Acceleration of unstructured implicit low-order finite-element earthquake simulation using OpenACC on Pascal GPUs

Takuma Yamaguchi*

Department of Civil Engineering,
Earthquake Research Institute,
The University of Tokyo,
Bunkyo, Tokyo, Japan
Email: yamaguchi@eri.u-tokyo.ac.jp
*Corresponding author

Kohei Fujita

Department of Civil Engineering,
The University of Tokyo,
Earthquake Research Institute,
Bunkyo, Tokyo, Japan
and
Advanced Institute for Computational Science,
RIKEN, Kobe, Hyogo, Japan
Email: fujita@eri.u-tokyo.ac.jp

Tsuyoshi Ichimura, Muneo Hori and Lalith Maddegedara

Department of Civil Engineering,
Earthquake Research Institute,
The University of Tokyo,
Bunkyo, Tokyo, Japan
Email: ichimura@eri.u-tokyo.ac.jp
Email: hori@eri.u-tokyo.ac.jp
Email: lalith@eri.u-tokyo.ac.jp

Abstract: We accelerate CPU-based unstructured implicit low-order finite-element simulations by porting to a GPU-CPU heterogeneous compute environment using OpenACC. We modified performance-sensitive parts of the code, such as sparse matrix-vector multiplication and MPI communication, so that computations would be suitable for GPUs. Other parts of the earthquake simulation code are ported by directly inserting OpenACC directives into the CPU code. This porting approach enables high performance with relatively low development costs. When comparing eight K computer nodes and eight NVIDIA Pascal P100 GPUs, we achieve 20.8 times speedup for the 3 × 3 block Jacobi preconditioned conjugate gradient finite-element solver. We show the effectiveness of the proposed method through many-case crust-deformation simulations and a large-scale computation using finite element model with 10^9 degrees-of-freedom on a GPU cluster.

Keywords: OpenACC; GPU; finite-element analysis; MPI; element-by-element method; conjugate gradient method.

Reference to this paper should be made as follows: Yamaguchi, T., Fujita, K., Ichimura, T., Hori, M. and Maddegedara, L. (2019) 'Acceleration of unstructured implicit low-order finite-element earthquake simulation using OpenACC on Pascal GPUs', *Int. J. High Performance Computing and Networking*, Vol. 13, No. 1, pp.3–18.

Biographical notes: Takuma Yamaguchi is a Master's student in the Department of Civil Engineering at the University of Tokyo and he has received his BE from the University of Tokyo. His research is high-performance computing using GPUs, which targets at earthquake simulation.

Kohei Fujita is an Assistant Professor of Civil Engineering at the University of Tokyo. He received his Dr. Eng. from the Department of Civil Engineering, University of Tokyo in 2014. His research interest is development of high-performance computing methods for earthquake engineering problems.

Tsuyoshi Ichimura is an Associate Professor of Civil Engineering at the University of Tokyo. His work focuses on huge-scale numerical simulation for earthquakes. He is the first author of SC14 and SC15 Gordon Bell Prize Finalist Papers on large-scale implicit unstructured finite-element earthquake simulations.

Muneo Hori is a Professor of Civil Engineering at the University of Tokyo and Unit Leader of Computational Disaster Mitigation and Reduction Research Unit in Advanced Institute of Computational Science, RIKEN. His work focuses on integrated earthquake simulation using computational science.

Lalith Maddegedara is an Associate Professor of Civil Engineering at the University of Tokyo. His research interests include agent-based simulation of large urban area with detailed model of environment and sophisticated agents, which solves the problem of optimum resource allocation in recovering lifeline networks.

This paper is a revised and expanded version of a paper entitled 'Acceleration of element-by-element kernel in unstructured implicit low-order finite-element earthquake simulation using OpenACC on Pascal GPUs' presented at Third Workshop on Accelerator Programming Using Directives (WACCPD), Salt Lake City, 14 November 2016.

1 Introduction

To improve the estimation reliability of empirical methods, we have been developing an integrated earthquake simulation based on physics-based numerical analyses of each phase of an earthquake disaster. Since such earthquake simulations require unstructured, low-order, implicit finite-element analyses for appropriate modelling of nonlinear phenomena in complex-shaped domains, we have been speeding up this method to enable realistic earthquake simulations [e.g., Gordon Bell Prize Finalists in Ichimura et al. (2014, 2015)]. This method is regarded as a prospective next-generation earthquake simulation by companies and governmental sectors, and many joint research projects are ongoing. Our development has targeted the K computer (Miyazaki et al., 2012), which is a CPU-based massively parallel supercomputer; however, companies commonly have different types of computing systems, and there is demand for simulations that can run on a wide range of computing equipment. In addition, we may be able to further speed up the analysis by using the rapidly developing GPU accelerators. OpenACC, which enables offloading of loops and regions of a code to various accelerators by compiler directives, is suitable for porting pre-developed CPU applications to heterogeneous compute environments without altering the performance of the original CPU code. Thus, in this study, we port the fast solver developed for the K computer, which was published in Ichimura et al. (2015), to GPU-CPU heterogeneous environment using OpenACC to make the simulation method available to a wider range of compute environments. We expect that this will also improve the runtime speed.

Compared to conventional finite-element solvers used for large-scale analyses, the unstructured implicit finite-element solver reported in the SC15 paper is designed to reduce communication and memory access and increase the ratio of single-precision computation to double-precision computation. This algorithm is suitable for current CPU architectures, and the algorithm attained 18.6% of peak (1.97 PFLOPS) for the whole solver when running on the full K computer, which is very high for an implicit, unstructured, low-order finite-element analysis. This single-precision computation-rich algorithm is also suitable for GPU architectures; however, the performance by GPU architectures can be greatly degraded due to difference in architecture designs between CPUs and GPUs. For matrix-vector multiplication, which accounts for the largest proportion of the computation cost, the algorithm designed for SIMD computation and large caches in CPU architectures cannot exhibit high performance in GPU architectures. In addition, performance of data transfer in GPU is much lower than that of computation. Hence, algorithm modifications to hide the data transfer cost is necessary to attain high performance. Thus, in this study, we first develop algorithms suitable for GPU platforms for the performance-sensitive parts, and later port them using OpenACC. Other parts of the code that do not require changing the algorithm are ported directly by adding OpenACC directives to CPU code. This porting approach enables high performance with relatively low development costs. To demonstrate the performance of the developed solver, we measure performance on the newest NVIDIA Pascal GPUs. We also present an application example of fault slip estimation of the 2011 Tohoku Earthquake by many-case crust-deformation simulations enabled using the ported earthquake simulation on a GPU cluster.

The remainder of this paper is organised as follows. Section 2 summarises the solver developed for the K computer in Ichimura et al. (2015). Section 3 presents the detailed algorithm suitable for GPUs and the OpenACC porting details. Section 4 shows an application example using finite-element models with $10^{8–9}$ degrees-of-freedom (DOF), and Section 5 summarises related work. Section 6 summarises the paper.

2 Baseline finite-element earthquake simulation on the K computer

2.1 Target problem

Targeting estimation of the distribution of fault slip, we solve the crust-deformation problem for fault slip as

$$\frac{\partial \sigma_{ij}}{\partial x_j} = f_i,$$

with

$$\sigma_{ij} = c_{ijkl}\epsilon_{kl}, \quad \epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right).$$

Here, σ, x, f, ϵ, c, and u indicate stress, coordinates, outer force, strain, elastic tensor, and displacement, respectively. To estimate the slip state properly, we must model the stress and strain change between plates as well as the three-dimensional geometry and material properties connecting the fault and observation points. Thus, we use second-order tetrahedral elements for discretisation of the problem, which enables accurate computation of stress and strain in complex-shaped domains. The target problem thus becomes

$$Ku = f. \tag{1}$$

Here, u and f are displacement and nodal force vectors, respectively, and K is the stiffness matrix, which is sparse and symmetric positive definite. Since nearly all of the computation cost is spent on solving this linear system of equations, acceleration of the solver is important.

2.2 Solver overview

Solving equation (1) is a common target problem arising from low-order implicit finite-element methods used in solid continuum mechanics problems in science and engineering fields. In these low-order (e.g., second-order) implicit simulations, highly efficient high-order methods cannot be used. Thus, iterative solvers, such as conjugate gradient (CG) methods, using matrices stored explicitly in memory are generally used. Since these solvers involve significant random access and intensive memory bandwidth access, it is difficult to achieve adequate performance with current CPU and GPU systems. Previous work (Ichimura et al., 2015) accelerated this solver by reducing the data transfer size, communication size, and computation cost compared to standard solvers. The main idea here is the combination of an inexact preconditioned CG method, a multigrid method, and mixedprecision computation. Below, we explain the contents following Algorithm 1.

- Inexact preconditioned CG method (Golub and Ye, 1997). An inexact preconditioned CG method is a preconditioned CG method that uses another solver to solve the preconditioning matrix equation rather than multiplying a fixed preconditioning matrix. As it is common to use another iterative solver to solve the preconditioning matrix equation, we refer to the original CG loop as the 'outer loop' (Algorithm 1, lines 8–32) and the preconditioning solver as the 'inner loop' (Algorithm 1, lines 10–19). As the inner loop does not need to be solved exactly, this makes room for improvements such as multigrid and mixed-precision arithmetic.
- Multigrid method. We reduce the computation cost and communication size of the inexact preconditioned CG method by using a multigrid solver for the inner loop. One limitation of multigrid solvers is the possibility of poor scalability in massively parallel compute environments due to the communication required to map between grids, or the difficulty in obtaining load balance for all grid levels. We circumvent this problem by using a two-step geometric multigrid that does not require communication for mapping and maintains good load balance in both grids. Here, we use the targeted second-order tetrahedral mesh FEmodel for the inner fine loop (Algorithm 1, line 17), and the same mesh without intermediate nodes of each element for the inner coarse loop (i.e., a first-order tetrahedral mesh FEmodel_c; Algorithm 1, line 15). We use a robust and efficient CG solver with 3 × 3 block Jacobi preconditioning to solve the inner fine and coarse loops (Algorithm 2).
- Mixed precision arithmetic. Even for a target problem in double precision, we only need to solve the preconditioning matrix equation roughly. Thus, we compute the whole preconditioner in single precision. This leads to halving the memory access and communication size, and enables the use of high-performance single-precision compute hardware used in most of the current HPC systems.

With Algorithm 1, we can move most of the compute cost of the outer loop to the inner loops by using appropriate threshold values ϵ_c^{in} and ϵ^{in} in the inner CG solvers. Thus, even though this algorithm may seem complicated, the performance of the whole solver is dependent on the matrix-vector multiplication kernel, inner product kernel, or saxpy kernels, which are commonly used with standard CG solvers. Below, we explain the matrix-vector multiplication kernel used in the inner fine loop, which is the most time-consuming kernel that requires algorithmic considerations for efficient computation on GPUs.

2.3 Design of element-by-element kernel for K computer

2.3.1 Element-by-element computation

As the relative memory transfer capability to floating point computation capability is decreasing, using an algorithm that can reduce memory access is becoming important for short time-to-solution. At the same time, low memory footprint solvers are desirable since the memory capacity per compute capability is decreasing. Thus, we use

element-by-element (EBE) computation (Winget and Hughes, 1985), which is a matrix-free matrix-vector multiplication method, to compute $f = Ku$. Here, u and f are displacement and nodal force vectors, respectively, and K is the global stiffness matrix generated by superimposing element stiffness matrices K_i^e. K_i^e can be computed using coordinates of nodes x and element material properties. In standard matrix-vector multiplication methods, $f = Ku$ is computed by reading the precomputed global stiffness matrix K from the memory and multiplying it by u. In contrast, in the EBE computation, matrix-vector multiplications are computed by computing the local matrix-vector multiplications

$$f_i^e = K_i^e u_i^e = K_i^e Q_i^{e\mathrm{T}} u, \tag{2}$$

and summing them as

$$f = \sum_i Q_i^e f_i^e. \tag{3}$$

Here, Q_i^e are matrices for mapping local nodal values to global nodal values. Since the coordinates, displacement, and force vectors x, u, f can be kept in cache by reordering nodes and elements, we can drastically reduce the memory access compared to methods that read the global matrix from the memory. On the other hand, the EBE computation involves more computation, and consequently, transfers memory access cost to computation cost.

Algorithm 1 Details of algorithm for solving **Ku** = **f**

The matrix-vector multiplication **Ky** is computed using an element-by-element method. *diag*[ ], $(\bar{\ })$, and ϵ indicate the 3 × 3 block Jacobi of [ ], single-precision variable, and tolerance for relative error, respectively. $(\ )_c$ indicates the calculation related to $\mathrm{FEMmodel_c}$, and the other is the related calculation of the FEMmodel. $\bar{\mathbf{P}}$ is a mapping matrix, from $\mathrm{FEMmodel_c^i}$ to $\mathrm{FEMmodel^i}$, which is defined by interpolating the displacement in each element of $\mathrm{FEMmodel_c^i}$. Solved on CPU except for lines 15 and 17.

1: set **f** according to boundary condition
2: $\mathbf{x} \Leftarrow 0$
3: $\bar{\mathbf{B}} \Leftarrow diag[\mathbf{K}]$
4: $\bar{\mathbf{B}}_c \Leftarrow diag[\mathbf{K}_c]$
5: $\mathbf{r} \Leftarrow \mathbf{f}$
6: $\beta \Leftarrow 0$
7: $i \Leftarrow 1$
8: (*outer loop start*)
9: **while** $\|\mathbf{r}\|_2 / \|\mathbf{f}\|_2 \geq \epsilon$ **do**
10: (*inner loop start*)
11: $\bar{\mathbf{r}} \Leftarrow \mathbf{r}$
12: $\bar{\mathbf{z}} \Leftarrow \mathbf{B}^{-1}\mathbf{r}$
13: $\bar{\mathbf{r}}_c \Leftarrow \bar{\mathbf{P}}^T\bar{\mathbf{r}}$
14: $\bar{\mathbf{z}}_c \Leftarrow \bar{\mathbf{P}}^T\bar{\mathbf{z}}$
15: $\bar{\mathbf{z}}_c \Leftarrow \bar{\mathbf{K}}_c^{-1}\bar{\mathbf{r}}_c$ (* Inner coarse loop: solved on $\mathrm{FEMmodel_c}$ by Algorithm 2 with ϵ_c^{in} and initial solution $\bar{\mathbf{z}}_c$ [using GPU] *)
16: $\bar{\mathbf{z}} \Leftarrow \bar{\mathbf{P}}\bar{\mathbf{z}}_c$
17: $\bar{\mathbf{z}} \Leftarrow \bar{\mathbf{K}}^{-1}\bar{\mathbf{r}}$ (* Inner fine loop: solved on FEMmodel byAlgorithm 2 with ϵ_c^{in} and initial solution $\bar{\mathbf{z}}$ [using GPU] *)
18: $\mathbf{z} \Leftarrow \bar{\mathbf{z}}$
19: (*inner loop end*)
20: **if** $i > 1$ **then**
21: $\beta \Leftarrow (\mathbf{z}, \mathbf{q}) / \rho$
22: **end if**
23: $\mathbf{p} \Leftarrow \mathbf{z} + \beta\mathbf{p}$
24: $\mathbf{q} \Leftarrow \mathbf{Kp}$
25: $\rho \Leftarrow (\mathbf{z}, \mathbf{r})$
26: $\alpha \Leftarrow \rho / (\mathbf{p}, \mathbf{q})$
27: $\mathbf{q} \Leftarrow -\alpha\mathbf{q}$
28: $\mathbf{r} \Leftarrow \mathbf{r} + \mathbf{q}$

29: $\mathbf{x} \Leftarrow \mathbf{x} + \alpha\mathbf{p}$
30: $i \Leftarrow i + 1$
31: **end while**
32: (*outer loop end*)
33: $\mathbf{u} \Leftarrow \mathbf{x}$
34: output results

Algorithm 2 Preconditioned CG method [computed on GPU]

This is used to roughly solve $\mathbf{z} = \mathbf{K}^{-1}\mathbf{r}$ and $\mathbf{z}_c = \mathbf{K}_c^{-1}\mathbf{r}_c$ in Algorithm 1. The matrix-vector multiplication $\mathbf{Ky}$ is computed using an element-by-element method with voxel: $\sum_i^{N^{\mathrm{V}}} \mathbf{K}_i^{\mathrm{V}}\mathbf{y}_i^{\mathrm{V}} + \sum_i^{N^{\mathrm{T}}} \mathbf{K}_i^{\mathrm{T}}\mathbf{y}_i^{\mathrm{T}}$, where $(\,)^{\mathrm{V}}$ indicates values related to merged voxel elements, $(\,)^{\mathrm{T}}$ indicates values related to tetrahedral elements, and the subscript i indicates the i-th element. $\mathbf{B}$ and ϵ^{in} are a 3 × 3 block Jacobi matrix and the tolerance for the relative error, respectively. All of the calculations are made using single-precision variables.

1: $\mathbf{x} \Leftarrow \mathbf{z}$ (* CPU to GPU transfer *)
2: $\mathbf{e} \Leftarrow \mathbf{r} - \mathbf{Kx}$ (* update boundary by CPU-GPU transfer *)
3: $\beta \Leftarrow 0$
4: $i \Leftarrow 1$
5: $\mathbf{z\_or\_q} \Leftarrow \mathbf{B}^{-1}\mathbf{e}$
6: $\rho_a \Leftarrow (\mathbf{z\_or\_q}, \mathbf{e})$ (* CPU-GPU transfer for allreduce *)
7: $r2 \Leftarrow \|\mathbf{r}\|_2$ (* CPU-GPU transfer for allreduce *)
8: $err \Leftarrow \|\mathbf{e}\|_2 / r2$
9: **while** $err \geq \epsilon^{in}$ **do**
10: $\mathbf{p} \Leftarrow \mathbf{z\_or\_q} + \beta\mathbf{p}$
11: $\mathbf{z\_or\_q} \Leftarrow \mathbf{Kp}$ (* update boundary by CPU-GPU transfer *)
12: $\alpha \Leftarrow \rho_a / (\mathbf{p}, \mathbf{z\_or\_q})$ (* CPU-GPU transfer for allreduce *)
13: $\rho_b \Leftarrow \rho_a$
14: $\mathbf{e} \Leftarrow \mathbf{e} - \alpha\mathbf{z\_or\_q}$
15: $\mathbf{x} \Leftarrow \mathbf{x} + \mathbf{p}$
16: $\mathbf{z\_or\_q} \Leftarrow \mathbf{B}^{-1}\mathbf{e}$
17: $\rho_a \Leftarrow (\mathbf{z\_or\_q}, \mathbf{e})$ (* CPU-GPU transfer for allreduce *)
18: $err \Leftarrow \|\mathbf{e}\|_2 / r2$ (* CPU-GPU transfer for allreduce *)
19: $\beta \Leftarrow \rho_a / \rho_b$
20: $i \Leftarrow i + 1$
21: **end while**
22: $\mathbf{z} \Leftarrow \mathbf{x}$ (* GPU to CPU transfer *)

2.3.2 Parallel computation of EBE kernel

As seen above, the EBE method has low algorithmic byte/FLOP and is suitable for current computers. However, it is not straightforward for parallel computation due to the data recurrence required for adding local force vectors of elements with shared nodes in equation (3). Thus, we have developed an algorithm for parallel computation on multi-core and SIMD-based CPUs (Figure 2).

- Multi-core parallelisation. We allocate and initialise temporary left-hand side vectors f_j^{core} (Figure 2, lines 5–11), sum element-wise results for each core in parallel (Figure 2, lines 31–40), and sum core-wise results into the global vector f (Figure 2, lines 52–62).
- SIMD parallelisation. Data recurrence may occur for elements allocated to each SIMD lane of compute cores. Thus, we split the innermost loop into two loops, i.e., the computation part of $f_i^e = K_i^e u_i^e$ (Figure 2, lines 14–30) and the addition part of f_i^e to f_j^{core} (Figure 2, lines 31–40). This enables SIMD computation of the computationally rich first loop. We reduce the loop splitting overhead by blocking the loop with a small block size (NL), which maintains the temporary buffers BDBu11-34 in cache, which reduces the cost involved in loop splitting.

2.3.3 Mixed structured/unstructured computation of EBE kernel

On the K computer, the bottleneck of multi-core and SIMD-enabled EBE kernels becomes random register-L1 cache access (Figure 2, lines 20–25 and 37–39). Thus, we have developed an integrated modelling-analysis environment for problem solving to reduce random access. Here, structured voxel elements are used to model homogeneous parts of the target domain, while unstructured tetrahedral elements are used to model the remaining complex parts [Figure 3(a)]. Since the voxel elements are equivalent to six tetrahedrons in a pure unstructured mesh, we can halve the amount of random access compared to a pure unstructured mesh by computing them together. In addition, because the geometry is fixed, the FLOP per element required to compute the local matrix-vector multiplication is reduced. Thus, the same matrix-vector multiplication results can be computed with less random access and less computation cost using mixed structured/unstructured computation. As the gap between random access performance and computation performance widens on GPUs, we can expect effective speedup by using mixed structured/unstructured algorithms on GPUs.

3 Accelerating finite-element earthquake simulation on multiple GPUs

3.1 Solver overview

We port the solver, both with and without mixed structured/unstructured computation, to a CPU-GPU heterogeneous compute environment. Most of the compute time is algorithmically designed to be spent on the inner fine and inner coarse loops; therefore, we can expect drastic reduction of overall compute time by applying a GPU to these preconditioners. Performance for double-precision computation is generally lower than that for single-precision computation on GPUs, and thus, the speedup of double-precision computation is small considering the cost of using large amounts of device memory. On the K computer, only 11% of the overall elapsed time is spent for this double-precision computation part; therefore, we compute the outer CG loop on the CPU. Thus, part of Algorithm 1 (indicated as [using GPU]) and all parts of Algorithm 2 are offloaded to the GPU.

In the default OpenACC settings, CPU-GPU data transfers on slow PCI Express are unnecessarily called before and after GPU kernel calls. We first use OpenACC directives, which explicitly designate data transfer such that only a minimum amount of data is transferred between the CPU and GPU (Algorithm 2 shows the data transfer parts).

The primary GPU computations become vector inner products, vector addition, vector multiplication, and matrix-vector multiplication. The vector inner product, vector addition, and vector multiplication kernels, which are memory bandwidth-limited kernels with simple loop structures, are ported directly by adding OpenACC directives to the Fortran77 (and partly Fortran90) CPU code. To instruct parallel computation, OpenACC requires three classes that are called gang, worker, and vector, respectively. Regarding NVIDIA GPUs, one gang corresponds to one block and one vector corresponds to one thread, so we need to specify only the number of gangs and the length of vectors to conduct parallel computation. The porting cost for this part is a few hours; thus, actual applications can be ported with small cost by using OpenACC. For the performance-sensitive part such as MPI communication and complex matrix-vector multiplication kernel, we modify the algorithm as shown in the following subsections.

3.2 Overlap of EBE computation and MPI communication

Since both MPI communication and CPU-GPU transfer are slow, these data transfers become bottlenecks when porting applications to multiple GPUs. Thus, reducing the cost for data transfer becomes important to obtain high performance with GPU-CPU heterogeneous compute environments. In the CG solvers, synchronisation during matrix-vector multiplication accounts for the largest proportion of the whole communication. This proposed solver overlaps EBE computation and MPI communication by employing Micikevicius' (2009) scheme and reduces data transfer cost. Nodes which require MPI communication between other processes are on the boundary of each sub-domain as shown in Figure 4. On the other hand, most part of nodes except boundary nodes can be computed without MPI communication. Thereby, we introduce a computation sequence as shown in Figure 5. First, we operate EBE computation for elements which include boundary nodes. Next, we operate EBE computation for other elements in the mesh using asynchronisation operations, which enables other computations during EBE computation in GPUs. We transfer data from GPUs to CPUs and conduct packing/unpacking and MPI communication in CPUs. After MPI communication, we transfer data from CPUs to GPUs and synchronise these communication and EBE computation by inserting 'wait' option in OpenACC directives. In this scheme, it is important to reduce the amount of data transfer between CPUs and GPUs. We minimise the amount and times of data transfer cost by reordering node numbers so that transferred nodes are put in one place and have consecutive numbers.

3.3 Detailed algorithms of EBE kernel for GPU

Since the GPU is designed to hide memory latency by launching many threads on each of its 10^3 physical cores, magnitudes of more threads are used for computation compared to the CPU. In the case of the EBE kernel, we launch threads for each element in the mesh, and conduct local matrix-vector multiplication in parallel. Although such computation is suitable for parallel processing, allocating temporary arrays for each thread for addition of local matrix-vector multiplication results becomes infeasible when considering the number of threads and the device

memory capacity. Thus, we must change the algorithm to add the local matrix-vector multiplication results to a global result vector in EBE kernels on GPUs. Since the performance of cache and atomics to global device memory has been relatively low until the development of recent GPUs, GPU computation of EBE-based finite-element analyses has often been conducted using colouring, which is an algorithm that does not depend on cache and atomics as discussed in Kiss et al. (2012). During colouring, elements that do not share nodes are grouped, and each group of elements is computed in parallel [Figure 6(a)]. On the other hand, cache and atomic hardware has improved in recent GPUs, e.g., NVIDIA Pascal architecture. Thus, in this study, we use an algorithm capable of taking advantage of these improved hardware capabilities. Here, we launch threads for the element loop i and use atomics when adding $Q_i^e f_i^e$ to the global result vector f [Figure 6(b)]. Although random data access is inevitable in unstructured mesh computation, its cost is reduced by taking advantage of the data locality in the element node connectivity by effective use of cache. At the same time, data access for element material properties and node connectivity information is coalesced; we expect hiding memory access latency by using a sufficient number of threads. Since the generated threads are computed automatically in parallel by SIMT on GPUs, the loop splitting and loop blocking used in the CPU algorithm are not required.

3.4 Performance measurement for EBE kernel

We discuss a most appropriate algorithm for EBE computation in GPU-based solver by measuring the performance of the developed EBE kernel on a single GPU. Here, we use NVIDIA K40 and NVIDIA P100 GPUs. Table 1 summarises the compute environment specifications. The memory bandwidth of a K40 GPU is 288 GB/s, with peak single-precision compute performance of 4.29 TFLOPS. The memory bandwidth of a P100 GPU is 720 GB/s, with peak single-precision compute performance of 10.6 TFLOPS. Since these GPUs have read-only cache, L1 cache, and L2 cache, using this fast memory considering data locality is expected to lead to fast computation.

First, we measure the performance of an EBE kernel for unstructured second-order tetrahedral elements. Here, a problem with 10,427,823 DOF and 2,519,867 tetrahedral elements is used. Figure 1 shows the elapsed time for a single matrix-vector multiplication using the colouring version and atomic version, both implemented using OpenACC. As can be seen in Figure 1, the atomics version is faster than the colouring version for both K40 and P100 GPUs. This could be due to the high data locality of nodal values in the atomics version, which leads to high cache utilisation and less memory access. The total amount of memory read from the device memory is reduced from 2.48 GB to 0.31 GB by switching from the colouring version to the atomics version for the P100 GPU case. Thus, it is evident that more data are reused in the atomics version. When comparing the P100/K40 performance ratio for the atomics version, we can see a 4.2 times speedup, which is higher than the increase in theoretical peak FLOPS ratio of 2.5. This could be due to the improvement in atomics by the new native 32-bit atomic add instructions or the increase in cache capacity. Another reason may be the increase in the number of registers per CUDA core. This means that more threads can be executed simultaneously on each CUDAcore, expecting to lead to better coverage of memory access latency. Although we try changing the number of threads per block or swapping the dimensions of arrays to achieve additional performance improvement, we find that the algorithmic change between colouring and atomics is the most effective. Details of the performance change due to the change in the number of threads per block and array dimension swapping are given in the Appendix.

One concern for application developers is choosing between OpenACC and CUDA for implementing performance critical kernels on GPUs. On the K40 GPU, the performance of the atomics version implemented using CUDA Fortran is nearly the same as that of OpenACC (CUDA Fortran: 18.06 ms; OpenACC: 18.96 ms). Here, we use 32 threads per block, which perform the best for both implementations. As can be seen, the OpenACC directive-based implementation can perform as well as the more general CUDA Fortran, which has high development cost, for this complex EBE kernel with a large loop body (504 lines) and many temporary variables (216 variables). In the subsequent sections, we will measure performance based on OpenACC implementation with atomic addition.

Next, we measure the effectiveness of mixed structured/unstructured computation. The models used in the previous measurements correspond to a mixed problem with 204,185 structured voxels and 1,294,757 unstructured tetrahedral elements. Figure 1 shows the elapsed time for each method using atomics. For the P100 GPU case, the elapsed time for the tetrahedral part is 2.15 ms and the elapsed time for the voxel part is 1.22 ms (total of 3.37 ms). Since one voxel is equivalent to six tetrahedral elements, structured computations results in 1.81 times speedup per tetrahedral element. The speedup ratio for the K40 GPU case is also approximately the same. Thus, it is evident that mixed structured/unstructured computation is effective for a variety of architectures.

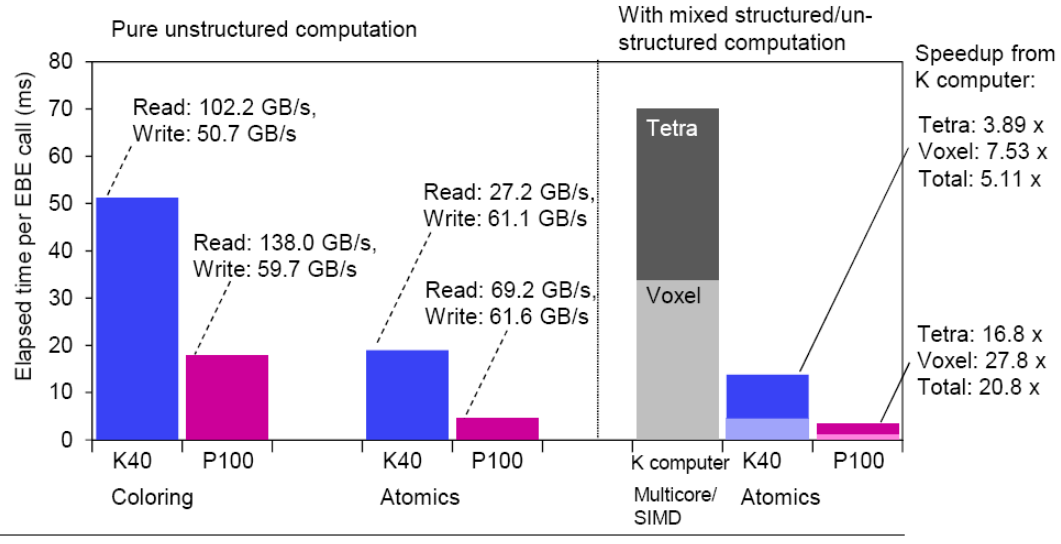
Last, we compare the performance of the mixed structured/unstructured EBE kernel with one node of the K computer (Table 1). The single-precision compute performance and memory bandwidth capability of the K40 GPU is 33.5 and 4.5 times that of the K computer, while the single-precision compute performance and memory bandwidth capability of P100 GPU is 82.8 and 11.3 times that of the K computer, respectively. On the K computer, the elapsed time for the tetrahedral part is 36.1 ms and the elapsed time for the voxel part is 33.9 ms, which totals to 70.0 ms (Figure 1). Thus, the EBE kernels are faster by 16.8 times and 27.8 times for the tetrahedral and voxel kernels on the P100 GPU, respectively. Although this is less than the hardware peak performance ratio of 82.8 times, we consider this a decent performance because a highly tuned

K computer implementation is used as a baseline for this complex kernel, which naturally suits CPUs. Indeed, the CPU implementation on the K computer obtains performance of 20.6% of hardware FLOPS peak for the total EBE kernel, which is considered high for an unstructured, low-order finite-element method. When comparing the speedup for tetrahedral and voxel kernels, it can be seen that the speedup for the voxel kernel is greater than that for the tetrahedral kernel. This is consistent with our expectation that the reduction in random access due to structure computing will be more effective on the GPU. We can see that the proper choice of algorithm is leading to large speedup.

Table 1 Comparison of hardware capabilities of K computer, GPU cluster, and NVIDIA DGX-1

	K computer	*K40 GPU cluster*	*NVIDIA DGX-1 (P100)*
# of nodes	8	4	1
CPU/node	1 × eight-core SPARC64 VIIIfx	2 × 12-core Intel Xeon E5-2695 v2	2 × 20-core Intel Xeon E5-2698 v4
Accelerator/node	-	2 × NVIDIA K40	8 x NVIDIA P100
# of MPI processes/node	1	2	8
Hardware peak (SP)	128 GFLOPS	4.29 TFLOPS (GPU only)	10.6 TFLOPS (GPU only)
FLOPS /process (DP)	128 GFLOPS	1.43 TFLOPS (GPU only)	5.3 TFLOPS (GPU only)
Bandwidth/process	64 GB/s	288 GB/s (GPU only)	720 GB/s (GPU only)
Interconnect	Tofu (4 lanes × 5GB/s in both directions)	InfiniBand FDR + PCI Express 3.0 × 16 (16GB/s in both directions)	4 × InfiniBand EDR + NVLink (20 GB/s)
Compiler	Fujitsu Fortran Driver Version 1.2.0	PGI compiler 16.7 (Sec. 3) PGI compiler 16.10 (Sec. 4)	PGI compiler 16.7 (Sec. 3) PGI compiler 16.10 (Sec. 4)
Compiler option		-ta=tesla:cc35,loadcache:L1	-ta=tesla:cc60,loadcache:L1
		-acc -memodel=medium -Minline=levels:10	-acc -memodel=medium -Minline=levels:10
		-Mvect=simd:256	-Mvect=simd:256
		-Mipa=fast -fastsse -O3	-Mipa=fast -fastsse -O3
MPI	Custom MPI	OpenMPI 1.10.2	OpenMPI 1.10.2

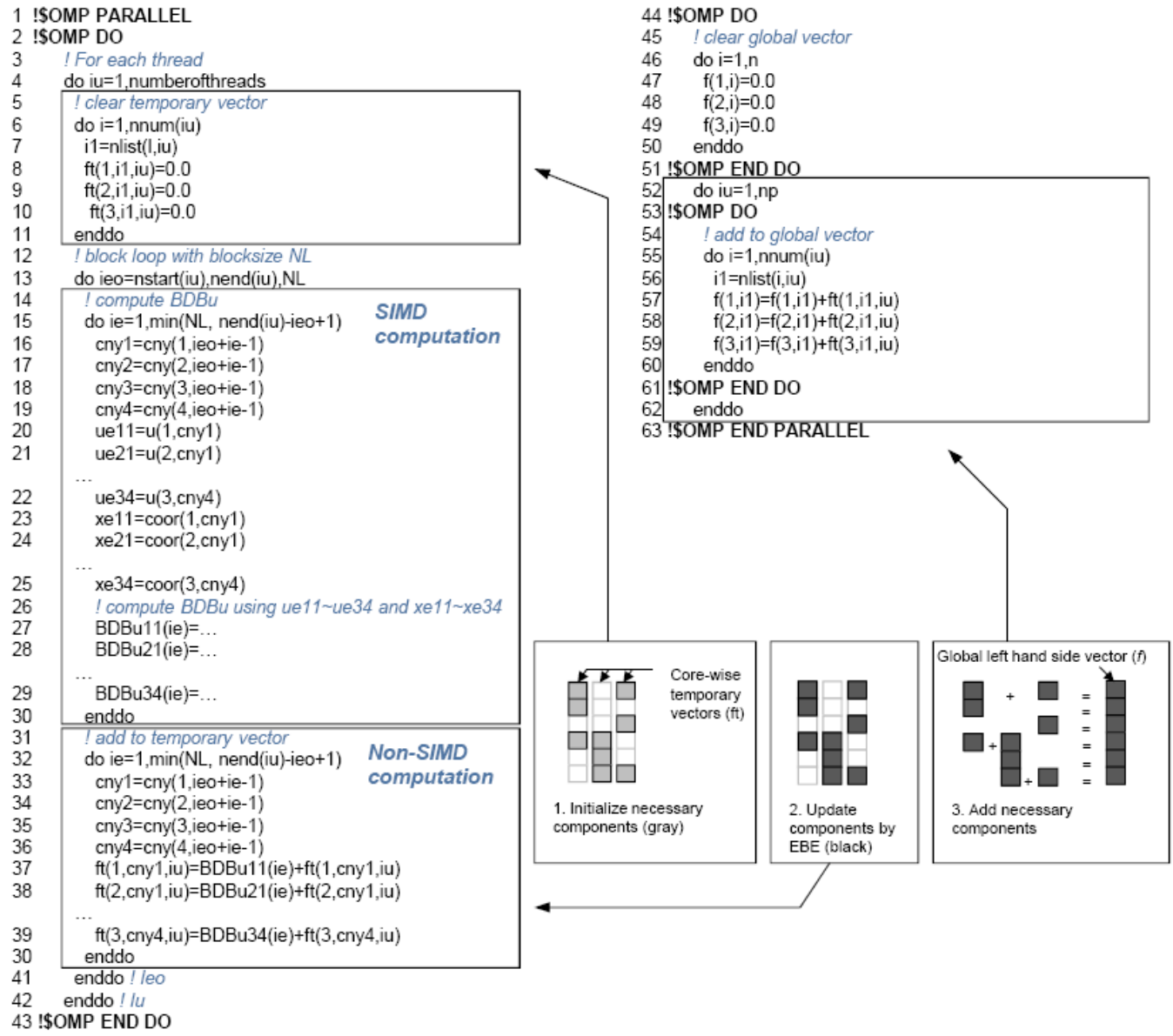
Figure 1 Performance of EBE kernel (see online version for colours)



Tetra (ms)	51.2	18.0	19.0	4.54	36.1	9.28	2.15
Voxel (ms)	-	-	-	-	33.9	4.50	1.22
Total (ms)	51.2	18.0	19.0	4.54	70.0	13.7	3.37

Notes: Insets indicate device memory access throughput of kernel. Here, one node of K computer with single SPARC64 VIIIfx CPU, one K40 GPU, and one P100 GPU is compared.

Figure 2 Parallelisation of EBE kernel on a CPU (see online version for colours)

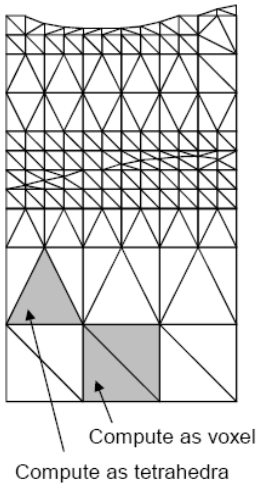


Notes: A linear tetrahedral element case is shown. Data races among multi-cores are avoided by using a temporary vector f_j^{core} (denoted ft). Only parts of the temporary vector updated by each core are initialised and added to the global vector. SIMD parallelisation is applied to element loop by loop splitting and loop blocking. Temporary buffer BDBu11-34 stays in cache as a small block size (i.e., NL) is used.

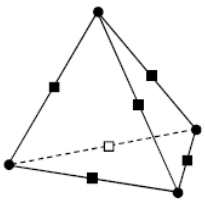
4 Application example

Estimating the distribution of earthquake fault slip is important for analysing past earthquakes and predicting future earthquakes. Although conducting 10^2 to 10^3 simulations of 10^8 DOF high-resolution finite-element crust-deformation simulations are required for precise estimation of underground fault-slip distribution from the data observed at the surface, the huge computational cost involved in such simulations has hindered realisation of such estimations. In this section, we show the effectiveness of the accelerated earthquake simulation method by conducting many-case high-resolution crustal-deformation analyses reflecting the available crust data, and use these results to estimate the slip distribution of the 2011 Tohoku Earthquake. Here, we model a 784 km (EW, x) × 976 km (NS, y) × 400 km (UD, z) region that includes the fault plane and observation points with a finite-element model at a 2,000 m resolution (Figure 7 and Figure 8). Figure 8(a) and Figure 8(b) show the generated 82,196,106 DOF and 19,921,530 second-order tetrahedral finite-element model. The material properties of this four-layered model are set according to a previous study in Sato et al. (2007) [Figure 8(d)]. This model is subdivided into eight partitions using METIS 5.1.0.

Figure 3 Use of mixed structured/unstructured mesh for EBE computation in finite-element analysis, (a) generated mesh for two-layered crust structure* (b) ten nodes are accessed during EBE computation of a tetrahedron element (c) 27 nodes are accessed during EBE computation of a voxel** (see online version for colours)

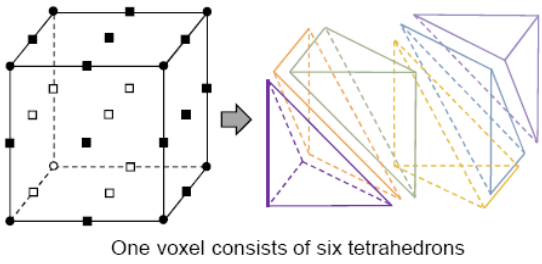


(a)



	load	store
● (linear nodes)	*u, f, x*	*f*
■ (second order nodes)	*u, f*	*f*
Total load/store nodes	24	10

(b)

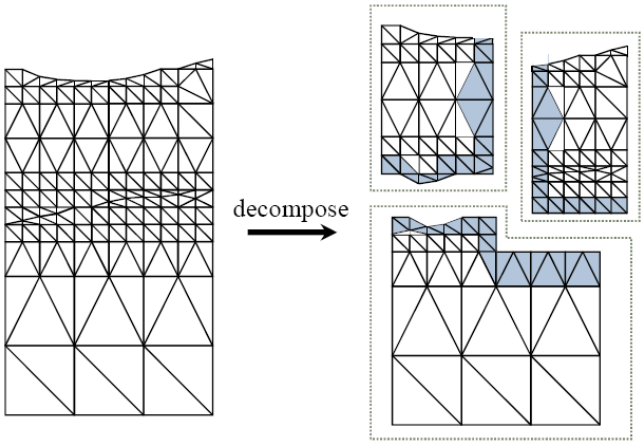


	load	store
● (linear nodes)	*u, f*	*f*
■ (second order nodes)	*u, f*	*f*
Total load/store nodes	54	27

(c)

Notes: *Voxels are used in homogeneous parts of the domain, and decomposed into six tetrahedrons with fixed geometry. Unstructured tetrahedral elements are used for the remaining parts of the domain with complex geometry. **Since each voxel is equivalent to six tetrahedrons, random access per tetrahedron can be reduced by 27 / (6 × 10) = 0.45 times.

Figure 4 Decomposition of the mesh for MPI parallel computation (see online version for colours)



Note: Coloured elements affect values on nodes requiring MPI communication.

We show the performance of the ported solver for one case of a crustal-deformation simulation on this finite-element model. We compare eight nodes of the K computer with an four-node GPU cluster with two K40 GPUs per node and the NVIDIA DGX-1 with eight P100 GPUs (Table 1). First we compare the performance of the inner fine loop (Table 2). The speedup for the non-memory bandwidth bound EBE tetra kernel is 14.5 times and that for the EBE voxel kernel is 25.4 times for DGX-1 (18.8 times speedup for total EBE kernel). These speedups are similar to the speedups obtained in the previous section; we can see that the developed EBE kernels attain high performance on actual application problems with complex mesh. Next, we discuss the effect of overlapping EBE computation and MPI communication in the inner fine loop for DGX-1. Figure 9 shows a rough process of the overlapping. Computation time of total EBE kernel is 3.13 ms per iteration. Out of the 3.13 ms, 3.01 ms is used for computation for elements that are not on the boundary of domains. Data transfer necessary for updating the boundary of domains between CPUs and GPUs requires 0.17 ms. This data transfer time increases six-fold (from 0.17 ms to 1.15 ms) if we do not reorder node numbering and transfer all of the vector. Communication between other GPUs including packing/unpacking and MPI operation in CPUs requires a total of 2.17 ms. This indicates that communication of the boundary nodes in the inner fine loop can be covered up by the EBE computation. For the memory bandwidth bound inner product kernel, near-linear speedup with increased hardware memory bandwidth is obtained by both K40 GPU cluster and DGX-1. Thus, the OpenACC implementations enable increased performance relative to the hardware improvements for this kernel with both compute environments. These increase in performance enables a 20.8 times speedup for the whole inner fine loop. As the inner fine loop is a standard CG finite-element solver with 3 × 3 block Jacobi preconditioning, by using the developed EBE kernel, we expect a similar speedup for solvers that are widely used in a variety of finite-element applications. The total speedup of the solver is 4.04 and 5.52 times for the K40 GPU cluster and DGX-1, respectively. As the peak performance of double-precision computation on P100 is improved from previous generations of NVIDIA GPUs, we can expect a large speedup of the outer loop by porting to a GPU.A conservative estimate of speedup of the outer loop running on a DGX-1 is eight times, which will lead to at least 12 times speedup of the whole solver.

Figure 5 Scheme for overlap of EBE computation and MPI communication (see online version for colours)

```
1     call EBE kernels for outer tetra elements
2     call EBE kernels for outer voxel elements
3   !ready for data necessary in MPI communication
4     call EBE kernels for inner tetra elements with !$acc async(1)
5     call EBE kernels for inner voxel elements with !$acc async(2)
6   !$acc update host(r(1:mpicomm))
7     do i=1, nproc                                   CPU computation
8       call MPI_Irecv(recvbuf…)
9     enddo
10  !pack a new array
11    do i=1, mpitot
12      sendbuf(i)=r(idx(i))
13    enddo
14    do i=1, nproc
15      call MPI_Isend(sendbuf…)
16    enddo
17    call MPI_Waitall
18  !unpack a new array
19    do i=1, mpitot
20      r(idx(i))=r(idx(i))+recvbuf(i)
21    enddo
22  !$acc update device(r(1:mpicomm)) async(3)
23  !$acc wait(1,2,3)
24  !proceed to next computation
```

Figure 6 Parallelisation of EBE kernel on GPU, (a) colouring add: elements are divided into groups with non-overlapping nodes* (b) atomic add: atomic operations are used to avoid data race** (see online version for colours)

```
1  !$ACC PARALLEL LOOP
2    ! clear global vector                          SIMT computation
3    do i=1,n
4      f(1,i)=0.0
5      f(2,i)=0.0
6      f(3,i)=0.0
7    enddo
8    do icolor=1,ncolor
9  !$ACC PARALLEL LOOP
10   do ie=ns(icolor),ne(icolor)
11     ! compute BDBu
12     cny1=cny(1,ie)
13     cny2=cny(2,ie)
14     cny3=cny(3,ie)                               SIMT computation
15     cny4=cny(4,ie)
16     ue11=u(1,cny1)
17     ue21=u(2,cny1)
       ...
18     ue34=u(3,cny4)
19     xe11=coor(1,cny1)
20     xe21=coor(2,cny1)
       ...
21     xe34=coor(3,cny4)
22     ! compute BDBu using ue11~ue34 and xe11~xe34
23     BDBu11=...
24     BDBu21=...
       ...
25     BDBu34=...
26     ! add to global vector
27     f(1,cny1)=BDBu11+f(1,cny1)
28     f(2,cny1)=BDBu21+f(2,cny1)
       ...
29     f(3,cny4)=BDBu34+f(3,cny4)
30   enddo
31   enddo
```

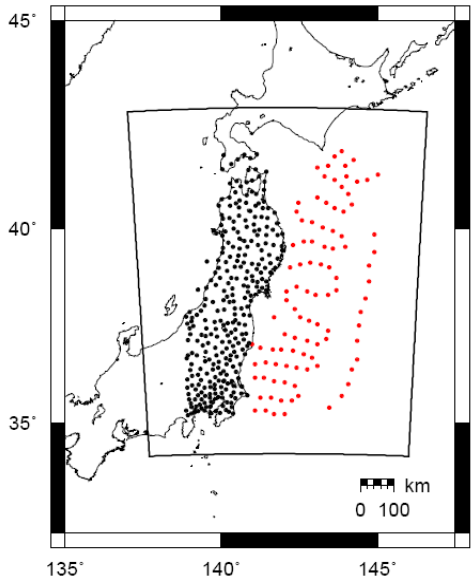
(a)

```
1  !$ACC PARALLEL LOOP
2    ! clear global vector                          SIMT computation
3    do i=1,n
4      f(1,i)=0.0
5      f(2,i)=0.0
6      f(3,i)=0.0
7    enddo
8  !$ACC PARALLEL LOOP
9    do ie=1,ne
10     ! compute BDBu
11     cny1=cny(1,ie)
12     cny2=cny(2,ie)
13     cny3=cny(3,ie)
14     cny4=cny(4,ie)                               SIMT computation
15     ue11=u(1,cny1)
16     ue21=u(2,cny1)
       ...
17     ue34=u(3,cny4)
18     xe11=coor(1,cny1)
19     xe21=coor(2,cny1)
       ...
20     xe34=coor(3,cny4)
21     ! compute BDBu using ue11~ue34 and xe11~xe34
22     BDBu11=...
23     BDBu21=...
       ...
24     BDBu34=...
25     ! add to global vector
26     !$ACC ATOMIC
27     f(1,cny1)=BDBu11+f(1,cny1)
28     !$ACC ATOMIC
29     f(2,cny1)=BDBu21+f(2,cny1)
       ...
30     !$ACC ATOMIC
31     f(3,cny4)=BDBu34+f(3,cny4)
32   enddo
```

(b)

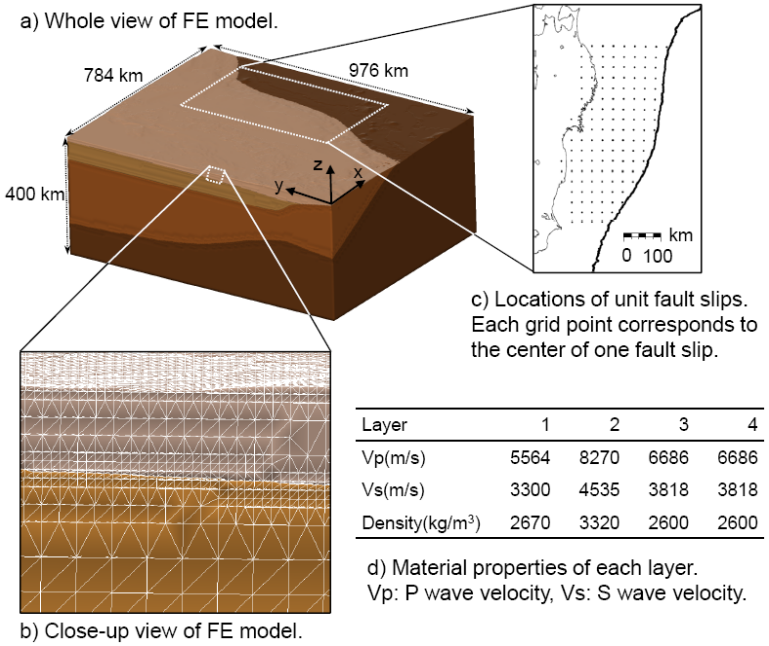
Notes: A linear tetrahedral element case is shown. *Threads update elements in each colour simultaneously, such that data race does not occur when adding local results $Q_i^e f_i^e$ to global left-hand side vector f. **Data-locality can be exploited by using on-chip cache in this case.

Figure 7 Target area of application example (black line) and location of observation points (see online version for colours)



Notes: The analysis region is almost identical as that of a previous report by Koketsu et al. (2011). x, y, and z components of GEONET (black points) and z component of S-Net (red points) are used for fault-slip estimation in this study.

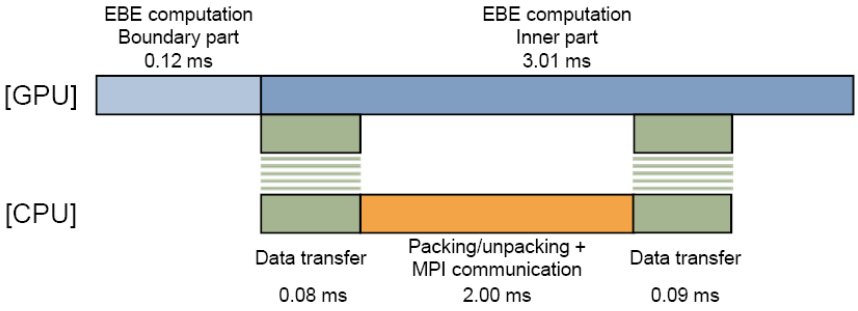
Figure 8 Finite-element (FE) model and location of unit fault slips used in the application example (see online version for colours)



Layer	1	2	3	4
Vp(m/s)	5564	8270	6686	6686
Vs(m/s)	3300	4535	3818	3818
Density(kg/m^3)	2670	3320	2600	2600

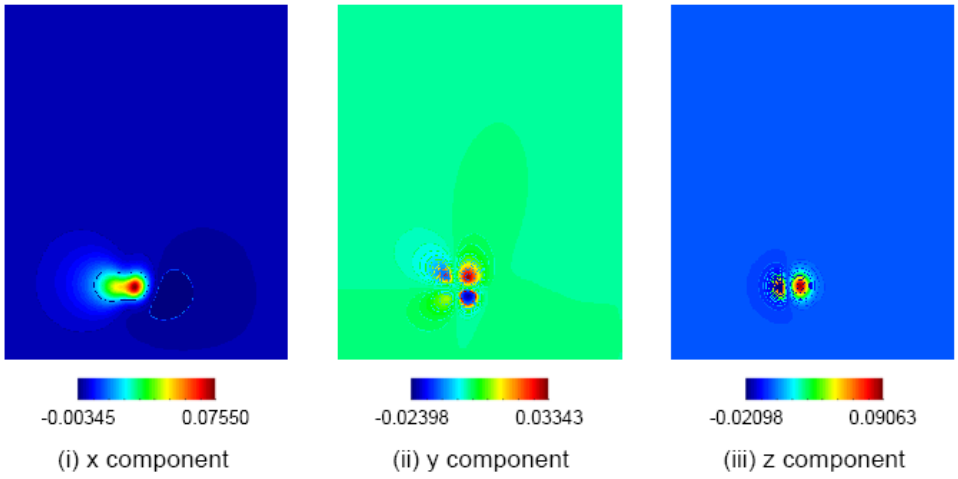
d) Material properties of each layer.
Vp: P wave velocity, Vs: S wave velocity.

Figure 9 Measurement result for overlapping EBE computation and MPI communication in inner fine loop (see online version for colours)

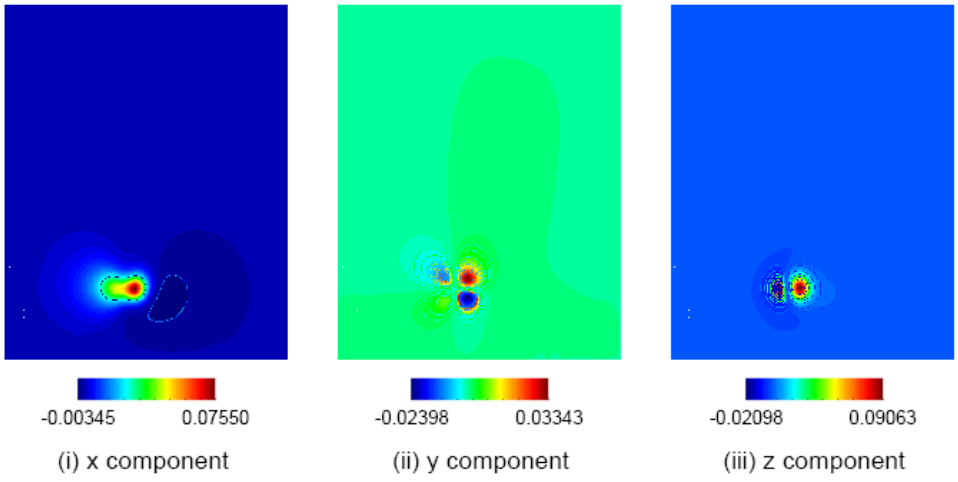


Note: Each time per iteration in DGX-1 is described.

Figure 10 Example of Green's function obtained by inputting a unit fault slip (m), (a) 2,000 m resolution (b) 500 m resolution (see online version for colours)

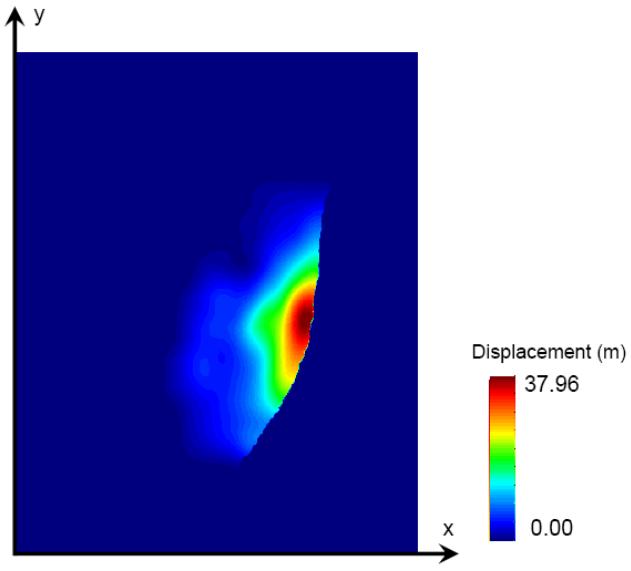


(a)



(b)

Figure 11 Estimated co-seismic fault slip distribution (m) (see online version for colours)



Using this accelerated finite-element application on NVIDIA DGX-1, we obtain Green's function for one unit fault slip as shown in Figure 10(a). We conducted 360 crust-deformation simulations to obtain Green's functions for unit slip in the x and y directions for 180 grid points, as shown in Figure 8(c). The total elapsed time for solving these Green's functions is 2,925 s. This corresponds to a speedup of 301 times compared to a previous study in Ichimura et al. (2013), which computed a single case of 157,897,032 DOF (approximately 1.9 greater than the present study) crustal deformation problem in 4,746 s using two Intel Xeon X5680 CPUs. Figure 11 shows the estimated fault-slip distribution using the computed Green's functions and observed data at the surface. Masterlark (2003) and Hughes et al. (2010) noted that simplification of the three-dimensional heterogeneity of crustal structure possibly has a significant impact on the analysis results in some cases; we can expect higher reliability in estimation results by using Green's functions that better reflects the crust structure. Additionally, we increased the resolution from 2,000 m to 500 m for the same domain and compute one of Green's functions, so as to check the convergency of the

solution. The generated model has 1,301,833,113 DOF and 316,478,123 tetrahedral elements. The computation time for this model was 118.9 s. The breakdown of the computation time is as follows: 29.1 s for the inner coarse loop, 8.9 s for the inner fine loop, and 80.9 s for the outer loop. In this computation, memory usage was 13.8 GB per GPU. Compared to the model with 2,000 m resolution, its DOF increased by 15.8 times and computation time for the solver increased by 13.0 times. Hence, the computation time per DOF was decreased by increasing the problem size per process. Obtained Green's function for the same input as Figure 10(a) is shown in Figure 10(b). These Green's functions are nearly the same, so we confirm that the solution has converged in spacial resolution. Since we can compute crust-deformation simulations in a very short time, we enable to introduce methods which evaluate uncertainties quantitatively by conducting multiple computation cases (e.g., Monte-Carlo method). We expect a more reliable slip distribution estimation considering uncertainties in crust and observation data through 10^6 computations within feasible time by using the proposed method on leading supercomputer systems.

5 Related work

Many studies into sparse matrix-vector multiplications on GPUs have been reported. For example, NVIDIA is developing and supplying the cuSPARSE library, which stores global matrices in formats suitable for GPUs and computes matrix-vector multiplications efficiently. The method used in this study targets a larger DOF problem per GPU; thus, a matrix-free method, i.e., EBE computation, is used. Targeting GPUs with small cache and slow atomics, an EBE kernel that uses colouring to sum the element-wise results has been developed in Kiss et al. (2012). The present study can be considered an update targeting newer GPUs with improved cache and atomics. Another study in Ljungkvist (2014) measured the performance of atomic/colouring addition in EBE computation on NVIDIA K20 GPUs. In this case, the elapsed time for colouring and atomics implementations were nearly the same. This is substantially different from the results of the present study, which can be due to the difference in the complexity of the mesh used. A structured mesh with eight colours has been used in Ljungkvist (2014), whereas the present study uses a three-dimensional unstructured mesh with up to 23 elements sharing one node, leading to 72 colours. It is evident that the atomics used in this study are more robust for complex three-dimensional problems.

The study by Gutzwiller et al. (2015) is an example of porting structured mesh-based solvers to a GPU-CPU heterogeneous environment. By decomposing the target domain with a set of structured meshes, computation can be performed efficiently on GPUs. However, its applicability to complex shapes is limited because it cannot be mixed with an unstructured mesh. By using a mesh generator capable of generating large mixed structured/unstructured meshes for complex shaped domains by Fujita et al. (2016), the present method is capable of fast analysis for a wider range of problems. Thus, the present method is characterised as a fast finite-element solver with low memory footprint and wide applicability to general problems with complex shapes.

6 Closing remarks

We accelerated the unstructured implicit low-order finite-element solver, by developing suitable algorithms for GPUs and porting to GPU-CPU heterogeneous environment using OpenACC. Sparse matrix-vector multiplication kernel, which accounts for the largest proportion of the computation cost, is computed with high data locality by using atomic operations. Expensive data transfer between other GPUs can be covered up by overlapping EBE computation and MPI communication. We also ported other parts of the CPU-based code by directly inserting OpenACC directives. When comparing eight K computer nodes and eight Pascal GPUs, we achieved 20.8 times speedup on a 3×3 block Jacobi preconditioned CG finite-element solver (i.e., the inner fine loop). This is expected to lead to at least 12 times speedup of whole solver. From these results, it is evident that by using OpenACC and minimal algorithmic development for performance-sensitive kernels, high performance can be achieved in a GPU-CPU heterogeneous compute environment with low development costs. Once suitable algorithms for GPUs were selected, we confirmed that complex kernels, such as the EBE kernel, can be ported to a GPU using OpenACC directives with performance that is nearly equivalent to that of CUDA Fortran implementations.

By taking the same approach to design suitable EBE kernels considering the underlying hardware, we can expect good performance with other architectures, such as the Post *K* supercomputer (Ishikawa, 2015) or Intel's Xeon Phi Knights Landing processors (Hotchips, 2015). On the other hand, there is the potential for reduction of more computational cost by changing the whole structure of solver for GPUs or other architectures. This consideration is our future task. By using the developed method on a GPU cluster, we enabled many-case crust-deformation simulations within short time. We also enabled a large-scale crustal deformation computation with a finite element model with 10^9 DOF. Such acceleration in finite-element simulation using GPUs will improve the applicability of earthquake simulations to wider compute environments and also help improve the reliability of simulation results. It is also expected to accelerate other finite-element simulations used in the industry. In the future, we hope for advances in programming environments that can exploit automatic insertion of host-device data transfer and automatic tuning of performance-critical parameters (e.g., number of threads), which will help accelerate development and porting of high-performance applications.

Acknowledgements

We thank Mr. Craig Toepfer (NVIDIA) and Mr. Yukihiko Hirano (NVIDIA) for the generous support concerning the use of NVIDIA DGX-1 (Pascal P100 GPU) environment. We also thank Mr. Shigeki Matsuo (NEC) for the generous support concerning the use of GPU environment. Part of the results were obtained using the K computer at the RIKEN Advanced Institute for Computational Science (Proposal numbers: hp160221, hp160160, and 160157). This work was supported by Post K computer project (priority issue 3: Development of Integrated Simulation Systems for Hazard and Disaster Induced by Earthquake and Tsunami), Japan Society for the Promotion of Science (KAKENHI Grant Numbers 15K18110, 26249066, 25220908) and FOCUS Establishing Supercomputing Center of Excellence.

References

Fujita, K., Katsushima, K., Ichimura, T., Hori, M. and Maddegedara, L. (2016) 'Octree-based multiple-material parallel unstructured mesh generation method for seismic response analysis of soil-structure systems', *Procedia Computer Science*, Vol. 80, pp.1624–1634.

Golub, G.H. and Ye, Q. (1997) 'Inexact conjugate gradient method with inner-outer iteration', *SIAM, Journal on Scientific Computing*, Vol. 21, No. 4, pp.1305–1320.

Gutzwiller, D., Srinivasan, R. and Demeulenaere, A. (2015) 'Acceleration of the FINE/Turbo CFD solver in a heterogeneous environment with OpenACC directives', in *Proceedings of the Second Workshop on Accelerator Programming using Directives (WACCPD '15)*, ACM, New York, NY, USA, Article 6, 8pp.

HotChips (2015) *Knights Landing: 2nd Generation Intel Xeon Phi Processor* [online] http://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.70-Processors-Epub/HC27.25.710-Knights-Landing-Sodani-Intel.pdf (accessed 26 June 2017).

Hughes, K., Masterlark, T. and Moonry, W. (2010) 'Poroelastic stress-triggering of the 2005 M8.7 Nias earthquake by the 2004 M9.2 Sumatra-Andaman earthquake', *Earth planet. Sci. Lett*., Vol. 293, Nos. 3, pp.289–299.

Ichimura, T., Agata, R., Hori, T., Hirahara, K. and Hori, M. (2013) 'Fast numerical simulation of crustal deformation using a three-dimensional high-fidelity model', *Geophysical Journal International*, Vol. 195, No. 3, pp.1730–1744.

Ichimura, T., Fujita, K., Quinay, P.E.B., Maddegedara, L., Hori, M., Tanaka, S., Shizawa, Y., Kobayashi, H. and Minami, K. (2015) 'Implicit nonlinear wave simulation with 1.08T DOF and 0.270T unstructured finite elements to enhance comprehensive earthquake simulation', *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'15)*.

Ichimura, T., Fujita, K., Tanaka, S., Hori, M., Lalith, M., Shizawa, Y. and Kobayashi, H. (2014) 'Physics-based urban earthquake simulation enhanced by 10.7 BlnDOF x 30 K time-step unstructured FE non-linear seismic wave simulation', *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'14)*, pp.15–26.

Ishikawa, Y. (2015) 'System software in post K supercomputer', *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis (InvitedTalk)*.

Kiss, I., Badics, Z., Gyimothy, S. and Pavo, J. (2012) 'High locality and increased intranode parallelism for solving finite element models on GPUs by novel element-by-element implementation', *2012 IEEE Conference on High Performance Extreme Computing (HPEC)*, Waltham, MA, pp.1–5, DOI: 10.1109/HPEC.2012.6408659.

Koketsu, K., Yokota, Y., Nishimura, N., Yagi, Y., Miyazaki, S., Satake, K., Fujii, Y., Miyake, H., Sakai, S., Yamanaka, T. and Okada, T. (2011) 'A unified source model for the 2011 Tohoku earthquake', *Earth and Planetary Science Letters*, Vol. 310, Nos. 3–4, pp.480–487.

Ljungkvist, K. (2014) 'Matrix-free finite-element operator application on graphics processing units', in *Euro-Par 2014: Parallel Processing Workshops, Lecture Notes in Computer Science*, Springer International Publishing, Vol. 8806, pp.450–461.

Masterlark, T. (2003) '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*, Vol. 108, No. B11, p.2540.

METIS 5.1.0 [online] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview (accessed 26 June 2017).

Micikevicius, P. (2009) '3D finite difference computation on GPUs using CUDA', *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ACM.

Miyazaki, H., Kusano, Y., Shinjou, N., Shoji, F., Yokokawa, M. and Watanabe, T. (2012) 'Overview of the K computer system', *FUJITSU Sci. Tech. J*., Vol. 48, No. 3, pp.302–309.

NVIDIA DGX-1 [online] http://www.nvidia.com/dgx1 (accessed 26 June 2017).

NVIDIA Kepler GPU [online] http://www.nvidia.com/object/tesla-k80.html (accessed 26 June 2017).

NVIDIA Pascal GPU [online] http://www.nvidia.com/object/tesla-p100.html (accessed 26 June 2017).

OpenACC [online] http://www.openacc.org/ (accessed 26 June 2017).

Sato, M., Minagawa, N., Hyodo, M., Baba, T., Hori, T. and Kaneda, Y. (2007) 'Effect of elastic inhomogeneity on the surface displacements in the north-eastern Japan: based on three-dimensional numerical modeling', *Earth Planets Space*, Vol. 59, No. 10, pp.1083–1093.

The NVIDIA CUDA Sparse Matrix Library (cuSPARSE) [online] https://developer.nvidia.com/cusparse (accessed 26 June 2017).

Winget, J.M. and Hughes, T.J.R. (1985) 'Solution algorithms for nonlinear transient heat conduction analysis employing element-by-element iterative strategies', *Computer Methods in Applied Mechanics and Engineering*, Vol. 52, Nos. 1–3, pp.711–815.

Appendix

Here, we present some details of EBE kernel implementations for GPUs.

Number of threads per block

The number of simultaneously running threads on GPUs and register pressure can be adjusted by changing the number of threads per block. Here, we compare the performance of an atomics based unstructured second-order tetrahedral element EBE kernel with various numbers of threads per block. Since 32 threads are grouped into warps in NVIDIA GPUs for SIMT computation, we vary the number of threads by units of 32 threads. Figure 12 shows the performance on a K40 GPU. We can see that (divisors of 8) × 32 threads obtain high performance as occupancy is obtained. Based on these observations, we use 32 threads per block for all EBE kernels in this study.

Effect of array dimension swapping

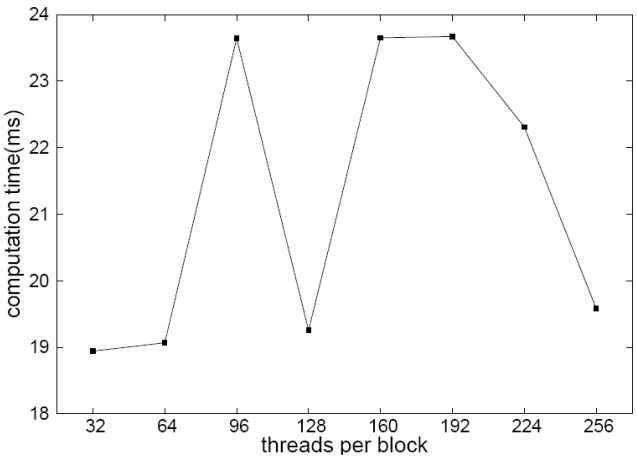
Table 3 shows the performance of array swapping for arrays coor, f, u in Figure 6(b). Here, we measure the performance for an atomics-based mixed structured/unstructured inner fine EBE kernel on a K40 GPU. It is evident that ordering coor(3, n) is slightly better than ordering coor(n, 3).

Table 3 Comparison of computation time on K40 GPU for different index ordering of two-dimensional arrays

Computation time (ms)	*Tetra part*	*Voxel part*
coor(3, n), etc.	9.28	4.50
coor(n, 3), etc.	9.49	4.70

Note: Case for mixed structured/unstructured EBE kernel computed using atomics.

Figure 12 Comparison of computation time by varying the number of threads per block



Note: Case for second-order tetrahedral EBE kernel computed using atomics.