TOTAL
COMMITTED TO BETTER ENERGY

USING OPENACC TO PARALLELIZE SEISMIC ONE-WAY BASED MIGRATION

Kshitij Mehta (Total E&P R&T)
Maxime Hugues (Total E&P R&T)
Oscar Hernandez (Oak Ridge National Lab)
Henri Calandra (Total E&P R&T)

GTC 2016

OAK RIDGE National Laboratory

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
TOTAL and Oak Ridge National Laboratory collaboration is done under the CRADA agreement NFE-14-05227

ONE-WAY IMAGING

- Classic depth imaging application
- Uses Fourier Finite Differencing
- Wave equation

$$\frac{\partial P(x,y,z,\omega)}{\partial z} = ik_z P(x,y,z,\omega)$$

- Approximation contains 3 terms

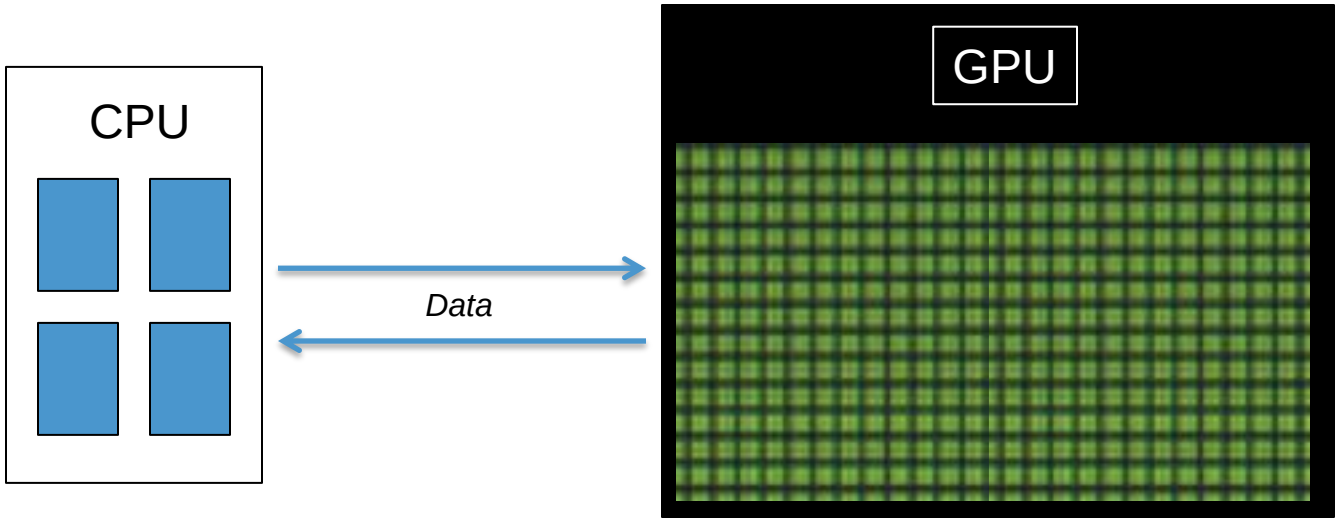
$$k_{z_n} = \sqrt{(\frac{\omega}{v_n})^2 + \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}} + (\frac{\omega}{v_n} - \frac{\omega}{c}) + (\frac{\frac{v^2}{w^2}\frac{\partial^2}{\partial x^2}}{a_1 + b_1 \frac{\omega^2}{v^2}\frac{\partial^2}{\partial x^2}} + \ldots)$$

Phase shift Lens correction Wide angle correction

- Iterative method where we compute the wavefield at every depth *z*
- Wavefield approximation takes 75-80% total time on a single shot

PARALLELIZING ONE-WAY MIGRATION USING OPENACC

1. Optimizing data transfer between CPU and GPU



- Copy wavefield and other data to GPU before we begin migration
- Only copy image to host for writing to file
- Copy slice of velocity model to GPU in every iteration if required

PARALLELIZING ONE-WAY MIGRATION USING OPENACC

2. Computation on the GPU

- Smaller components such as adding signal to wavefield, applying damping etc. are simple and straight-forward

- Parallelizing Phase-Shift and Wide-Angle require more work

WIDE-ANGLE ALGORITHM

```
for each row
    for each frequency
        wavefield(:) —> wave

        create_sparse_matrix()

        tridiagonal_solver()

        rhs —> wavefield
    enddo
enddo
```

WIDE-ANGLE ALGORITHM

parallelizable
for each row
    parallelizable
    for each frequency

        parallelizable
        wavefield(:) —> wave

        parallelizable
        create_sparse_matrix()

        sequential
        tridiagonal_solver()

        parallelizable
        rhs —> wavefield
    enddo
enddo

WIDE-ANGLE OPENACC I

```
!$acc loop collapse(2)
for each row
      for each frequency

            !$acc loop vector
            wavefield(:) —> wave

            !$acc loop vector
            create_sparse_matrix()

            tridiagonal_solver()

            !$acc loop vector
            rhs —> wavefield
      enddo
enddo
```

- Parallelize outer loops as gang
- Parallelize inner loops as vector
- Performance is very poor
- Reason: Solver is executed by a single thread

WIDE-ANGLE OPENACC II

```
!$acc loop collapse(2) gang vector
for each row
      for each frequency

            wavefield(:) —> wave

            create_sparse_matrix()

            tridiagonal_solver()

            rhs —> wavefield
      enddo
enddo
```

- Parallelize outermost loops as gang vector
- Inner loops are sequential
- Much better than previous version
- Solver code run by multiple threads

- Still not as good as CPU 8-cores
- Primary reason: non-coalesced memory access
- VERY expensive on GPUs

WIDE-ANGLE OPENACC II

wavefield (nx,ny,nw,n_wave) → wavefield_2 (nw,nx,ny,n_wave)

!$acc loop
for each row
    !$acc loop
    wavefield2(1:nw,) —> wave(1:nw,)

    !$acc routine (inside)
    create_sparse_matrix(1:nw,)

    !$acc routine (inside)
    tridiagonal_solver(1:nw,)

    !$acc loop
    rhs(1:nw,) —> wavefield2(1:nw,)
enddo

wavefield2 → wavefield

- Create temporary wavefield in wide angle where innermost dimension is *w* (frequencies) and vectorize along *w*
- This way we have coalesced memory access
- Work on this wavefield and copy it back to the original wavefield at the end of the subroutine
- All local arrays must have *w* as the inner dimension

- Requires acceptable amount of code change
- This is how directive-based programming models are expected to work

WIDE ANGLE OPENACC III CONTD.

- *Pros:*
- Can control the no. of gangs if you run out of memory

- *Cons:*
- Inner dimension must be large enough

```
!$acc parallel num_gangs(100)
do ix=1,nx
 …
enddo
```

WIDE ANGLE OPENACC IV (BATCHED)

```
!$acc parallel
for each row
  for each frequency
    wavefield —> wave

!$acc parallel
for each row
  for each frequency
    create_sparse_matrix()

!$acc parallel
for each row
  for each frequency
    tridiagonal_solver()

!$acc parallel
for each row
  for each frequency
    rhs —> wavefield
```

- Batching or grouping of operations
- Break up a loop into many loops performing similar operations
- Instead of privatizing local arrays, pad arrays with additional dimensions
- Create a large system of sparse matrices, solving a large system of solvers etc.
- Requires significant code changes
- On the original wavefield, performance only as good as 8-core CPU
- Again due to non-coalesced memory access

WIDE ANGLE OPENACC V

wavefield (nx,ny,nw,n_wave) →
wavefield_2 (nw,nx,ny,n_wave)

```
!$acc parallel
for each row
  for each frequency
    wavefield_2 —> wave

!$acc parallel
for each row
  for each frequency
    create_sparse_matrix()

!$acc parallel
for each row
  for each frequency
    tridiagonal_solver()

!$acc parallel
for each row
  for each frequency
    rhs —> wavefield_2
```

wavefield2 → wavefield

- Create temporary wavefield which has *w* as the inner dimension, then use batched operations
- Optimize usage of local arrays
- Use solution of X direction as input to Y direction
- Don't copy output of X direction back to wavefield
- Now memory access is coalesced
- Leads to 2.47x performance improvement over 8-core CPU case on Titan K20 with PGI 15.7 and CUDA 7.0
- 2.62x with PGI 15.9

WIDE ANGLE: FURTHER OPTIMIZATIONS

- Use CUDA code for transpose and some data copy operations in wide-angle
- Leads to ~3x performance improvement over 8-core CPU case
- We lose portability here due to use of CUDA

PARALLELIZING PHASE-SHIFT

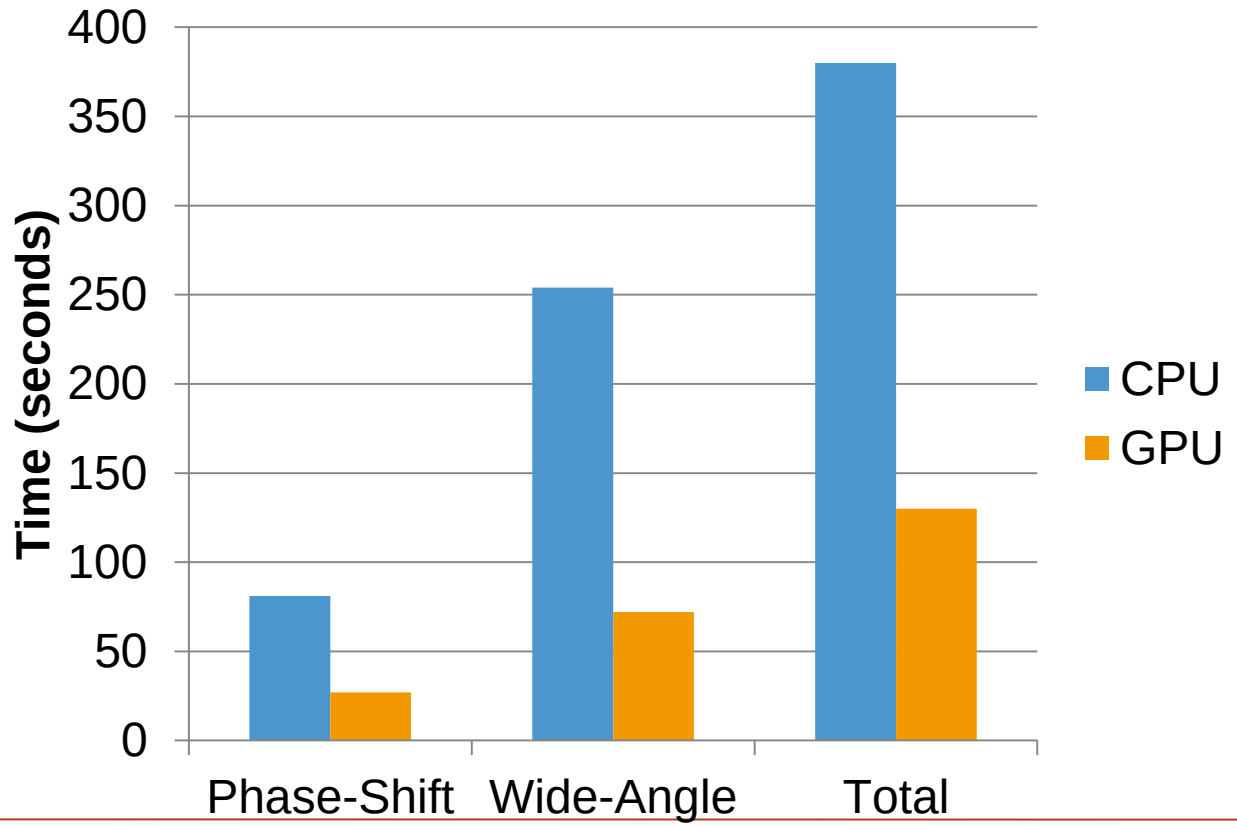
- Phase-Shift:
  1. 2D FFT Forward
  2. Phase-Shift computation
  3. 2D FFT Backward
  4. Thin lens correction

- In OpenACC, operations such as FFT require using optimized vendor libraries
  - NVIDIA's CuFFT

- Modified code to group operations so that we perform as much work as possible in the FFT call (batched FFT operations)

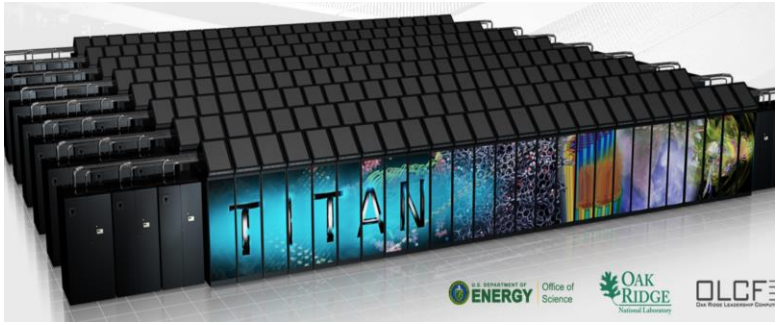
BENCHMARK PERFORMANCE

- 1 shot of the SEGSALT dataset
- K20X GPU on Titan vs. 8-core Intel Sandybridge CPU
- 3x speedup



ONE-WAY MIGRATION AT SCALE

- Titan supercomputer at Oak Ridge National Lab
  - 2nd in list of top 500 supercomputers as of March, 2016
  - 18,688 nodes, each with an NVIDIA K20X GPU
  - Lustre file system
  - PGI 15.10.0 compiler
  - CUDA 7.0



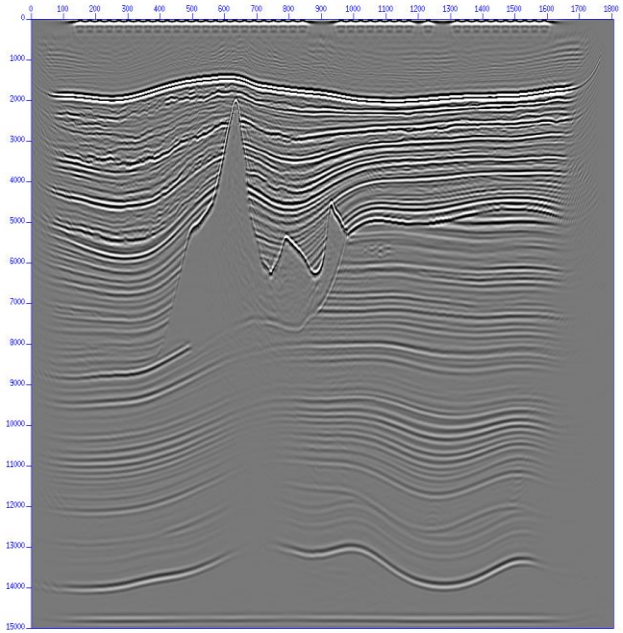
- K20X GPU
  - 6GB memory
  - 14 SMX, 192 single-precision FP cores in each (2688 total)

LARGE RUN CONFIGURATION

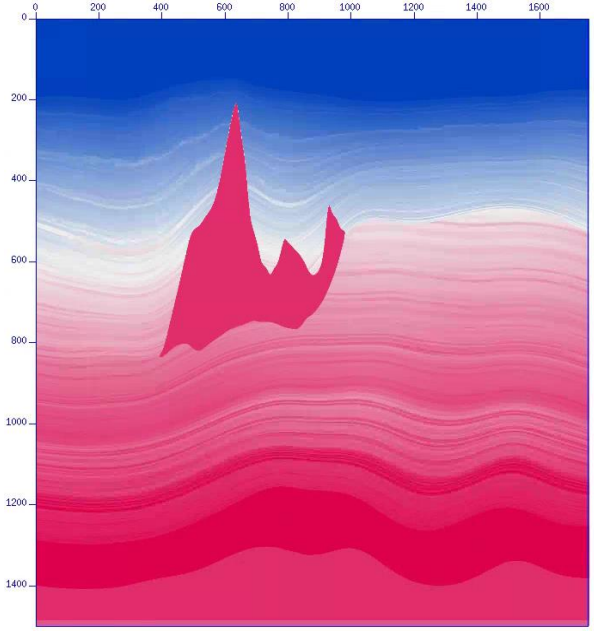
- Dataset:
  - SEAM Isotropic Phase I
  - 2793 shots
- Used 99% of nodes on Titan (18,508/18,688 nodes)
- 28 GPUs per shot (considering memory requirement and load balancing within group)
- 661 process groups (= shots) running simultaneously
- Flat MPI mode: shot distribution to process groups is static

RESULTS

- Large run took 54 minutes to complete

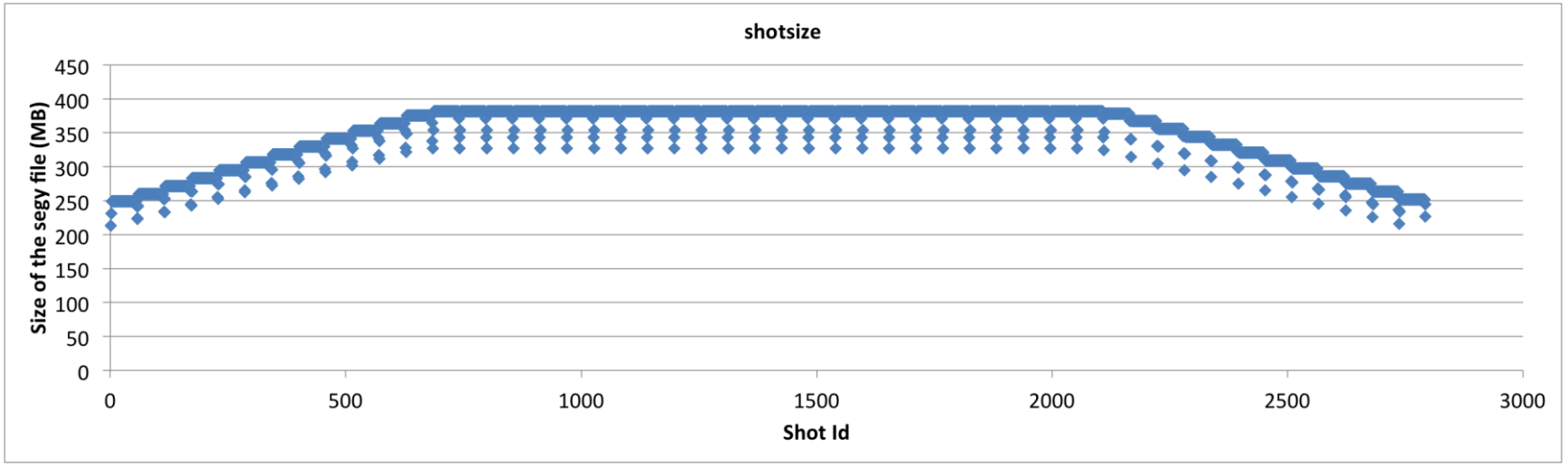


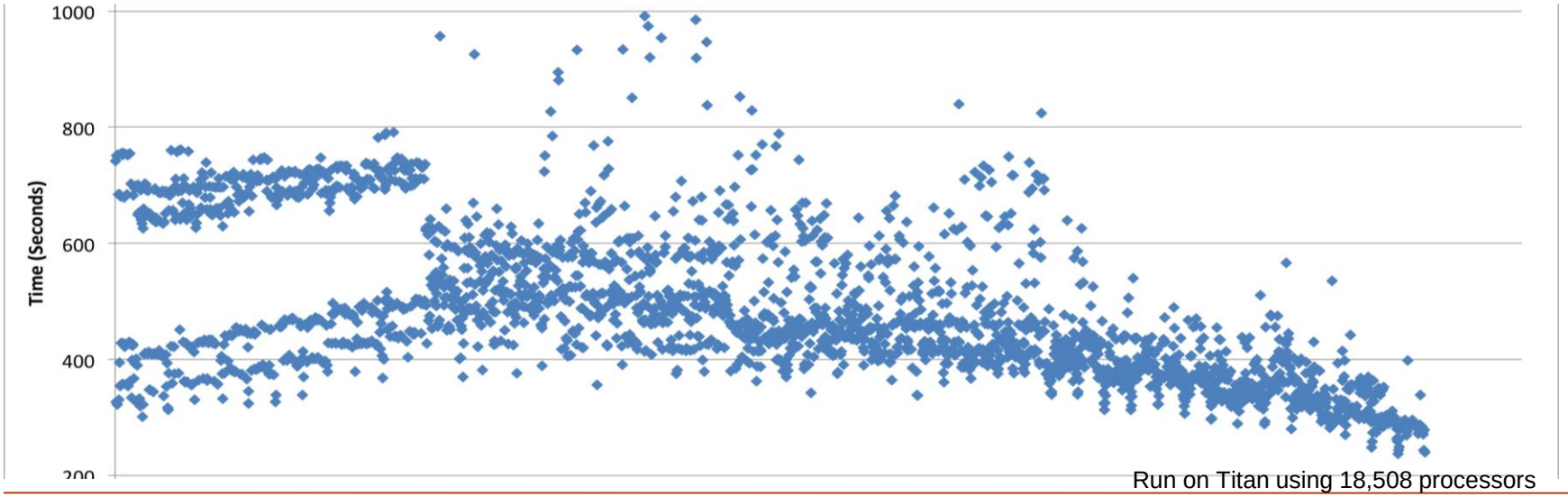
Partial stacked images



Velocity model

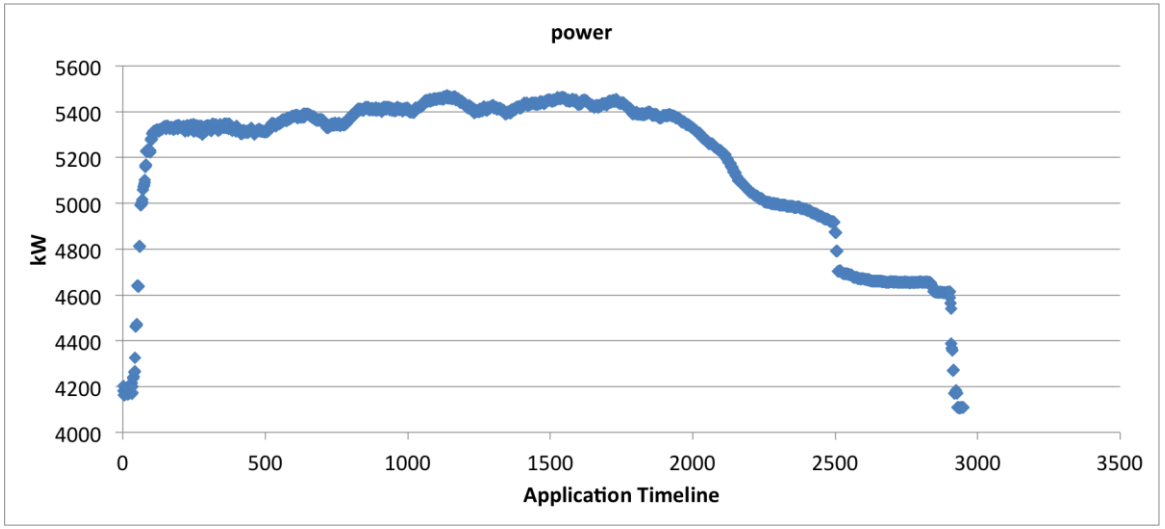
SHOT SIZE AND SHOT TIMES

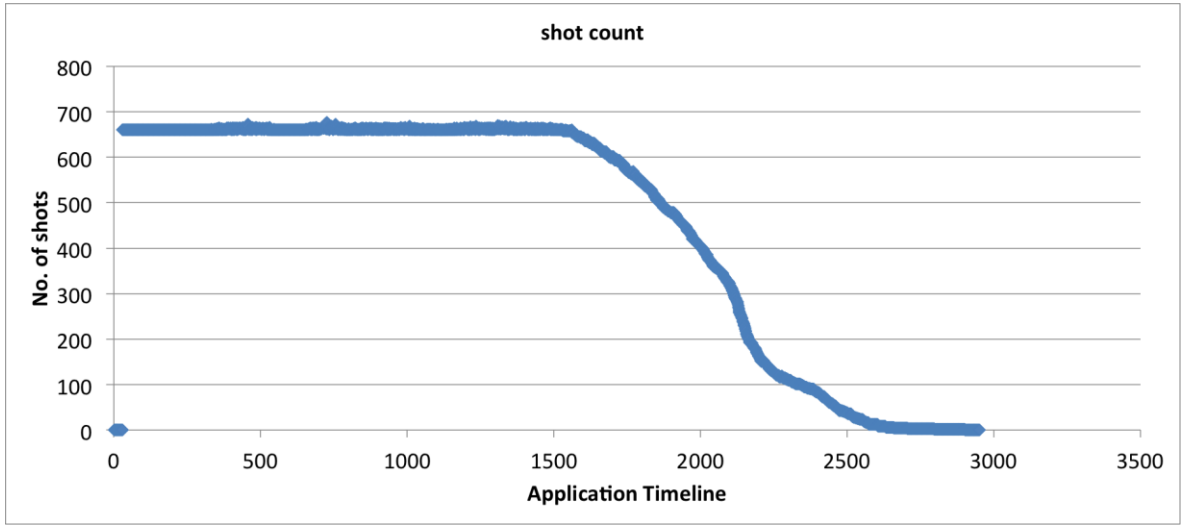




Run on Titan using 18,508 processors

POWER MEASUREMENT ON TITAN





Run on Titan using 18,508 processors

SUMMARY

- We have ported One-Way Migration to GPUs using OpenACC
- Porting to GPU requires some code modifications, but directive based model is highly preferred
- 3x speedup on benchmark dataset

- Ran One-Way Migration at large scale on Titan supercomputer
- Processed 2793 shots in less than an hour
- Running at scale yields interesting points of discussion

- Future Work:
  - How to scale I/O
  - Run more complicated applications such as RTM