Automatic Differentiation in the Devito Domain-Specific Language

Navjot Kukreja$^1$$^a$

Also:
Jan Hückelheim$^1$  Fabio Luporini$^1$  Sri Hari Krishna Narayanan$^2$  Gerard J. Gorman$^1$  Paul Hovland$^2$

October 18, 2019

$^1$Department of Earth Science and Engineering, Imperial College London, UK
$^2$Argonne National Laboratory, Lemont, IL, USA

$^a$This work was funded by the Intel Parallel Computing Centre at Imperial College London and EPSRC EP/R029423/1
Seismic Imaging - Motivation

Full Waveform Inversion (FWI): A PDE-constrained optimisation problem to understand the earth

**Figure 1:** Offshore seismic survey

Source: http://www.open.edu/openlearn/science-maths-technology/science/environmental-science/earths-physical-resources/petroleum/content-section-3.2.1
Problem Statement - the Forward Problem

Given source signal $q_s$ (at a given location) and the earth’s physical parameters $m$, the wave propagation can be simulated using the equation:

$$
\begin{align*}
& m \frac{d^2 u(x,t)}{dt^2} - \nabla^2 u(x,t) = q_s \\
& u(.,0) = 0 \\
& \frac{du(x,t)}{dt} \bigg|_{t=0} = 0
\end{align*}
$$

(1)

The function $u$ describes the entire wavefield. The signal received at the specific (given) receiver locations could be seen as:

$$
d_{sim} = P_r u = P_r A(m)^{-1} P_s^T q_s
$$

(2)

where $A$ is the action of the equation 1, $P_r$ is the receiver restriction operator, and $P_s$ is the source projection operator.
FWI can be defined as Virieux and Operto [2009]:

\[
\min_{\bm{m}} \Phi_s(\bm{m}) = \frac{1}{2} \left\| \bm{d}_{\text{sim}} - \bm{d}_{\text{obs}} \right\|_2^2
\]  

(3)

The gradient of the objective function \( \Phi_s(\bm{m}) \) with respect to the model parameter \( \bm{m} \) is given by Plessix [2006]:

\[
\nabla \Phi_s(\bm{m}) = \sum_{t=1}^{n_t} \bm{u}[t] \bm{v}_{tt}[t]
\]  

(4)

where \( \bm{u}[t] \) is the wavefield in the forward problem and \( \bm{v}_{tt}[t] \) is the second-derivative of the adjoint (reverse) field.
Why does it need to be fast?

• Large number of operations: \(\approx 6000\) FLOPs per loop iteration of a 16th order TTI kernel

• Realistic problems have large grids: \(1580 \times 1580 \times 1130 \approx 2.82\) billion points (SEAM benchmark \(^1\))

• \(2.82 \times 10^9 \times 6000 \times 3000(t) \times 2\) (forward-reverse) \(\approx 10^{17}\) FLOPs per shot

• Typically \(\approx 30000\) shots (\(\approx 3 \times 10^{21} = 3 \times 10^9\) TFLOPs per FWI iteration)

• Typically \(\approx 15\) FWI iterations (\(\approx 4.6 \times 10^{22} = 46\) billion TFLOPs total)

\(\approx\) days/weeks/months even on supercomputers

---

\(^1\)
Traditional Approach

\[
\begin{aligned}
\frac{m}{\partial t^2} u(x,t) - \nabla^2 u(x,t) &= q_s \\
\left. \frac{\partial u(x,t)}{\partial t} \right|_{t=0} &= 0 \\
u(.,0) &= 0
\end{aligned}
\]

void finite_difference_solver(...) {
    //...impenetrable "performance_optimised" code
}
<table>
<thead>
<tr>
<th>Computer science</th>
<th>Geophysics</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Fast code is complex</td>
<td></td>
</tr>
<tr>
<td>• Loop blocking</td>
<td></td>
</tr>
<tr>
<td>• OpenMP clauses</td>
<td></td>
</tr>
<tr>
<td>• Vectorization - intrinsics</td>
<td></td>
</tr>
<tr>
<td>• Memory - alignment, NUMA</td>
<td></td>
</tr>
<tr>
<td>• Common sub-expression elimination</td>
<td></td>
</tr>
<tr>
<td>• ADD/MUL balance</td>
<td></td>
</tr>
<tr>
<td>• Denormal numbers</td>
<td></td>
</tr>
<tr>
<td>• Elemental functions</td>
<td></td>
</tr>
<tr>
<td>• Non temporal stores</td>
<td></td>
</tr>
<tr>
<td>• Fast code is platform dependent</td>
<td></td>
</tr>
<tr>
<td>• Intrinsics</td>
<td></td>
</tr>
<tr>
<td>• CUDA/OpenCL</td>
<td></td>
</tr>
<tr>
<td>• Data layouts</td>
<td></td>
</tr>
<tr>
<td>• Fast code is error prone</td>
<td></td>
</tr>
<tr>
<td>• Change of discretizations - Numerical analysis</td>
<td></td>
</tr>
<tr>
<td>• Change of physics</td>
<td></td>
</tr>
<tr>
<td>• Anisotropy - VTI/TTI</td>
<td></td>
</tr>
<tr>
<td>• Elastic equation</td>
<td></td>
</tr>
<tr>
<td>• Boundary conditions</td>
<td></td>
</tr>
</tbody>
</table>

Not everyone is a polymath
Why automated

Computer science

• Fast code is complex
  • Loop blocking
  • OpenMP clauses
  • Vectorization - intrinsics
  • Memory - alignment, NUMA
  • Common sub-expression elimination
  • ADD/MUL balance
  • Denormal numbers
  • Elemental functions
  • Non temporal stores

• Fast code is platform dependent
  • Intrinsics
  • CUDA/OpenCL
  • Data layouts

• Fast code is error prone

Geophysics

• Change of discretizations - Numerical analysis

• Change of physics
  • Anisotropy - VTI/TTI
  • Elastic equation

• Boundary conditions

Not everyone is a polymath
Raising the abstraction with Devito

\[
\begin{cases}
    m \frac{d^2 u(x,t)}{dt^2} - \nabla^2 u(x,t) = q_s \\
u(., 0) = 0 \\
\frac{du(x,t)}{dt} |_{t=0} = 0
\end{cases}
\]

```python
pde = m * u.dt2 - u.laplace
stencil = Eq(u.forward, solve(pde, u.forward)[0])
fwd_op = Operator([stencil], ...)
```

```python
void finite_difference_solver(...) {
    //...impenetrable "performance Optimised" code
}
```
Under the hood of Devito

Equations lowering
Input Equations → Lowered Equations

Local analysis

Clustering
Lowered Equations → Clusters

Symbolic optimization [DSE]
Clusters → Clusters

IET construction
Clusters → IET [abstract syntax tree]

IET analysis
IET → IET

IET optimization [DLE/YLE]
IET → IET

Synthesis
IET → CGen AST → C/C++ string

JIT Compilation
C/C++ string → kernel.c → kernel.so

Declarations
Instrumentation for profiling
Header files, globals, macros, …

Enforcement of iteration direction
Grouping

Invariants extraction
Aliases detection
Common sub-expressions elimination
Factorization

Loop blocking
SIMD vectorization
Shared-memory (hierarchical) parallelism
Low-level optimization (e.g., sw prefatching)
**Figure 3:** Roofline plot of achieved performance on Skylake 8180 for different discretisations of the Acoustic wave equation
The Gradient again

\[ \nabla \Phi_s(m) = \sum_{t=1}^{n_t} u[t] v_{tt}[t] \]

Hand-derived adjoint equation

Can we automate this?
Finite difference generates a stencil

\[
pde = m \times u.\text{dt2} - u.\text{laplace}
\]

\[
stencil = Eq(u.\text{forward}, \text{solve(pde, u.\text{forward})}[0])
\]

\[
fwd_op = \text{Operator([stencil], ...)}
\]

```c
void finite_difference_solver(...) {
    ...
    for (int t=t_m; t<t_M; t++) {
        for (int x=x_m; x<x_M; x++) {
            u[t+1][x] = c1*u[t][x-1] + c2*u[t][x] + c3*u[t][x+1] + c4*u[t-1][x];
        }
    }
    ...
}
```
Figure 4: Finite difference generates a 3/4 dimensional stencil. The radius depends on the order of discretisation
AD on a Stencil
Figure 5: AD on a gather produces a scatter
Figure 6: The Stencil is originally a gather
Figure 7: AD converts it to a scatter
1D Stencil Example

Figure 8: The scatter can be split into individual updates
Figure 9: Shift the indices so every update is writing to the $ith$ element.
Figure 10: The intersection of the three iteration spaces can be combined into a single loop. The update inside the loop looks like a stencil again. The left and right remainder sections have “partial” stencils.
PerforAD

• Extends naturally to n dimensions
• Does not make any assumptions about the nature of the stencil
• Holds even when multiple stencils being applied to different parts of the domain
• Does not require zero-padding
Where should PerforAD live?

- Equations lowering
  - Input Equations → Lowered Equations

- Local analysis

- Clustering
  - Lowered Equations → Clusters

- Symbolic optimization [DSE]
  - Clusters → Clusters

- IET construction
  - Clusters → IET [abstract syntax tree]

- IET analysis
  - IET → IET

- IET optimization [DLE/YLE]
  - IET → IET

- Synthesis
  - IET → CGen AST → C/C++ string

- JIT Compilation
  - C/C++ string → kernel.c → kernel.so

- Enforcements of iteration direction
  - Grouping

- Invariants extraction
  - Aliases detection
  - Common sub-expressions elimination
  - Factorization

- Declarations
  - Instrumentation for profiling
  - Header files, globals, macros, …

- Enforcement of iteration direction

- Loop blocking
  - SIMD vectorization
  - Shared-memory (hierarchical) parallelism
  - Low-level optimization (e.g., sw prefetching)
Figure 12: Speedups for the wave equation solver on a Broadwell processor, using up to 12 threads. The Tapenade-generated code with manual parallelisation does not scale at all. The primal and PerforAD-generated adjoint benefit from using all 12 cores.
**Figure 13:** Absolute runtimes for wave equation primal and adjoint stencils and conventional adjoints in serial, as well as best observed primal and adjoint stencil run time in parallel. The best-observed performance of adjoint stencils was with 12 threads and is faster than the conventional adjoint by a factor of $3.4\times$. 
Automatic/Manual Differentiation produces code that has a very high requirement of memory.

Previously we introduced PyRevolve, an execution environment that implements memory-compute tradeoffs through:

- Revolve\(^1\)-based checkpoint-recomputation
- Lossy/Lossless Compression
- Combination of the above two

\(^1\) Griewank and Walther [2000]
Future Work

- Integrate this in Devito so it works for complex stencils (e.g. TTI)
- Implement AD for sources/receivers (high-level)
Thank you

Questions?
