Accelerating ideation & innovation cheaply in the Cloud
the power of abstraction, collaboration & reproducibility

Felix J. Herrmann

4th EAGE Workshop on High-performance Computing
Dubai, October 8, 2019
Accelerating ideation & innovation cheaply in the Cloud
the power of abstraction, collaboration & reproducibility

Charles Jones✉, Gerard Gorman†, Jan Hückelheim†, Keegan Lensink★, Paul Kelly†,
Navjot Kukreja†, Henryk Modzelewski★, Michael Lange†, Mathias Louboutin♛, Fabio
Luporini†, James Selvages✉, Phillipp Witte♛
Accelerating ideation & innovation cheaply in the Cloud
the power of abstraction, collaboration & reproducibility

Charles Jones*, Gerard Gorman†, Jan Hückelheim†, Keegan Lensink★, Paul Kelly†,
Navjot Kukreja†, Henryk Modzelewski★, Michael Lange†, Mathias Louboutin★, Fabio
Luporini†, James Selvages*, Phillipp Witte★

Georgia Institute of Technology

SLIM

Microsoft Azure

Amazon Web Services

Google Cloud Platform

UBC
THE UNIVERSITY
OF BRITISH COLUMBIA

osokey

Imperial College
London

Georgia Tech
Disclaimer

We worked w/ Google Cloud Computing Services, Amazon Web Services (AWS) and Microsoft Azure. We therefore refer to services & product names related to these platforms.

Technology presented is not tied to one specific Cloud provider and has been replicated on all major cloud platforms (AWS, Azure, Google Cloud)

I am not trying to sell anything... I am talking from the perspective of an entrepreneurial academician who wants to

- drive innovations more rapidly
- bring codes close to at scale technology validation
- deal w/ intermittent workloads
Early attempt — 1 y ago

```
for j=1:n
    r = J*x - d_obs
    g = J'*r
    x = x - alpha*g
end
```

Julia code

Google Cloud Platform
Early attempt

Seismic imaging on GCP: Fall 2018

- Lift & shift approach
- 2D LS-RTM on BP Synthetic 2004 model
- 32,000 cores on 1000 nodes
- Parallel pool using Ethernet
- SLURM + parallel Julia session
- 2 hours to launch pool
- Frequent interruptions and restarts of pool

- **Total cost in 10 days: 170,000$**
- **But we were able to hack something in a matter of weeks...**
Tiny marmousi
62k gridpoints
1.5MFlop/time-step
$10^{-8}$/time-step $\Rightarrow < $10

Sigsbee
2.2M gridpoints
56MFlop/time-step
$4 \times 10^{-7}$/time-step $\Rightarrow < $100

3D overthrust
222M gridpoints
6GFlop/time-step
$4 \times 10^{-5}$/time-step
$\Rightarrow $3000

Full marmousi
640k gridpoints
15MFlop/time-step
$10^{-7}$/time-step $\Rightarrow < $20

SEAM elastic
5.3G gridpoints
2.8TFlop/time-step
$0.02$/time-step $\Rightarrow $14M
full azimuth 35k shots

Cloud
$\times$
Recent success

ML & AI have been responsible for major breakthroughs
- rapid rate of innovation & radical performance improvements
- sharing of ideas & code
- modern abstracted code bases & tools

HPC developments in Oil & Gas
- relatively slow
- proprietary attitudes
- too small a community

We are lagging behind & operating at too high costs!
Rapid developments

Training Resnet-50 on Imagenet

<table>
<thead>
<tr>
<th></th>
<th>Facebook Caffe2</th>
<th>UC Berkeley, TACC, UC Davis Tensorflow</th>
<th>Preferred Network ChainerMN</th>
<th>Tencent TensorFlow</th>
<th>Sony Neural Network Library (NNL)</th>
<th>Fujitsu MXNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1 hour</td>
<td>31 mins</td>
<td>15 mins</td>
<td>6.6 mins</td>
<td>2.0 mins</td>
<td>1.2 mins</td>
</tr>
<tr>
<td>GPU Configuration</td>
<td>Tesla P100 x 256</td>
<td>1,600 CPUs</td>
<td>Tesla P100 x 1,024</td>
<td>Tesla P40 x 2,048</td>
<td>Tesla V100 x 3,456</td>
<td>Tesla V100 x 2,048</td>
</tr>
<tr>
<td>Date</td>
<td>Apr</td>
<td>Sept</td>
<td>Nov</td>
<td>July</td>
<td>Nov</td>
<td>Apr</td>
</tr>
<tr>
<td>Year</td>
<td>2017</td>
<td>2017</td>
<td></td>
<td></td>
<td>2018</td>
<td>2019</td>
</tr>
</tbody>
</table>

- short development cycle
- almost exclusively 2D

Thanks to Azure
So far

Our successes in FWI & RTM relied on hand code for
- FD stencils on CPUs/GPUs
- sensitivities & “adjoints”
- memory & IO handling

Remarkable achievement RTM/FWI=DCNN w/ 10k layers on $1k^3$ grids

Unfortunately, this approach
- does not scale very well to different wave physics
- is error prone, and
- impedes rapid innovation
Research questions

“How can we exploit ML & JIT compiler technology in the Cloud?”

- manage complexities of often monolithic code bases
- be more agile, reduce development time & (running) costs
- use serverless technology that removes need to touch all data all the time

Today’s agenda:

- abstractions for FD-based FWI & RTM w/ Devito* + Judi*
- serverless implementations* in the Cloud
- case study & road ahead

Not a lift & shift solution!

*open source under MIT license
Our approach

Create performant open source platform in the Cloud

<table>
<thead>
<tr>
<th></th>
<th>modeling</th>
<th>Q</th>
<th>gradient</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D acoustic</td>
<td>✔️</td>
<td>✘</td>
<td>✔️</td>
</tr>
<tr>
<td>3D acoustic TTI</td>
<td>✔️</td>
<td>✘</td>
<td>✔️</td>
</tr>
<tr>
<td>3D elastic</td>
<td>✔️</td>
<td>✔️</td>
<td>✘</td>
</tr>
<tr>
<td>3D elastic TTI</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
</tbody>
</table>

Will be finalized next months
Solution

DEVITO – Domain specific language for stencil-based finite difference code generation for PDEs w/ explicit time stepping in Python using SymPy.

https://www.devitoproject.org
Open-source software

Devito:

- Open-source MIT license
- High-level Python interface for discretization of ODEs + PDEs using finite differences
- Automatic performance optimization and JIT code generation
- https://github.com/opesci/devito
Raising the level of abstraction

\[ m \frac{\partial^2 u}{\partial t^2} + \eta \frac{\partial u}{\partial t} - \Delta u = 0 \]

```c
void kernel(...) {
    ...
    <impenetrable code with crazy performance optimizations>
    ...
}
```
Raising the level of abstraction

\[ m \frac{\partial^2 u}{\partial t^2} + \eta \frac{\partial u}{\partial t} - \Delta u = 0 \]

void kernel(...) {
    ...
    impenetrable code with crazy performance optimizations>
    ...
}
Raising the level of abstraction

\[ m \frac{\partial^2 u}{\partial t^2} + \eta \frac{\partial u}{\partial t} - \Delta u = 0 \]
Raising the level of abstraction

\[ m \frac{\partial^2 u}{\partial t^2} + \eta \frac{\partial u}{\partial t} - \Delta u = 0 \]

eqn = m * u.dt2 + eta * u.dt - u.laplace
solve(eqn, u.forward)
Raising the level of abstraction

\[ m \frac{\partial^2 u}{\partial t^2} + \eta \frac{\partial u}{\partial t} - \Delta u = 0 \]

```cpp
void kernel(...) { ... }
```

```cpp
eqn = m * u.dt2 + eta * u.dt - u.laplace
solve(eqn, u.forward)
```
Raising the level of abstraction

\[ m \frac{\partial^2 u}{\partial t^2} + \eta \frac{\partial u}{\partial t} - \Delta u = 0 \]

Devito

```python
eqn = m * u.dt2 + eta * u.dt - u.laplace
solve(eqn, u.forward)

void kernel(...) { ... }
```
u = TimeFunction(..., space_order=so)

```
for (int time = time_m, t0 = (time + 1)%(3), t1 = (time + 2)%(3); time <= time_M; time += 1, t0 = (time)%(3), t1 = (time + 1)%(3), t2 = (time + 2)%(3)) {
    for (int y = y_m; y <= y_M; y += 1) {
        for (int x = x_m; x <= x_M; x += 1) {
            eqn = m * u.dt2 + eta * u.dt - u.laplace
        }
    }
}
```

```
for (int x = x_m; x <= x_M; x += 1) {
    for (int y = y_m; y <= y_M; y += 1) {
        for (int z = z_m; z <= z_M; z += 1) {
            u[t1][x + 4][y + 4][z + 4] = 2*pow(dt, 3)*(-2.08333333333333e-4F*u[t0][x + 2][y + 4][z + 4] + 3.33333333333333e-3F*u[t0][x + 3][y + 4][z + 4] - 2.08333333333333e-4F*u[t0][x + 4][y + 2][z + 4] + 3.33333333333333e-3F*u[t0][x + 4][y + 3][z + 4] - 2.08333333333333e-4F*u[t0][x + 4][y + 4][z + 2] + 3.33333333333333e-3F*u[t0][x + 4][y + 4][z + 3] - 1.875e-2F*u[t0][x + 4][y + 4][z + 4] + 3.33333333333333e-3F*u[t0][x + 4][y + 4][z + 5] - 2.08333333333333e-4F*u[t0][x + 4][y + 4][z + 6] + 3.33333333333333e-3F*u[t0][x + 4][y + 5][z + 4] - 2.08333333333333e-4F*u[t0][x + 4][y + 6][z + 4])/(pow(dt, 2)*damp[x + 1][y + 1][z + 1] + 2*dt*m[x + 4][y + 4][z + 4]) + pow(dt, 2)*damp[x + 1][y + 1][z + 1]*u[t2][x + 4][y + 4][z + 4]/(pow(dt, 2)*damp[x + 1][y + 1][z + 1] + 2*dt*m[x + 4][y + 4][z + 4]) + 4*dt*m[x + 4][y + 4][z + 4]*u[t0][x + 4][y + 4][z + 4]/(pow(dt, 2)*damp[x + 1][y + 1][z + 1] + 2*dt*m[x + 4][y + 4][z + 4]) - 2*dt*m[x + 4][y + 4][z + 4]*u[t2][x + 4][y + 4][z + 4]/(pow(dt, 2)*damp[x + 1][y + 1][z + 1] + 2*dt*m[x + 4][y + 4][z + 4]));
    }
}
```

For time discretization:

- `so=4`: Using a time step of `dt=1` and spatial steps of `x_M, y_M, z_M`.

For space discretization:

- `so=12`: Using a time step of `dt=1` and spatial steps of `x_M, y_M, z_M`.

Flexibility in space/time discretization.
OMP/MPI scaling
Setup

- 300 x 300 x 300 grid
- 16th order FD
- 100 Time-steps
- Xeon E5-2670 8C
- Single socket for OMP scaling
- one MPI rank per socket per node for MPI scaling
Strong scaling OMP threads – near optimal
Strong scaling MPI – TTI compute bound

GFlops/s scaling according to node number

- Acoustic
- Optimal
- Viscoelastic
- Elastic
- TTI

Normalized GFlops/s

Number of nodes

$2^0$, $2^1$, $2^2$, $2^3$
Setup

• 512 x 512 x 512 grid
• varying FD order
• 1000ms modeling
• Intel Skylake 8180
• Single socket, OMP only
20 X flop reduction

<table>
<thead>
<tr>
<th>FD order</th>
<th>Flops noop</th>
<th>Flops basic</th>
<th>Flops advanced</th>
<th>Flops 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>5489</td>
<td>2839</td>
<td>2131</td>
<td>276</td>
</tr>
</tbody>
</table>
produces C code
compiled w/ -O3 + standard flags
Single-socket — TTI on Skylake 8180

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

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

Trend: fewer flops (higher OI), better runtime

Best speedup: ~3x aggressive vs basic

- 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
Solution

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

https://github.com/slimgroup/JUDI.jl
Open-source software

Julia Devito Inversion framework:
- JUDI.jl - MIT license
- Abstract linear operators and objective functions for FWI + LS-RTM
- Parallel out-of-core SEG-Y reader interface
- Interface to ML library Flux.jl
- URL: 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
**Algorithm 1** Preconditioned LS-RTM with SGD

for \( j = 1 \) to \( n \) do

\[
\begin{align*}
\mathbf{r}_j &= \mathbf{M}_I^{-1} \mathbf{J}^{(j)} \mathbf{M}_I^{-1} \mathbf{x}_j - \mathbf{M}_I^{-1} \mathbf{d}_{r(j)} \\
\mathbf{g}_j &= \mathbf{M}_I^{-1} \mathbf{J}^{(j)\top} \mathbf{M}_I^{-1} \mathbf{r}_j \\
\mathbf{t}_j &= \frac{\|\mathbf{r}_j\|^2}{\|\mathbf{g}_j\|^2} \\
\mathbf{x}_{j+1} &= \mathbf{x}_j - \mathbf{t}_j \mathbf{g}_j
\end{align*}
\]

end for

---

# Stochastic gradient descent

\[
\text{batchsize} = 10 \\
\text{niter} = 32 \\
\text{for } j=1: \text{niter} \\
\quad \# \text{Select batch} \\
\quad \text{idx} = \text{randperm(dD.nsrc)}[1: \text{batchsize}] \\
\quad \text{Jsub} = \text{subsample(J, idx)} \\
\quad \text{dsub} = \text{subsample(dD, idx)} \\
\quad \# \text{Compute residual and gradient} \\
\quad \mathbf{r} = \mathbf{M}_I \text{Jsub} \mathbf{M}_I \mathbf{x} - \mathbf{M}_I \text{dsub} \\
\quad \mathbf{g} = \mathbf{M}_I \text{Jsub} \mathbf{M}_I \mathbf{r} \\
\quad \# \text{Step size and update variable} \\
\quad \mathbf{t} = \frac{\text{norm(r)^2}}{\text{norm(g)^2}} \\
\quad \mathbf{x} -= \mathbf{t} \mathbf{g} \\
\text{end}
\]
Algorithm 2 Preconditioned LS-RTM with elastic average SGD

\begin{algorithm}
  \For{$j = 1$ to $n$ do}
    \For{$k = 1$ to $p$ do}
      \State $r_j = M_l^{-1} J_{jk} M_r^{-1} x_j^k - M_l^{-1} d_{jk}$
      \State $g_j = M_r^{-\top} J_{jk}^\top M_l^{-\top} r_j$ and $x_j^{k+1} = x_j^k - \eta g_j^k(x_j^k) - \alpha(x_j^k - \bar{x}_j)$
    \EndFor
    \State $\bar{x}_{j+1} = (1 - \beta) \bar{x}_j + \beta \left( \frac{1}{p} \sum_{i=1}^{p} x_j^i \right)$
  \EndFor
\end{algorithm}

---

# Gradient function

```julia
@everywhere function update_x(M_l, J, M_r, x, d, eta, alpha, xav)
    r = M_l * J * M_r * x - M_l * d
    g = M_r' * J' * M_l' * r
    return x - eta * g - alpha * (x - xav)
end
```

# Parallel function wrapper

```julia
update_x_par = remote(update_x)
```

---

# Select batch

```julia
idx = randperm(dD.nsrc)[1:batchsize]
Jsub = subsample(J, idx)
dsub = subsample(dD, idx)
```

---

# Calculate x update in parallel

```julia
xnew[:,k] = update_x_par(M_l, Jsub, M_r, x[:,k], dsub, eta, alpha, xav)
```

---

# Update average variable

```julia
xav = (1 - beta) * xav + beta * (1/p * sum(x, 2))
x = copy(xnew)
```
Least-squares migration

Making Broadband Least-squares Reverse-Time Migration affordable

Iteratively refining the output toward true reflectivity

Suppressing migration artefacts, wavelet sidelobes, incorrect amplitudes, poor illumination

Move LS-RTM from 10 x RTM compute cost to 2 x RTM compute cost

Applicable to all existing datasets

Sparsity-promoting LS-RTM: 2 data passes

RTM: 1 data pass

Cloud based 2D LS-RTM image of BP Salt model for < $100
Observations

Demonstrated power of abstractions towards data & compute intensive tasks
- flexibility w.r.t hardware (GPU, different CPUs etc.)
- exploits data-space parallelism
- exploits model space parallelism w/ multithreading & domain decompositions
- allows for reproducibility
- ready to scale technology to 3D TTI in Cloud?

“Not” ready for
- elastic
Solution

Serverless Cloud – Large-scale event-driven seismic imaging w/ automatic resource allocations, resilient nested levels of parallelization, and containerization

https://www.devitoproject.org
Seismic inversion in the cloud

Cloud computing:

✅ Pros
- Theoretically unlimited scalability
- High flexibility (hardware, jobs)
- No upfront + maintenance costs: pay-as-you-go
- Available to anyone
- No compromise – latest hardware & architectures available (GPUs, ARM)

❌ Cons
- Slower inter-node connections (depending on platform)
- Oftentimes larger MTBF
- High costs if not used properly
- Need to transition software
- Steep learning curve
Moving to the cloud

Lift & shift

Legacy Fortran or C code
Moving to the cloud

Go serverless (and re-engineer)

Automatic code generation

\[ pde = \text{model.m} \times \text{u.dt}^2 - \text{u.laplace} + \text{model.damp} \times \text{u.dt} \]
Moving to the cloud

Go serverless (and re-engineer)

- Save cost (up to 10x): no idle instances, lower start-up time
- Resilience managed by cloud platform
- Requires re-engineering of software

\[
pde = \text{model.m} \cdot \text{u.dt2} - \text{u.laplace} + \text{model.damp} \cdot \text{u.dt}
\]
Nested levels of parallelization:

- Parallelize shot records (Azure Batch)
- Domain decomposition (MPI)
- Multithreading (OpenMP)
- Each gradient computed on individual instance or cluster of instances (cluster of clusters)
RTM/FWI gradients

Software stack:

- Batch runs docker containers
- Solve wave equations using Devito*
- Automated performance optimizations (loop blocking, vectorization, refactoring, OMP, MPI, etc.)

* Luporini et al., 2018; Louboutin et al., 2019
RTM/FWI gradients

Azure - Batch Shipyard:

- Tool for executing + monitoring batch jobs
- Many templates for docker + **singularity** containers
- Pre-existing containers for MPI, **Infiniband**, ML, various compilers, etc.
- Configure pools + jobs using high-level yaml files
- Developed by Microsoft + **open source**: https://github.com/Azure/batch-shipyard
Summation:

- Gradients stored in object storage (blob)
- Virtually unlimited I/O scalability
- Send object IDs to message queue
- Event-driven gradient summation using Azure functions
RTM/FWI gradient computations

Event-driven gradient reduction:

- Azure functions
- Cheaper than pay-as-you-go nodes
- Asynchronous & parallel
- Invoked as soon as at least 2 gradients are available
- Stream gradients from blob → sum → write back
- Update image after final summation
Strong scaling - OpenMP

- Fixed workload: 1 gradient
- Runtime as function of no. of threads
- Performance on bare metal vs. container similar (w/o hyperthreading)
Strong scaling - MPI

- Fixed workload: 1 gradient
- Runtime as function of no. of instances (per gradient)
- Good speed-up **but** significant cost increase

![Graph showing runtime and speedup for different instances and services](image)
## Multi platform approach

<table>
<thead>
<tr>
<th></th>
<th>Azure</th>
<th>AWS</th>
<th>GCP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute instances</strong></td>
<td>Virtual machines</td>
<td>EC2</td>
<td>Compute engine</td>
</tr>
<tr>
<td><strong>Object storage</strong></td>
<td>Blob</td>
<td>S3</td>
<td>Cloud storage</td>
</tr>
<tr>
<td><strong>Batch computing</strong></td>
<td>Azure Batch</td>
<td>AWS Batch</td>
<td>Pipelines</td>
</tr>
<tr>
<td><strong>Serverless functions</strong></td>
<td>Azure functions</td>
<td>Lambda functions</td>
<td>Cloud functions</td>
</tr>
<tr>
<td><strong>Message queues</strong></td>
<td>Queue storage</td>
<td>SQS</td>
<td>Cloud Pub/Sub</td>
</tr>
<tr>
<td><strong>Distributed file system</strong></td>
<td>Azure files</td>
<td>EFS</td>
<td>Cloud filestore</td>
</tr>
</tbody>
</table>

https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services
https://cloud.google.com/docs/compare/aws/
Multi platform approach

Serverless batch computing

Azure
- Azure functions
- Azure Batch
- Queue
- Blob

AWS
- Lambda
- Step Functions
- AWS Batch
- SQS
- S3
Multi platform approach

Event-driven gradient summation

Azure

AWS
Numerical examples

Sparsity-promoting LS-RTM of the BP Synthetic 2004 model:

- 1348 shot records
- Velocity model: 67.4 x 11.9 km (10,789 x 1,911 grid points)
- 20 iterations of linearized Bregman method
- Batchsize of 200 shot records per iteration
- Curvelet-based sparsity promotion
Numerical examples

Sparsity-promoting LS-RTM on the BP Synthetic 2004 model

Image after $\approx 3$ data passes (total cost of $< 120$ $\$$)
Numerical examples

Reverse-time migration of the BP TTI model:

- 1641 shot records
- Velocity model: 78.7 x 11.3 km (12,596 x 1,801 grid points)
- Anisotropic modeling using pseudo-acoustic TTI equations*
- True adjoints of linearized Born scattering operator
- Domain-decomposition to compute gradients
- Each gradient computed on MPI cluster of 6 instances (no spot instances)

*Zhang et al., 2011
Numerical examples

Reverse-time migration of the BP TTI model:

RTM image (total cost of \( \approx 420 \text{ $} \))
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 RTM on Azure

Acquisition geometry:

OBN receiver grid
50 X 50 m

Source vessel grid
12.5 X 12.5 m
3D TTI RTM on Azure

3D Overthrust + Salt model:

Velocity

Delta
3D TTI RTM on Azure

3D Overthrust + Salt model:

Epsilon

Azimuth
3D TTI RTM on Azure

Observed data: 1,500 shots

Shot records in xline
3D TTI RTM on Azure

Depth slice 725 m

Depth slice 1250 m
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure
3D TTI RTM on Azure

Azure setup:

- E64 and ES64 VMs
- 2.3 GHz Intel Xeon® E5-2673 v4 (Broadwell)
- 432 GB RAM, 64 vCPUs per VM
- 100 VMs → 6,400 vCPUs
- 2 VMs per gradient (1 MPI rank per socket)
- 600 GB per wavefield
- Peak performance: 140 GFLOPS per VM (14 TFLOPS total)

- **Total cost for RTM:** ≈17,000$ (dedicated/on-demand)
**Timings:**
- 100 nodes
- 2 nodes per gradient
- 1500 source positions
- Average runtime: 110 minutes per gradient
- **Average cost per gradient: 11\$ (dedicated)**
- **Peak performance: 140 GFLOPS per VM (14 TFLOPS total)**
Azure Batch:

- Jobs start as VMs are added to pool
- Do not need to wait for full pool
- No long idle times
- At least **6X cost reduction** when using low priority...
Future directions

Go large:
- Ongoing collaboration Azure to run at industry-scale
- Iterative LS-RTM on large-scale 3D TTI
- SEAM model: long offset data acquisition w/ 3D elastic modeling

Check for updates on our website:

https://slim.gatech.edu/
Elastic
3D SEAM elastic, full offset, full azimuth

- 35km x 40km x 15 km
- 20m x 20m x 10m grid
- 12th order FD
- Velocity-stress formulation (9 coupled PDEs)
- Full offset & full azimuth
- 16s recording
- Modeling only (elastic imaging requires more than just compute)

.5Tb RAM
5.3 Gpoints
2.8TFlop/time-step
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))))
Setup

- Out of the box Devito
- Source at the centre (17.5km, 20km)
- 16s recording of Vx, Vy, Vz at ocean bottom
- 32 compute nodes (small node on azure, no InfiniBand)
- 3s per time-step
- 7 TFlop/s
Observations

Built a scalable reproducible imaging solution in the Cloud in 1y timeframe
  ‣ leveraging abstractions, open source, and collaboration
  ‣ using serverless Cloud native tools
  ‣ levels the play field

Proved that focussed
  ‣ industry-supported Consortia & public-private partnerships deliver
  ‣ needs to be sustainably funded

Contrast w/ industry-wide initiatives
  ‣ integration of different systems often fail
  ‣ suffer from scope creep
Observations

Created an industry-strength low-cost TensorFlow/Pytorch-like environment

- makes research findings directly available & reproducible
- changes how we spend our research budgets & interact with Consortia & Startups
- that drives innovations more rapidly by giving everybody a chance
Conclusions

Seismic imaging in the Cloud:

- feasible in Cloud but requires rethinking algorithms & implementations
- take advantage of high-throughput batch computing, serverless/event-driven computations, object storage, spot instances
- access to hardware w/o compromise w/ potential of hyperscaling
- only pay what you use: up to 10x cost reductions
- software based on separation of concerns + abstractions is prerequisite to go serverless
Acknowledgments

This project was made possible through the help of:

- Microsoft Azure
- Sverre Brandsberg-Dahl
- Evan Burness
- Kadri Umay
- Alexander Morris
- Steve Roach
- Hussein Shel
- Georgia Research Alliance & Georgia Institute of Technology