Forward and Adjoint Simulations of Seismic Wave Propagation on Emerging Large-Scale GPU Architectures

Max Rietmann*, Peter Messmer†, Tarje Nissen-Meyer‡, Daniel Peter§, Piero Basini‡, Dimitri Komatitsch¶, Olaf Schenk*, Jeroen Tromp§, Lapo Boschi‡, and Domenico Giardini‡

*USI Lugano, Institute of Computational Science
Via Giuseppe Buffi 13
CH-6900 Lugano, Switzerland
Email: [max.rietmann, olaf.schenk]@usi.ch

†NVIDIA Corp.
Zurich, Switzerland
Email: pmessmer@nvidia.com

§Princeton University,
Dept. of Geosciences, Guyot Hall
Princeton, New Jersey 08544, USA
Email: [dpeter, jtromp]@princeton.edu

‡ETH Zurich, Institute of Geophysics
Sonneggstrasse 5, CH-8092 Zurich, Switzerland
Email: tarjen@ethz.ch,
piero.basini@gmail.com,
lapo@erdw.ethz.ch,
domenico.giardini@sed.ethz.ch

¶CNRS Marseille
Laboratory of Mechanics and Acoustics
31 chemin Joseph Aiguier
13402 Marseille cedex 20, France
Email: komatitsch@lma.cnrs-mrs.fr

***Abstract*—Computational seismology is an area of wide sociological and economic impact, ranging from earthquake risk assessment to subsurface imaging and oil and gas exploration. At the core of these simulations is the modeling of wave propagation in a complex medium. Here we report on the extension of the high-order finite-element seismic wave simulation package SPECFEM3D to support the largest scale hybrid and homogeneous supercomputers. Starting from an existing highly tuned MPI code, we migrated to a CUDA version. In order to be of immediate impact to the science mission of computational seismologists, we had to port the entire production package, rather than just individual kernels. One of the challenges in parallelizing finite element codes is the potential for race conditions during the assembly phase. We therefore investigated different methods such as mesh coloring or atomic updates on the GPU. In order to achieve strong scaling, we needed to ensure good overlap of data motion at all levels, including internode and host-accelerator transfers. Finally we carefully tuned the GPU implementation. The new MPI/CUDA solver exhibits excellent scalability and achieves speedup on a node-to-node basis over the carefully tuned equivalent multi-core MPI solver. To demonstrate the performance of both the forward and adjoint functionality, we present two case studies run on the Cray XE6 CPU and Cray XK6 GPU architectures up to 896 nodes: (1) focusing on most commonly used forward simulations, we simulate seismic wave propagation generated by earthquakes in Turkey, and (2) testing the most complex seismic inversion type of the package, we use ambient seismic noise to image 3-D crust and mantle structure beneath western Europe.**

I. Introduction

The Computational Infrastructure for Geodynamics (CIG) hosts the community code package SPECFEM3D [12], [13], [19] since 2002. This open-source package simulates seismic wave propagation on local to regional scales. A sister package called SPECFEM3D_GLOBE simulates wave propagation on larger global and regional scales. Both code packages use the continuous Galerkin spectral-element method, which is closely related to the discontinuous Galerkin technique with optimized efficiency owing to its tensorized basis functions. While SPECFEM3D focuses on mesh flexibility and local-scale simulations, SPECFEM3D_GLOBE is based on a cubed-sphere mesh for the full Earth and includes effects like gravity and Earth's self-rotation which is only important at global scales. SPECFEM3D has, with ∼70,000 source lines of code, a substantial but still relatively small code size compared to other high-performance computing (HPC) community codes (the sister package SPECFEM3D_GLOBE has ∼100,000 source lines due to additional material implementations). This manageable code size and large community usage makes SPECFEM3D an ideal candidate to extend support for graphics processing units (GPUs) embedded in HPC clusters.

The SPECFEM3D package simulates forward and adjoint coupled acoustic-(an)elastic seismic wave propagation on arbitrary conforming unstructured hexahedral meshes. Simulations are separated into a mesher and solver, to account for solving many earthquake simulations on the same mesh region. The code package benefits from advances in hexahedral meshing, load balancing, and code optimizations. Meshing may be accomplished using a mesh generation tool kit, such as CUBIT, GiD, or Gmsh, and load balancing is facilitated by graph partitioning based on the SCOTCH [4] library. Topography, bathymetry, and Moho undulations are readily included in a mesh, and physical dispersion and attenuation associated with anelasticity are accounted for using a series of standard linear solids. The solver accommodates coupling between fluid and solid regions using domain decomposition, thereby facilitat-

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

ing off-shore or ocean acoustics simulations. Finite-frequency Fréchet derivatives for earthquake and seismic interferometric data are calculated based on adjoint methods in both fluid and solid domains, thereby facilitating "adjoint tomography" with earthquakes or seismic noise.

We extend this SPECFEM3D package to support CUDA devices on large clusters. The original Fortran90 code already includes message passing interface (MPI) support for simulations running on large clusters and multi-core central processing units (CPUs). Using this code basis, we added an additional Fortran90/CUDA bridge to enable switching between existing code and newly implemented CUDA kernels. The main effort behind this code development consists of optimizing a widely used, complex code which includes many features and very simulation specific I/O routines. Our major goal was to optimize data management for simulations on heterogeneous memory systems and fine-tuning spectral-element calculations for GPU acceleration devices.

We choose to use CUDA as the GPU-programming language and environment, which includes a major development effort and implies additional costs for careful code testing and quality control. Our team, consisting of geophysicists, computer scientists and NVIDIA specialists, demonstrates that this represents a multidisciplinary effort. The final result, however, is a highly tuned GPU code that displays excellent performance results. Using automatic schemes to port such a multi-core code to GPUs will most likely fail due to its increased code complexity. Also, semiautomatic schemes using accelerator directives, such as OpenACC, will become nontrivial and will probably need a similar amount of time to carefully implement effective optimizations. We are, however, further exploring this in future comparisons.

Based on previous work on a partial port of SPECFEM3D_GLOBE to GPU clusters using CUDA and MPI [8]–[11], we improve and incorporate previous CUDA routines and optimizations into a full-featured community code, representing a larger challenge to achieve critical performance gains than porting individual routines and functionality for simplified GPU-only simulations. Our extensions enable us to run the full-featured package, including forward and adjoint simulations, on large GPU clusters. They will be included in the next release of the SPECFEM3D package and users may then install the same code package on different hardware systems and are required only to set a single flag as the simulation input parameter in order to use GPUs for the computations or run on CPUs only. Thus our implementation effectively hides the complexity of the hardware system and facilitates the handling for end users. This enables acoustic/elastic seismic wave simulations for local-scale applications to be run on GPUs by a wide user community, which is unique so far.

To evaluate the new code optimizations, we show an example of a forward problem based on earthquakes in Turkey, and a nonlinear inverse scenario based on the adjoint method using ambient noise around the Alpine region in Europe. These two cases represent the most common and most complex simulation types, respectively, of the SPECFEM3D package.

A. Contributions

The main contributions of this article with respect to the cited works and previous publications are:

- We report on our experiences in extending the wave-propagation software SPECFEM3D to efficiently execute on large-scale GPU clusters, such as the Cray XK6 at CSCS and the Cray XK6 (Titan) at ORNL. In particular, we extend previous research on that topic [8]–[11] by studying several mesh coloring options and by adding highly asynchronous memory copy to further boost the scalability of the MPI/CUDA implementation.
- We present a model for inverse wave propagation based on the adjoint method and implement a fully MPI/CUDA parallelized version of SPECFEM3D for both the forward and the adjoint process (sensitivity kernels for inverse problems). We evaluate and prove the accuracy of our performance model and show scalability.
- We simulate wave propagation for the 2011 Mw 7.1 Turkey earthquake and two more scenario events with about 19 million spectral elements based on a state-of-the-art background model for these scales. Simulations of this kind need to be run on a routine basis to gain statistical confidence in seismic hazard assessment; thus any significant speedup as demonstrated here shall enable such endeavors.
- We postulate this implementation as being optimally capable of exploiting GPU resources for weak scaling and compare this on a cost-performance basis to the equally carefully tuned CPU-only solver, spanning the relevant range of realistic seismic applications including bottleneck peculiarities, such as disk I/O. Thus, our new SPECFEM3D package is well prepared to act as a comparative, prototype application for future systems following either type of architecture.
- This project not only demonstrates the impact of high performance hybrid architectures, but also demonstrates a fruitful collaboration between domain scientists, computer scientists, and industry partners in an increasingly important model for taking advantage of emerging extreme-scale supercomputers.

The remainder of this article is organized as follows. Section II highlights the mathematical background for this work. Section III describes the spectral-element software code package SPECFEM3D from a code development standpoint. Section IV presents our GPU implementation and discusses several optimizations. In Section V we evaluate the performance of the new package for weak and strong scaling benchmarks. Section VI presents examples of realistic user simulations, i.e., large-scale wave propagation with an application on earthquake scenarios in Turkey and a tomographic problem using seismic ambient noise to image 3-D structure beneath Europe. We conclude with a discussion in Section VII.

II. Seismic Wave Propagation and Inversion

A. Forward wave propagation

Far-field seismic wave propagation is well described by linear elastodynamics, either formulated as a first-order velocity-stress system [16] or as a second-order displacement system [12]. For mass density ρ and excitation source $\mathbf{f}$ at $\mathbf{x}_s$, the second-order strong form of the elastodynamic momentum equation in displacement $\mathbf{u}$ is given by

$$\rho(\mathbf{x})\,\partial_t^2\mathbf{u}(\mathbf{x},t) - \boldsymbol{\nabla}\cdot\mathbf{T}(\mathbf{x},t) = \mathbf{f}(\mathbf{x}_s,t) \text{ in } \oplus, \tag{1}$$

subject to kinematical and dynamical boundary conditions

$$[\hat{\mathbf{r}}\cdot\mathbf{u}]^+_- = 0 \tag{2}$$

and $\hat{\mathbf{r}}\cdot\mathbf{T} = \mathbf{0}$ on the free outer surface $\partial\oplus$ with unit outward normal $\hat{\mathbf{r}}$. The stress $\mathbf{T}(\mathbf{x},t)$ is related to the displacement gradient $\boldsymbol{\nabla}\mathbf{u}$ via Hooke's constitutive law

$$\mathbf{T}(\mathbf{x},t) = \mathbf{C}(\mathbf{x}) : \boldsymbol{\nabla}\mathbf{u}(\mathbf{x},t), \tag{3}$$

where $\mathbf{C}$ is the fourth-order elasticity tensor. The weak form for test functions $\mathbf{w}$ with square-integrable derivatives reads

$$\int_\oplus \left[\rho\mathbf{w}\cdot\partial_t^2\mathbf{u} + \boldsymbol{\nabla}\mathbf{w} : \mathbf{C} : \boldsymbol{\nabla}\mathbf{u}\right] d^3\mathbf{x} = \int_\oplus \mathbf{w}\cdot\mathbf{f}\, d^3\mathbf{x}, \tag{4}$$

which naturally honors the free-surface boundary condition. Coupling terms at internal fluid/solid domain interfaces may be explicitly added with boundary conditions which imply continuity between pressure and normal displacements at such surfaces.

Upon (continuous or discontinuous) Galerkin discretization, we arrive at the ordinary differential equation in time [18]

$$\mathbf{M}\ddot{\mathbf{u}}(t) + \mathbf{K}\mathbf{u}(t) = \mathbf{F}(t), \tag{5}$$

where $\mathbf{M}$ is the mass matrix and $\mathbf{K}$ the stiffness matrix. This can be solved using a fully explicit, conditionally stable, time scheme if $\mathbf{M}$ is diagonal, as for instance constructed by a Gauss-Lobatto-Legendre basis. Forward problems are thus solved by computing the displacement $\mathbf{u}(\mathbf{x},t)$, given Earth model $(\rho,\mathbf{C})$, and excitation source $\mathbf{f}$.

B. Inverse wave propagation based on the adjoint method

Seismic tomography is an inverse problem in which one infers structural control parameters $\mathbf{m} = (\rho,\mathbf{C})$ on observations $\mathbf{u}_{\text{obs}}$ and physical laws (1)–(3) by solving

$$\mathbf{u}_{\text{obs}} = \mathbf{G}(\mathbf{m}). \tag{6}$$

This necessitates solutions to the forward problem $\mathbf{u}_0 = \mathbf{G}(\mathbf{m}_0)$ upon assumed Earth $\mathbf{m}_0$ and source model $\mathbf{f}_0$. Let

$$\boldsymbol{\chi} = \sum_r \int_0^T \mathcal{F}\left[\mathbf{u}_{\text{obs}}(\mathbf{m},\mathbf{x}_r,t), \mathbf{u}_0(\mathbf{m}_0,\mathbf{x}_r,t)\right] dt \tag{7}$$

be the misfit of generic misfit measurements $\mathcal{F}$ between $\mathbf{u}_{\text{obs}}$ and $\mathbf{u}_0$, and T the maximal time. To solve the inverse problem, one may expand the misfit in a Taylor series [23] around $\mathbf{m}_0$

$$\boldsymbol{\chi}(\mathbf{m}_0+\delta\mathbf{m}) = \boldsymbol{\chi}(\mathbf{m}_0) + \frac{\partial\boldsymbol{\chi}}{\partial\mathbf{m}}\Big|_{\mathbf{m}_0}\delta\mathbf{m} + \tfrac{1}{2!}\delta\mathbf{m}^T\frac{\partial^2\boldsymbol{\chi}}{\partial\mathbf{m}^2}\Big|_{\mathbf{m}_0}\delta\mathbf{m} + \ldots \tag{8}$$

The first-order term encapsulates the Fréchet derivative $\mathbf{g}$,

$$\mathbf{g}(\mathbf{m}_0) = \frac{\partial\boldsymbol{\chi}}{\partial\mathbf{m}}\Big|_{\mathbf{m}_0}, \tag{9}$$

with elements given by a projection onto basis functions B_i,

$$g_i = \frac{\partial\chi}{\partial m_i} = \int_\oplus K(\mathbf{x}) B_i\, d^3\mathbf{x}. \tag{10}$$

In the case of $\mathbf{m} = \mathbf{C}$, this is computed by

$$K(\mathbf{x}) = \int_0^T \mathbf{E}_0(\mathbf{x},T-t) : \mathbf{E}_0^\dagger(\mathbf{x},t)\, dt, \tag{11}$$

where $\mathbf{E} = \frac{1}{2}\left(\boldsymbol{\nabla}\mathbf{u} + (\boldsymbol{\nabla}\mathbf{u})^T\right)$ is the strain tensor, $*$ denotes temporal convolution, and $\mathbf{E}^\dagger$ is the strain tensor upon solution of the adjoint system [25].

In the case of self-adjoint elastodynamics, this can be computed using the same solver as for the forward problem by inserting the time-reversed adjoint source

$$\mathbf{f}^\dagger(\mathbf{x},t) = \sum_r \partial_{\mathbf{u}}\mathcal{F}\left(\mathbf{x}_r, T-t, \mathbf{m}_0\right)\delta\left(\mathbf{x}-\mathbf{x}_r\right) \tag{12}$$

at all receivers (seismic recording stations) $\mathbf{x}_r$ simultaneously. Thus, the overall number of forward and adjoint simulations scales with the number of earthquake sources, but is independent of the number of receivers [25].

III. Spectral-element package SPECFEM3D

SPECFEM3D performs both forward and inverse seismic wave propagation on unstructured meshes in (visco-)elastic, acoustic, anisotropic, and highly heterogeneous Earth models. It is based on a continuous Galerkin implementation using a Gauss-Legendre-Lobatto polynomial basis (optimized for Lagrange polynomials of degree 4) and Gaussian quadrature, yielding a diagonal mass matrix by construction. As mentioned above, this results in explicit time marching and high scalability, where the bulk computational cost is confined to the calculation of the stiffness term and assembly stage (including message passing).

Such explicit spectral-element methods lend themselves very well to realistic, accurate, and scalable seismic wave propagation: the combination of good computational scalability due to the explicit time integration, spectral convergence, and accurate representation of discontinuous interfaces and surface topography make the code fall into a regime of many applications ranging from earthquake rupture dynamics, to hydrocarbon imaging [27], to global wave propagation [18].

The sister code, SPECFEM3D_GLOBE, is dedicated to whole-earth wave propagation, rather than local and regional models. That version of the code won the Gordon Bell Prize in 2003 [14], and was a finalist again in 2008 [2]. Many optimizations carried out in one package have been incorporated into the other package as well, as both packages share common routines, such as indirect addressing and sorting algorithms. An explicit 3D spectral-element solver for incompressible flows including elliptic problems also won the Gordon Bell Prize in 1999 [26].

Running a simulation using the SPECFEM3D package requires three stages: (1) mesh decomposition, which splits the mesh into partitions for each MPI-rank, (2) mesh preprocessing, which prepares the mesh constants and earth model, and finally, (3) the solver, performing the seismic wave propagation simulation on this mesh. Both mesh preprocessing and solver are MPI parallel applications, written in Fortran90 with a few I/O routines written in C. The solver can be run using either single or double precision values and calculations, but single precision generally provides sufficient accuracy, improving performance and memory usage.

Meshes for simulations of interest can easily range from 10^6 to 10^9 degrees of freedom, such that memory and node allocation must be tailored to each specific application. While the global-scale code SPECFEM3D_GLOBE uses analytical, hard-coded meshes for the full Earth, SPECFEM3D can work with arbitrary, user-provided hexahedral meshes generated with a program such as the CUBIT Mesh Generation Toolkit [5], offering modeling flexibility to the scientist. Partitioning of the unstructured mesh is accomplished by the SCOTCH graph partitioning library, resulting in high-quality domain decompositions that ensure low surface to volume ratio, edge-cut minimization and optimized load balancing.

The spatial partitioning has the requirement that, at each time step, the local elements that border another partition, or *halos* (also called "outer" elements), need to be exchanged amongst nearest neighbors, introducing a loose global coupling. To implement this exchange, each time step is split into an outer and inner phase and MPI is used to send messages containing the updates of the outer elements to neighboring partitions. Thus the solver algorithm follows the following procedure:

for $t = 1$,NSTEPS **do**
  Time-step update ()
  for phase=outer,inner **do**
    Stiffness Assembly (phase)
    Absorbing Boundaries (phase)
    Source Forcing (phase)
    MPI-Communications (phase) // *sent asynchronously*
  end for
  Time-step finalize ()
  Calculate Seismograms ()
end for

where the "outer" and "inner" phases correspond to elements on the partition boundary halo and the inner, nonhalo region, respectively. For example, *Stiffness Assembly (phase=outer)*, calculates the stiffness update only for elements in the outer halo region. The MPI communications for the outer phase are nonblocking, which allows the inner-element stiffness, boundary, and source contributions to be computed while the MPI communications are being completed on a concurrent process managed by the MPI implementation. This overlap of computation and communication results in excellent scalability of the code to large processor counts.

IV. Optimizations for hybrid node architectures

In order to take advantage of large clusters with hybrid, GPU-accelerated nodes, optimizations at both the implementation and algorithmic level needed to be performed. For example, overlapping the GPU-CPU memory transfers, required for MPI communications, was critical for scalability. On the algorithmic level, an efficient method to avoid race conditions during the concurrent update of nodes shared amongst elements is required. Finally, device specific optimizations taking into account the particularities of the GPU architecture are required. In the following, we describe these individual optimizations in detail.

A. Execution model

The standard mode of execution for SPECFEM3D on a CPU-only cluster is running a single MPI rank per processor core. This model scales well even on nodes with a large number of cores. Performance experiments using OpenMP + MPI hybrid parallelism have not yet been able to surpass the excellent MPI scalability shown on the systems used in this study. Additionally, a benefit of a pure MPI implementation is that it enables future implementation of fault-tolerance mechanisms which may require a separate MPI process [1] on a reserved CPU core on each node.

A similar execution model is used for running on a cluster with GPU nodes. However, unlike the case of a pure CPU cluster, where the number of MPI ranks per node is determined by the number of available CPU cores, the number of MPI ranks per hybrid node is determined by the number of GPUs attached to the node. While the code can be run on systems with multiple GPUs per node and shows excellent scalability within a single node, all of our large-scale experiments were run on systems with only a single GPU per node.

B. Asynchronous memory transfers

As seen in the details of the CPU implementation above, SPECFEM3D is already capable of using MPI efficiently. For the GPU version, the synchronization at each time step requires a GPU→CPU memory copy before the MPI communications can begin. By using page-locked pinned memory for the MPI buffer, we can send the required updates to the CPU from the GPU asynchronously using the `cudaMemcpyAsync()` function. The computational timeline in Figure 1 outlines our strategy to hide most of the communications required by the GPU version. The "outer" elements that share a boundary or node with elements in a different partition (shaded in blue) are computed first. An asynchronous memory copy is launched once these outer elements are finished, which is directly followed by the launching of the CUDA kernel to compute the inner elements.

Both the asynchronous memory copy and CUDA kernel launches are nonblocking, which allows the main thread to actively wait for the memory copy to finish. Upon completion of the copy to the CPU, nonblocking MPI communications can be started. As soon as both the inner element computation and MPI have finished, the MPI-transferred updates are

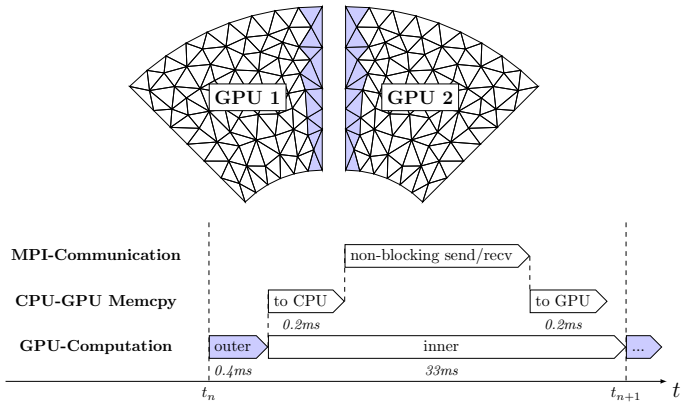

Fig. 1. A view of the computational time line, outlining our overlapping strategy. The shaded region contains elements in the "outer" domain, which share a boundary, an edge or a point with another partition and are computed first. The time line elements are not displayed to scale, but their approximate timing for the 303,116 element mesh on two GPUs is included directly below each, respectively.

asynchronously copied back to the GPU and added to their respective elements. The next time step can begin once this update finishes, repeating the same process. If the ratio of inner to outer elements is high enough, the code will have effectively hidden both the memory copy and communication required to synchronize the two partitions.

C. Mesh coloring

As seen in the computational timeline, the most time consuming part of the SPECFEM3D solver is the assembly of the stiffness matrix, often using $> 65\%$ of the run time per time step. In this calculation, the elastic forces of each element are determined and finally added to the global force field. While in the CPU version, this is accomplished by a loop nest, in the GPU version this is performed by a large number of hardware threads.

Following previous research [8]–[11], we utilize one block per element, and one thread per node. For order $N = 4$ polynomials, resulting in 125 nodes per element, 125 threads are needed. In order to take advantage of the GPU hardware, 128 threads are used instead, leading to three idle threads per element [7].

The main challenge in parallelizing the assembly phase is updating the degrees of freedom shared between multiple elements, resulting in a race condition unless properly guarded. The CPU-only MPI-parallel version avoids this by computing the intrapartition element contributions in serial, with MPI synchronization applied to perform the sum on shared nodes that span two partitions or more. In a multithreaded shared-memory model, such as OpenMP on the CPU or CUDA on the GPU, one must guarantee that the assembly operation on a shared node is done one at a time. Two possible ways of doing this are by using hardware supported synchronization primitives, such as atomic operations, or alternatively by using mesh coloring.

Atomic memory operations offer a convenient mechanism for avoiding race conditions, however, often coming at a significant cost. While the performance of atomic memory operations on GPUs have improved with each generation of CUDA capable GPUs, they are still associated with a significant cost.

Mesh coloring [9], [11] solves the concurrent assembly problem by algorithmically guaranteeing that element contributions to shared nodes are calculated and updated at different points in time. A mesh coloring algorithm takes the elements in a partition Ω, and creates K disjoint subsets Ω_k, such that the

$$\bigcup_{k}^{K} \Omega_k = \Omega$$

and that the elements in each Ω_k do not share any boundaries or points. This partitioning guarantees that elements can be computed in parallel and that their boundary nodes are updated safely without synchronization primitives, such as `atomicAdd()` in CUDA. Note that mesh coloring adds a serial loop on all the colors in the solver, with CUDA kernel calls for each color, and is thus only a good option when the number of colors obtained is small from a given finite-element mesh, which is fortunately always the case in practice.

In general, an optimal coloring is one with the fewest possible colors; however, a coloring with a balanced number of elements/color could have better performance due to better load balancing. Currently two coloring methods are available within SPECFEM3D:

- *First Fit (FF)*: A simple greedy algorithm that always chooses the first possible color [11], [15]. This produces a coloring with a near-optimal coloring in terms of numbers of colors, but is often quite unbalanced due to its greedy nature.
- *Droux*: A balanced, but nonoptimal coloring following a "Least Used" (LU) coloring principle, which is based on a modified implementation of [6].

The greedy coloring method uses fewer colors than the Droux algorithm, but the balance of elements per color is not perfect. The number of colors and their balancing, however, does not seem to affect performance as noted in the top panel of Figure 2. In addition to performance, another factor in the choice of coloring algorithm is the time required for the coloring itself. The FF coloring requires 15x less time than Droux to generate. Given the identical performance at run time for the hexahedral meshes in this study, we recommend using the simplest greedy algorithm (FF), if only to save time and complexity in preprocessing.

In Figure 2 we further compare the two coloring methods against the standard atomic update along with the noncoloring, nonatomic benchmark, which by definition produces incorrect results as mentioned above, but proves useful as a comparison because it provides an upper bound on the performance of any such operation. Both coloring options produce performance similar to this incorrect method, illustrating that the overhead for using a colored mesh is negligible and that any reasonable coloring method can and should be used.

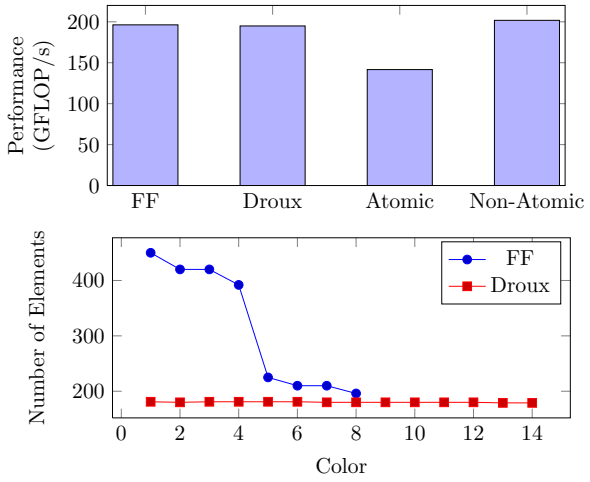

Fig. 2. (top) The total performance of two GPUs on a 303,116 resolution element mesh. The first two bars represent two separate coloring algorithms. The third bar represents a noncolored version, which uses CUDA's `atomicAdd()` operation to guarantee correctness. All schemes are compared to the final bar, which provides an upper performance bound as it uses neither coloring nor atomic updates, and thus cannot ensure correct results, but runs at maximum speed. (bottom) The distribution of colors from a small test mesh to demonstrate the balance of colors between FF and Droux. In this example, the FF algorithm uses the optimal number of colors for a regular hexahedral mesh (8), but with an uneven distribution. Droux requires more colors, but distributes them evenly.

Cecka et al. [3] determined that doubling the number of colors yields less than a 10% performance loss, and our own experiments depicted in Figure 2 show that neither the number of colors, nor the balance of elements per color have a strong effect on performance. In fact, the incorrect, noncoloring, but also nonatomic, code performs only slightly better than the coloring versions, indicating both that assembly via mesh coloring can achieve near peak performance, and that any reasonable coloring is a good coloring.

D. Spectral-element calculations: CUDA-specific optimizations

The stiffness assembly portion of the code received the most optimization attention as it represents the largest fraction of run time, and also is the most complex operation performed in each time step. The next important kernels from a run time perspective simply represent operations required by the time-stepping algorithm and are nothing more than vector arithmetic, which is memory bound and does not respond well to optimization work.

For the stiffness assembly kernel, each spectral element is mapped to a CUDA block, and each node within each element is mapped to a CUDA thread. The stiffness assembly update contains several small matrix-matrix operations. Each resulting matrix entry is calculated by a single thread and stored in shared memory, which other threads can use later after a thread synchronization is performed. Calculations follow the CPU-version and run using single-precision values, but can be recompiled to use double precision values. The arithmetic intensity of this kernel is only $\sim$2.0, making it generally memory bound like finite difference and finite volume methods.

Taking full advantage of the GPU's memory bandwidth requires the accessed memory to be word aligned. The mesh constants that are unique to each node are 128-padded to make use of coalesced memory transfers. Beyond this, it was discovered that using global memory for certain element constants yields a 10% performance gain compared with using the cached constant memory. Constant memory and its cache produce better performance when many threads access a single constant memory location; however we have many threads accessing multiple locations in constant memory, which instead produced slower performance.

In order to gain a further 10%, the displacement and acceleration vectors are bound to texture memory [9]. The texture cache does not maintain coherency within a kernel launch, and with mesh coloring, this guarantees that each acceleration value updated during the stiffness assembly is only read and written by a single thread. The texture cache is flushed between kernel launches, ensuring the correctness of the next shared-node acceleration update.

V. Performance Evaluation

We calculate the performance by measuring the time loop for 1,000 time steps. The number of floating point operations per element for a single time step was counted manually for the European mesh, and used to estimate floating point performance for all meshes used in this study as

$$\text{Performance (GFLOP/s)} = \left(\frac{41,833 \text{ FLOPs/element/time step}}{\text{time-stepping run time}}\right) \times \left(\frac{\text{elements} \times K \text{ steps}}{1}\right) \times \left(\frac{1 \text{ GFLOP}}{1 \times 10^9 \text{ FLOPs}}\right).$$

Note that we ignore the time required to set up and initialize memory on the CPU and/or GPU, which for shorter time iterations would lead to a significant portion of the time-to-solution performance, especially for GPUs. However, for almost all user applications, time iterations of several tens of thousands of time steps are required, making such an effect of memory initialization negligible in practice.

For our benchmarks, we will plot the parallel efficiency of the simulation which is the speedup $S = T_s/T_r$, where T_r is the measured run time and T_s the serial run time (or 2-GPU run time in our case), divided by the number of parallel processes N (or in our case the number of nodes). In an attempt to model how well our code scales, we further compare the simulation run time T_r against a simple model based on Amdahl's law:

$$T_\alpha = \alpha \frac{T_s}{N} + (1-\alpha)T_s, \tag{13}$$

where T_α is the effective total run time and parameter α indicates the fraction of code which was parallelized effectively.

To run our simulations, we use three different Cray systems:

- a Cray XK6 system (Tödi) located at the Swiss National Supercomputing Center (CSCS) that features 176 nodes,

each equipped with a 16-core AMD Opteron 6272 CPU, 32 GB of DDR3 memory, and one NVIDIA Tesla X2090 GPU with 5.25 GB of available GDDR5 memory.
The GPU device delivers 1331 GFlops of peak single-precision performance from its 16 streaming multiprocessors (SM), which have a total of 512 CUDA cores and 178 GB/sec of memory bandwidth.

- a Cray XE6 system (Monte Rosa) also at CSCS to run our CPU scaling experiments on, with 2×AMD Opteron 6272 2.1 GHz 16-core CPUs, 32 GB per compute node, and high performance networking with Gemini 3D torus interconnect. Each Opteron 6272 core shares its floating-point hardware with a second core, limiting the linear scalability of SPECFEM3D to only 8 cores per socket. Monta Rosa features a total of 1,496 nodes for a total of 47,872 cores, with a theoretical peak performance of 402 TFlops.
- a Cray XK6 system (Titan), which is still in development at the Oak Ridge Leadership Computing facility, where we conducted our largest GPU simulations on 896 nodes. It comprises the same XK6 compute nodes as used at CSCS.

Benchmarking GPU versus CPU simulations

We chose to compare the performance for a single GPU node (Cray XK6) against a single CPU-only node (2x16-core Cray XE6). We believe this node-to-node comparison is relatively fair, from both a cost and power perspective. From a scientific and HPC perspective, cost should prove to be the final deciding factor as hybrid architectures move from development to production at many sites worldwide. Ideally, a project would outline their total computational requirements for a specific scientific milestone and work with a supercomputing center to detail the total cost for the project in terms of hardware and power usage. This discussion, however, should also consider the higher programmer and maintenance effort required by CUDA programming, which may be a significant factor in cost decisions. The recent development of OpenACC accelerator directives may help lower these development costs, and is currently being explored as an alternative and addition to CUDA for SPECFEM3D. Additionally, The long-term power and cooling for a system should also be considered, i.e. costs associated with actually running the system for a few years once it is installed.

A. Weak scaling

In weak scaling benchmarks, we filled ∼75% of each GPU memory, keeping the load per MPI process constant while varying the number of total processes. Each XK6 GPU node has 413,952 elements (3.9 GB of GPU memory each) where, on the XE6, each 32-core CPU node has 422,400 elements. Figure 3 depicts both CPU and GPU scaling performance for an increasing problem size. We also plot the parallel efficiency, i.e., the measured speedup T_s/T_r divided by the number of nodes N. Parallel efficiency is normalized to 2 XE6 and XK6 nodes, respectively. Comparing weak scaling performances

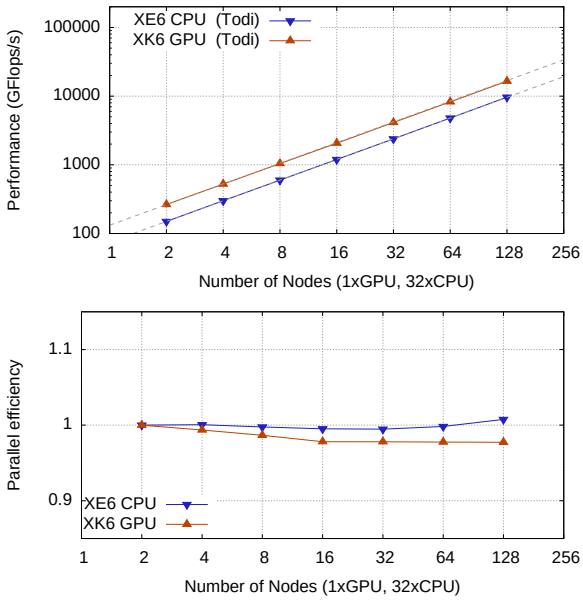

Fig. 3. Weak scaling results on XK6 GPU and XE6 CPU nodes: (top) Performance results for 400K element meshes, showing a speedup factor of ∼1.7x to 1.75x between CPU and GPU. (bottom) Parallel efficiency for weak scaling results displayed on top.

between GPU and CPU nodes, we see a speedup factor of ∼1.7x to 1.75x for the chosen mesh sizes.

The parallel efficiency for both GPU and CPU simulations is excellent, scaling within 98% of ideal runtimes across all benchmark sizes. This indicates that asynchronous message passing is nearly perfect in hiding the network latency on these systems. Note that this becomes more of a challenge for smaller mesh sizes, where the percentage of outer elements increases compared to inner elements. As both the GPU and CPU simulations scale very well with an increasing problem size, we can expect to run geographically large simulations with high resolution on either GPUs or CPUs very efficiently.

B. Strong scaling

For strong scaling benchmarks, we keep the simulation mesh size fixed while partitioning it onto a varying number of MPI processes. Figure 4 depicts scaling performance experiments on a European mesh with 303,116 elements and a 110 million synthetic mesh running a purely forward simulation. The experiments were conducted on a Cray XK6 with up to 128 nodes for the smaller mesh and up to 896 nodes for the larger mesh. We also ran the same experiments on a Cray XE6 with 32-cores per node. Parallel efficiency is normalized to 2 XE6 and XK6 nodes, respectively. The GPU version crosses the 80% parallel efficiency mark at 32 nodes (119 MB/GPU) for the 300K mesh size and at 64 nodes for a 2.2M mesh size. The largest 896-GPU performance run was able to achieve 135 TFLOP/s of floating point performance.

Analyzing strong scaling in more detail, we find that two GPUs simulating a mesh of 300,000 elements spend only 1 to 4% of their run time doing all communications —

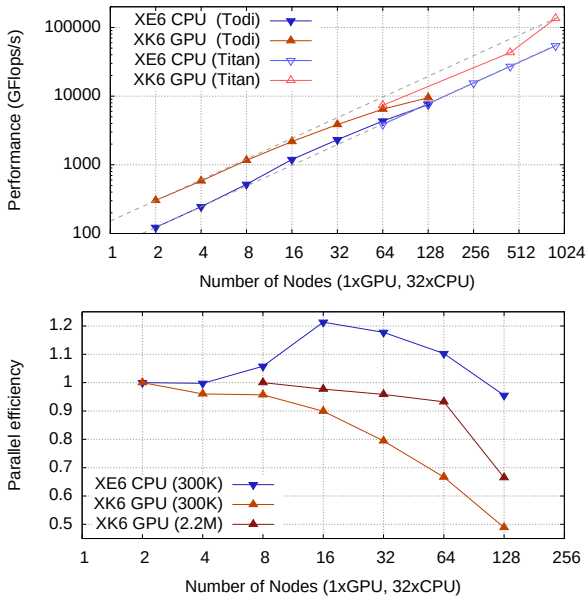

Fig. 4. Strong scaling results on XK6 GPU and XE6 CPU nodes: (top) Performance results for 300K and 110M element meshes, showing a speedup factor of ∼2.5x to 1.7x between CPU and GPU up to 32 nodes. (bottom) Parallel efficiency for strong scaling results displayed on top, with additional GPU results using a 2.2M element mesh. We note the greater than 1 initial CPU parallel efficiency, thought to be a result of increased cache efficiency.

GPU→CPU, MPI, and CPU→GPU. The outer "halo" region makes up only 1 to 2% of the total elements, matching the run time. However, if we scale up to 32 GPUs, the outer region becomes >10% of the total elements. By hiding these communications through overlap, we save 15% or more in run time, and maintain high parallel efficiency. Beyond CPU-GPU communications, any serial overhead or imbalance in the code will cause scaling inefficiencies:

- *Efficiency for different number of nodes*: Note the super-linear scaling performance of the CPU version shown in the bottom panel of Figure 4. We assume this increase in performance is due to increased cache efficiency as more of the mesh is able to fit within the various levels of cache. Our tests show that with such an improved cache usage on XE6 nodes, CPU simulations can run up to 20% faster. This would indicate that a type of sorting algorithm or space-filling curve could be very effective to improve cache and overall performance of the CPU version.
- *Efficiency for different mesh sizes*: Figure 4 (bottom) also shows the parallel efficiency of GPU simulations with two different meshes (300K and 2.2M elements). Best fitting scaling curves with Amdahl's model (13) set the effectively parallelizable code fraction $\alpha = 0.985$ to fit the 300K mesh and $\alpha = 0.975$ for the 2.2M mesh. We see that, despite the effectiveness of overlapped communications, strong scaling on GPUs remains a challenge. However, in many applications, such extreme strong scaling is avoided and the simulations are run with much higher per-node memory utilization, yielding higher efficiency and further resources for additional parallel simulations. Profiling using built-in profiling tools by NVIDIA indicates that overlapped communications are functioning correctly. Previously unoptimized functionality that was a very small percentage of overall run time at a smaller number of nodes, is causing inefficiencies at scales of 64+ nodes for a mesh of 300K elements. Note that scaling efficiency is dependent on the mesh size determined by the application. We find that loading a single GPU card with less than 15,000 spectral elements can lead to efficiency below 80%.

Concerning speedup values between GPU and CPU simulations, we see at 2 nodes a speedup factor of 2.5x, and at 16 nodes the speedup is 1.8x. At 32 nodes and further, both versions lose parallel efficiency due to sequential portions of the code.

In order to further compare the performance of the 32-core AMD nodes, we also tested two Intel nodes: a 2.6 GHz 2x6-core X5650 Westmere node, and a newer 2x8-core E5-2670 Sandy Bridge node. Using the Intel compiler, the 12-core Westmere node achieved 32 GFLOP/s and the 16-core Sandy Bridge node achieved 67 GFLOP/s, both with good scaling from 2 to 12 (or 16) cores. This is compared to 62 GFLOP/s on the AMD-based XE6 using the Cray Fortran compiler and all 32 Interlagos cores.

VI. Results of Realistic Large Simulations

A. Turkey scenario earthquake simulations

A common example of seismic forward modeling lies in the quantification of ground shaking due to earthquakes. Such simulations are important for comparing synthetic seismograms based on current knowledge of the structural region against observations and lead to better assessments in seismic hazard analysis. We focus on earthquakes in Turkey, a country that is subjected to high seismic hazard. Our settings simulate ground motions resulting from significant fault ruptures close to Istanbul.

We construct a spectral-element mesh that honors surface topography (Figure 5, top panel) and the undulating crust-mantle interface (called the "Moho" interface in geophysics, Figure 5, bottom panel), spanning from Turkey to Portugal, North Africa to Scandinavia, and down to a depth of 1,500 km (i.e., in the mid-mantle of the Earth). The mesh contains about 19 million spectral elements, leading to 3.8 billion degrees of freedom and collective run-time GPU-memory occupation of 260 GB with 290 MB per GPU for 896 XK6 nodes. We model three earthquake scenarios and run the simulations on 896 and 448 Titan nodes to test scalability for this realistic example. On 896 nodes, the solver requires 1682 s to complete the necessary 75,000 time steps for 1500-second seismograms (i.e. almost realtime simulation), yielding 35 TFLOP/s of floating point performance. For the same setup on 448 GPUs, we have 26 TFLOP/s of performance. In order to evaluate the resulting seismic waves, seismograms were recorded at stations in Turkey and Europe including a synthetic grid of stations

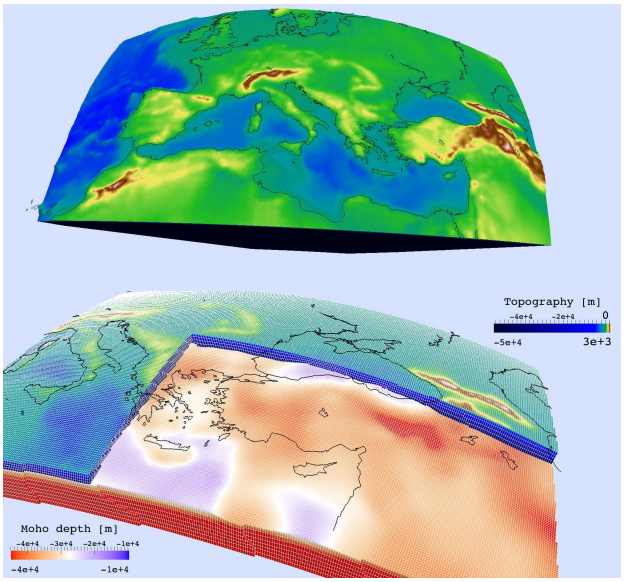

Fig. 5. A 3-D unstructured crust and mantle model in Europe including 19 million spectral elements, honoring surface topography (top) and the undulating crust-mantle Moho interface (bottom).

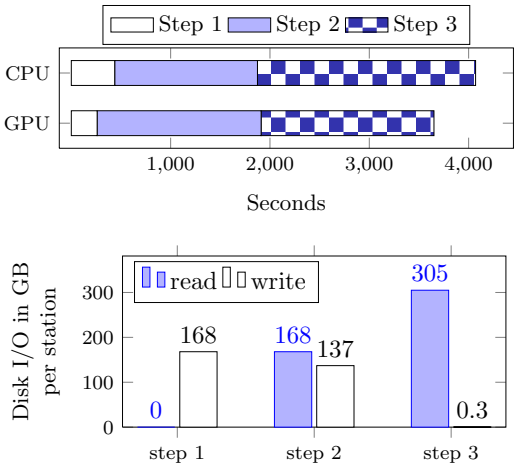

Fig. 6. Run time comparison of the three steps for a single ambient-noise sensitivity kernel. All three steps were run on 8 XE6 nodes and 8 XK6 nodes for the CPU and GPU versions, respectively. The run times shown measure the time step loop only (top). (bottom) The disk I/O read and write profile is shown for a single station on a 192,892 element mesh.

encompassing Istanbul. These recording stations tend to be clustered, and are very unbalanced between the 896 partitions of the 19M element mesh seen in Figure 5. Repeating the 896-GPU simulation with only a single station required less than half the time, yielding 78 TFLOP/s of performance. This type of problem may not be as pronounced in the CPU version because the individual mesh partitions would be 32x smaller, allowing for a better distribution of stations among processors.

We investigated the ground motion in Istanbul produced by a hypothetical magnitude 7.3 earthquake located in the Sea of Marmara fault region, 20 km southwest of Istanbul. As part of the analysis, a 21×21 grid of stations was placed on a 60 km grid surrounding Istanbul, in order to evaulate vertical and horizontal seismic ground shaking for the city following the study done in [20].

Beyond this hypothetical earthquake scenario, we simulated the 1999 Izmit magnitude 7.8 earthquake and the 2011 magnitude 7.3 earthquake in Eastern Turkey in order to test the long-range quality of our Earth model for typical seismic frequency ranges recorded at stations in Switzerland and Germany. Because Western Europe produces few earthquakes, improving the earth model using tomography must rely on earthquakes that occur at a relatively large distance from the area in question. This motivates the use of ambient noise as a replacement for naturally occuring earthquakes in this region.

B. Ambient-noise tomography simulations in Europe

To enhance seismic resolution in locations with little seismicity such as Western Europe, one can conveniently exploit information by cross-correlating stacked ambient noise between two seismographs [22]. Such cross correlations deliver signals reminiscent of surface-wave propagation from one seismograph to the other. In principle, one can thus use any pair of seismic stations as a combination of source and receiver, enabling seismic tomography without earthquakes to see a dramatic growth in seismic coverage and resolving power.

The noise-tomography adjoint approach [24] requires three forward/adjoint simulations per optimization iteration. This three-step procedure can be run independently for each station involved in such studies. We estimate that it is necessary to run 20 conjugate-gradient iterations to converge to a final model with sufficient misfit reduction, resulting in 9000 simulations of about 4.5 Million CPU-hours. Each simulation produces a significant amount of disk I/O, which considerably affects the performance on GPUs. Such ambient-noise simulations are computationally the most demanding case of adjoint-based inverse simulations implemented in the SPECFEM3D package. For our simulations, we use a database of 150 stations of measurements from station-station cross correlations filtered at periods between 8 to 35 s.

As a starting point, we create a mesh that combines a high-resolution crustal model [17] with a large-scale model of the 3-D seismic wave speeds in the upper mantle [21]. That mesh covers Western Europe and includes all seismic stations of the database, extending to a depth of 200-km. It also honors the topography of both the free surface of the Earth and crust-mantle interface, and is designed to resolve periods of 8 s with elemental average dimension of 24 km. It contains 303,116 elements, resulting in 0.13 billion degrees of freedom occupying 3 GB of GPU run-time memory.

Figure 6 depicts the three-step run time profiles for ambient-noise simulations running on CPU only or on GPUs. Disk I/O required for each of the three steps to calculate a station sensitivity kernel varies and has a measurable effect on the overall performance of the simulations. The GPU simulation gains primarily on the first and third step, but is slower on

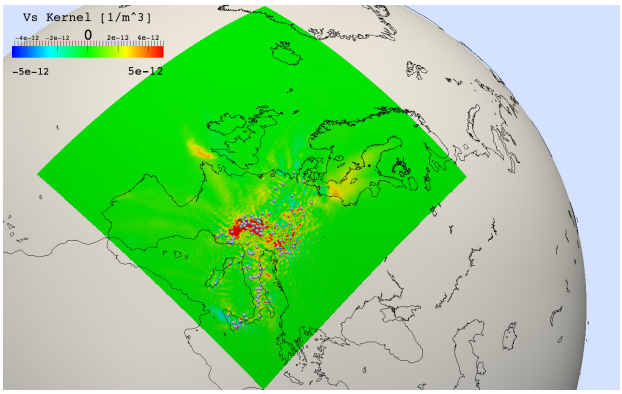

Fig. 7. Sensitivity kernel for shear-wave velocity, which is the fundamental building block for the first model update in a conjugate-gradient inversion. This kernel was obtained by summing contributions from 3×104 simulations, each obtained considering a different reference station (blue dots). Slice is taken at 15 km depth, i.e., within the crust.

the second step due to the large I/O output required by the second step. Note that the third step is similar to conventional earthquake-based sensitivity kernel simulations supported in the SPECFEM3D package; the first and second step, however, are specific to ambient-noise simulations chosen as an example here.

Each three-step simulation per station can be run in parallel (up to 150 for our case), but due to the I/O bottleneck only a fixed number can be run at once depending on the I/O subsystem. In order to reduce the time-to-solution, we scaled each simulation out to eight GPUs, and note that strong scaling is important to maintain high efficiency along with total performance. The output of each 3-step process is a sensitivity kernel, each of which is then summed among all stations depicted in Figure 7 to produce the overall sensitivity kernel needed to update the model. Such kernels are the fundamental building blocks to guide a nonlinear conjugate-gradient optimization model update.

VII. Conclusions

We have extended the full-featured SPECFEM3D software package to run on large GPU clusters. We compared the most recent Cray XE6 32-core CPU nodes against single-GPU XK6 nodes and demonstrated weak scaling up to 128 nodes, with a GPU performance gain of 1.7x. The excellent weak scaling performance indicates that GPU simulations with SPECFEM3D can be very efficient when each GPU node has enough elements in its partition, i.e. when granularity is chosen thoughtfully.

We also compared the strong scaling performance of different-sized meshes on the GPU and CPU, which highlights a level of inefficiency when scaling the GPU version past a number of elements/GPU. These results, especially the change in performance by varying the number of elements per GPU, are important to guide future optimization work. The weak scaling results do indicate, however, that if the GPU memory is kept full, high performance and efficiency can be attained at scale.

Because the GPU version only runs a single MPI rank per node (for the XK6), it is able to reduce the overall communication load when compared to the CPU version, which we expect to be an advantage at very large scale. This advantage can in principle be emulated by a hybrid OpenMP + MPI version, which has not been shown to perform well for SPECFEM3D so far and would require further exploration and optimization. We are also exploring the use of accelerator directives such as OpenACC to lower the programmer effort required by CUDA programming and the Fortran90/CUDA bridge required by our current solution.

The performance of our GPU simulations seems to be limited by GPU memory bandwidth. The arithmetic intensity of the longest running kernel per time step is 2.0, and we hope that future NVIDIA Kepler platforms will have an increase in memory bandwidth among other optimizations, boosting the performance and making an even stronger case for scientific GPU computing.

This work was done in the context of European-scale tomography, which requires 9,000 simulations and about 4.5 million CPU hours. Using GPUs, along with our current speedup of 1.7 to 2.5x and any future optimization gains will reduce the required resources and time to solution in order to complete such an iterative inversion project. The scientific problems driving this performance work allow for a further degree of parallelism across simulations, encouraging efficiency over single-simulation scaling performance.

Acknowledgments

We acknowledge the support of the Swiss National Supercomputing Center and ORNL for their hardware and support. The work done in Switzerland is funded through the Petaquake project, under the HP2C (High Performance and High Productivity Computing) initiative as well as the Initial Training Network QUEST by the European Commission. The work done in France and in the USA was supported in part by the French ANR and American NSF under G8 grant #2010G8EX00203 'Seismic Imaging'. D.K. thanks Christophe Merlet for early discussion about GPUs.

References

[1] L. Bautista-Gomez, N. Maruyama, D. Komatitsch, S. Tsuboi, F. Cappello, S. Matsuoka, and T. Nakamura. FTI: high performance fault tolerance interface for hybrid systems. In *Proceedings of the Supercomputing 2011 Conference*, 2011.

[2] L. Carrington, D. Komatitsch, M. Laurenzano, M.M. Tikir, D. Michéa, N. Le Goff, A. Snavely, and J. Tromp. High-frequency simulations of global seismic wave propagation using SPECFEM3D GLOBE on 62K processors. In *Proceedings of the Supercomputing 2008 Conference*, 2008.

[3] C. Cecka, A.J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *International journal for numerical methods in engineering*, 85(5):640–669, 2011.

[4] C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(68):318 – 331, 2008.

[5] T. CUBIT. Cubit 13.2 users manual. *Sandia National Laboratories, Albuquerque, NM*, 2012.

[6] J.-J. Droux. An algorithm to optimally color a mesh. *Computer Methods in Applied Mechanics and Engineering*, 104(2):249–260, 1993.

[7] D. Kirk, W.H. Wen-mei, and W. Hwu. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[8] D. Komatitsch. Fluid-solid coupling on a cluster of GPU graphics cards for seismic wave propagation. *Comptes Rendus de l'Acadmie des Sciences Mécanique*, 339:125–135, 2011.

[9] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Joural of Computational Physics*, 229(20):7692–7714, 2010.

[10] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa. Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs. *Computer Science Research and Development*, 25(1-2):75–82, 2010.

[11] D. Komatitsch, D. Michéa, and G. Erlebacher. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing*, 69(5):451–460, 2009.

[12] D. Komatitsch and J. Tromp. Introduction to the spectral element method for three-dimensional seismic wave propagation. *Geophysical Journal International*, 139(3):806–822, December 1999.

[13] D. Komatitsch and J. Tromp. Spectral-element simulations of global seismic wave propagation-I. Validation. *Geophysical Journal International*, 149(2):390–412, May 2002.

[14] D. Komatitsch, S. Tsuboi, J. Chen, and J. Tromp. A 14.6 billion degrees of freedom, 5 teraflop, 2.5 terabyte earthquake simulation on the Earth Simulator. *Proceedings of the ACM/IEEE Supercomputing SC'2003 conference*, 2003. on CD-ROM.

[15] M. Krivelevich. Coloring random graphs–an algorithmic perspective. In *Proc. 2nd Colloquium on Mathematics and Computer Science*, pages 175–195, 2002.

[16] R. Madariaga. Dynamics of an expanding circular fault. *Bulletin of the Seismological Society of America*, 66(3):639–666, 1976.

[17] I. Molinari and A. Morelli. Epcrust: a reference crustal model for the european plate. *Geophysical Journal International*, 185(1):352–364, 2011.

[18] T. Nissen-Meyer, A. Fournier, and F. A. Dahlen. A 2-D spectral-element method for computing spherical-earth seismograms–I. Moment-tensor source. *Geophysical Journal International*, 168:1067–1093, 2007b.

[19] D. Peter, D. Komatitsch, Y. Luo, R. Martin, N. Le Goff, E. Casarotti, P. Le Loher, F. Magnoni, Q. Liu, C. Blitz, T. Nissen-Meyer, P. Basini, and J. Tromp. Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes. *Geophysical Journal International*, 186(2):721–739, August 2011.

[20] N. Pulido, A. Ojeda, K. Atakan, and T. Kubo. Strong ground motion estimation in the sea of marmara region (turkey) based on a scenario earthquake. *Tectonophysics*, 391(1):357–374, 2004.

[21] J. F. Schaefer, L. Boschi, T. W. Becker, and E. Kissling. Radial anisotropy in the European mantle: Tomographic studies explored in terms of mantle flow. *Geophysical Research Letters*, 38(23):L23304, December 2011.

[22] N.M. Shapiro, M. Campillo, L. Stehly, and M.H. Ritzwoller. High-resolution surface-wave tomography from ambient seismic noise. *Science*, 307(5715):1615, 2005.

[23] A. Tarantola. *Inverse Problem Theory*. Society for Industrial and Applied Mathematics, Philadelphia, 2005.

[24] J. Tromp, Y. Luo, S. Hanasoge, and D. Peter. Noise cross-correlation sensitivity kernels. *Geophysical Journal International*, 183(2):791–819, 2010.

[25] J. Tromp, C. Tape, and Q. Liu. Seismic tomography, adjoint methods, time reversal and banana-doughnut kernels. *Geophysical Journal International*, 160:195–216, 2005.

[26] H. M. Tufo and P. F. Fischer. Terascale spectral element algorithms and implementations. *Proceedings of the ACM/IEEE Supercomputing SC'1999 conference*, 1999. on CD-ROM.

[27] H. Zhu, Y. Luo, T. Nissen-Meyer, C. Morency, and J. Tromp. Elastic imaging and time-lapse migration based upon adjoint methods. *Geophysics*, 74:WCA167–WCA177, 2009.