博 士 論 文 （要約）

Fast Low-Order Finite Element Earthquake Simulations on GPUs with Transprecision Computing

(精度変動演算を用いた GPU による

高速な低次有限要素地震シミュレーション手法開発)

山口　拓真

ABSTRACT

The content is not included in the abridged version.

Acknowledgment

First and foremost, I would like to express my deep gratitude to Prof. Tsuyoshi Ichimura, who took me a trip to the farthest edge, or leading edge of computer science. Participation to various international conferences, collaborative researches, and shortening of the doctor course period were fruitful and exciting experience for me.

I also would like to thank Prof. Kengo Nakajima for his guidance as a sub-chief examiner. Advices form the perspective in computational science were insightful for this research. In addition, I appreciate the feedback offered by Prof. Muneo Hori, Prof. Lalith Wijerathne, and Prof. Kei Yoshimura.

Prof. Kohei Fujita made enormous and invaluable contribution to my studies. Without his supports, this dissertation would not have materialized.

I am grateful and thankful to my fellow lab members whose continuous supports have motivated me to keep moving forward.

Part of our results were obtained using the Summit at Oak Ridge Leadership Computing Facility, a US Department of Energy, Office of Science User Facility at Oak Ridge National Laboratory (ORNL). We thank Yukihiko Hirano (NVIDIA) for coordination of the collaborative research project. We thank Christopher B. Fuson, Don E. Maxwell, Oscar Hernandez, Scott Atchley, Veronica Melesse-Vergara (ORNL), Jeff Larkin, Stephen Abbott (NVIDIA), Lixiang Luo (IBM), Richard Graham (Mellanox Technologies) for generous support concerning use of Summit. We thank Noda Tomoyuki and Hikaru Inoue (Fujitsu Limited) for support in program development. We acknowledge support from Post K computer project (Priority Issue 3 - Development of integrated simulation systems for hazards and disasters induced by earthquakes and tsunamis: proposal numbers hp180217, hp190177), and Japan Society for the Promotion of Science (18H05239, 18H05239, 26249066, 25220908, and 17K14719). Part of our results were obtained using Computational resource of AI Bridging Cloud Infrastructure (ABCI), National Institute of Advanced Industrial Science and Technology (AIST). We thank Prof. Olaf Schenk (Institute of Computational Science, Università della Svizzera italiana) for the generous support concerning the use of Piz Daint.

Contents

List of Tables

List of Figures

1 Introduction

When designing countermeasures for earthquake disasters, sophisticated damage estimation is one of key issues. Physical processes of earthquake disasters mainly consist of crustal deformation, wave propagation, and soil amplification. Traditional approach for damage estimation is based on statistical analysis [1]; however, available data is not enough for earthquake disasters. Thus, the introduction of numerical simulations is assumed to be more effective than traditional approaches.

However, these simulations require massive computation cost. Numerical simulations targeting earthquake disasters come down to the problem with $\sim 10^{10-11}$ DOF. In reference [2], finite element model was generated for the $10.25\,\mathrm{km} \times 9.25\,\mathrm{km} \times 0.24\,\mathrm{km}$ domain with $2\,\mathrm{m}$ resolution to simulate soil amplification around Tokyo area. Degrees of freedom of the model reached 1.3×10^{11}.

In addition, there are uncertainties in the earthquake simulation (e.g. earthquake scenarios, mechanical characteristics, and geometries). Some studies proposed methods to include these uncertainties. Regarding crustal deformation, [3] considered the effect of error in Green's functions for inverse analyses of slip distribution. Also, [4] introduced Markov chain Monte Carlo approach for the estimation of the posterior distribution of fault parameters. In wave propagation analysis, [5] demonstrated a case study for inversion in a tomographic problem. In soil amplification analysis, [6] applied Monte Carlo method in the regional seismic damage prediction accounting for uncertainty in building structure parameters. These simulations using stochastic computing often come up in this field. It is assumed that the demand for consideration of uncertainties in earthquake simulations has also increased. Typical methods to consider uncertainties generally require multiple forward analyses. Altogether, stochastic computing with high-fidelity 3-D models can lead to more reasonable damage estimation. Jointly with observation networks, physics-based simulations using high resolution data will enhance the capability of monitoring and evaluating earthquake hazard.

Due to the massive computation cost, we couldn't execute simulations mentioned above within a realistic timeframe. Conventional damage estimation is conducted by using deterministic problem settings or simplified models (e.g. homogeneous elastic half-space model for crustal deformation computation [7]). In our previous studies, fast implicit low-order finite element solver had been developed for CPU-based systems [8]. The solver part accounts for the largest proportion of the entire computation cost in our target simulations; thus, acceleration of the solver is important to reduce the computation cost.

Recently, the computation environment has been rapidly improved and diverged. Graphic Processing Unit (GPU) is known as one of most widely used accelerators. GPU enables parallel computations with more than 1,000 cores and it is commonly used in scientific computing. Programming models to introduce GPU computations have been established. For instance, CUDA [9] enables us to write GPU-based codes with some extension to C. OpenACC [10], which can port CPU-based codes to GPU-based systems with directives, is also available. Using GPU computation is expected to reduce the computation cost required in our target application.

On the other hand, recent GPUs tend to have cores specialized for AI applications. These cores have much higher peak performance (e.g. Tensor Core on NVIDIA Volta GPU [11]); however, they can work only for matrix-matrix multiplication in lower-precision data types. Applications without corresponding computations cannot gain the benefit from the high arithmetic performance of AI-specific hardware. In addition, it is generally known that some specific operations limit the performance of GPU computations. Data transfer between CPU and GPU is major bottleneck of the performance. Also, we have to consider that the rate of arithmetic performance increase is smaller than that of memory bandwidth increase. Low-order finite element solver has many memory bound computations; therefore, straightforward implementations cannot exhibit high arithmetic performance of GPUs. To utilize AI-specific hardware and reduce data transfer size and memory footprint, the introduction of transprecision computing with lower-precision data types [12] is important. As lower-precision numbers have smaller dynamic range and less accuracy, convergence of the solver without overflow or underflow is a big challenge. To accelerate our target application sufficiently using GPUs, it is important to design appropriate algorithm and implementation.

Therefore, this thesis aims at the followings: (1) Introduction of GPU computations to a baseline solver with single and double precision numbers, (2) Design of framework to compute multiple forward analyses using CPU-GPU heterogeneous computing, and (3) Further acceleration of the solver with transprecision computing in finite element earthquake simulations. The remainder of the thesis is organized as follows: In Chapter 2, we design our baseline codes for GPU-based systems and evaluate the performance. In Chapter 3, we apply our baseline solver for the estimation of 3-D inner structure with heuristic optimization. The content of Chapter 4 and Chapter 5 is not included in the abridged version. In Chapter 6, we describe conclusions and future prospects.

2 Baseline unstructured low-order finite-element solver targeting viscoelastic crustal deformation computations on GPU-based systems

2.1 Background

One of the targets of earthquake disaster reduction is the prediction of the place, magnitude, and time of earthquakes. One approach is estimate earthquake occurrence probability by comparing plate conditions at past occurrence of earthquakes with plate deformation conditions estimated by crustal deformation observation data at surface measurement points [13]. In this process, inverse analysis is required to estimate inter-plate displacement distribution using crust deformation data observed at the surface. In order to realize this inverse analysis, forward simulation tools computing elastic and viscoelastic crust deformation for a given fault slip distribution are under development.

In previous crust deformation analyses, simplified models such as horizontally stratified layers [14] were used. However, recent studies point out that the simplification of crust geometry has significant effects on the response [15]. Thus, computation considering the crust heterogeneity by methods such as finite-element methods is suitable. Recently, crust property data for such analysis purposes is being known, and crust deformation data measured at observation stations can be used. Thus, crust deformation analyses based on these high resolution data is being anticipated.

On the other hand, the computational domain of the crust deformation analysis is large, and considering that 1 km resolution is available, the computational size of the target problem becomes larger than the order of 10^8 degrees-of-freedom. For this simulation, introduction of GPU computation is assumed to be effective to reduce the required computation cost.

The target finite-element analysis has many parts that are memory bandwidth bound. Especially many random memory accesses are included in the sparse matrix-vector product kernel which is most computationally expensive. Compared to its high floating point performance, general GPU has a smaller memory bandwidth, which is further decreased when coalesced memory access cannot be performed. Thus, it is difficult to utilize the large arithmetic hardware capability of GPU architectures in conventional finite-element solvers. Reducing random access is important to improve efficiency of the GPU computation. In this study, we reduce random access of the major computational kernel targeting GPUs, and also introduce an algorithm that reduce the solver iterations, to accelerate from the previous solver. Here we use a multi-time step method together with a predictor to improve the initial solution of the iterative solver to improve the convergency of the iterative solver. In addition, by using several vectors for computation, we can reduce random memory access in the major sparse matrix-vector kernel and improve performance.

Section 2.2 explains the developed method. Section 2.3 shows the performance of the developed method on Piz Daint [16], which is a P100 GPU based supercomputer system. Section 2.4 shows an application example using the developed method. Section 2.5 summarizes this chapter and gives future prospects.

2.2 Methodology

We target elastic and viscoelastic crust deformation to a given fault slip. Following [14], the governing equation is

$$\sigma_{ij,j} + f_i = 0, \tag{2.1}$$

with

$$\begin{aligned} \dot{\sigma} &= \lambda \dot{\epsilon}_{kk} \delta_{ij} + 2\mu \dot{\epsilon}_{ij} - \frac{\mu}{\eta} \left(\sigma_{ij} - \frac{1}{3} \sigma_{kk} \delta_{ij} \right), & (2.2) \\ \epsilon_{ij} &= \frac{1}{2}(u_{i,j} + u_{j,i}), & (2.3) \end{aligned}$$

where σ_{ij} and f_i are the stress tensor and outer force. $(\dot{\ })$, $(\ )_{,i}$, δ_{ij}, η, ϵ_{ij}, and u_i are the first derivative in time, that in the ith direction in space, Kronecker delta, viscosity coefficient, strain tensor, and displacement, respectively. λ and μ are Lame's constants. Discretization of this equation by the finite-element method leads to solving a large system of linear equations. The developed method is based on viscoelastic analysis by [17], which can be described as follows (Algorithm 1 and 2).

An adaptive preconditioned conjugate gradient solver with Element-by-Element method [18], multi-grid method, and mixed-precision arithmetic is used in Algorithm 2. As most of the computational cost is in the inner loop of Algorithm 2, and it can be computed in single precision, we can reduce computational cost. In addition, we can reduce the data transfer size, and thus we can expect it to be suitable for GPU systems. Below, we call line 8 of Algorithm 2(a) as the inner coarse loop and line 10 of Algorithm 2(a) as the inner fine loop. The most computational costly kernel is the Element-by-Element kernel which computes sparse matrix-vector products. The Element-by-Element kernel computes the product of element stiffness matrix by vectors element wise, and adds the results for all elements to compute a global matrix vector product. As element matrices are computed on the fly, the data transfer size from memory can be reduced significantly. This leads to circumventing the memory bandwidth bottleneck, and thus is suitable for GPU architectures with relatively low memory bandwidth compared with its arithmetic capability. On the other hand, since many random data access is required when adding up element wise results, this data access becomes the bottleneck in this kernel. In order to improve the efficiency of this kernel, we introduce the methods below to reduce random accesses.

2.2.1 Parallel computation of multiple time steps

In the developed method, we solve four time steps in viscoelastic analysis in parallel following [19]. As the stress of the step before needs to be obtained before solving the next step, only one time step can be solved exactly. Here we compute until the error of the next time step (displacement) becomes smaller than prescribed threshold ϵ. The next three time steps are solved using the solutions of the steps before to estimate the solution. The estimated solution of the step before is used to update the stress state and outer force vector. By using this method, we can obtain estimated solutions for improving the convergency of the solver. In this method, four vectors can be computed simultaneously. In the Element-by-Element kernel, four values corresponding to the four time steps will be consecutive in memory address space. Therefore we can improve the computational efficiency when compared with conducting Element-by-Element kernel of one vector for four times. That is, the arithmetic count per iteration increases by approximately four times, but by the decrease in the number of iterations and the improvement of computational efficiency of the Element-by-Element kernel is expected to reduce the time-to-solution.

In order to improve convergency, it is important to estimate the initial solution of the fourth time step accurately. Thus we use the predictor as below. Until the 5th time step, no predictor is used. For predicting the 6, 7, and 8th time steps, we use the second order Adams-Bashforth predictor. For predicting the 9th step and on, we use a linear predictor. In this linear predictor, a linear regression based on the accurately computed 7 time steps are used to predict the future time step. As regressions based on higher order polynomials or exponential base functions may lead to jumps in the prediction, we will not use them in this study.

2.2.2 Reduction of atomic access

In the Element-by-Element kernel for GPUs in previous study [20], we need to add up element wise results to the global vector. We use a buffering method to reduce the number of accesses to the global vector to improve the efficiency of the Element-by-Element kernel. Regarding for NVIDIA GPU, we can utilize a shared memory, in which values can be referred among threads in the same `block`. The computation procedure is as below and also described in Fig. 2.1.

1. Group elements in to blocks, and store element wise results into a shared memory
2. Add up nodal values in shared memory using a precomputed table
3. Add up nodal values to global vector

We can expect performance improvement as summation of temporal results is mainly performed in preliminary reduction in a shared memory, which has wider bandwidth. In this scheme, the setting of block size is assumed to have some impact on its performance. By allocating more elements in a `Block`, we can improve the number of reduction of nodal values in shared memory. However, the total number of threads is constraint by the shared memory size. In addition, we need to synchronize threads in a `Block` when switching from element wise matrix-vector multiplication to data addition part, using large number of threads in a `Block` leads to increase in synchronization cost. Under this circumstances, we allocate 128 threads (32 elements $\times$ four time steps) per `Block`.

In GPU computation, SIMT composing of 32 threads is used [21]. When the number of computation differs between the 32 threads, warp divergence occurs and is expected to lead to decrease in performance. In reduction phase, we need to assign threads per node. However, since the number of elements per node differs significantly between nodes, we can expect large load imbalance among the 32 threads. Thus we sort the nodes according to the number of elements to be added up as described in Fig. 2.2. This leads to good load balance among the 32 threads, leading to higher computational efficiency.

This method on shared memory requires implementation by CUDA. We also use CUDA for inner product computation to improve the memory access pattern and thus improve efficiency. On the other hand, other computation such as vector addition and subtraction are memory bound, and thus use of OpenACC is expected to perform as well. Thus we use CUDA for these performance sensitive kernels, and use OpenACC for the other parts. The CUDA part is called via a wrapper function.

2.3 Performance Measurement

We measure performance of the developed method on hybrid nodes of Piz Daint[1].

2.3.1 Performance measurement of the Element-by-Element kernel

We use on P100 GPU on Piz Daint to measure performance of the Element-by-Element kernels. The target mesh is a tetrahedral mesh used for finite-element analysis with 959,128 elements. Its degrees-of-freedom is 4,004,319 in second-order tetrahedral mesh and 522,639 in first-order tetrahedral mesh. Here we compare four versions of the kernels. Table 2.1 summarizes the kernels. Case A corresponds to the conventional Element-by-Element kernel, and Case D corresponds to the proposed kernel.

Figures 2.3-a, 2.3-b show the normalized elapsed time per vector of the kernels in inner fine and coarse loops.

We can see that the use of four vectors, reduction, and reordering significantly changes the performance. In order to assess the time spent for data access part, we also indicate the time measured for the Element-by-Element kernel without computing the element wise matrix-vector products. We

[1] Piz Daint comprises of 1,431 $\times$ multicore compute node (Two Intel Xeon E5-2695 v4) and 5,320 $\times$ hybrid compute node (Intel Xeon E5-2690 v3 + NVIDIA Tesla P100) connected by Cray Aries routing and communications ASIC, and Dragonfly network topology.

can see that the data access is dominant in the Element-by-Element kernel on P100 GPUs, and thus it is the bottleneck. With the decrease in memory access by reduction, we can see that the elapsed time of the kernel has decreased. When compared to the performance in second-order tetrahedral mesh, the performance in first-order tetrahedral mesh was much more improved by reduction using shared memory. This effect can be confirmed by the number of call for atomic add to the global vector. In second-order tetrahedral mesh, atomic addition is performed 115,095,360 times in Case B and 43,189,848 times in Case D; thereby the number of calls is reduced by about 37%. Regarding first-order tetrahedral mesh, atomic addition is performed 46,038,144 times in Case B and 10,786,920 times in Case D; thus the number of calls is reduced by about 23%. In total, we can see that the computational performance of the kernel has improved by 3.3 times in first-order tetrahedral mesh and 2.2 times in second-order tetrahedral mesh when comparing the conventional kernel (Case A) and the developed kernel (Case D).

2.3.2 Comparison of solver performance

We compare the developed solver to the original solver without time parallel algorithm and reduction in shared memory access costs. The same tolerances of solvers is used for both methods, $\epsilon = 10^{-8}$ is used for the outer loop, $(\bar{\epsilon}_c^{in}, N_c) = (0.1, 300)$ is used for the inner coarse loop, and $(\bar{\epsilon}^{in}, N) = (0.2, 30)$ is used for the inner fine loop. These tolerance numbers are selected to minimize the elapsed time for both solvers. We use time step increment $dt = 2592000$ s with $N_t = 300$ time steps, and measure performance of the viscoelastic computation part (time step 2 to 300).

A model with 41,725,739 degrees-of-freedom and 30,720,000 second-order tetrahedral elements is computed using 32 Piz Daint nodes. Figure 2.4 shows the number of iterations and elapsed time of the solvers. By using the multistep predictor, the number of iterations of the most computationally costly inner coarse loop has decreased by 2.3 times. Together, as the Element-by-Element kernel performance has improved significantly, the total elapsed time has decreased by 2.79 times.

2.3.3 Weak scaling

Next we measure weak scalability up to 1024 Piz Daint nodes. Table 2.2 shows the models used for measurement.

Using the 32 GPU problem in the previous subsection as one unit, these models are made by making a periodic problem set with problem size according to the number of Piz Daint nodes. The elapsed time for the viscoelastic computation (time step 2 to 300) is shown in Fig. 2.5-a. We can see reasonable scalability, however, it is lower than the results of previous studies such as [22]. Main reason for this is the performance decrement in MPI_Allreduce. In this solver, we call MPI_Allreduce to add up results of inner vector product, whose size is four Bytes or eight Bytes $\times$ four time steps. We measured elapsed time required only for this MPI_allreduce and estimated the time required for MPI_Allreduce considering the number of calls in the solver. The estimated breakdown is described in Fig. 2.5-b. The scalability of MPI_Allreduce part is getting worse as the number of compute nodes increases; while the other part is roughly constant among all models. To improve this scalability, further discussion for MPI_Allreduce via other GPU-based supercomputer such as Summit in which reduction operation is supported at the hardware level is needed. Also we are now trying to use communication avoiding CG methods [23] to circumvent this bottleneck.

2.4 Application Example

We apply the developed solver to a viscoelastic deformation problem following a hypothetical earthquake on the Hellenic arc subduction interface, which affects deformation measured in Greece and across the Eastern Mediterranean. We selected this Hellenic region, because recent analysis of time-scale bridging numerical models suggests that the large amount of sediments subducting could mean that a larger than anticipated M 9 earthquake might be able to occur in this highly populated region [24]. To model the complete viscoelastic response of the system we simulate a large depth range, including the Earth's crust, lithosphere and complete mantle down to the core boundary. The target domain is of size 3,686 km $\times$ 3,686 km $\times$ 2,857 km. Geometry data of layered structure is given in spatial resolution of 1 km [25].

To fully reflect the geometry data into the analysis model, we set resolution of finite-element model to 0.9 km (second-order tetrahedral element size is 1.8 km). As this becomes a large scale problem, we use a parallel mesh generator capable of robust meshing of large complex shaped multiple material problems [26,19]. This leads to a finite-element model of 589,422,093 second-order tetrahedral elements, 801,187,352 nodes, and 2,403,562,056 degrees-of-freedom shown in Figure 2.6a-d. We can see that the layered structure geometry is reflected into the model. We input a hypothetical fault slip in the direction of the subduction, that is, slip with (dx, dy, dz) = (25, 25, -10) m, at the subduction interface separating the continental crust of Africa and Europe in the center of the model with diameter of 250 km. Following this hypothetical M 9 earthquake we compute the elastic coseismic surface deformation and postseismic viscoelastic deformation due to viscoelastic relaxation of the crust, lithosphere and mantle.. Following [17], a split node method is used to input the fault dislocation, and time step increment dt is set to 30 days (2,592,000 s). The analysis of 2,000 time steps took 4587 s using 512 P100 GPUs on Piz Daint.

Figure 2.6e,f shows the surface deformation snapshots. We can see that elastic coseismic response as well as the viscoelastic response is computed reflecting the 3D geometry and heterogeneity of crust. We can expect more realistic response distribution by inputting fault slip distributions following current solid earth science knowledge.

2.5 Summary

We developed a fast unstructured finite-element solver for viscoelastic crust deformation analysis targeting GPU-based computer. The target problem becomes very computationally costly since it requires solving a problem with more than 10^8 degrees-of-freedom. In order to improve performance of this analysis, the random data access in Element-by-Element method in matrix-vector products was the bottleneck. In this study, we proposed a reduction method to use shared memory of GPUs. We also introduced multi-step predictor and linear predictor to improve the convergency of the solver. Performance measurement on Piz Daint showed 2.79 times speedup from the previous solver. We also seen reasonable scalability up to 1024 nodes of Piz Daint.

Algorithm 1 Coseismic/postseismic crustal deformation computation against given fault displacement. $(\ )^n$ is the variables in the nth timestep. dt is time increment and $\boldsymbol{\beta}^n = \mathbf{D}^{-1}\mathbf{A}\boldsymbol{\sigma}^n$, where $\boldsymbol{\sigma}^n = (\sigma^n_{11}, \sigma^n_{22}, \sigma^n_{33}, \sigma^n_{12}, \sigma^n_{23}, \sigma^n_{13})^{\mathrm{T}}$. $\mathbf{B}$ is the displacement-strain transformation matrix and $\mathbf{D}$ and $\mathbf{A}$ are 6×6 matrices indicating material properties. $\mathbf{D}^v = (\mathbf{D}^{-1} + \alpha dt \boldsymbol{\beta}')$, where α is a controlling parameter and $\boldsymbol{\beta}'$ is the Jacobian matrix of $\boldsymbol{\beta}$.

```
1:  Compute $\mathbf{f}^1$ by split-node technique
2:  Solve $\mathbf{K}\mathbf{u}^1 = \mathbf{f}^1$
3:  $\{\boldsymbol{\sigma}^j\}_{j=1}^4 \Leftarrow \mathbf{D}\mathbf{B}\mathbf{u}^1$
4:  $\{\delta\mathbf{u}_j\}_{j=1}^4 \Leftarrow \mathbf{0}$
5:  $i \Leftarrow 2$
6:  while $i \le N_t$ do
7:      if $5 \le i \le 8$ then
8:          Compute initial guess solution by 2nd-order Adams-Bashforth method $\delta\mathbf{u}^{i+3} \Leftarrow \mathbf{u}^i - 3\mathbf{u}^{i+1} +$
            $2\mathbf{u}^{i+2}$
9:      end if
10:     if $i \ge 9$ then
11:         Compute initial guess solution by linear predictor $\delta\mathbf{u}^{i+3} \Leftarrow (-17\delta\mathbf{u}^{i-7} - 10\delta\mathbf{u}^{i-6} - 3\delta\mathbf{u}^{i-5} +$
            $4\delta\mathbf{u}^{i-4} + 11\delta\mathbf{u}^{i-3} + 18\delta\mathbf{u}^{i-2} + 25\delta\mathbf{u}^{i-1})/28$
12:     end if
13:     while $\|\mathbf{K}^v\mathbf{u}^i - \mathbf{f}^i\| > \epsilon$ do
14:         $\{\mathbf{f}^j\}_{j=i}^{i+3} \Leftarrow \sum_k \int_{\Omega_e^k} \mathbf{B}^T(dt\mathbf{D}^v\{\boldsymbol{\beta}^j\}_{j=i}^{i+3} - \{\boldsymbol{\sigma}^j\}_{j=i}^{i+3})d\Omega_e + \mathbf{f}^0$
15:         Solve $\mathbf{K}^v\{\mathbf{u}^j\}_{j=i}^{i+3} = \{\mathbf{f}^j\}_{j=i}^{i+3}$ using Algorithm 2
16:         $\{\boldsymbol{\sigma}^j\}_{j=i+1}^{i+3} \Leftarrow \{\boldsymbol{\sigma}^j\}_{j=i}^{i+2} + \mathbf{D}^v(\mathbf{B}\{\delta\mathbf{u}^j\}_{j=i}^{i+2} - dt\{\boldsymbol{\beta}^j\}_{j=i}^{i+2})$
17:         $\mathbf{u}^i \Leftarrow \mathbf{u}^{i-1} + \delta\mathbf{u}^i$
18:         $\boldsymbol{\sigma}^{i+4} \Leftarrow \boldsymbol{\sigma}^{i+3} + \mathbf{D}^v(\mathbf{B}\delta\mathbf{u}^{i+3} - dt\boldsymbol{\beta}^{i+3})$
19:         $i \Leftarrow i + 1$
20:     end while
21: end while
```

Algorithm 2 The iterative solver to obtain a solution $\mathbf{u}$. $(\ )_c$ are variables in first-order tetrahedral model, while others are in second-order tetrahedral model. $(\ \bar{}\ )$ represents single-precision variables, while the others are double-precision variables. The input variables are $\mathbf{K}, \overline{\mathbf{K}}, \overline{\mathbf{K}}_c, \overline{\mathbf{P}}, \mathbf{u}, \mathbf{f}, \overline{\epsilon}_c^{in}, N_c, \overline{\epsilon}^{in}$, and N. The other variables are temporal. $\overline{\mathbf{P}}$ is a mapping matrix from the coarse model to the target model. This algorithm computes four vectors at the same time, so coefficients have the size of four and vectors have the size of $4 \times$ DOF. All computation steps in this solver, except MPI synchronization and coefficient computation, are performed in GPUs.

(a) Outer loop

```
1:  $\mathbf{r} \Leftarrow \sum \mathbf{K}_e \mathbf{u}_e$
2:  $\mathbf{r} \Leftarrow \mathbf{f} - \mathbf{r}$
3:  $\beta \Leftarrow 0$
4:  $\overline{\mathbf{u}} \Leftarrow \overline{\mathbf{M}}^{-1} \mathbf{r}$
5:  $\overline{\mathbf{r}}_c \Leftarrow \overline{\mathbf{P}}^T \mathbf{r}$
6:  $\overline{\mathbf{u}}_c \Leftarrow \overline{\mathbf{P}}^T \overline{\mathbf{u}}$
7:  Solve $\overline{\mathbf{u}}_c = \overline{\mathbf{K}}_c^{-1} \overline{\mathbf{r}}_c$ in (b) with $\overline{\epsilon}_c^{in}$ and $N_c$
8:  $\overline{\mathbf{u}} \Leftarrow \overline{\mathbf{P}} \overline{\mathbf{u}}_c$
9:  Solve $\overline{\mathbf{u}} = \overline{\mathbf{K}}^{-1} \overline{\mathbf{r}}$ in (b) with $\overline{\epsilon}^{in}$ and $N$
10: $\mathbf{u} \Leftarrow \overline{\mathbf{u}}$
11: $\mathbf{p} \Leftarrow \mathbf{z} + \beta \mathbf{p}$
12: $\mathbf{q} \Leftarrow \sum \mathbf{K}_e \mathbf{p}_e$
13: $\rho \Leftarrow (\mathbf{z}, \mathbf{r})$
14: $\gamma \Leftarrow (\mathbf{p}, \mathbf{q})$
15: $\alpha \Leftarrow \rho / \gamma$
16: $\mathbf{r} \Leftarrow \mathbf{r} - \alpha \mathbf{q}$
17: $\mathbf{u} \Leftarrow \mathbf{u} + \alpha \mathbf{p}$
```

(b) Inner loop

```
1:  $\overline{\mathbf{e}} \Leftarrow \sum \overline{\mathbf{K}}_e \overline{\mathbf{u}}_e$
2:  $\overline{\mathbf{e}} \Leftarrow \overline{\mathbf{r}} - \overline{\mathbf{e}}$
3:  $\overline{\beta} \Leftarrow 0$
4:  $i \Leftarrow 1$
5:  $\|\overline{\mathbf{e}}_1\|^2 / \|\overline{\mathbf{r}}_1\|^2 > \overline{\epsilon}$ and $N > i$
6:  $\overline{\mathbf{z}} \Leftarrow \overline{\mathbf{M}}^{-1} \overline{\mathbf{e}}$
7:  $\overline{\rho}_a \Leftarrow (\overline{\mathbf{z}}, \overline{\mathbf{e}})$
8:  if $i > 1$ then
9:      $\overline{\beta} \Leftarrow \overline{\rho}_a / \overline{\rho}_b$
10: end if
11: $\overline{\mathbf{p}} \Leftarrow \overline{\mathbf{z}} + \overline{\beta} \overline{\mathbf{p}}$
12: $\overline{\mathbf{q}} \Leftarrow \sum \overline{\mathbf{K}}_e \overline{\mathbf{p}}_e$
13: $\overline{\gamma} \Leftarrow (\overline{\mathbf{p}}, \overline{\mathbf{q}})$
14: $\overline{\alpha} \Leftarrow \overline{\rho}_a / \overline{\gamma}$
15: $\overline{\rho}_b \Leftarrow \overline{\rho}_a$
16: $\overline{\mathbf{e}} \Leftarrow \overline{\mathbf{e}} - \overline{\alpha}\, \overline{\mathbf{q}}$
17: $\overline{\mathbf{u}} \Leftarrow \overline{\mathbf{u}} + \overline{\alpha}\, \overline{\mathbf{p}}$
18: $i \Leftarrow i + 1$
```

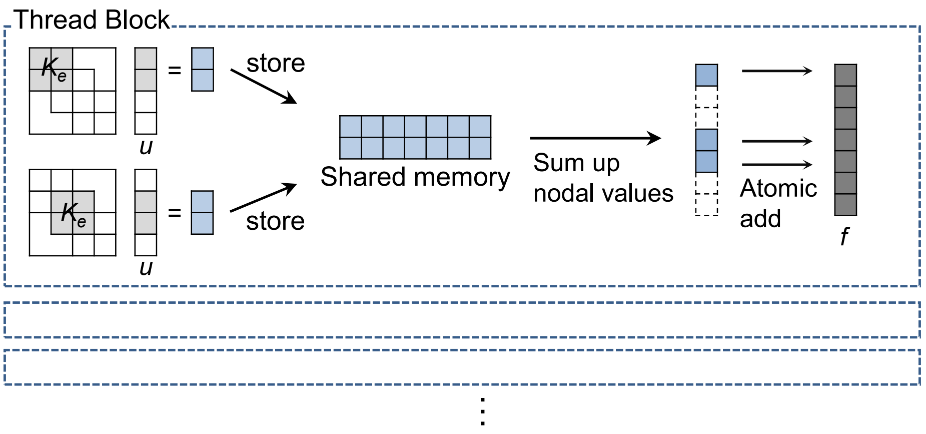

Fig.2.1: Rough scheme for reduction in Element-by-Element kernel to compute $\mathbf{Ku} = \mathbf{f}$. This figure mainly shows the computation in one `block`.

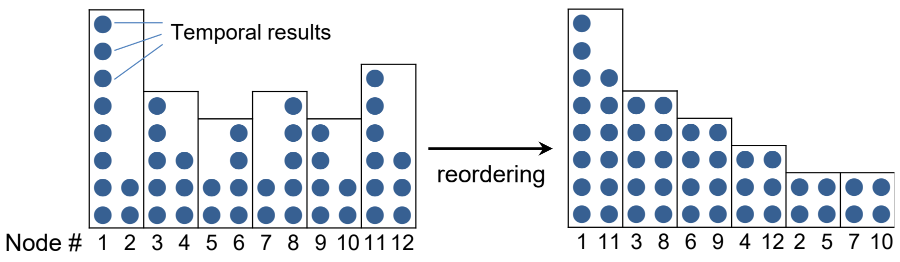

Fig.2.2: Reordering of reduction table. Temporal results are aligned in corresponding node number. For simplicity, we assume there are two threads per warp and 12 nodes in the thread block. Load balance in warp is improved by reordering.

Table2.1: Configuration of the Element-by-Element for performance comparison

Case	# of vectors	Reduction using shared memory	Reordering of nodes in reduction
A	1	x	-
B	4	x	-
C	4	o	x
D	4	o	o

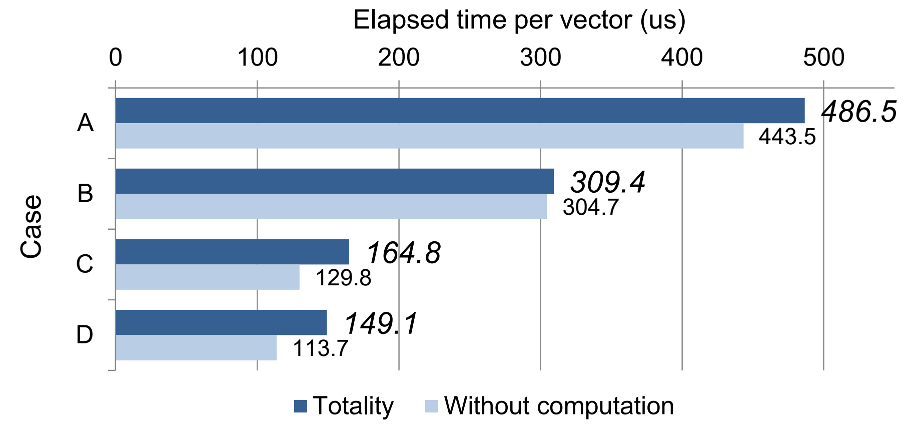

Fig.2.3-a: first-order tetrahedral mesh

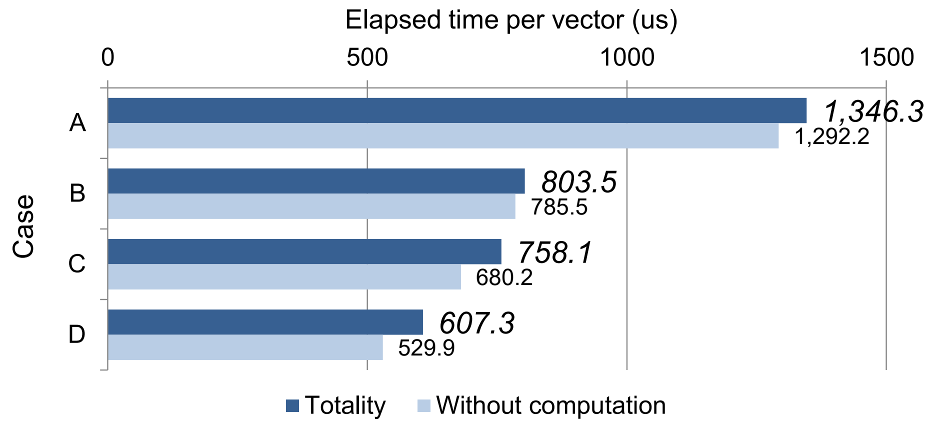

Fig.2.3-b: second-order tetrahedral mesh

Fig.2.3: Elapsed time per Element-by-Element kernel call. Elapsed times are divided by four when using four vectors.

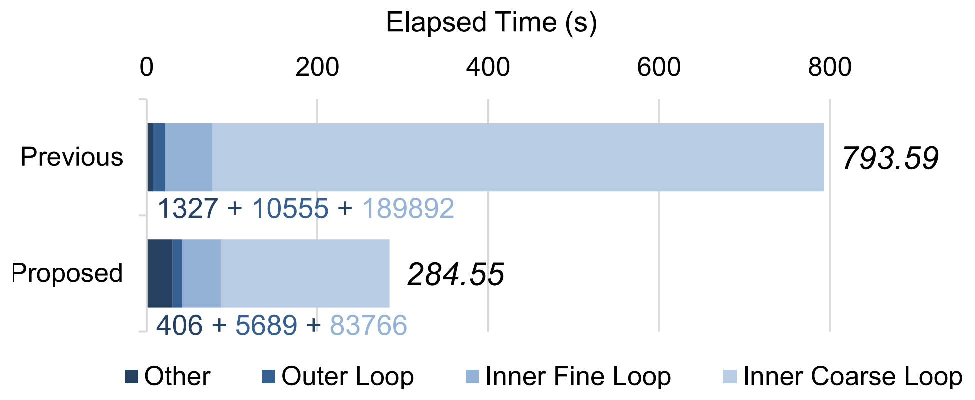

Fig.2.4: Performance comparison of the entire solver. The numbers of iteration for outer loop, inner fine loop, and inner coarse loop are described below each bar.

Table2.2: Model settings for weak scaling on Piz Daint. The number of GPU equals the number of compute nodes.

Model	# of GPUs	Degrees of Freedom	# of elements	# of elements/GPU
No.1	32	125,177,217	30,720,000	960,000
No.2	128	496,736,817	122,880,000	960,000
No.3	256	992,038,737	245,760,000	960,000
No.4	512	1,978,979,217	491,520,000	960,000
No.5	1024	3,955,080,657	983,040,000	960,000

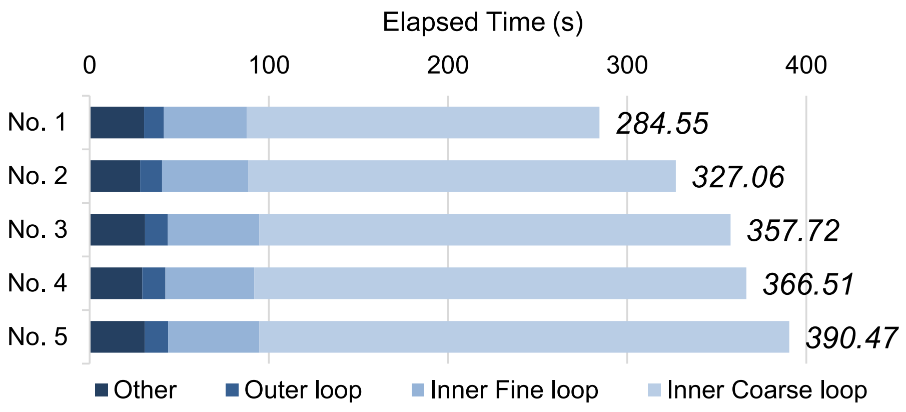

Fig.2.5-a: Breakdown for each loop in the solver

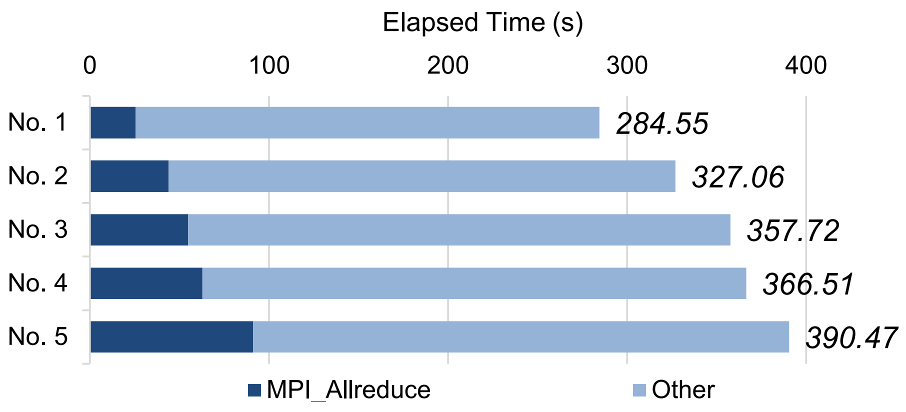

Fig.2.5-b: Breakdown for MPI_Allreduce

Fig.2.5: Weak scaling for models indicated in Table 2.2 on Piz Daint

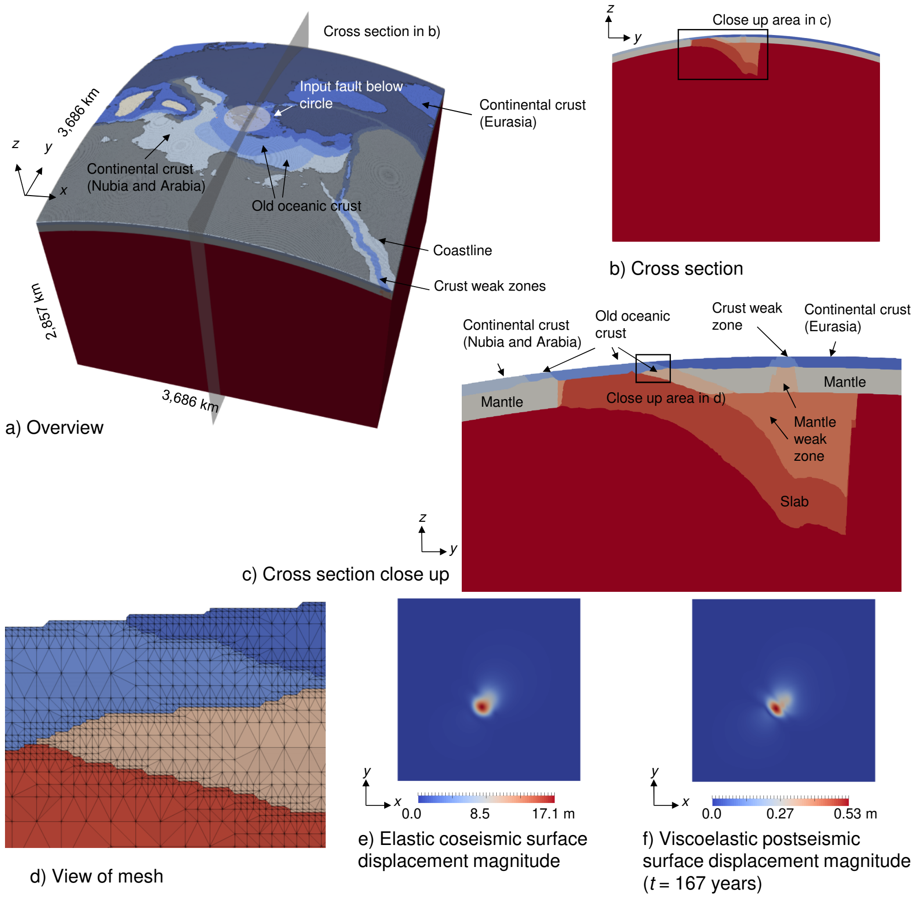

Fig.2.6: Finite-element mesh for application problem. The 10 layered crust is modeled using 0.9 km resolution mesh. Elastic coseismic and viscoelastic postseismic displacements. a) Overview of finite-element mesh with position of input fault and position of cross section. b) Cross section of finite element mesh. c) Close up area in the cross section. d) Close up view of mesh. e) Elastic coseismic response and f) viscoelastic postseismic response.

Acknowledgment

Reprinted by permission from Springer Nature Customer Service Centre GmbH: Springer, Lecture Notes in Computer Science, vol 10861, Viscoelastic Crustal Deformation Computation Method with Reduced Random Memory Accesses for GPU-Based Computers, Takuma Yamaguchi, Kohei Fujita, Tsuyoshi Ichimura, Anne Glerum, Ylona van Dinther, Takane Hori, Olaf Schenk, Muneo Hori, and Lalith Wijerathne, ©(2018)

3 Heuristic optimization of three-dimensional inner structure with CPU-GPU heterogeneous wave computing

3.1 Background

Numerical simulations with large degrees of freedom are becoming feasible due to the development of computation environments and algorithms. Accordingly, more reliable models are required to obtain more reliable results for the target domain with complex structures. This approach has been discussed in various fields including biomedicine [27], [28], and it is also important for the numerical simulation of earthquake disasters. It is rational that we allocate resource and take countermeasures after detecting an area with potentially substantial damage. We can apply numerical simulation for estimating damages. [29] found that the geometry of the target domain significantly affects the distribution of displacement on the ground surface and strain in underground structures. To undertake well-suited countermeasures, three-dimensional unstructured finite element analysis is preferred, as it considers complex geometry. This analysis results in problems with large degrees of freedom because it targets large domains with high resolution. The computation mentioned above has become more attainable due to the development of the computation environment and analysis methods for CPU-based large-scale systems [2]. However, inner soil structure is not available with high resolution, which hampers the generation of finite element models. On the ground surface, [30] is used as elevation data for Japan. With the advance of sensing technology, it is possible to observe earthquake waves on many points on the ground surface. It is desirable that we estimate a finite element model which can reproduce observational data on the ground surface and conduct analyses using an estimated model. On the other hand, it is difficult to measure the underground structure with high accuracy and resolution.

One of realistic ways to address this issue is introduction of an optimization method using observation data on the ground surface for a micro earthquake. If we can generate many finite element models and conduct wave propagation analyses for each model, it is possible to select a model with the maximum likelihood for available observation data. Using optimized models will increase the reliability of the analyses. There are some gradient-based methods for optimization as [31] proposed for three-dimensional crustal structure optimization. These methods have the advantage that the number of trials is small; however, they may be difficult to escape from a local solution if control parameters have low sensitivity to an error function. Thus, this study focuses on heuristic methods such as simulated annealing so that we can reach the global optimal solution robustly. The expected optimization requires many forward analyses, and the challenge is an increase in the computation cost for many analyses with large degrees of freedom.

We use our solver algorithm proposed in Chapter 2. The computation time can be reduced by using parallel computation with many GPU cores. However, it is known that GPU computation requires the consideration of memory access and communication cost to attain better performance. This chapter proposes an algorithm that combines very fast simulated annealing and wave propagation analyses and repeats generation of finite element models and the computation of the solver for estimation of inner soil structure. Some computations in our optimizer are not suitable for GPU computation. Thus, computer resources are allocated so that we can benefit further from the introduction of GPU computation. A finite element solver appropriate for GPU computation is used to reduce the computation time in the solver, which is the most computationally expensive part. At the same time, generation of finite element models, which requires serial operations, is computed on CPUs so that computation time for model generation can be overlapped. We confirm that the inner soil structure has a large effect on the results and that our proposed method can estimate the soil structure with sufficient accuracy for damage estimation.

3.2 Methodology

For estimating the inner structure of the target domain, this study proposes a method to conduct many wave propagation analyses and accept the inner structure with its maximum likelihood. In this study, optimization targets the estimation of boundary surfaces of the domain that has different material properties. For simplicity and for the purposes of this study, we have assumed that the target domain has a stratified structure and that target parameters for optimization are an elevation of the boundary surface on control points which are located at regular intervals in the x and y directions. A boundary surface is generated in the target domain by interpolating elevation on control points using linear functions. Figure 3.1 depicts the scheme for optimization. In this scheme, we conduct finite element analyses for evaluation of parameters many times in very fast simulated annealing. Our optimizer is designed so that the generation of a finite element model and the finite element solver, which account for the large proportion of the whole computation time, can be computed at the same time by CPUs and GPUs, respectively. We describe the details for each part of our optimizer in the following parts.

3.2.1 Very Fast Simulated Annealing

Very fast simulated annealing, which is a heuristic optimization method for problems with many control parameters [32], is applied. Simulated annealing has a parameter that corresponds to temperature, and the temperature decreases as the number of trials increases. We search and evaluate parameters in the following manner. First, trial parameters are selected randomly based on current parameters. The search parameter domain is wider when the temperature is higher. The evaluation value of trial parameters and that of previous ones are compared, and if the evaluation value is improved, parameters are always updated. Even if the evaluation value is worse, parameters are updated with a high degree of probability while the temperature is high. By repeating this procedure, this method can move out of the local optimal solution and find the global optimal solution robustly. To evaluate the parameters, finite element analysis is conducted. We assume that we have many observation points on the surface of the target domain. Our error function is defined by the time histories of displacement in the analyses and observation data on observation points. The actual error function is defined in Section 3.

In very fast simulated annealing, temperature at the k-th trial is defined using initial temperature T_0 and the number of control points D as $T_k = T_0\exp(-ck^{\frac{1}{D}})$, where parameter c is defined by T_0, D, lowest temperature T_f, and the number of trials k_f as $T_f = T_0\exp(-m)$, $k_f = \exp n$, and $c = m\exp(-\frac{n}{D})$. The initial temperature, lowest temperature, and number of iterations depend on problems. A certain number of iterations are conducted for this simulation, though we can stop searching by other conditions, including acceptance frequency of new solutions.

3.2.2 Finite Element Analyses

In the scheme, we must conduct more than 10^3 finite element analyses; thus, it is essential to conduct these analyses in a realistic timeframe. We target linear wave propagation analyses. The target equation is $\left(\frac{4}{dt^2}\mathbf{M} + \frac{2}{dt}\mathbf{C} + \mathbf{K}\right)\mathbf{u}_n = \mathbf{f}_n + \mathbf{C}\mathbf{v}_{n-1} + \mathbf{M}\left(\mathbf{a}_{n-1} + \frac{4}{dt}\mathbf{v}_{n-1}\right)$, where $\mathbf{u}$, $\mathbf{v}$, $\mathbf{a}$, and $\mathbf{f}$ are displacement, velocity, acceleration, and force vector, respectively, and $\mathbf{M}$, $\mathbf{C}$, and $\mathbf{K}$ are mass, damping, and stiffness matrix, respectively. In addition, dt is the time increment, and n is the number of time steps. For the damping matrix $\mathbf{C}$, we use Rayleigh damping and compute it by linear combination as $\mathbf{C} = \alpha\mathbf{M} + \beta\mathbf{K}$. Coefficients α and β are set so that $\int_{f_{\min}}^{f_{\max}}(h - \frac{1}{2}(\frac{\alpha}{2\pi f} + 2\pi f\beta))^2 df$ is minimized, where $f_{\max}$, $f_{\min}$, and h are maximum targeting frequency, minimum targeting frequency, and damping ratio. We apply Newmark-β method with $\beta = 1/4$ and $\delta = 1/2$ for time integration. Vectors $\mathbf{v}_n$ and $\mathbf{a}_n$ can be described as $\mathbf{v}_n = -\mathbf{v}_{n-1} + \frac{2}{dt}(\mathbf{u}_n - \mathbf{u}_{n-1})$, $\mathbf{a}_n = -\mathbf{a}_{n-1} - \frac{4}{dt}\mathbf{v}_{n-1} + \frac{4}{dt^2}(\mathbf{u}_n - \mathbf{u}_{n-1})$. We obtain displacement vector $\mathbf{u}_n$ by solving the equation above and updating vectors $\mathbf{v}_n$ and $\mathbf{a}_n$. Computation in the finite element solver and generation of finite element models are most computationally expensive parts in our optimizer.

Finite Element Solver

We extend our finite element solver for crustal deformation computation in Chapter 2.
The solver combines a conjugate gradient method with adaptive preconditioning, geometric multigrid method, and mixed precision arithmetic to reduce the amount of arithmetic counts and data transfer size. In the solver, sparse matrix vector multiplication is computed by the Element-by-Element (EbE) method. It computes element matrix on-the-fly and reduces the memory access cost. Specifically, the multiplication $\mathbf{y} = \mathbf{A}\mathbf{x}$ is computed as $\mathbf{y} = \sum_{i=1}^{ne}(\mathbf{Q}^{(i)T}(\mathbf{A}^{(i)}(\mathbf{Q}^{(i)}\mathbf{x})))$, where ne is the number of elements in the domain, $\mathbf{Q}^{(i)}$ is a mapping matrix from local node numbers in the i-th element to

the global node numbers, and $\mathbf{A}^{(i)}$ is the i-th element matrix and satisfies $\mathbf{A} = \sum_{i=1}^{ne} \mathbf{Q}^{(i)T}\mathbf{A}^{(i)}\mathbf{Q}^{(i)}$. In this problem, $\mathbf{A}^{(i)} = \frac{4}{dt^2}\mathbf{M}^{(i)} + \frac{2}{dt}\mathbf{C}^{(i)} + \mathbf{K}^{(i)}$. The entire part of the solver is implemented in the multiple GPUs using CUDA. To exhibit higher performance using GPUs, we have to reduce the operations that are not suitable for GPU computation; thus, we modify the algorithm of the solver. We overlap computation and communication as described in [33]. In the domain of each MPI process, some elements are adjacent to domains of other MPI processes and require point-to-point communications, and others do not require these communications. First we compute elements that require data transfer among other GPUs. Next we communicate with other GPUs while we are computing elements that do not require data transfer. By following this procedure, it is possible to overlap MPI point-to-point communication in the solver.

In the conjugate gradient solver, the coefficients are derived from the result of inner product calculations so that orthogonal residual vector and A-orthogonal searching vector can be generated to those in the previous iteration, respectively. When multiple GPUs are used with MPI, calculations of these coefficients require data transfer and synchronization among MPI processes such as MPI_Allreduce. Thus, they become relatively time-consuming taking into account that other computations including vector operations and sparse matrix vector multiplication are accelerated by GPUs. In our solver, we employ the method described in [23]. This algorithm requires one MPI_Allreduce per iteration, which halves the number of MPI_Allreduce per iteration in the original conjugate gradient method. The amount of vector operation increases in this scheme. However, the reduction of calculations of coefficients is more effective for GPU-based systems.

Generation of the Finite Element Model

We automatically generate finite element model using the method by [17]. Its procedure includes serial computation; thus, we cannot apply GPU computation for this part. Generation of finite element models can account larger proportion of the whole elapsed time, which is not negligible compared to the computation time in the finite element solver. Therefore, we design our optimizer so that it is possible to generate a finite element model for the next trial on CPUs while wave propagation analysis is computed on GPUs. All of the main computation in the solver can be computed in GPUs, so we can assign only one core of CPUs for each GPU and this has little effect on the performance of the solver. Other cores in CPUs are assigned for the generation of finite element models. Program of the model generation is created separately from that of the finite element solver and we executed them asynchronously using a shell-script. Output files are shared in the file system and controlled so that they are updated in correct timing. By allocating heterogeneous computer resource as mentioned above, it is possible to overlap model generation with GPU computation.

3.3 Application Example

We use our developed optimizer to estimate soil structure. Our target domain has two layers, and we define their boundary surface. We use IBM POWER System AC922 for computation, which has two POWER9 CPUs (16 cores, 2.6 GHz) and four NVIDIA Volta V100 GPUs. We assign one CPU core to each GPU for finite element analysis with MPI and we use the remaining 28 CPU cores for the model generation with OpenMP.
The target domain is 300 m $\times$ 400 m $\times$ 75 m and the resolution is 2.5 m at the maximum. Elevation data at the surface are available. They are flat and we set them as $z = 0$ m. A finite element model with approximately 3,000,000 degrees of freedom is generated. Figure 3.3 shows one of the FE models in the analysis. Control points are located at regular intervals in the x and y directions. We notate the elevation of the hard layer on points $(x, y) = (100i, 100j)$(i=0-3, j=0-4) as α_{ij}(m). The points $x = 0$, $x = 300$, $y = 0$, and $y = 400$ are the edges of the domain, and we assume $\alpha_{ij} = 0$ for these points. The parameters for optimization are α_{ij}(i=1-2, j=1-3). Initial parameters and reference parameters, which are true, are shown in Table 3.1.

We assume the information from the boring survey are available at points $(x, y) = (50, 50)$, $(150, 350)$, $(200, 100)$. Elevations at these points are -7.01 m, -5.97 m, and -16.3 m, respectively, and these elevations are interpolated to make the initial boundary surface. The distributions of the boundary surface for initial and reference models are described as Fig. 3.2.

In this problem, material properties of the soil structure are deterministic. These properties are described in Table 3.2.

Input waves for wave propagation analyses can be obtained by pulling back observed waves on the ground surface. In this chapter, we assume that input waves are generated by micro earthquakes, and linear analysis can be applied. It is then possible to use Ricker wave as our input wave. We derive amplification functions from observation data and pulled back waves. Using these functions, it is possible to estimate observation data when we input Ricker wave. These operations reduce time steps for wave propagation analyses and the entire computation time. Ricker waves, represented as $(1 - 2\pi^2 f_c^2 (t - t_c)^2)\exp(-\pi^2 f_c^2 (t - t_c)^2)$, are input as x and y components of velocity at the bottom of models. t is time in second, f_c is central frequency and t_c is central time. For this application example, the target frequency is as much as 2.5 Hz and we set the period of each analysis to 2.56 seconds. Considering these settings, we set $(f_c, t_c) = (0.8, 1.2)$. Time increment of the analysis is 0.01 second; thus each wave propagation analysis requires computation for 256 times steps. We set two cases for observation points. In case 1, we allocate 35 observation points defined as $(x, y) = (50i, 50j)$ (i=1-5, j=1-7) and in case 2, observation points are $(x, y) = (-50+100i, -50+100j)$ (i=1-3, j=1-4) and the number of points is 12. We use an error function as follows; $Error = \frac{1}{np}\sum_{i=1}^{np}\sum_{j=1}^{3}\int_0^{f_{\max}} |F[v_{ij}] - F[\bar{v_{ij}}]| df$, where np is the number of observation points, and $f_{\max}$ is the maximum targeting frequency, which is 2.5 Hz in our paper. v is the time history of x, y, and z components of velocity on each observation point. Values with an over-line corresponds to the observation data. In addition, $F[\ ]$ corresponds to the discrete Fourier transformation. In other words, this error function is the total sum of absolute values of difference for frequency components on observation points. These settings mentioned above are the same as settings in [34].

In our proposed method, we generate finite element models for the next trial and conduct wave propagation analysis at the same time. In simulated annealing, we generate next trial parameters after current trial parameters are adopted or rejected. It is desirable that we generate two models in cases that trial parameters are adopted and rejected while we are conducting wave propagation analysis; however, generation of finite element model twice takes more time than the computation in our finite element solver. Parameters in these problem settings are thought to be rejected with high probability. Thus, we generate a finite element model with prediction that trial parameters will be rejected. When trial parameters are adopted, we regenerate next finite element models for updated parameters. This regeneration has a small effect on the whole computation time. The number of control points in very fast simulated annealing $D = 6$. Also, we set the number of trials $k_f = 1500$ and $c = 4.2975$. This c satisfies that parameters which increase the value of the error function by ΔE are adopted with the probability of 80% at the initial temperature and parameters which increase the value of the error function by $\Delta E \times 10^{-5}$ are adopted with the probability of 0.1% at the lowest temperature, where ΔE is the value of error function obtained in the initial model. The history of error function is described in Figure 3.4 and parameters are estimated as Table 3.3.

Optimization of both case 1 and 2 adopted trial parameters 51 times in 1,500 trials. Trial parameters are rejected with the probability of more than 90% and we find that the generation of finite element model is mostly overlapped by the computation in the solver. Compared to case 2, case 1 with more observation points estimated the soil structure more accurately. Our previous study [34] used a multigrid stochastic search algorithm and optimized the same parameters in meters. The numbers of iteration were 3,000 in case 1 and 1,300 in case 2; thereby we found that parameters were efficiently optimized with higher accuracy as the number of trials by the very fast simulated annealing was 1,500.

For confirmation of the optimized model, we conduct wave propagation analysis with parameters obtained in case 1. Figure 3.5 is the distribution of the displacement on the ground surface at time $t = 2.20$ s and Fig. 3.6 is the time history of the velocity on point $(x, y, z) = (150, 200, 0)$.

Judging from these figures, we can confirm that the results by optimized model and reference model are consistent.

Here we evaluate the performance of the computation in our optimization. The elapsed time for our solver part is about 18 s per trial. [34] computed wave propagation analysis in 263 s for a finite element model with 274,041 degrees of freedom using Intel Xeon E5-4667 v3 CPU. Our GPU-based solver has achieved about 160-fold speeding up per problem size, although it is difficult to compare the performance on different systems. Here we use peak memory bandwidth to evaluate the speeding up ratio, as general finite element analyses are memory bandwidth bound computations. Intel Xeon E5-4667 v3 CPU has 68 GB/s and four NVIDIA V100 GPUs have 900 GB/s $\times$ 4 = 3,600 GB/s of memory bandwidth. We attained higher speeding up ratio than the ratio of peak memory bandwidth; this indicates that we utilize GPU computation. The optimization in case 1 was computed in 13 h 32 min. It took about 10 s for the generation of each model; thus, the whole elapsed time would be 13 h 32 min + 10 s $\times$ 1500 = 17 h 42 min and increase by 30% if model generation and finite element solver were computed sequentially. Thereby we confirmed that efficient allocation of computer resources is important for this optimization.

Finally, we conduct a non-linear ground shaking analysis using the optimized model. The methods are the same as [2]. We input wave observed in the 1995 Kobe Earthquake at the Kobe Local Meteorological Office and its time increment is 0.005 s and the number of time steps is 16,384. We used the modified Ramberg-Osgood model and Masing Rule for non-linear constitutive models. We assume that a gas pipeline is buried as shown in Fig. 3.7 (a). Figure 3.7 (b) shows the maximum axial strain of the pipeline.

We can confirm that the strain distributions obtained by our optimized model and initial model, which is derived from boring survey, are completely different. This analysis is used for screening of underground structures which will be damaged and its result shows that this optimization is important to assure the reliability of the result.

3.4 Summary

To increase the reliability of numerical simulations, it is essential to use more reliable models. Our proposed optimizer searches for a finite element model that can reproduce observation data by combining very fast simulated annealing and finite element analyses. As an application example, we estimated soil structure using observation data with 1,500 wave propagation analyses with a finite element model with 3,000,000 degrees of freedom. The finite element solver, which accounted for the large proportion of the whole computation time, was accelerated by utilizing the GPU computation. Compared to the previous study, the elapsed time per problem size was decreased by 1/160. Generation of a finite element model was difficult to compute on GPUs. We designed our algorithm so that the computation in model generation on CPUs was overlapped by the computation in the solver on GPUs and enhanced the effect of GPU acceleration. For future prospects, more trials will be required for larger problem size, as the convergence of simulated annealing gets worse. To reduce the computation time, we must attain more speedup ratio for the solver or design a faster algorithm of our optimizer.

Table3.1: Parameters. The units of α are meters.

	α_{11}	α_{21}	α_{12}	α_{22}	α_{13}	α_{23}
initial model	-9.190	-16.260	-6.490	-11.660	-4.980	-7.050
reference model	-28.030	-16.260	-25.550	-21.140	-12.790	-11.090

Table3.2: Material properties in target domain. V_p, V_s, and ρ are primary and secondary wave velocity, and density, respectively. h is the damping ratio used in the linear wave field calculation, $h_{\max}$ is maximum damping ratio, and γ is the reference strain used in the non-linear wave analyses.

	V_p(m/s)	V_s(m/s)	ρ (kg/m^3)	h	$h_{\max}$	γ
soft layer	700	100	1500	0.001	0.23	0.007
hard layer	2100	700	2100	0.001	0.001	-

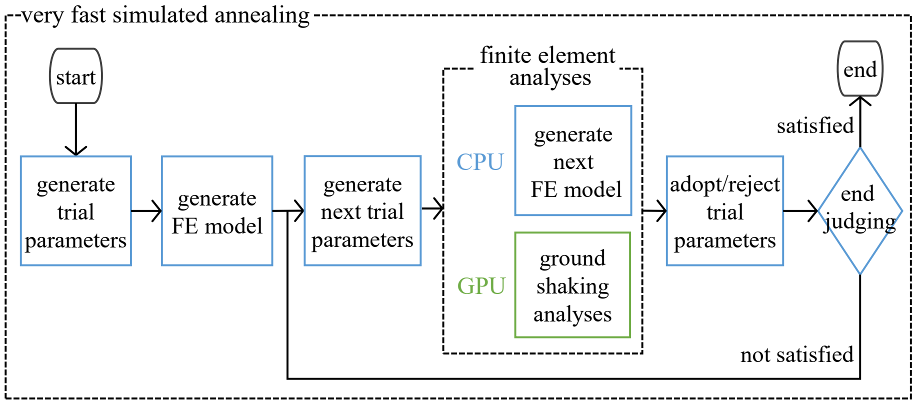

Fig.3.1: Rough scheme for our proposed optimizer for an estimation of soil structure.

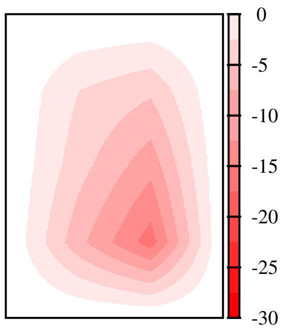

Fig.3.2-a: Initial model

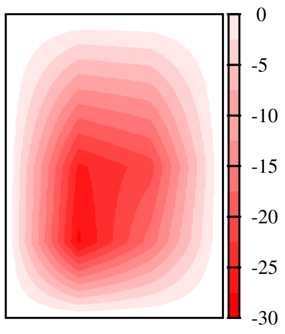

Fig.3.2-b: Reference model

Fig.3.2: Distribution of elevation (m) of the hard layer.

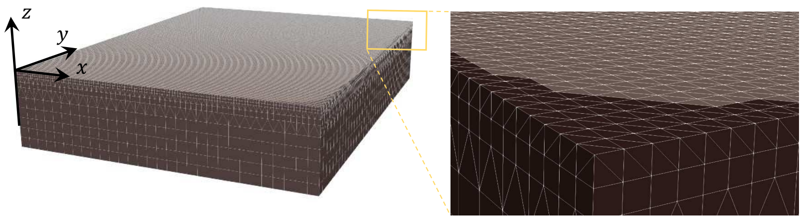

Fig.3.3: One of finite element models in the analysis.

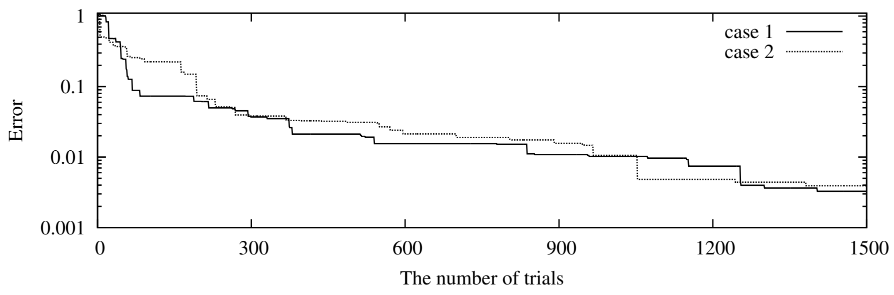

Fig.3.4: Time history of error function. Each value is normalized by the error of the initial model.

Table3.3: Parameters obtained by the optimizer for each case. The units of α are meters. RSS is the residual sum of squares based on the reference model and defined as $\sum_{ij}(\alpha_{ij}/\bar{\alpha}_{ij}-1)^2$, where $\bar{\alpha}_{ij}$ are parameters of the reference model.

	α_{11}	α_{21}	α_{12}	α_{22}	α_{13}	α_{23}	RSS
reference	-28.030	-16.260	-25.550	-21.140	-12.790	-11.090	-
case 1	-28.007	-16.290	-25.611	-21.092	-12.770	-11.100	1.6×10^{-5}
case 2	-28.119	-16.133	-25.484	-21.182	-12.849	-11.086	1.2×10^{-4}

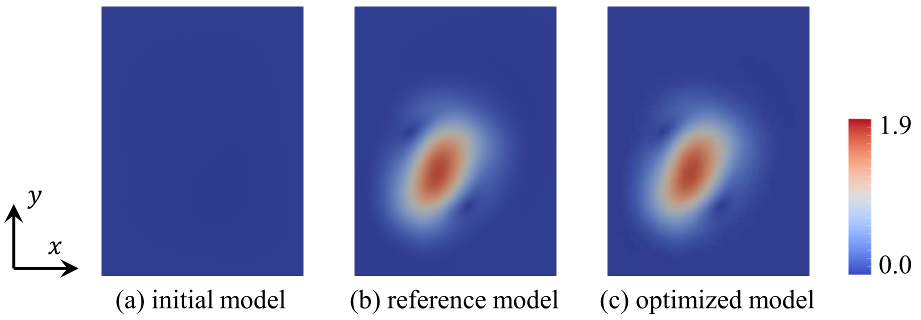

Fig.3.5: Norm distribution of displacement (m) on the ground surface at $t = 2.20s$ in the linear ground shaking analysis.

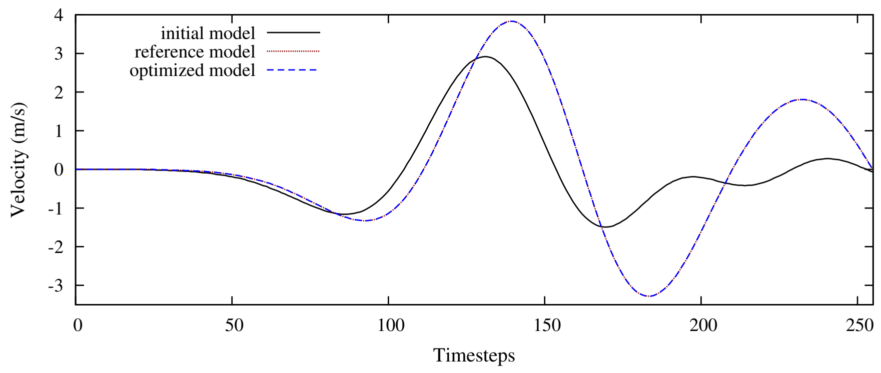

Fig.3.6: x component of the velocity at $(x, y) = (150, 200)$ on the ground surface in the linear ground shaking analysis.

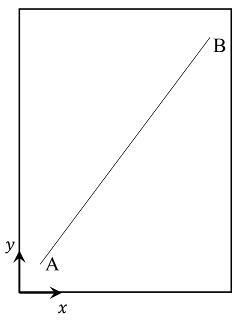

Fig.3.7-a: Location

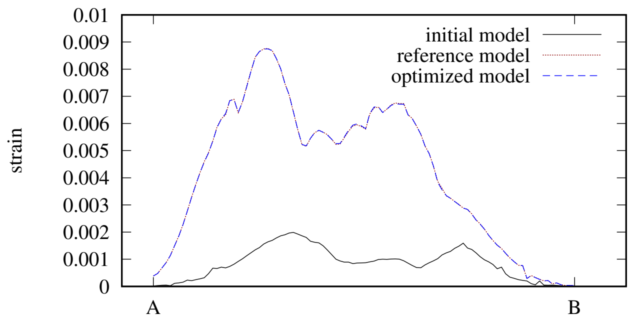

Fig.3.7-b: Maximum axial strain of the pipeline

Fig.3.7: Maximum distribution of axial strain along a buried pipeline for each model in the non-linear ground shaking analysis. The buried pipeline is located between point A $(x, y, z) = (30, 40, -1.2)$ and point B $(x, y, z) = (270, 360, -1.2)$.

Acknowledgment

Reprinted by permission from Springer Nature Customer Service Centre GmbH: Springer, Lecture Notes in Computer Science, vol 11537, Heuristic Optimization with CPU-GPU Heterogeneous Wave Computing for Estimating Three-Dimensional Inner Structure, Takuma Yamaguchi, Tsuyoshi Ichimura, Kohei Fujita, Muneo Hori, and Lalith Wijerathne, ©(2019)

4 Fast solver for crustal deformation computation

The content of this chapter is submitted to an international conference (The Platform for Advanced Scientific Computing Conference).

5 Fast solver for earthquake city simulation with FP21-FP32-FP64 transprecision computing

The content of this chapter will be published.

6 Conclusion

We developed a fast implicit low-order finite element solver with transprecision computing on GPU-based systems aiming at stochastic computing for large-scale earthquake simulations within a realistic timeframe.

Firstly, in Chapter 2, we designed our baseline codes for GPU-based systems and evaluated the performance. We introduced the solver algorithm that could reduce memory footprint and data transfer size as well as the computation amount. We also considered the modification of the algorithm so that random accesses to the memory was reduced. The reasonable speedup ratio was attained in the performance measurement on Piz Daint. In Chapter 3, we applied our baseline solver for the estimation of 3-D inner structure with heuristic optimization. Some serial computations in the optimization were not suitable for GPU computations; thus we introduced a framework where computations on CPUs were overlapped by executing other computations on GPUs simultaneously.

We achieved further speedup in Chapter 4 and Chapter 5.

From the perspective in computer science, hardware which can attain high performance for some specific operations has been popularly developed, and data transfer and memory footprint in computations will continue to be the bottleneck of the performance. Our approaches to introduce transprecision computing are expected to be helpful to exhibit higher performance on the latest systems.

The proposed methods are expected to be applied for earthquake simulations with actual observation data. For instance, we estimated the inner soil structure by computing many forward analyses in toy problem as described in Chapter 3. This estimation is expected to be applied for actual soil structures. Besides, targeting the 2011 Tohoku earthquake, [35] conducted inversion of slip distribution with uncertainties only in material properties and [36] conducted the similar inversion with uncertainties only in geometries by many crustal deformation computations. It was confirmed that these uncertainties had non-negligible effects on the results when we discussed the slip distribution and stress change distribution. These studies originally used GPU computations, but more forward analyses required in more complex problem settings can be computed by proposed algorithms and the improvement of computation environments. It is expected that our proposed methods activate stochastic computing approaches for better understanding of earthquake disaster processes.

For future tasks, settings of maximum iteration numbers and tolerances in preconditioning loops are currently set by humans in every problem, which must be automatically optimized. Also, it is possible that non-standard low-precision data types other than FP21 are more efficient for our target computations according to their required accuracy. The impact of lower-precision numbers on the convergence of the solver should be evaluated quantitatively.

Appendix A Acceleration of Matched Filtering by Tensor Core on Volta GPUs

Appendix A.1 Background

Matched Filtering [37] is a process of detecting specific pattern in a wave with noise, and it has been applied to various fields, which include signal detection of radar [38], detection of gravitational waves [39], and detection of earthquake events [40]. With the improvement of measurement technology, massive observation data have been accumulated; thus, reduction of the computation cost in Matched Filtering becomes an important issue. Methods using GPUs are proposed by [41]; however, knowledge based on latest computer architectures can achieve further speeding-up.

Recently, NVIDIA Volta GPU [42] has Tensor Core [11] for acceleration of dense matrix-matrix multiplication, which is one of the biggest features of the architecture. Two problems to accelerate computation with Tensor Core are identified. The first point is that the performance is often memory bandwidth bound as Tensor Core has extremely high peak theoretical performance. The second point is that current Tensor Core supports only lower precision data types, i.e., a 16-bit floating point number, an 8-bit integer, and an 1-bit integer. For example, the 16-bit floating point number is unable to guarantee the accuracy of more than 4 digits. In some specific fields where very high accuracy are not required, it is easy to apply these data types; however, it is challenging to apply them in general numerical simulations. Numerical error in Matched Filtering can lead to detection of unnecessary events or overlook of events; thus, the effect of numerical error should be minimized. If we design the algorithm which satisfies conditions mentioned above, we can achieve benefits of very high performance by Tensor Core operations.

This chapter proposes an algorithm to accelerate the core computation in Matched Filtering using

Tensor Core with 16-bit floating point number. We issue Tensor Core operations with lower memory access cost. Besides, we locally normalize the components of matrices to reduce the effect of using lower precision data types. We demonstrate that our algorithm attains a reasonable speeding up and improvement in accuracy when compared to cuBLAS [43], the common library for Tensor Core. Matched filtering is mathematically a normalized 1D convolution, so our approach can be beneficial for other implementations targeting convolutional neural networks [44].

Appendix A.2 Methodology

Matched Filtering detects waves similar to templates from the observation data by calculating correlation coefficient as follows:

$$CC(i,j) = \frac{\sum_{k=1}^{K} T_j(k) S(k+i)}{\sqrt{\sum_{k=1}^{K} T_j^2(k) \sum_{k=1}^{K} S^2(k+i)}}, \tag{A.1}$$

where T_j is the j-th template wave, S is the observation wave, i is the initial time step of clipped observation wave, and K is the length of template wave. We focused on the computation in the time domain while the computation in the frequency domain is also available. When computing $CC(i,j)$ for many templates and for many time steps, calculation of the dot product in the numerator of Eq. A.1 accounts for the largest computation cost. These dot products are a matrix-matrix multiplication when we calculate them with many i and j at the same time. Therefore, we can introduce Tensor Core operations for this computation.

In this chapter, we use warp matrix multiply-accumulate (wmma) API [11] for Tensor Core operations to optimize the performance and improve the accuracy using low-level descriptions. This API facilitates the computation of the two 16×16 matrices multiplication using 32 threads.

As GPU has high peak theoretical performance, we must provide data to cores rapidly to prevent memory bandwidth from binding the performance.

In matrix-matrix multiplications, it is efficient to use shared memory as a buffer and to reduce the amount of memory access to global memory. We assume that K is at most 256; therefore, we calculate a correlation coefficient by conducting 16 multiplications of 16×16 matrices. As the number of templates depends on problems, we construct our algorithm to compute correlation coefficients for 16 templates at a time, which is the smallest configuration for Tensor Core operations. We assign a

16×256 template matrix and a $256 \times N$ matrix to each thread block in computing on GPU. While we have to read all components for template matrices, we can reduce the memory access cost for reading observation matrix by applying the same method as [41]. Observation matrix has duplicated components as the matrix consists of multiple observation vectors which slide initial time steps. Given that specific characteristic in this problem, we store observation data required for each thread block in shared memory as described in Fig. Appendix A.1. Then, we reduce the number of memory access to global memory.

The number of components of the observation matrix per thread block is $N \times 256$, proportional to the floating operation counts in the multiplication. To generate this observation matrix, we require $N + 255$-time steps of observation data. Since the size of the observation matrix for each thread block N increases, higher performance is expected as the amount of memory access per the computation is reduced. However, the memory resources per thread increase, and it is harder to overlap latencies in the computation owing to a decline in the number of available threads. We decide an appropriate matrix size through a performance comparison of different matrix size. Our algorithm can lead to a reduction in access cost of global memory.

However, computation using Tensor Core tends to be a shared memory bandwidth bound for the following reasons. Tensor Core requires cooperation of 32 threads for one matrix-matrix multiplication. Here components of the matrices must be stored in registers of the corresponding thread. Then, the data mapping between threads is required before and after Tensor Core operations. This mapping is so complex that wmma API provides functions to map values of registers. These functions are using shared memory for the data distribution, that is, transferring data between shared memory and registers. If we issue these functions too frequently, the amount of memory access to shared memory increases. To exhibit high peak performance on Tensor Core, we must reduce the memory access mentioned above.

We construct observation matrix reading values in shared memory. The components of the matrix in shared memory are transferred to registers using load_matrix_sync function of wmma API; however, we can map values to registers based on the distribution of thread mapping as analyzed by [45]. Thus, we use PTX assembly to pass the variables, required for each thread, for the Tensor Core operations.

This implementation has skipped the mapping of matrices from shared memory to registers that would occurred when we used the function load_matrix_sync in wmma API, so very high peak performance on Tensor Core can be utilized since the memory access cost in shared memory as well as the memory access cost in global memory is reduced.

In the Tensor Core multiplication, 16×16 input matrices are in half precision whereas the output matrix is stored in single precision to avoid numerical error in the summation of the results. Furthermore, we introduce localized normalization to reduce numerical error when input matrices are converted into half precision. Template matrices are normalized per 16 components, and the observation matrix is normalized for the components stored in shared memory, as shown in Fig. Appendix A.2. When we call the kernel, template waves are stored in half precision and the observation wave is stored in single precision. The observation data are converted into half precision with local normalization in each thread block. In our computations, thread block includes only 32 threads; thus normalizations including searches of the maximum value can be computed only by using warp shuffle functions.

We must add results of 16×16 matrix multiplication in single precision after reflecting the values of scaling factors, involved in the template matrices. Components of 16×16 matrix are distributed among registers in 32 threads and usage of shared memory via the function load_matrix_sync makes it easy to identify rescaling factors; however, data transfer between shared memory and registers for rescaling also decreases the performance. Therefore, we have to rescale the results on registers. We specify which scaling factor must be multiplied with registers, considering distribution of matrix and using PTX assembly. It is not until we introduce low-level description considering register allocation between threads that we can carry out our fine normalization without decreasing the performance greatly.

Appendix A.3 Performance Measurement

We apply our proposed kernel to the seismic waveforms and evaluate the performance and the accuracy via comparison with a common library. In recent years, nation-wide seismic observation networks have been operated (e.g., MOWLAS [46] in Japan) with a lot of continuously recorded data; besides, the amount of data is expected to increase. For instance, MOWLAS is currently providing about million template waves and observation data of around 2,100 channels consisting of 4.32×10^6

time steps per day for approximately 10 years. Using Matched Filtering for these massive data requires much computation cost; thus, a faster algorithm is necessary to lower the cost. We use a subset of observation data provided by MOWLAS. We target calculation using 16 template waves and observation wave with 4.32×10^6 time steps; we also target matrices with the sizes of 16×256 and $256 \times (4.32 \times 10^6)$.

NVIDIA Tesla V100 GPU is used as our computing environment. Its peak FLOPS are 7.8 TFLOPS in double precision, 15.7 TFLOPS in single precision, 31.4 TFLOPS in half precision, and 125 TFLOPS in half precision with Tensor Core. The peak memory bandwidth is 900 GB/s. Our code is written with CUDA Fortran/C and complied with PGI 18.10 and nvcc 10.0.130. Elapsed time and actual memory bandwidth of kernels are measured by nvprof, whereas FLOPS are counted manually. cuBLAS is provided by cuda 10.0.130. With multiplication using cuBLAS, we generate the entire matrices explicitly. A function cublasGemmEx is provided by cuBLAS for dense matrix-matrix multiplication [47], and we prepare four types of kernels in which data types of matrices and precision in each operation are different. We must note that the elapsed time required to construct the input matrices including normalization prior to cuBLAS functions is not included. We entirely normalize input matrices when we use cuBLAS with half precision variables, by searching the maximum value in matrices as a scaling factor. Without normalization, multiplication caused an overflow. We prepare two versions of our proposed kernel: the first computes without PTX assembly, and the second skips the data transfer between shared memory and registers using PTX assembly. Targeting kernels are shown in Table. Appendix A.1. In the next section, we compare the performance and accuracy of each kernel.

Appendix A.3.1 Evaluation of performance

The middle part of Table Appendix A.1 summarizes the elapsed time of the kernel and actual bandwidth of shared memory and global memory. When comparing cuBLAS ver. 1, ver. 2, and ver. 3 kernels, the computation time decreased as the precision of input data reduced. General dense matrix-matrix multiplication is known as dense computation; however, the performance of our targeting multiplication was not arithmetic bound and limited by the memory access cost. While cuBLAS ver. 2 and ver. 3 partly used half precision variables and required data conversion cost, cuBLAS ver. 4 kernel entirely used half precision variables and improved the performance as the data conversion was unnecessary and Tensor Core operations were enabled. In our case, Tensor Core operations were disabled even when we specified options to use Tensor Cores except for cuBLAS ver. 4. The actual bandwidth of global memory in the kernel cuBLAS ver. 4 reached 768 GB/s. This was close to the result of the benchmark by [48], which was 900 GB/s $\times$ 83.3% = 750 GB/s; thus, the performance of this kernel was limited by the global memory bandwidth. On the other hand, the proposed kernels increased the bandwidth of shared memory and reduced the bandwidth of global memory. This was because we took components of the observation matrix from shared memory instead of global memory, reducing the memory access cost of global memory and increasing the cost of shared memory. Our proposed kernel used Tensor Core operations; thus, data transfer between shared memory and registers was issued to distribute components of matrices and additional memory access cost was required if we used only wmma API. Accordingly, the performance was bound by the bandwidth of shared memory, increasing the elapsed time. Contrarily, when we used assembly shown in Section 2, the performance significantly improved as the memory access to shared memory reduced.

We chose the optimal size of matrix per thread block in the kernel. Elapsed time and the register usage for different sizes of matrices are described in Fig. Appendix A.3. By increasing the size of the matrix per thread block, memory access cost for computation cost was reduced as we reused template matrices many times. However, the register usage per thread increased as the size of the matrix increased because the results of multiplication must be stored in registers. This made it difficult to overlap latencies involved in memory accesses because the number of available threads is declined. For our developed kernel, $N = 96$ was the equilibrium point of these factors.

Our kernel attained 28.4 TFLOPS that was higher than the peak FP32 FLOPS. This performance was 22.7% of 125 TFLOPS, which was theoretical peak performance when using Tensor Core on V100 GPU. This was because our kernel included operations without Tensor Core required for normalization and data conversions required for PTX assembly. Performance measurement by [11] showed that Tensor Core on V100 GPU targeting 512×512 matrices attained no more than 20 TFLOPS in any implementations. In that multiplication, the number of components of matrices was $3 \times 512 \times 512$ and the number of floating point operations was $2 \times 512 \times 512 \times 512$. On the other hand, for our targeting matrices, the number of components of matrices was $16 \times 256 + 256 \times (4.32 \times 10^6) + (4.32 \times 10^6) \times 16$, and the number of floating point operations was $2 \times 16 \times 256 \times (4.32 \times 10^6)$. As the number of matrices components was 1,500 times larger and computation cost was only 132 times larger, it was more difficult to attain higher performance with our targeting matrices. Considering these conditions, we demonstrated that our developed kernel attained reasonable performance. We computed each template that had 256-time steps in 1.243ms / 16 = 77.7us with observation data that had 4.32×10^6-times steps.

Appendix A.3.2 Evaluation of accuracy

We evaluated the numerical error in the result $CC_{i,j}$ obtained by each kernel based on the result $CC_{i,j}^{FP64}$ computed in double precision variables. We used an $Error$ defined below:

$$Error = \max_{i,j} |CC_{i,j} - CC_{i,j}^{FP64}|, \tag{A.2}$$

which is the absolute maximum value of each error. We used the same data as the previous subsection. We refer to this data as actual data. In addition, we generated data from a uniform pseudorandom number with the interval $[-50, 50]$. We refer to this data as dummy data. The numerical error of cuBLAS functions and proposed methods using actual data and dummy data, respectively are shown in the lower part of Table Appendix A.1. Error in actual data was larger than in dummy data for all kernels. Targeting waves in MOWLAS had a wide dynamic range and their values increased locally; thus, results of actual data were more affected by numerical errors. The proposed method reduced the numerical error by introducing local normalization with low-level description. This approach can work for improving the accuracy unless values change too rapidly even within tens of time steps. By contrast, numerical error in cuBLAS increased as the precision of variables decreased. To attain the same degrees of accuracy as our proposed method, internal single-precision computation such as cuBLAS ver. 2 was required. This kernel cuBLAS ver. 2 took 5.894 ms; therefore, we evaluated that our kernel attained 4.74-fold speeding up (5.894 ms/1.243 ms) compared to the common library with returned equally accurate results. When we detected patterns that had $CC_{i,j} > 0.7$, there was no erroneous detection in all kernels. This result can change depending on the problem settings; therefore more verification for accuracy is required as a future task.

Appendix A.4 Summary

We focused on Matched Filtering in the time domain. The largest proportion of the computation cost of Matched Filtering is in matrix-matrix product. Considering massive observation data, the reduction of computation cost was a critical issue. Using Tensor Core on NVIDIA Volta GPUs, we designed an algorithm to use fast matrix-matrix product. When we computed using Tensor Core, memory access to global memory or shared memory became the bottleneck for the performance. Thus, we reduced the memory access cost reusing the data and skipping unnecessary data movement. In addition, current Tensor Core only supported lower precision data types; thus, we had to reduce the effect of numerical errors in the computation. We introduced localized normalization for the target matrices to improve the accuracy of the computation. This normalization was issued by low-level description to minimize the data transfer cost. With the appropriate algorithm design, we achieved 28.4 TFLOPS in the kernel, which was reasonable performance for the sizes of our targeting matrices. We confirmed that our proposed kernel had a smaller numerical error than matrix-matrix multiplication on Tensor Core in cuBLAS, which was a common linear algebra library on NVIDIA GPUs. When we compared our kernel and a function of cuBLAS that exhibited the same degree of accuracy, our kernel was 4.47 times faster.

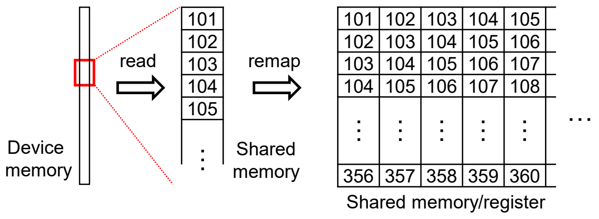

Fig.Appendix A.1: Memory transaction for an observation matrix. The matrix has many duplicated components. Shared memory is used as a buffer, and components of the actual matrix are read from shared memory.

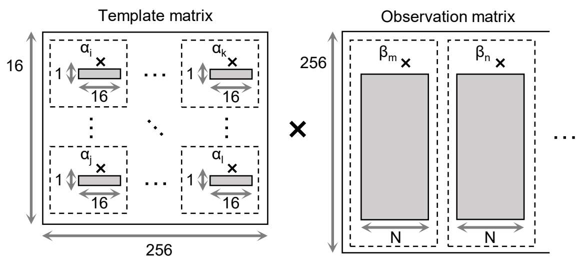

Fig.Appendix A.2: Rough scheme of normalization. α and β are single-precision scaling factors for matrices surrounded by dash lines.

TableAppendix A.1: Performance using cuBLAS and our developed kernel. Matrices are input or output in the precision noted in the row of "Input" or "Output" and multiplication is done in the precision noted in the row of "Computation".

	cuBLAS				proposed	
	ver.1	ver.2	ver.3	ver.4	w/o PTX	with PTX
Input (Template wave)	FP32	FP16	FP16	FP16	FP16	FP16
Input (Observation wave)	FP32	FP16	FP16	FP16	FP32	FP32
Output	FP32	FP32	FP16	FP16	FP32	FP32
Computation	FP32	FP32	FP32	FP16	FP16	FP16
Tensor Core operation	Disabled	Disabled	Disabled	Enabled	Enabled	Enabled
Elapsed Time	12.775 ms	5.894 ms	5.540 ms	3.032 ms	3.365 ms	1.243 ms
Device Memory Bandwidth	447 GB/s	619 GB/s	594 GB/s	768 GB/s	89 GB/s	232 GB/s
Shared Memory Bandwidth	3296 GB/s	8159 GB/s	8468 GB/s	6384 GB/s	10978 GB/s	9882 GB/s
Error in actual data	7.5×10^{-7}	3.0×10^{-4}	5.4×10^{-4}	4.3×10^{-3}	3.0×10^{-4}	1.8×10^{-4}
Error in dummy data	2.9×10^{-7}	9.2×10^{-5}	1.6×10^{-4}	1.3×10^{-3}	9.2×10^{-5}	9.2×10^{-5}

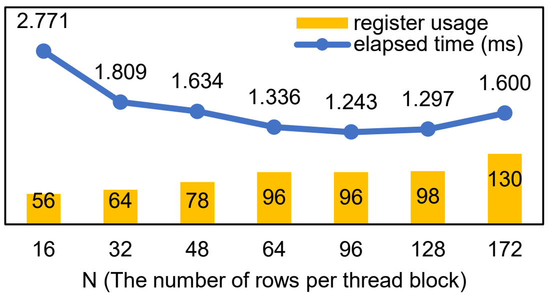

Fig.Appendix A.3: Performance of our proposed kernel when the size of matrix per thread changes.

Acknowledgment

This appendix is reproduced from “Takuma Yamaguchi, Tsuyoshi Ichimura, Kohei Fujita, Aitaro Kato, and Shigeki Nakagawa. Matched Filtering Accelerated by Tensor Cores on Volta GPUs with Improved Accuracy using Half Precision Variables. IEEE Signal Processing Letters, 26(12):1857-1861, 2019,” licensed under Creative Commons Attribution 4.0 License.

REFERENCES

[1] P Somerville, N Collins, N Abrahamson, R Graves, and C Saikia. Ground motion attenuation relations for the central and eastern united states. *US Geological Survey, Award 99HQGR0098, final report*, 2001.

[2] Tsuyoshi Ichimura, Kohei Fujita, Pher Errol Balde Quinay, Lalith Maddegedara, Muneo Hori, Seizo Tanaka, Yoshihisa Shizawa, Hiroshi Kobayashi, and Kazuo Minami. Implicit nonlinear wave simulation with 1.08 T DOF and 0.270 T unstructured finite elements to enhance comprehensive earthquake simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 4. ACM, 2015.

[3] Yuji Yagi and Yukitoshi Fukahata. Introduction of uncertainty of green's function into waveform inversion for seismic source processes. *Geophysical Journal International*, 186(2):711–720, 2011.

[4] Fukuda, Jun'ichi and Johnson, Kaj M. A fully bayesian inversion for spatial distribution of fault slip with objective smoothing. *Bulletin of the Seismological Society of America*, 98(3):1128–1146, 2008.

[5] Max Rietmann, Peter Messmer, Tarje Nissen-Meyer, Daniel Peter, Piero Basini, Dimitri Komatitsch, Olaf Schenk, Jeroen Tromp, Lapo Boschi, and Domenico Giardini. Forward and adjoint simulations of seismic wave propagation on emerging large-scale gpu architectures. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[6] Shunsuke Homma, Kohei Fujita, Tsuyoshi Ichimura, Muneo Hori, Seckin Citak, and Takane Hori. A physics-based monte carlo earthquake disaster simulation accounting for uncertainty in building structure parameters. In *ICCS*, pages 855–865, 2014.

[7] Yoshimitsu Okada. Internal deformation due to shear and tensile faults in a half-space. *Bulletin of the Seismological Society of America*, 82(2):1018–1040, 1992.

[8] Tsuyoshi Ichimura, Kohei Fujita, Seizo Tanaka, Muneo Hori, Maddegedara Lalith, Yoshihisa Shizawa, and Hiroshi Kobayashi. Physics-based urban earthquake simulation enhanced by 10.7 blndof$\times$ 30 k time-step unstructured fe non-linear seismic wave simulation. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 15–26. IEEE, 2014.

[9] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[10] OpenACC, [Online]. `http://www.openacc.org/` (Accessed: 2019-08-20).

[11] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. NVIDIA Tensor Core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[12] A Cristiano I Malossi, Michael Schaffner, Anca Molnos, Luca Gammaitoni, Giuseppe Tagliavini, Andrew Emerson, Andrés Tomás, Dimitrios S Nikolopoulos, Eric Flamand, and Norbert Wehn. The transprecision computing paradigm: Concept, design, and applications. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1105–1110. IEEE, 2018.

[13] Takane Hori, Mamoru Hyodo, Shin'ichi Miyazaki, and Yoshiyuki Kaneda. Numerical forecasting of the time interval between successive M8 earthquakes along the Nankai Trough, southwest Japan, using ocean bottom cable network data. *Marine Geophysical Research*, 35(3):285–294, 2014.

[14] Yukitoshi Fukahata and Mitsuhiro Matsu'ura. Quasi-static internal deformation due to a dislocation source in a multilayered elastic/viscoelastic half-space and an equivalence theorem. *Geophysical Journal International*, 166(1):418–434, 2006.

[15] Timothy Masterlark. Finite element model predictions of static deformation from dislocation

sources in a subduction zone: sensitivities to homogeneous, isotropic, Poisson-solid, and half-space assumptions. *Journal of Geophysical Research: Solid Earth*, 108(B11), 2003.

[16] Piz Daint [Online]. `https://www.cscs.ch/computers/piz-daint/` (Accessed: 2019-08-20).

[17] Tsuyoshi Ichimura, Ryoichiro Agata, Takane Hori, Kazuro Hirahara, Chihiro Hashimoto, Muneo Hori, and Yukitoshi Fukahata. An elastic/viscoelastic finite element analysis method for crustal deformation using a 3-D island-scale high-fidelity model. *Geophysical Journal International*, 206(1):114–129, 2016.

[18] James M Winget and Thomas JR Hughes. Solution algorithms for nonlinear transient heat conduction analysis employing element-by-element iterative strategies. *Computer Methods in Applied Mechanics and Engineering*, 52(1-3):711–815, 1985.

[19] Kohei Fujita, Keisuke Katsushima, Tsuyoshi Ichimura, Masashi Horikoshi, Kengo Nakajima, Muneo Hori, and Lalith Maddegedara. Wave Propagation Simulation of Complex Multi-Material Problems with Fast Low-Order Unstructured Finite-Element Meshing and Analysis. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2018, pages 24–35, New York, NY, USA, 2018. ACM.

[20] Kohei Fujita, Takuma Yamaguchi, Tsuyoshi Ichimura, Muneo Hori, and Lalith Maddegedara. Acceleration of element-by-element kernel in unstructured implicit low-order finite-element earthquake simulation using openacc on pascal gpus. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*, pages 1–12. IEEE, 2016.

[21] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[22] Kohei Fujita, Tsuyoshi Ichimura, Kentaro Koyama, Hikaru Inoue, Muneo Hori, and Lalith Maddegedara. Fast and Scalable Low-Order Implicit Unstructured Finite-Element Solver for Earth's Crust Deformation Problem. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, page 11. ACM, 2017.

[23] AT Chronopoulos and Charles William Gear. s-Step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153–168, 1989.

[24] S. Brizzi, Iris van Zelst, Ylona van Dinther, Francesca Funiciello, and Fabio Corbi. How long-term dynamics of sediment subduction controls short-term dynamics of seismicity. In *American Geophysical Union*, 2017.

[25] Peter Bird. An updated digital model of plate boundaries. *Geochemistry, Geophysics, Geosystems*, 4(3):n/a–n/a, 2003. 1027.

[26] Kohei Fujita, Keisuke Katsushima, Tsuyoshi Ichimura, Muneo Hori, and Lalith Maddegedara. Octree-based multiple-material parallel unstructured mesh generation method for seismic response analysis of soil-structure systems. *Procedia Computer Science*, 80:1624 – 1634, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[27] R Clement, J Schneider, H-J Brambs, A Wunderlich, Martin Geiger, and Franz Günter Sander. Quasi-automatic 3d finite element model generation for individual single-rooted teeth and periodontal ligament. *Computer methods and programs in biomedicine*, 73(2):135–144, 2004.

[28] Rolf M Koch, Markus H Gross, Friedrich R Carls, Daniel F von Büren, George Fankhauser, and Yoav IH Parish. Simulating facial surgery using finite element models. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 421–428. ACM, 1996.

[29] Jianwen Liang and Shaoping Sun. Site effects on seismic behavior of pipelines: a review. *Journal of pressure vessel technology*, 122(4):469–475, 2000.

[30] Geospatial Information Authority of Japan Tokyo ward area. 5m mesh digital elevation map. `http://www.gsi.go.jp/MAP/CD-ROM/dem5m/index.htm`.

[31] Pher Errol B Quinay, Tsuyoshi Ichimura, and Muneo Hori. Waveform inversion for modeling three-dimensional crust structure with topographic effects. *Bulletin of the Seismological Society of America*, 102(3):1018–1029, 2012.

[32] Lester Ingber. Very fast simulated re-annealing. *Mathematical and computer modelling*, 12(8):967–973, 1989.

[33] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings*

of 2nd workshop on general purpose processing on graphics processing units, pages 79–84. ACM, 2009.

[34] Tsuyoshi Ichimura, Kohei Fujita, Atsushi Yoshiyuki, Pher Errol Quinay, Muneo Hori, and Takashi Sakanoue. Performance enhancement of three-dimensional soil structure model via optimization for estimating seismic behavior of buried pipelines. *Journal of Earthquake and Tsunami*, 11(05):1750019, 2017.

[35] Takuma Yamaguchi, Tsuyoshi Ichimura, Yuji Yagi, Ryoichiro Agata, Takane Hori, and Muneo Hori. Fast crustal deformation computing method for multiple computations accelerated by a graphics processing unit cluster. *Geophysical Journal International*, 210(2):787–800, 2017.

[36] Takuma Yamaguchi, Kohei Fujita, Tsuyoshi Ichimura, Takane Hori, Muneo Hori, and Lalith Wijerathne. Fast finite element analysis method using multiple gpus for crustal deformation and its application to stochastic inversion analysis with geometry uncertainty. *Procedia Computer Science*, 108:765–775, 2017.

[37] George Turin. An introduction to matched filters. *IRE transactions on Information theory*, 6(3):311–329, 1960.

[38] Philip Mayne Woodward. *Probability and Information Theory, with Applications to Radar: International Series of Monographs on Electronics and Instrumentation*, volume 3. Elsevier, 2014.

[39] Bernard F Schutz. Gravitational wave astronomy. *Classical and Quantum Gravity*, 16(12A):A131, 1999.

[40] Steven J Gibbons and Frode Ringdal. The detection of low magnitude seismic events using array-based waveform correlation. *Geophysical Journal International*, 165(1):149–166, 2006.

[41] Eric Beaucé, William B Frank, and Alexey Romanenko. Fast matched filter (FMF): An efficient seismic matched-filter search for both CPU and GPU architectures. *Seismological Research Letters*, 89(1):165–172, 2017.

[42] NVIDIA. NVIDIA Tesla V100 GPU Architecture, [Online]. `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

[43] NVIDIA. cuBLAS library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31, 2008.

[44] Hoo-Chang Shin, Holger R Roth, Mingchen Gao, Le Lu, Ziyue Xu, Isabella Nogues, Jianhua Yao, Daniel Mollura, and Ronald M Summers. Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning. *IEEE transactions on medical imaging*, 35(5):1285–1298, 2016.

[45] Md Aamir Raihan, Negar Goli, and Tor M Aamodt. Modeling Deep Learning Accelerator Enabled GPUs. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92. IEEE, 2019.

[46] National Research Institute for Earth Science and Disaster Resilience. NIED MOWLAS. `https://doi.org/10.17598/NIED.0009` (Accessed: 03 July 2019), 2019.

[47] NVIDIA. cuBLAS, [Online]. `https://docs.nvidia.com/cuda/cublas/index.html`.

[48] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.