solves a 3D velocity-stress wave equation using a staggered-grid finite difference method to achieve fourth-order accurate in space and second-order accurate in time. The coupled system of partial differential equations includes the particle velocity vector ν and the symmetric stress tensor σ [26]. Suppose:

$$v = (v_x, v_y, v_z) \qquad \sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix} \tag{1}$$

Then the governing elastodynamic equations are [45]:

$$\partial_t v = \frac{1}{\rho}\nabla \cdot \sigma \tag{2}$$

$$\partial_t \sigma = \lambda(\nabla \cdot v)\mathrm{I} + \mu(v + \nabla v^{\mathrm{T}}) \tag{3}$$

Where λ and μ are the lame coefficient and ρ is the constant density. Simplifying formula (2) and (3) leads to three scalar-valued equations for velocity vector components and three scalar-valued equations for the stress tensor components. Three equations are listed below including one for velocity vector equation and two for stress equations:

$$\frac{\partial v_x}{\partial t} = \frac{1}{\rho}\left(\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{zz}}{\partial z}\right) \tag{4}$$

$$\frac{\partial \sigma_{xx}}{\partial t} = (\lambda + 2\mu)\frac{\partial v_y}{\partial y} + \lambda\left(\frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}\right) \tag{5}$$

$$\frac{\partial \sigma_{xy}}{\partial t} = \mu\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}\right) \tag{6}$$

AWP-ODC is a memory-intensive application and twenty-one 3D variable arrays are involved in the main computation loop. In addition to the three velocity vector components and six symmetric stress tensor components, 6 temporary variables (*r*1, *r*2, *r*3, *r*4, *r*5, *r*6) and 6 constant coefficients (λ, μ, ρ, S for quality factor wave Qs and P wave Qp, boundary condition variable Cerjan Cj [27]) are used in the numerical modeling. Fig.1 presents the pseudo-code of the computation kernels in the main loop for formula (4) and (5). The algorithm mainly includes 5 steps:

Step 1: This step initializes the original model's parameters, sets the 3D grid's size and the time step and calculates the wavelet for simulation.

Step 2: In processing the *t*th time, differential of the stress along the spatial direction is computed, update values of velocities along the surface.

Step 3: Surface differential of the velocity field along the axis direction is calculated, update values of stress along the surface.

Step 4: Check if *t>max*: when termination condition is not satisfied, *Step* 4 will be returned with *t*=*t*+1 and add the wavelet source into simulation to *Step* 2; Otherwise, jump to *Step* 5.

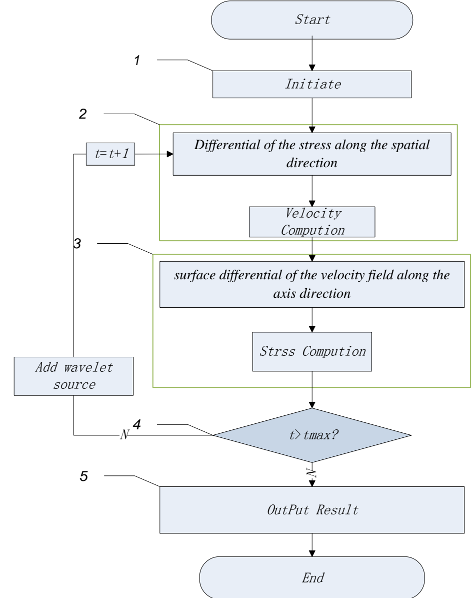

Fig. 1. AWP-ODC algorithm

Step 5: Output the results of simulation.

The 13 point asymmetric stencil computation for v_x in AWP-ODC main loops reads occur 12 times in three different 3D arrays and writes occur only once in one 3D array. Stress component calculation includes two 1D asymmetric stencils, with only 4 reads from a single 3D array and no writes. TABLE I. summarizes the analysis of kernels in Fig.1. It shows that AWP-ODC is a memory-bound application because of the low FLOPS to bytes ratio (the average operation intensity is around 0.5), which means the application has poor temporal locality and the performance is dominated by the memory system or arithmetic throughput [28].

TABLE I. ANALYSIS OF COMPUTATION KERNELS IN AWP-ODC MAIN LOOP

Kernels	***Reads***	***Writes***	***FLOPS***	***FLOPS/Bytes***
Velocity Computation Kernel	51	3	86	0.398
Stress Computation Kernel	85	12	221	0.569
Total	136	15	307	0.508

A. General-purpose Computing on Graphics Processing Units (GPGPU) and Compute Unified Device Architecture (CUDA)

Graphics processing unit (GPU) is specialized for compute-intensive and highly parallel computations [29], especially those problems that can be expressed as data-parallel computations (e.g., the Matrix multiplication). Owing to their highly-parallel architecture, modern GPUs are capable of a theoretical peak performance that is an order of magnitude higher than mainstream CPUs [30]. This feature together with the high performance-to-price ratio and widespread availability have propelled GPUs to the forefront of high-performance computing which are now accessible even on most commodity computers. General-purpose Computing on Graphics Processing Units (GPGPU) has recently boomed with the enhanced programmability of GPUs rather than having to depend on standard graphics APIs or shadier language, which forces the developer to re-formulate algorithms in graphics metaphors, general purpose programming languages.

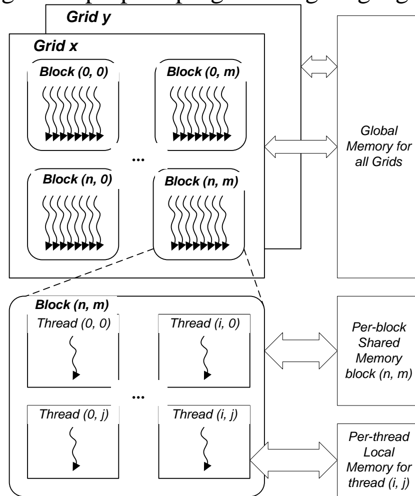

Fig. 2. Hierarchy of Threads and Memory in CUDA

Relatively high abstraction levels have emerged, and NVIDIA's Compute Unified Device Architecture (CUDA) is a salient example. The current CUDA fosters a software environment which allows programming on GPU with a slightly extended version of C, therefore a CUDA-based application can execute on the GPU in a massively parallel fashion. A CUDA application defines C functions as "kernels" to be explicitly executed on the GPU (referred to as the "device"). Normally the kernels are invoked from the main executable of the application on the CPU (the ''host''), which will be executed for N times via N individual CUDA "threads". As illustrated in Fig. 2, CUDA threads are organized in one to three-dimensional manner, where threads are grouped to blocks and blocks are again grouped into grids. Threads in a block can cooperate together by efficiently sharing data through a fast shared memory and synchronizing their execution to coordinate memory access. Threads, memories and synchronization are exposed to developers for very fine-grained data and thread parallelisms, e.g., dealing with a single variable of float type per thread. Application developers should partition the problem into a set of fine-grained subtasks which can be mapped to threads accordingly. A more detailed introduction to GPGPU and CUDA can be seen in [29].

III. Parallelization

A. GPGPU-aided 3D Staggered-grid Finide-difference

This study focus on a GPGPU-aided implementation of the 3D Staggered-grid Finite-difference method. The main difficulty when implementing a finite-difference code on a GPU comes from the highly compute-intensive. For a fourth-order spatial operator, the thread that handles the calculation of point (i, j, k) needs to know the fields (and therefore access the arrays) at points: (i, j, k) , (i + 1, j , k), (i + 2, j , k), (i − 1, j , k), (i − 2, j , k), (i , j + 1, k) and so on, including the point that it handles and its neighboring points. This implies that 13 accesses to global memory are needed on the GPU to handle each grid point, which causes a high access overhead due to the longer access delay of global memory.

However threads that belong to the same block can access data in faster shared memory, it is possible to significantly reduce this number of memory accesses per grid point and thus drastically improve performance. Since the number of threads and the size of shared memory per block run at the same time are limited, and the number of threads points is growing with an order of $O(n^3)$ as the growth of grid size, there is not enough shared memory to allow one to use 3-D blocks large enough to sufficiently reduce this ratio, and therefore this approach cannot be efficiently implemented, so we have to split the 3D model to 2D approach.

We therefore turn to a more efficient 2-D approach, which increases the computation workload of each thread to reduce the number of threads. We can parallelize the 3D Staggered-grid Finite-difference application at the grid level. As indicated in Fig.3, each grid point of *3*D Staggered-grid Finite-difference is mapped to an individual task, but the number of points is too big and the workload of each point is too small, so one row data point mapping to one thread task is the right design. And also we design the program accessing the global memory in coalesced way.

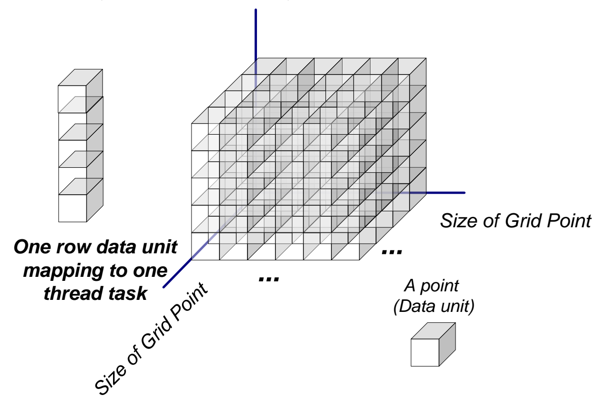

Fig. 3. Parallelization at threads level, one row data mapping to one thread task

Full utilization of on-chip memory is the key to achieve high performance for GPU-based applications. Our optimization mechanisms can be summarized into four steps to achieve this goal:

Step 1: Read-Only Memory Cache: Texture and Constant memory in the GPU device has its own cache. As discussed

above, six 3D variable arrays hold constant coefficients. We put them into the texture memory to take advantage of the texture cache. All scalar constants are put into the constant memory to save registers. The other 15 arrays are stored in global memory because their values are updated during iteration.

Step 2: 2D Domain Decomposition: CUDA is an extended language of C/C++, so memory storage for 3D arrays will be fast along the z axis while be slow along x axis. To obtain a better cache hit rate and allow all threads in the same wrap to access data along the fast axis z instead of the slow axis x, we decompose the 3D Grid only along axis y and axis z. Each thread will calculate the entire NX for a given 2D (y, z) location as shown in Fig.4.

Since the thread block is also a 3-dimension structure and presents the best performance in the x direction, threads in the x direction must correspond to the 3D Grid z direction, while threads in the y direction correspond to the 3D Grid y direction. Since 32 threads in a wrap are executed at the same time, the number of threads in the x direction is considered to be a multiple of 32 for better performance.

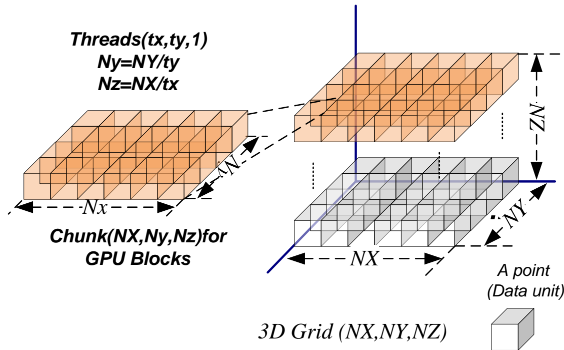

Fig. 4. Decomposition for GPU Kernels

Step 3: Global Memory Coalesced: For maximum performance, memory access need to be coalesced in global memory. CUDA provides two functions "cudaMallocPitch" and "cudaPitchedPtr" to help ensure aligned memory access. However, instead of using these library functions, we manually pad zeros onto the boundaries in the z axis of the 3D grid to align memory to the inner region. The padding size plus NZ should also be a multiple of 32, and the two layers of ghost cells are included in the padding.

Step 4: Memory Management: GPU memory is allocated and managed by CPU. Memory optimizations are the most important area for performance optimization. Minimizing data transfer between the host and the device is an effective way. Our method is to batch multiple small transfers into one transfer to significantly reduce the communication costs between GPU and CPU.

Since GPU has limited memory resources, we design the program with only one transfer from CPU to GPU during the whole computation procedure and reuse the GPU memory to make the program calculate much larger 3D seismic wave modeling than original methods. Since GPU has more cores and compute faster than CPU, all calculations are done simultaneously by GPU except memory copy, see Fig.5 for more details:

Step 1: Initialization phase. This step sets the size of grid and defines the boundaries of seismic wave model. Thus we can obtain the size of each parameter, and allocate the exact size of GPU memory for next step's computation.

Step 2: After initialization, this step deals with the computation on GPU, a two-dimensional thread block is defined to identify the CUDA threads. After that, the algorithm focus on PML boundary set, Wave field variable initialization, the source sequence generation.

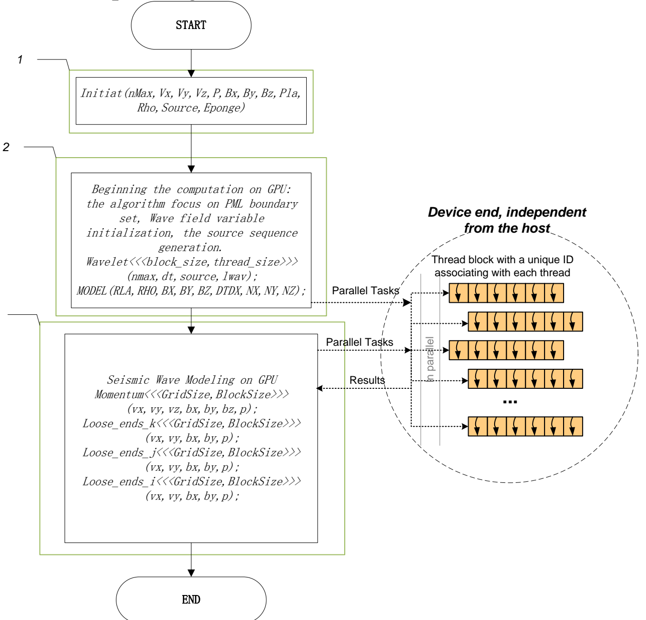

Fig. 5. The flow of the *GPGPU-Aided 3D Staggered-grid Finite-difference*

Step 3: This step calculates the Seismic Wave Modeling on GPU to speed up the calculation of wave field, including space differential calculation, matrix or vector product and sum. Meanwhile, in order to save and map results, the needed results from GPU must be copied back to the CPU. Of course, we can use GPU rendering technology directly to the output wavelength snapshot as well.

IV. Experiment and Performance Comparisons

The new CUDA method implementation has been carefully validated for correctness. We compare our version of GPU with the traditional method base on CPU. The results are shown in Fig.6. The first 200 time steps of the stress of two methods are recorded at the source point. It can be seen that the results are almost identical with tiny differences caused by different programming languages and compilers.

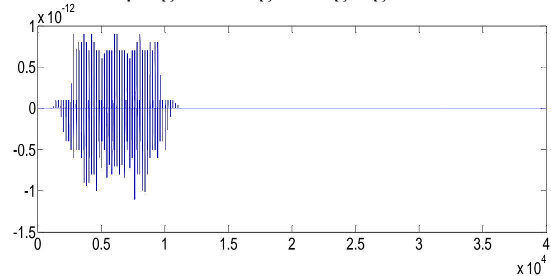

Fig. 6. Accuracy comparison of waveforms computed on GPU and CPU

On a single GTX680 card with memory of 2GB, in order to compute as large size as possible, we use a simple model. It has nine 3D grid parameters. By trying to allocate increasingly

larger chunks of memory, almost 1.9 GB memory is available to the user. The largest 3D model that we can use has a size of 37.6km × 37.6km × 37.6km discretized using a grid of 376×376×376 points, that is with grid cells of size 100m in the three spatial directions.

To test the performance of the CUDA code running on GPU, we compare it with the performance of the C code running on CPU. The GPU card we use is NVIDIA GTX 680 and the CPU is Intel(R) Core(TM) i7-2600. We compile the C code in VS2008, and the code for the GPU with the standard NVIDIA nvcc compiler of CUDA version 5.0. To investigate the impacts of different factors on the proposed method, we use a few different measurements on CPU and GTX680 card. TABLE II. is the performance comparisons between GPU and CPU, all benchmark experiments run for 200 time steps and the measurement is focused on the whole time used during the modeling.

From TABLE II. we can observe that:

1) With the growing of grid size, the amount of computation is on exponential growth. The time used by CPU to calculate the size larger than 370*370*370 is more than an hour. It's a big challenge for us to compute large 3D gird Seismic Wave Modeling on CPU.
2) GPU is good at fine-grained computation especially matrix operations. It only costs half a minute for GPU to compute 368*368*368 3D grid, which is 88-fold accelerated than CPU.
3) Even though GPU can provide an amazing speedup, the resource on GPU is limited, so the size we can calculate and the speedup we can get depend on heavily GPU resources. TABLE II. shows that when the grid size is less than 368*368*368, the speedup is increasing with the size growth, but when the 3D size is 376*376*376 the speedup only reaches 84, the reason is that the limited GPU resources has been used up.

TABLE II. PERFORMANCE COMPARISON BETWEEN CPU AND GPU IN 200 TIME STEPS

Grid Size	***Time(s)***		***Speedup***
	CPU(i7)	***GPU***	
128*128*128	131.125	2.300	57.019
192*192*192	424.684	6.091	69.723
256*256*256	970.876	13.150	73.831
368*368*368	2961.970	33.524	88.354
376*376*376	3029.589	36.160	83.783

From the analysis above, under the premise of correctness, a single GPU is able to bring more than 80 times speedup, which gives us a perfect alternative for solving highly compute-intensive algorithm and problem.

V. CONCLUSIONS AND FUTURE WORK

In this study, the feasibility and efficiency of 3D Staggered-grid Finite-difference for seismic wave modeling approach was explored. Finite difference is a simple, fast and effective numerical method for seismic wave modeling, and has been widely used in forward waveform inversion and reverse time migration. However, intensive calculation of three-dimensional seismic forward modeling has been restricting the industrial application of 3D pre-stack reverse time migration and inversion.

In order to satisfy the intensive calculation of three-dimensional seismic forward modeling, this study proposes an approach which uses CUDA with thousands of threads, the complex matrix operations can be divided into simple and easy calculations which map the thread's structure exactly, i.e., each thread only calculates one element of the result matrix. With several optimization technologies used, we can get much more obvious promotion.

Experiments have been carried out to evaluate the performance of the parallel variants of 3D Staggered-grid Finite-difference using examples of different size simulation. We have shown that the GPU code is accurate by comparing our results to results obtained running the same simulation on a CPU core. We have accelerated a 3-D finite-difference wave propagation code by a factor of from 57 to 88 compared to a serial implementation using one NVIDIA GPU graphics cards and the CUDA programming language, which reduces the simulation time from about one hour into half a minute.

As the size growing even larger, more efficient methods must be taken into account, such as multi-GPU program or redesign algorithm, which one is better, our future work will focus on this.

ACKNOWLEDGMENT

This work was sponsored in part by the Hundred University Talent of Creative Research Excellence Programme (Hebei, China), the National Natural Science Foundation of China (grants No. 61272314), the Programme of High-Resolution Earth Observing System (China), the Fundamental Research Funds for the Central Universities (No.CUGL100608, No.CUGL100231 and G1323521289, China University of Geosciences, Wuhan), Specialized Research Fund for the Doctoral Program of Higher Education (20110145110010), and the Program for New Century Excellent Talents in University (grant No. NCET-11-0722)

REFERENCES

[1] K.S. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations," *IEEE Transactions on Antennas and Propagation,* vol. 14, no. 3, pp. 302-307, 1966.

[2] J. Virieux, "P-SV wave propagation in heterogeneous media: velocity stress finite-difference method," *Geophysics*, vol. 51, no. 4, pp. 889-901, 1986

[3] A. J. Chorin, "Numerical solution of the Navier-Stokes equations," *American Mathematical Society*, vol. 22, no. 104, pp. 745-762, 1968.

[4] P. Moczo and J. Robertsson, L. Eisner, "The finite-difference time-domain method for modeling of seismic wave propagation," *Advancesin in Geophysics*, vol. 48, no. 8, pp. 421-516, 2007.

[5] J. M. Carcione and P. J. Wang, "A Chebyshev collocation method for the wave equation in generalized coordinates," *Computers and Fluid Dynamics Journal*, no. 2, pp. 269-290, 1993.

[6] D. Komatitsch, F. Coutel and P.Mora, "Tensorial formulation of the wave equation for modelling curved interfaces," *Geophysical Journal International*, vol. 127, no. 1, pp. 156-168, 1996.

[7] H. Kawase, "Time-domain response of a semi-circular canyon for incident SV, P and Rayleigh waves calculated by the discrete wavenumber boundary element method," *Bulletin of the Seismological Society of America*, vol. 78, pp. 1415-1437, 1988.

[8] R. Vai, J. M. Castillo-Covarrubias, F. J. Sánchez-Sesma, D. Komatitsch and J. P. Vilotte, "Elastic wave propagation in an

irregularly layered medium," *Soil Dynamics and Earthquake Engineering*, vol. 18, no. 1, pp. 11-18, 1999.

[9] Q. Liu, J. Polet, D.Komatitsch and J. Tromp, "Spectral-element moment tensor inversions for earthquakes in Southern California," *Bulletin of The Seismological Society of America-BULL SEISMOL SOC AMER*, vol. 94, no. 5, pp. 1748-1761, 2004.

[10] E. Chaljub, D. Komatitsch, J. P. Vilotte, Y. Capdeville, B. Valette and G. Festa, "Spectral element analysis in seismology," *Advances in Wave Propagation in Heterogeneous Media*, vol. 48, pp. 365-419, 2007.

[11] J. Tromp, D. Komatitsch and Q. Liu, "Spectral-element and adjoint methods in seismology," *Communications in Computational Physics*, vol. 3, no. 1, pp. 1-32, 2008.

[12] W. H. Reed and T. R. Hill, "Triangular mesh methods for the neutron transport equation," *Alamos Scientific Laboratory, Los Alamos, USA*, 1973.

[13] M. J. Grote, A. Schneebeli and D. Schötzau, "Discontinuous Galerkin finite element method for the wave equation," *Siam Journal on Numerical Analysis-SIAM J NUMER ANAL*, vol. 44, no. 6, pp. 2408-2431, 2006.

[14] J. P. Bérenger, "A Perfectly Matched Layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, vol. 114, no. 2, pp. 185-200, 1994.

[15] F. Collino and C. Tsogka, "Application of the PML absorbing layer model to the linear elastodynamic problem in anisotropic heterogeneous media, " *Geophysics*, vol. 66, no. 1, pp. 294-307, 2001.

[16] J. A. Roden and S. D. Gedney, "Convolution PML (CPML): an efficient FDTD implementation of the CFS-PML for arbitrary media, " *Microwave and Optical Technology Letters*, vol. 27,no. 5, pp. 334-339, 2000.

[17] R. Martin, D. Komatitsch and S. D. Gedney, "A variational formulation of a stabilized unsplit convolutional perfectly matched layer for the isotropic or anisotropic seismic wave equation," *Computer Modeling in Engineering & Sciences*, vol. 37, no. 3, pp. 274-304, 2008b.

[18] S. D. Gedney and B. Zhao, "An auxiliary differential equation formulation for the complex-frequency shifted PML," *IEEE Transactions on Antennas and Propagation*, vol. 58, no. 3, pp. 838-847, 2010.

[19] R. Martin, D. Komatitsch, S. D. Gedney and E. Bruthiaux, "A high-order time and space formulation of the unsplit perfectly matched layer for the seismic wave equation using Auxiliary Differential Equations (ADE-PML)," *Computer Modeling Engineering and Science*, vol. 56, no. 1, pp.17-42, 2010.

[20] K. C. Meza-Fajardo and A. S. Papageorgiou, "A nonconvolutional, split-field, perfectly matched layer for wave propagation in isotropic and anisotropic elastic media: stability analysis," *Bulletin of the Seismological Society of America*, vol. 98, no. 4, pp. 1811-1836, 2008.

[21] L. Nyland, M. Harris and J. Prins, "Fast N-body simulation with CUDA," GPU Gems 3, Chapter 31, pp. 677-695, Addison-Wesley Professional, Boston, MA, USA , 2007.

[22] D. Komatitsch, G. Erlebacher, D. Göddeke and D. Michéa, "High-order finite-element seismic wave propagation modeling with MPI on a large-scale GPU cluster," Journal of Computer Physics, vol. 229, no. 20, 2010.

[23] D. Komatitsch, D. Göddeke, G. Erlebacher and D. Michéa, "Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs," *Computer Scienc-Research and Development*, vol.25, no. 1-1, pp. 75-82, 2010.

[24] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman and G. Latu, "Fast seismic modeling and reverse time migration on a GPU cluster," *The 2009 High Performance Computing & Simulation*, pp. 36-44, 2009.

[25] J. Zhou, D. Unat , D. J. Choi, C. C. Guest and Y. F. Cui, "Hands-on Performance Tuning of 3D Finite Difference Earthquake Simulation on GPU Fermi Chipset," Procedia Computer Science, vol. 9, pp. 976-985, 2012.

[26] Y. Cui, K. B. Olsen, T.H.Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. Panda, A. Chourasia, J. Levesque, S. M. Day, and P. Maechling, Scalable Earthquake Simulation on Petascale Supercomputers, In Proceedings of the 2010 ACM/IEEE conference on Supercomputing, SC'10, pp. 1-20, Nov. 2010.

[27] A. Simone, and S. Hestholm, "Instabilities in applying absorbing boundary conditions to high-order seismic modeling algorithms," *Geophysics*, vol. 63, pp. 1017- 1023, 1998.

[28] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. Communications of The ACM, vol. 52, no. 4, pp. 65-76, 2009.

[29] NVIDIA, "NVIDIA CUDA Programming Guide," http://www.nvidia.com/object/cuda_develop.html, 2008.

[30] O. Schenk, M. Christena and H. Burkharta, "Algorithmic performance studies on graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 68, pp. 1360-1369, 2008.