Accelerated Reverse Time Migration with optimized IO

1. Motivations, Value Proposition & Challenges

- Have a fast 2D seismic processing chain
  - For R&D purposes in both HPC and geophysics
  - For industrial purposes: quick reproducibility before full new 3D
  - Enabling and Collaboration with Oil Companies for technology adoption
- In this work we evaluate how to limit or avoid IO for Reverse time migration, depending on the workload size
- And we try to figure out what would be the impact of Intel Persistent Memory for a ‘fast 2D RTM on a workstation’

2. Seismic imaging

Seismic imaging is defined as an inverse problem. The Reverse Time Migration (RTM) is a time reversal scheme that solves the iterative 0 of the mistfit function minimization, while a Full Waveform Inversion (FWI) will do several iterations. In practice we correlate 2 wave fields to get the final image. One simulation does ‘forward’ propagation from source point, while the other does the backwards propagations using the seismic data, starting from the receiver positions up until the source point.

3. Memory Hierarchies

Seismic imaging will take advantage of new memory hierarchies, benefiting from both the bandwidth and size increases. First of all the SSD and SSD on PCIe provide extra bandwidth (BW) for local storage. Secondly, the persistent memory as 3D XPoint™ technology will also provide larger sizes that will fit the RTM’s needs for temporary snapshots during the execution (around 5 TB in 3D).


4. BP model and data size

We used the 2004 BP velocity model with only 200 Shots


Dataset specs:
- 200 shot gathers
- dx=0.5 m in time sampling
- rt = 20001
- (12 seconds trace length)

Velocity Model:
- dxy = 12.5 meters
- nx, ny = 5455, 1931 (42 MB)

Section of the velocity model after adding the random boundaries (Clapp, 2011, 2015)

5. Pseudo code for the 3 versions

Code 1: Snapshots are kept in memory increasing memory footprints

Input physical fields
Loop over seismic data
- Read prestack seismic traces
- Loop over time steps
  - Memory Weave propagation from source point
  - Optional Compute wavefield
- Store wavefield in memory (DRAM or NVMe)
End loop
Loop over time steps
- Mask propagates real data from source locations
- Mask previously simulated wavefield from memory
- Apply imaging condition (correlation of both wavefields)
End loop
Output seismic image (Sigs to PostHمت)
End loop

Code 2: Snapshots are stored on disk and rely on IO bandwidth and size

Input physical fields
Loop over seismic data
- Read prestack seismic traces
- Loop over time steps
  - Memory Weave propagation from source point
  - Optional Compute wavefield
- Store wavefield on disk (SSD, 800 GB or shared PCIe system)
End loop
Loop over time steps
- Mask propagates real data from source locations
- Mask previously simulated wavefield from disk
- Apply imaging condition (correlation of both wavefields)
End loop
Output seismic image (Sigs to PostHمت)
End loop

Code 3: Nothing stored except the last 2 Snapshots. That required a third full propagation increasing computation with neither IO nor extra memory.

6. Comparison of the versions

<table>
<thead>
<tr>
<th>Code</th>
<th>Snapshot remain in memory</th>
<th>Memory footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code 1</td>
<td>Yes</td>
<td>Memory footprint =&gt; Limited Workloads</td>
</tr>
<tr>
<td>Code 2</td>
<td>No WL limit</td>
<td>IO BW Penalty</td>
</tr>
<tr>
<td>Code 3</td>
<td>Reduced memory footprint</td>
<td>No WL limit</td>
</tr>
</tbody>
</table>

Code 1 in NVMe
- No extra DRAM footprint
- Snapshot remain in NVMe
- Speed if enough BW
- Almost no WL limit even in 3D
- Extra HBW in the node (NVMe DIMM)

7. Roofline example

Cache Aware roofline from Intel® Advisor for the code version 3 that is doing one forward and 2 backward propagations. Thanks to the vectorization report we are able to find out where the data movement (the blue and dark dots should be between 12 and 11 roots here) is limiting the performance (mostly ineffective pooled/remainder loops). Optimizing this performance will also impact the IO and will increase the IO BW needed for snapshots.

8. Elapsed time distribution for the 3 versions

Elapsed time distribution for each code when running 1 shot on an Intel® Xeon® Gold 6148 codenamed Skylake (2 sockets server, 40 cores @ 2.4 GHz). The number of time steps is limited to 4000 to allow code 1 to keep snapshots in memory. Firstly, we see that doing 3 propagations in code 3 is as fast as code 1 due to larger memory management that currently impacts the performance. For code 2, we see the impact of the IO bandwidth when testing a standard SSD and on a SSD on PCIe that should be improved with hardware tuning. This is why we want to find what hardware (and BW) would make IO totally transparent.

9. IO BW prediction and Memory footprint

Isotropic acoustic RTM case. Memory footprints (Blue line). To fit in a 192GB node and Snapshot size (to fit in Persistent memory) as a function of model (or domain size). The Black, red, green and orange lines represent 20k, 10k, 5k and 1k snapshots, respectively. We also show the sizes for the BP model to underlines how snapshots can grow rapidly.

Projected IO BW that would be needed to get rid of the IO bottleneck slowing down code 2. This plot shows IO BW as a function of the code performance expressed in GB/s. Firstly, we can see that performance of the 3 versions could be improved by a factor of 2 to 3 to get closer to the Stream Triad BW (about 200 GB/s on that platform). This confirms the results of the roofline analysis. Secondly, we see that an IO BW of about 10 GB/s (and up to 35 GB/s for a fully optimized code) would be needed per node. This is a very important conclusion for a minimum Persistent memory BW. If we would like to use it as a temporary storage without impacting the performance and without any limitations in terms of workload size.

10. Conclusions

Handling snapshots in memory has clear limitations due to large footprint requirements and penalty of moving extra data. This would work well for a very small workload. Doing IO for the snapshots makes sense if the hardware can deliver about 10 GB/s per node and probably more (50 GB/s should be a safe value) as we have shown our code still has room for optimization. This is why we believe that using persistent memory within the node would make sense if the bandwidth meets our needs.

Doing the 3 propagations without keeping snapshots is compelling and benefits from any kernel optimization. This version follows today’s trend of doing more FLOPS (even recomputed) than moving the data (or keeping extra arrays). Adding compression to the snapshots (more FLOPS, less IO) would certainly relieve these conclusions; "WIP"