Modeling for inversion in exploration Geophysics

Mathias Louboutin
Ph.D Defense of dissertation
February 26th 2020
Wave-equation based geophysical exploration
Seismic inversion

Infer 3D images from massive multi-experiment data:

- $\mathcal{O}(10^9)$ unknowns
- $\mathcal{O}(10^{15})$ datapoints
- propagate $\mathcal{O}(10^2)$ wavelengths
- >10k time steps
- acoustic only
- elastic much larger
- extreme compute & IO
Equivalent to (computationally)

- 10000 Layers CNN
- 1000x1000x1000 input (ie HD gifs)
- Tens of iterations
- Thousands of source experiments (training pair)
- Interested in the parameters (physical meaning and interpretation) not the data fit only
Motivations and problem statement

Complex mathematics
Mathematical problem

\[
\text{minimize } \frac{1}{2} \| \mathbf{P}_r \mathbf{A}^{-1}(\mathbf{m}) \mathbf{P}_s^T \mathbf{q} - \mathbf{d} \|^2_2 \quad \text{(Virieux and Operto, 2009)}
\]

\( \mathbf{m} \): squared slowness  
\( \mathbf{d} \): field recorded data  
\( \mathbf{A}(\mathbf{m}) \): discretized wave-equation  
\( \mathbf{q} \): source term  
\( \mathbf{P}_r \): projection onto the receivers locations  
\( \mathbf{P}_s \): projection onto the source location
Challenges

Non-convex problem

- Requires complex algorithms

Problem sizes are huge

- Seismic surveys consist of tens of thousands of individual experiments
- Model wave propagation over thousands of time steps in large domains
- Typical size of modeling matrix $A(m) \in \mathbb{R}^{n \times n}$, $n = \mathcal{O}(10^{13})$

Gradient based optimisation

- Requires adjoints, gradients (and Hessian approximations)
Motivations and problem statement

Complex mathematics

Complex Physics with multiple representations
Scalar acoustic wave-equations

\[
\frac{1}{c^2} \frac{d^2 p(x, t)}{dt^2} - \Delta p(x, t) = 0
\]

**Acoustic isotropic**

\[
O(10) \text{ Flops/grid point}
\]

\[
\frac{1}{\rho c^2} \frac{d^2 p(x, t)}{dt^2} - \nabla \cdot \left( \frac{1}{\rho} \nabla p(x, t) \right) = 0
\]

**Acoustic isotropic with density**

\[
O(100) \text{ Flops/grid point}
\]

\[
m(x) \frac{d^2 p(x, t)}{dt^2} - (1 + 2\epsilon(x)) H_{\bar{x}\bar{y}} p(x, t) - \sqrt{1 + 2\delta(x)} H_{\bar{z} \bar{r}} r(x, t) = 0,
\]

**Acoustic anisotropic**

\[
m(x) \frac{d^2 r(x, t)}{dt^2} - \sqrt{1 + 2\delta(x)} H_{\bar{x}\bar{y}} p(x, t) - H_{\bar{z} \bar{r}} r(x, t) = 0
\]

**Tilted transverse isotropic (VTI)**

\[
O(1000) \text{ Flops/grid point}
\]
Challenges

Flexible representation of the physics

Flop expensive and complex implementations

Approximation (TTI) not always self-adjoint
  - Derivation and
  - implementation of the true adjoint necessary
\[
\frac{1}{c^2} \frac{d^2 p(x, t)}{dt^2} - \Delta p(x, t) = 0
\]

Self-adjoint

\[
m(x) \frac{d^2 p(x, t)}{dt^2} - (1 + 2\epsilon(x)) H_{\bar{x}y} p(x, t) - \sqrt{1 + 2\delta(x)} H_{\bar{z}} r(x, t) = 0,
\]

Not self-adjoint

\[
m(x) \frac{d^2 r(x, t)}{dt^2} - \sqrt{1 + 2\delta(x)} H_{\bar{x}y} p(x, t) - H_{\bar{z}} r(x, t) = 0
\]

\[
m(x) \frac{d^2 p_a(x, t)}{dt^2} - H_{\bar{x}y} (1 + 2\epsilon(x)) p_a(x, t) - H_{\bar{x}y} \sqrt{1 + 2\delta(x)} r_a(x, t) = 0,
\]

\[
m(x) \frac{d^2 r_a(x, t)}{dt^2} - H_{\bar{z}} \sqrt{1 + 2\delta(x)} p_a(x, t) - H_{\bar{z}} r_a(x, t) = 0
\]
Effect of incorrect adjoint
Contributions

Complex Physics with multiple representations (Chapter 3 and 4)
  ▸ Separation of concern with Devito

Non trivial Performance evaluation (Chapter 2)
  ▸ The roofline model for performance prediction and evaluation

Enabling at-scale research and development with separation of concern
The Roofline model for PDEs

Chapter 2 Contributions:

- Performance prediction

- Solver design from predictions

- Portable and absolute performance metric


S. Williams, A. Waterman, D. Patterson, “The Roofline model offers insight on how to improve the performance of software and hardware.”, communications of the ACM 52
Roofline model

Measure hardware level performance

No source to source comparison ("I achieved 200x speedup")

Enables theoretical estimations of performance
Operational intensity

\[ \mathcal{OI}_{alg}(k) = \frac{\mathcal{F}_{kernel}(k)}{\mathcal{B}_{global}} = \frac{\mathcal{F}_{kernel}(k)}{4(l + s)} \]

\( \mathcal{OI}_{alg}(k) \) : Operational Intensity
\( \mathcal{B}_{global} \) : Data movement
\( \mathcal{F}_{kernel}(k) \) : number of floating point operation (Add/Mul)
\( l \) : number of load/read
\( s \) : number of store/write
Operational intensity, example

\[ f[x] = 0.5 \times (f[x + 1] + f[x - 1]) \]

\( l \): Two loads \( f[x + 1], f[x - 1] \)
\( s \): One store \( f[x] \)
\( F_{kernel}(k) \): One addition + One multiplication

\[ OI_{alg}(k) = \frac{1 + 1}{4 \times 1 + 2} = 0.17 \]
Standard algorithms

Theoretical SP performance
Theoretical memory bandwidth
LINPACK SP performance
STREAM memory bandwidth

Performance (GFLOPS)

Operational intensity (FLOPs/Byte)

SpMV
Stencil
3DFFT
Finite-differences estimates, acoustic

Theoretical SP performance
Theoretical memory bandwidth

STREAM memory bandwidth

Memory bound for realistic orders

Operational intensity (FLOPs/Byte)

Performance (GFLOPS)

Do people know what these orders are. Did you talk about implementation of Laplacian yet?

It isn’t yet yes, was gonna refer to the thesis for the details to avid making it too heavy. Equations are defined in intro part though (both acoustic and TTI)
Finite-differences estimates, TTI

- Theoretical SP performance
- Theoretical memory bandwidth
- LINPACK SP performance
- STREAM memory bandwidth

Operational intensity (FLOPs/Byte)

Performance (GFLOPS)

- 2nd order ($k = 3$)
- 6th order ($k = 7$)
- 12th order ($k = 13$)
- 24th order ($k = 25$)

Compute bound for realistic orders
Finite-differences

- Theoretical Operational Intensity
- Discretization dependent
- Physics dependent
- Architecture dependent
Conclusions

Performance prediction:
- can be estimated before implementation
- depends on wave-equation choice
- depends on discretization

One-fits-all not adapted to finite-difference codes

(Relative) Runtime not fully representative of performance

Needs flexible framework to take advantage of architecture
Separation of concerns

Chapter 3 contributions:
- High level symbolic language
- Finite difference DSL

Chapter 4 contributions:
- Code generation
- Just-in-time compiler

https://www.devitoproject.org
Finite-difference DSL

Separation of Concerns:

- Geophysicists focus on physics
- Computer scientists focus on software
- Mathematicians focus on numerical analysis
Devito


---

**Model**

**Discrete setup**

**Symbolic PDE**

**Operator**

**Compiler**

**C**

**Result**

**Physical setup**

**Symbolic discretization**

**Symbolic stencil**

**Input/output processing and setup**

**Stencil optimizations, memory setup, parallelisation, ..**

**Compile (gcc/clang/icc/..) and run**
Chapter 3


Physical setup

Symbolic discretization

Symbolic stencil

Input/output processing and setup

Stencil optimizations, memory setup, parallelisation, ..

Compile (gcc/clang/icc/...) and run
Contributions

Design of the symbolic DSL for finite-differences
Implementation of a high-level user interface
Implementation and enabling of non standard operations
- Interpolation/injection
- Imaging condition
Design and implementation of finite-difference automation
Design and implementation of geophysical propagators
Implementation of inversion operators (adjoint/gradients/..)
Wave-equation setup

\[ m(x, y, z) \frac{d^2 u(t, x, y, z)}{dt^2} - \Delta u(t, x, y, z) = 0 \]

Only parameter to change for a different spatial order

In Devito translates as:

```python
u = TimeFunction(name="u", grid=grid, time_order=2, space_order=spc_order)
m = Function(name="m", grid=model.grid)
equation = m * u.dt2 - u.laplace
```

[In [16]: equation = m * u.dt2 - u.laplace]

[In [17]: sympy.pprint(eq)]
```
def forward(model, source, receiver, 
    space_order=2):
    m, eta = model.m, model.damp
    # Allocate wavefield and auxiliary fields
    u = TimeFunction(name='u', grid=model.grid, 
        time_order=2, 
        space_order=space_order)

    # Derive stencil from symbolic equation
    eqn = m * u.dt2 - u.laplace + eta * u.dt
    stencil = solve(eqn, u.forward)
    update_u = Eq(u.forward, stencil)

    # Source injection and receiver interpolation
    src = source.inject(field=u.forward, 
            expr=src * dt**2 / m)
    rec = receiver.interpolate(expr=u)

    op = Operator([update_u] + src + rec, 
        subs=model.spacing_map)
```
# Run the forward propagator op()

```c
#include <cassert>
#include <cstdlib>
#include <cassert>
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <fstream>
#include <vector>
#include <cstdio>
#include <string>
#include <inttypes.h>
#include <sys/time.h>
#include <math.h>

struct profiler {
    double loop_stencils_a;
    double loop_body;
    double kernel;
};

struct flops {
    long long loop_stencils_a;
    long long loop_body;
    long long kernel;
};

extern "C" int ForwardOperator(double *m_vec, double *u_vec, double *damp_vec, double *src_vec, float *src_coords_vec, double *rec_vec, float *rec_coords_vec, long i1block, struct profiler *timings, struct flops *flops) {
    struct timeval start_kernel, end_kernel;
    gettimeofday(&start_kernel, NULL);

    for (int i3 = 0; i3<3; i3++) {
        flops->kernel += 2.000000;
        {
            t0 = (i3)%3;
            t1 = (t0 + 1)%3;
            t2 = (t1 + 1)%3;
        }
    }

    for (int i1b = 1; i1b<279; i1b+=i1block) {
        for (int i1 = i1b; i1<i1b+i1block; i1++) {
            ...
        }
    }
```
Vectorial extension

Earth is elastic (better physics = better inversion)
Elastic is expensive
- Low velocities (half of acoustic) => 16 time bigger
- Nine fields
- Nine scalar equations (15 with viscosity)

Complex and tedious implementation
Vectorial extension

grid = Grid((1751, 2001, 1501), extent=(35000., 40000., 15000.))

# Elastic parameters
lam = Function(name="lam", grid=grid, space_order=0, is_parameter=True)
mu = Function(name="mu", grid=grid, space_order=0, is_parameter=True)
rho = Function(name="rho", grid=grid, space_order=0, is_parameter=True)

# Absorbing mask
damp = Function(name="damp", grid=grid, space_order=0, is_parameter=True)

# Stress and particle velocities
v = VectorTimeFunction(name="v", grid=grid, space_order=so, time_order=1)
tau = TensorTimeFunction(name="tau", grid=grid, space_order=so, time_order=1)

# symbol for dt
s = grid.time_dim.spacing

# Velocity stress formulation in its vectorial form
u_v = Eq(v.forward, damp * (v + s / rho * div(tau)))
u_t = Eq(tau.forward, damp * (tau + s * (lam * diag(div(v.forward)) + mu * (grad(v.forward) + grad(v.forward).T))))
Without vectorial support

# Stress and particle velocities

\[
v = \text{VectorTimeFunction}(\text{name}="v", \text{grid}=\text{grid}, \text{space\_order}=\text{so}, \text{time\_order}=1) \\
\tau = \text{TensorTimeFunction}(\text{name}="\tau", \text{grid}=\text{grid}, \text{space\_order}=\text{so}, \text{time\_order}=1)
\]

\[
\begin{align*}
\text{# Velocity stress formulation in its vectorial form} \\
u_v &= Eq(v.forward, \text{damp} \times (v + s / \rho \times \text{div}(\tau))) \\
u_t &= Eq(\tau.forward, \text{damp} \times (\tau + s (\lambda \times \text{diag(div}(v.forward)) + \mu \times (\text{grad}(v.forward) + \text{grad}(v.forward).T))))
\end{align*}
\]

9 fields

9 equations
Chapter 4

Contributions

Interfacing with the symbolic API
Testing framework and continuous integration
Performance evaluation and performance metric
Compiler components implementation
Symbolic manipulation and lowering
Devito compiler architecture

**Analysis**

**Equation lowering**

Preliminary transformations, such as
- Constant folding
- Index shifting (to account for halo and padding)
- Subdimensions (to implement subdomains)

**Clustering**

Group equations according to data dependencies

**Symbolic optimization**

Flop-reduction optimisation, such as:
- Common sub-expressions elimination
- Aliases detection and precomputation
- Factorization

**Tree-fication**

Abstract Syntax Tree (AST)

**AST analysis**

Detect loop nest properties using advanced data dependence analysis, such as:
- Parallel vs sequential
- SIMD-vectorizability
- Affine vs non-affine
- Communication-avoiding opportunities

**AST optimization**

Loop nest optimization, such as:
- Loop blocking
- SIMD vectorization
- Distributed-memory parallelism (e.g., comp/comm overlap)
- Shared-memory parallelism

**Finalization**

Type and variable declarations, instrumentation for profiling, header files, globals, macros, ...

---

(eqn0, eqn1, injection, ...)

\[ eqn0 = m \times u.dt^2 - u.laplace \]

\[ u = \text{TimeFunction(name='u', grid=grid)} \]

\[ \text{op} = \text{injection} = \text{src.inject(field=u.forward, expr=src*s*s**2/m)} \]
Example step: Data dependency analysis

Represents data (array) accesses as (labelled) vectors in $\mathbb{Z}^n$

$$u[t + 2, x - 3] \rightarrow \begin{bmatrix} t + 2 \\ x - 3 \end{bmatrix}$$

Currently uses a simple in-house framework

The Devito compiler relies on data dependence analysis for many tasks

- Inferring the iteration direction (i++ or i-- ?)
- Loop scheduling (convert list of equations into a tree of loops)
- Loop optimizations (e.g., discovery of parallel loops)
Example step: topological sorting for maximal “loop fusion”

\[ u[t+1,x] = F0(u[t,x], u[t-1,x], v[t,x], \ldots) \]

\[ u[t+1,s] = F1(u[t+1,u coords[s]], \ldots) \]

\[ v[t+1,x] = F2(v[t,x], v[t-1,x], u[t,x], \ldots) \]

\[ v[t+1,s] = F3(u[t+1,v coords[s]], \ldots) \]

Only flow-dependences in the time (t) dimension!
Example step: Generalized Common Sub-expressions Elimination

\[ a[t, i, j] = \sin(\phi[i, j]) + \sin(\phi[i-1, j-1]) + \sin(\phi[i+2, j+2]) \]

Observations:
- Same operators (\(\sin\)), same operands (\(\phi\)), same indices (\(i, j\))
- Linearly dependent index vectors ([\(i, j\], [\(i-1, j-1\], [\(i+2, j+2\])

\[ B[i,j] = \sin(\phi[i,j]) \]

# 20 X flop reduction for TTI

<table>
<thead>
<tr>
<th>FD order</th>
<th>Flops dse=noop</th>
<th>Flops dse=basic</th>
<th>Flops dse=advanced</th>
<th>Flops dse=aggressive</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>501</td>
<td>217</td>
<td>175</td>
<td>95</td>
</tr>
<tr>
<td>4</td>
<td>539</td>
<td>301</td>
<td>238</td>
<td>102</td>
</tr>
<tr>
<td>8</td>
<td>1613</td>
<td>860</td>
<td>653</td>
<td>160</td>
</tr>
<tr>
<td>16</td>
<td><strong>5489</strong></td>
<td><strong>2839</strong></td>
<td><strong>2131</strong></td>
<td><strong>276</strong></td>
</tr>
</tbody>
</table>

dse: Devito Stencil Engine, performs stencil optimisations
MPI support — for free

Python-level features:

- Domain decomposition based on MPI Cartesian grid abstraction.
- New package numpy4mpi: NumPy arrays automatically split and distributed according to domain decomposition.
- Parallel data slicing dealt with efficiently under the hood
- Sources/receivers (i.e. “sparse data”) distributed automatically.
C-level features:
- Generated code contains the MPI halo exchanges. Required halo exchanges identified through data dependence analysis.
- Optimizations (e.g., reshuffling halo updates, computation/communication overlap) also exploit data dependence analysis.
- Data packing/unpacking is threaded for performance.
Performance evaluation

Roofline analysis

Strong scaling
Single-socket — TTI on Skylake 8180 for multiple optimization levels

**TTI<grid=[512,512,512], TO=[2], sim=1000ms>, varying<dse>, arch<skl8180>, backend<core>**

- **<basic>**
- **<advanced>**
- **<aggressive>**

**Best speedup:** ~3x aggressive vs basic

**Trend:** fewer flops (higher OI), better runtime

4th order FD
8th order FD
12th order FD
16th order FD
3D TTI performance:
- 768x768x768 grid points
- 1000ms propagation (416 time steps)

Scales linearly!

512³ → 768³
3.4 X larger grid

Order 16:
72s → 287s
3.9 X slower

Order 8:
93s → 292s
3.2 X slower
OpenMP (multi-threading) strong scaling

GFlops/s scaling according to the number of threads

- Optimal
- Acoustic
- TTI
- Elastic
- Viscoelastic

Normalized GFlops/s

Number of OMP threads
MPI strong scaling

GFlops/s scaling according to the number of nodes

- **Optimal**
- **Acoustic**
- **TTI**
- **Elastic**
- **Viscoelastic**

Number of nodes:
- $2^0$
- $2^1$
- $2^2$
- $2^3$
Contributions

Theoretical analysis of finite-difference solvers
Flexible and portable performance metric
Symbolic finite-difference DSL
Enables research and development for large scale inverse problems
Reproducible and verifiable research
Just-in-time compiler for high-performance computing
Hardware portability
JUDI

JUDI – Domain specific language for linear algebra abstractions, data parallelism & meta data in Julia

https://github.com/slimgroup/JUDI.jl
JUDI – true vertical integration

Julia
- Linear operators, data containers, IO
- Parallel modeling function
- Parallelization: distribute sources, data
- Serial modeling function
- Interface to Devito (Python)

Python
- Devito: symbolic definition of PDE
- Automatic code generation and JIT compilation

C
- Solve PDE w/ OpenMP parallelism

---

students
- Math/optimizers/cs/seismic practitioners

students
- CS/math/physics people

polyhydral compiler people
Example: LS-RTM w/ serial & parallel SGD

Algorithm 1 Preconditioned LS-RTM with SGD

for $j = 1$ to $n$ do
    $r_j = \mathbf{M}_l^{-1} \mathbf{J}_{r(j)} \mathbf{M}_l^{-1} x_j - \mathbf{M}_l^{-1} \delta r(j)$
    $g_j = \mathbf{M}_r^{-1} \mathbf{J}_{r(j)} \mathbf{M}_r^{-1} r_j$
    $t_j = \frac{\|r_j\|^2}{\|g_j\|^2}$
    $x_{j+1} = x_j - t_j g_j$
end for

# Stochastic gradient descent
batchsize = 10
niter = 32

for $j=1:niter$
    # Select batch
    idx = randperm(dd.nsrc)[1:batchsize]
    Jsub = subsample(J,idx)
    dsub = subsample(dd,idx)

    # Compute residual and gradient
    $r = \mathbf{M}_l \mathbf{J}_{sub} \mathbf{M}_r x - \mathbf{M}_l \mathbf{d}_{sub}$
    $g = \mathbf{M}_r' \mathbf{J}_{sub}' \mathbf{M}_r' r$

    # Step size and update variable
    $t = \text{norm}(r)^2 / \text{norm}(g)^2$
    $x -= t * g$
end
3D TTI RTM on Azure

Synthetic model based on SEG Overthrust + Salt models:

- Domain: 10 x 10 x 3.325 km
- Grid: 881 x 881 x 347 (12.5 m grid + ABCs)
- **Wide-azimuth acquisition w/ 1,500 randomly distributed OBNs**
- 799 x 799 dense source grid (12.5 m)
- Anisotropic TTI models + density
- Used source-receiver reciprocity
3D TTI imaging on Azure

Industry scale imaging

Realistic physical representation

Demonstrates Devito portability
Conclusions

Modeling for inversion with separation of concern

- High level abstractions enable research and development
- Just-in-time compiler provides portability and reproducibility
- Roofline model gives rigorous and absolute benchmarks


