High Performance Seismic Wave Propagation with SPECFEM3D

Max Rietmann\textsuperscript{1}
Olaf Schenk\textsuperscript{1}
Daniel Peter\textsuperscript{2}

\textsuperscript{1}Institute for Computational Science
USI Lugano, Switzerland

\textsuperscript{2}Institute of Geophysics,
ETH Zürich, Switzerland

HPC Advisory Council — April 3, 2014
Abstract

SPECFEM3D is classic example of the state of current scientific HPC software.

- MPI + Fortran = Performance
**Abstract**

*SPECFEM3D is classic example of the state of current scientific HPC software.*

- MPI + Fortran = Performance
- MPI + GPUs + Clever Algorithms = More Performance
**Abstract**

*SPECFEM3D is classic example of the state of current scientific HPC software.*

- MPI + Fortran = Performance
- MPI + GPUs + Clever Algorithms = More Performance

Today’s outline:

- Elastic Wave Propagation & Seismic Imaging
- SPECFEM3D
- GPU SPECFEM3D
- Conclusions and Perspectives
Forward vs. Inverse “Modeling”

Our general nonlinear setup:

\[ G(m) = d \]  \hspace{1cm} (1)

where

- \( G() \): “simulation function”
- \( m \): earth model \((v_p, v_s, \rho)\)
- \( d \): “data”, e.g., recorded field traces at surface

Assuming \( G() \),

- **Forward** modeling uses \( m \) to produce \( d \). (easy)
- **Inverse** modeling uses \( d \) to produce \( m \). (hard)
The Forward Problem

\[ \rho \frac{\partial^2 \vec{u}}{\partial t^2} - \nabla \cdot T(\vec{x}, t) = f(\vec{x}_s, t), \quad (2) \]

with stress
\[ T(\vec{x}, t) = C(\vec{x}) : \nabla \vec{u}(\vec{x}, t), \quad (3) \]

and \( C \) is the fourth-order elasticity tensor.
The Forward Problem

The physics of wave propagation.

\[ \rho(\vec{x}) \frac{\partial^2 \vec{u}}{\partial t^2} - \nabla \cdot \mathbf{T}(\vec{x}, t) = f(\vec{x}_s, t), \quad (2) \]

with stress

\[ \mathbf{T}(\vec{x}, t) = \mathbf{C}(\vec{x}) : \nabla \vec{u}(\vec{x}, t), \quad (3) \]

and \( \mathbf{C} \) is the fourth-order elasticity tensor.
Spectral Element Method (SEM)

Spatial discretization:

- Continuous Galerkin FEM
- Nodal basis functions on GLL colocation points (Lagrange polys)

Finite element treatment yields

\[
M \ddot{u} + Ku = F \tag{4}
\]

Using Gaussian quadrature, \( M \) is diagonal!

\[
\ddot{u} = -M^{-1}(Ku - F) \tag{5}
\]
More SEM properties

- Only Hexahedra
- Hexahedra ideal for wave-propagation
- Meshing can be difficult (Trelis née CUBIT)
- Codes using Tetrahedra use *discontinuous* Galerkin (DG) methods (ADER-DG of Käser et al.)
Time-Stepping

Explicit Newmark Stepping

\[ u_{n+1} = u_n + \Delta t \ v_n + \frac{\Delta t^2}{2} \ a_n \]
\[ a_{n+1} = M^{-1} K u_{n+1} \]
\[ v_{n+1} = v_n + \frac{\Delta t}{2} (a_{n+1} + a_n) \]

Properties:

\[ \text{Conservative} \]
\[ \text{\( \Delta t \) limited by CFL} \]
\[ \text{Straightforward MPI-Parallelization (MPI-Sync after } K u) \]
Quick Movie
“Inverse Problems”, i.e., Seismic Imaging of the Earth’s Interior Structure

\[ \arg\min(J(m)); J(m) = \int_0^T \left( \sum_{i=1}^{\text{stations}} \| G(m)(x_i)(t) - d_i(t) \|_2 \right) + \alpha R(m) \] (6)

Adjoint Tomography yields \( \nabla_m J \), simplified we update using:

\[ m_{n+1} = m_n + \beta \nabla_m J \]
Seismic Imaging

Fig. 8. Sensitivity kernel for shear-wave velocity, representing the first model update in the conjugate-gradient inversion. This kernel was obtained by summing contributions from 104 sets of simulations, each obtained considering a different reference station (blue dots). Top: 15km depth, i.e. within the crust. Bottom: 40km i.e. in the shallowest part of the mantle.

surface-wave tomography of the uppermost mantle down to periods of 35 seconds. Our mesh (Figure 7) covers Western Europe and includes all seismic stations of the database, extending to 200 km depth. 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 seconds with elemental average dimension of 24 km.

B. Computational cost

The mesh contains $303,116$ elements, resulting in $1.3 \times 10^8$ degrees of freedom occupying 3GB of GPU run-time memory. The noise-tomography adjoint approach requires 3 forward/adjoint simulations for 150 stations per optimization iteration. The 3-step procedure can be run independently for each station, creating a further level of parallelism. We estimate 20 necessary 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 can become a bottleneck if too many stations are run simultaneously.

Given a fixed amount of I/O throughput, strong scaling is used to reduce the time-to-solution. Given a fixed set of resources, the shortest time-to-solution is achieved through an experimentally chosen balance of simulation-level and station-level parallelism. As evident in the next sections, the GPU weak and strong scaling results indicate that GPUs are most efficient when given enough work (i.e., their memories are full). CPUs, on the other hand, scale superlinearly up to a point and the work allotment should take advantage of this higher efficiency. The code acceleration presented here is a general addition to the widely-used SPECFEM3D source code for inverse problems, and thus not restricted to noise tomography (which is computationally the most demanding case of adjoint-based inverse problems), but useful for any case of large-scale forward and inverse wave propagation.

C. Strong scaling

Figure 9 depicts scaling performance experiments upon a European mesh with $303,116$ elements running a purely forward simulation. The experiments were conducted on the Cray XK6 with up to 128 nodes and, with a single GPU per node, 128 GPUs. We also ran the same experiments on the Cray XE6 with 32-cores per node. The GPU version crosses the 80% parallel efficiency mark at 32 nodes (119MB/GPU).

Note the superlinear scaling performance of the CPU version, which is best noted in the bottom panel of Fig. 9. 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. This would also 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.

At 2 nodes, we see a speedup of 2.5x, and at 16 nodes, the speedup is 1.8x. At 32 nodes and further, the CPU version continues to scale very well, especially compared to the...
Seismic Imaging is Expensive

For each gradient

- 150+ sources
- 3 “forward” simulations per source
- 20–30+ iterations to converge
- 15 mins per simulation on 324 cores (mesh with 300K elements)

\[(150 \times 3 \times 30 = 13500 \text{ total simulations!})\]

= 1M CPU hours
Computationally, *forward* modeling is mostly solved. *Inverse* modeling is still new and has a lot of attack surfaces:

- Wait for faster computers
- More nodes (e.g., parallel source simulations)
- New architectures (GPUs & MIC)
- Better Algorithms (LTS, Improved model updates)
Outline

▶ Introduction
▶ **Specfem3D**
▶ GPU-Specfem3D
▶ Perspectives
Specfem3D Cartesian (vs. Globe)

- Fortran95
- MPI
- GPU (CUDA)

3 stages:
1. Mesh decomposition (once per mesh)
2. Mesh preparation (once per velocity model)
3. Source simulation (once per source)
do it=1,NSTEP

! -- simulation
call timestep_initial(d,v,a)
do iphase = outer,inner
   call stiffness_kernel(d,a,iphase)
call mpi_sync(a,iphase)
enddo
call timestep_finalize(d,v,a)

! -- save seismograms and gradient
call seismogram_io(d,v,a)
call gradient_io(d,v,a)

enddo
subroutine timestep_initial(d,v,a)
    d(:) = d(:) + deltat*v(:) + deltatsqover2*a(:)
    v(:) = v(:) + deltatover2*a(:)
subroutine stiffness_kernel(d,a,iphase)
  do ispec=1,NSPEC(iphase)
    do ijk = 1,125
      iglob = ibool(ijk,ispec,iphase)
      loc = d(iglob) ! gather
      ! ... stiffness kernel ...
      ! ...
      ! scatter (+ assembly)
      accel(iglob) = accel(iglob) + contrib
    enddo
  enddo
Outline

- Introduction
- Specfem3D
- GPU-Specfem3D
- Perspectives
GPU Version: Fortran + C/CUDA + MPI for GPU clusters

To use:

```
./configure --with-cuda
```

[1]: Rietmann et al., “Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures”, 2012
GPU Version: Fortran + C/CUDA + MPI for GPU clusters

To use:

```
./configure --with-cuda
```

[1]: Rietmann et al., “Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures”, 2012
### GPU Version in Brief

<table>
<thead>
<tr>
<th>F95</th>
<th>CUDA C</th>
<th>CUDA Kernel</th>
</tr>
</thead>
</table>
| if(GPU_MODE) then  
call K_gpu(sp)  
else  
call K_cpu(d,a)  
endif |
| void  
K_cpu_(Ptr* sp)  
{  
dim3 bk(num_elem);  
dim3 tds(125);  
K_kernel  
    <<<bk,tds>>>(...); } |
| __global__  
void K_kernel(...) {  
    int elem =  
        blockIdx.x;  
    int node =  
        threadIdx.x; } |
Adding GPU-support is a **major** effort.

- High development and maintenance cost.
- All or nothing (CPU-GPU transfers kill effective speedup)
Performance experiments conducted on:

- Cray XK6: 16-core AMD Opteron + X2090 GPU (Fermi)
Performance experiments conducted on:

- Cray XK6: 16-core AMD Opteron + X2090 GPU (Fermi)
- Cray XK7: 16-core AMD Opteron + K20X GPU (Kepler)
Performance Comparison

Performance experiments conducted on:

- Cray XK6: 16-core AMD Opteron + X2090 GPU (Fermi)
- Cray XK7: 16-core AMD Opteron + K20X GPU (Kepler)
- Cray XE6: 2 × 16-core AMD Opteron

We compare performance **node-to-node.** (XK6/7 vs. XE6)
Performance Comparison

Performance experiments conducted on:

- Cray XK6: 16-core AMD Opteron + X2090 GPU (Fermi)
- Cray XK7: 16-core AMD Opteron + K20X GPU (Kepler)
- Cray XE6: $2 \times 16$-core AMD Opteron

We compare performance node-to-node. (XK6/7 vs. XE6)

Systems:

- Rosa (XE6) and Tödi (XK6,XK7)
- Titandev (XK6)
300 \times 10^3 \text{ element mesh:}

![Graph showing performance (GFLOP/s) vs. number of nodes (1xGPU, 32xCPU) for XE6, XK6, and XK7. The graph indicates linear scalability and a performance improvement of 3.8x.]
300 \times 10^3 \text{ element mesh:}

![Graph showing performance and efficiency of different systems](image-url)
Case Study: Noise tomography

Experiment using seismic-noise interferometry, using standard adjoint-tomography techniques.

Initial 3D structure.

Structure update from 150 station contributions.
Case Study: Performance

The tomography algorithm requires 3-steps, with significant I/O in each step.

Simulations run on Tödi (XK6) and Rosa (XE6) at the Swiss Supercomputing Center CSCS.
Outline

- Introduction
- Specfem3D
- GPU-Specfem3D
- Perspectives
SPECFEM3D, is well developed.

- F95 a good choice at time of creation
- Now, I choose C++ with Eigen matrix library: Performance + Readability

Current development pathway:

1. Matlab 1-D Prototype
2. Reimplement everything in C/C++/Fortran (6-months)
3. Fix Bugs (1 month)

*However, a taste of developing in NumPy/SciPy has ruined me forever. Ideally, development flow looks like:*

1. Prototype in Python/Julia/Haskell
2. Profile and fix slow routines (1 month)
3. There is no step 3
Conclusions

A brief view of Seismology code SPECFEM3D

▶ Forward and Inverse modeling
▶ SEM, Time-Stepping, and Code layout
▶ GPU Version (don’t forget I/O)
▶ Perspectives

Takeaway: Don’t forget the developers who have to maintain fast code. Tooling/Languages/Libraries are important!
Backup Slides

Extra Slides
Hexahedra vs. Tetrahedra
Hexahedra vs. Tetrahedra
Hexahedra vs. Tetrahedra