Large scale linearised inversion with multiple GPUs

Chris Leader* and Robert Clapp

SEP149 - 333

Wednesday June 19th

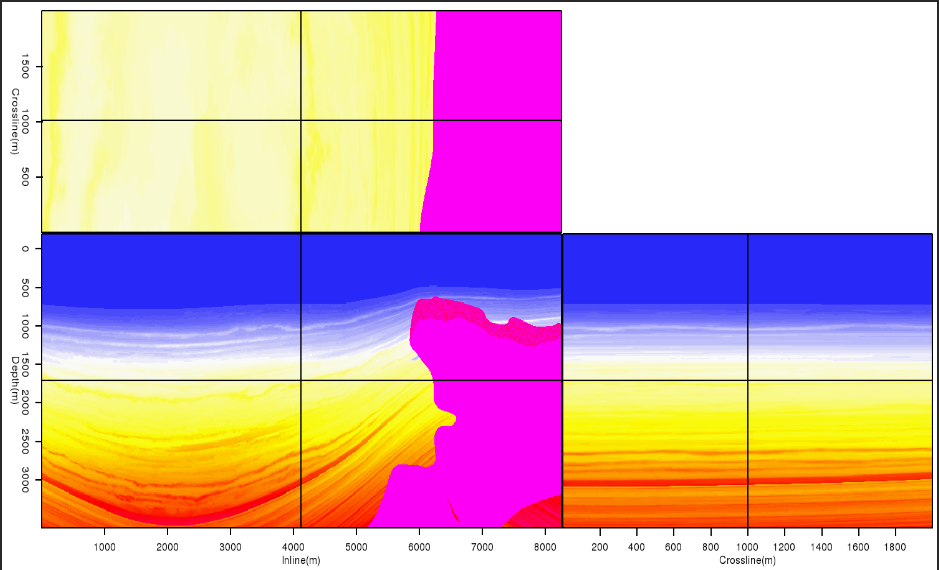
Research goals

Accelerate the imaging of seismic data through inverse methods

Create a solution which:

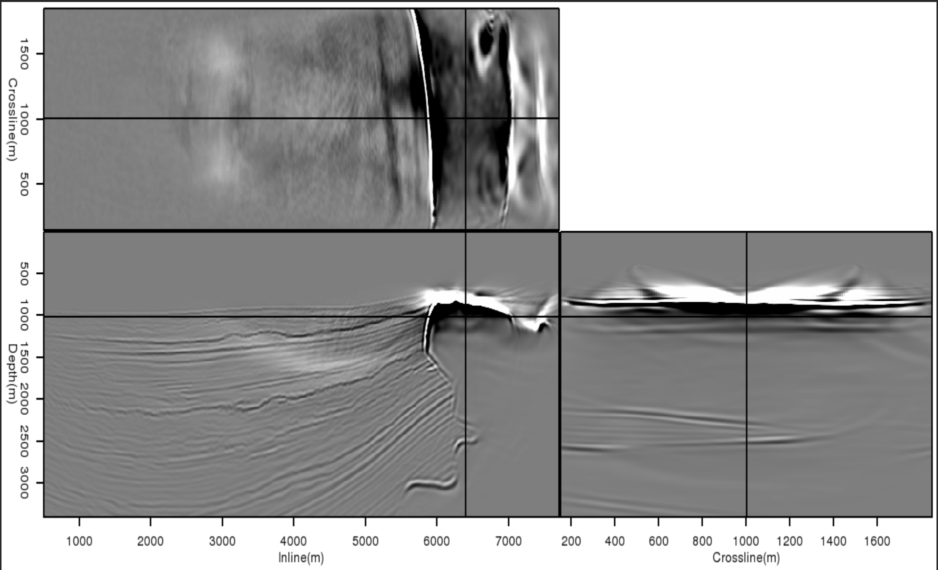
- Produces high fidelity seismic images
- Is not limited by the global memory of a single GPU
- Scales (close to) linearly with model/problem size

Earth model

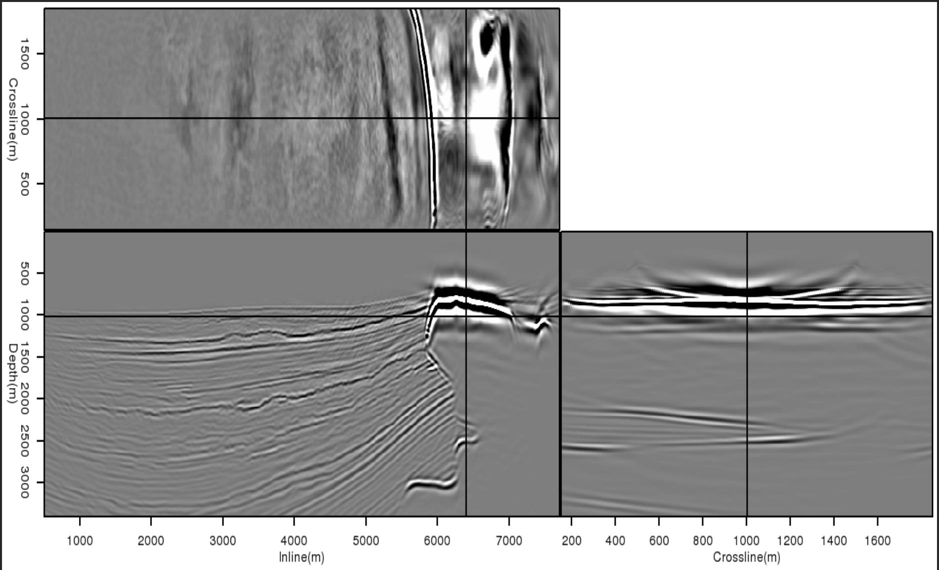


1500
1000
500
Crossline(m)
0
500
1000
1500
2000
2500
3000
Depth(m)
1000
2000
3000
4000
5000
6000
7000
8000
Inline(m)
200
400
600
800
1000
1200
1400
1600
1800
Crossline(m)

RTM image (adjoint approach)



Filtered RTM image



After 10 iterations (20x RTM cost)

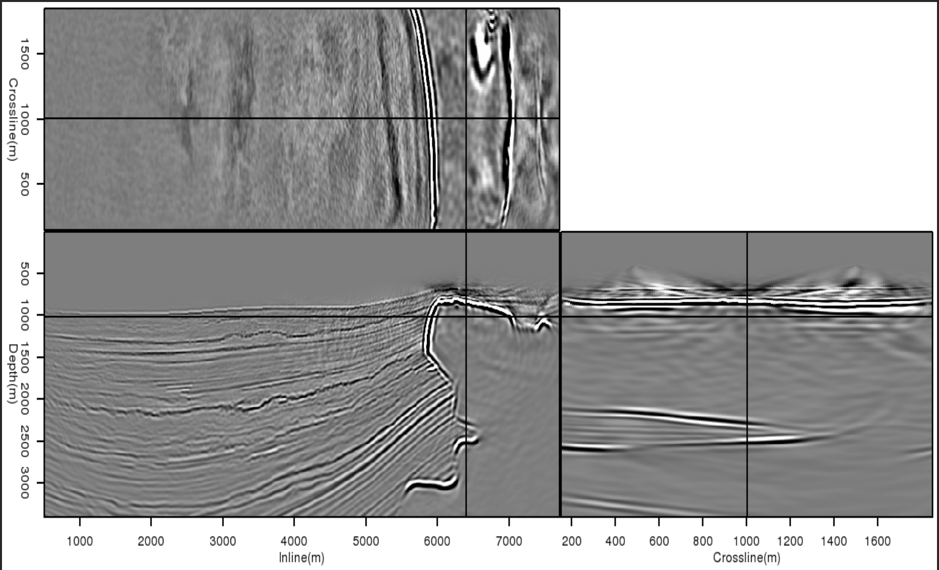


Table of contents

Adjoint imaging

For imaging, we are trying to solve:

- Computational bottleneck
- I/O bottleneck

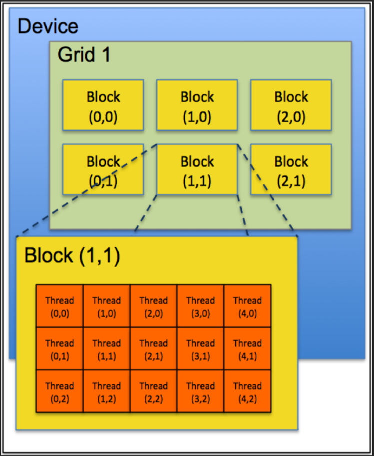
We can do this by:

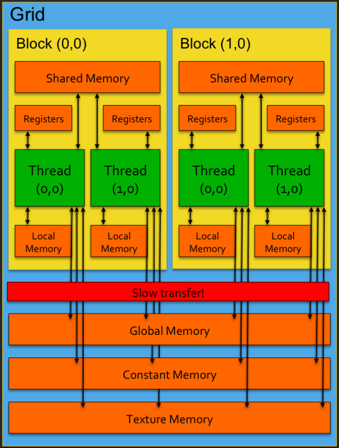
- Using optimised GPU wave propagation kernels
- Using random boundaries to remove I/O from the RTM loop

Memory heirarchy - multi-core CPU

- Cores share L3 and main memory
- No explicit control over which memory is used

Memory heirarchy - GPU





Conventional imaging algorithm

Forward model the source wavefield

- Save this to disk (z, x, y, t)

Back propagate recorded data

- Read the relevant source wavefield snapshot
- Multiply source and receiver wavefields
- Sum result to image estimate

Conventional imaging algorithm

Forward model the source wavefield

- Save this to disk

Computational bottleneck

Back propagate recorded data

- Read the relevant source wavefield snapshot
- Multiply source and receiver wavefields
- Sum result to image estimate

Conventional imaging algorithm

Forward model the source wavefield

- Save this to disk

Computational bottleneck

Back propagate recorded data

- Read the relevant source wavefield snapshot
- Multiply source and receiver wavefields
- Sum result to image estimate

IO bottleneck

GPU wave propagation

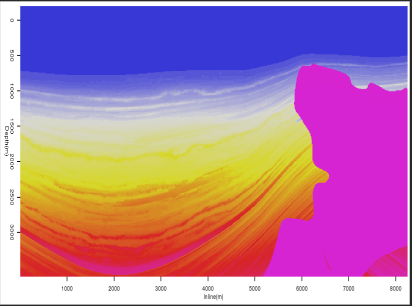
Follow Micikevicius, 2009

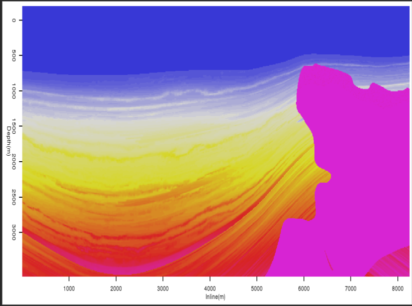
- Minimise global memory read redundancy
- Break wavefield into blocks, store in shared memory

Use texture memory for velocity array

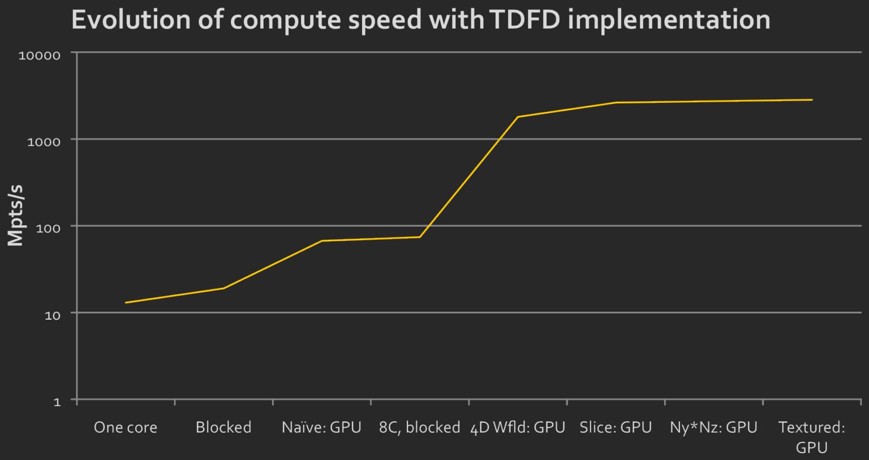
- Cached (useful for adjoint propagation)
- Normalised indexing option
- Out of boundary clamping $\Longrightarrow$ reduce boundary allocation

CPU vs GPU


Depth(m)
0
500
1000
1500
2000
2500
3000
1000
2000
3000
4000
5000
6000
7000
8000
Inline(m)


Depth(m)
0
500
1000
1500
2000
2500
3000
1000
2000
3000
4000
5000
6000
7000
8000
Inline(m)

GPU implementation



Conventional algorithm

Forward model the source wavefield

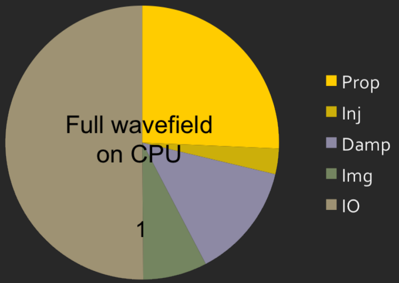
- Save this to disk (z, x, y, t)

Back propagate recorded data

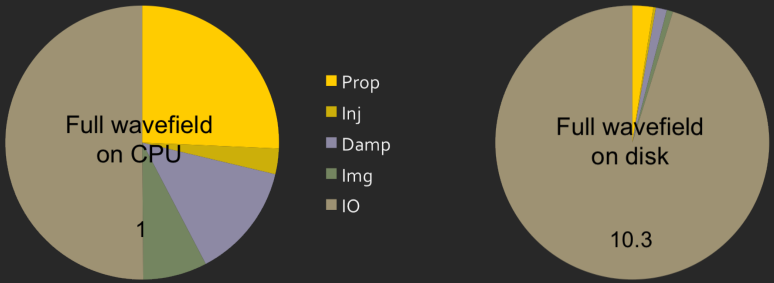
- At imaging time step?
  - Read the relevant source wavefield snapshot
  - Multiply source and receiver wavefields
  - Sum result to image estimate

IO bottleneck

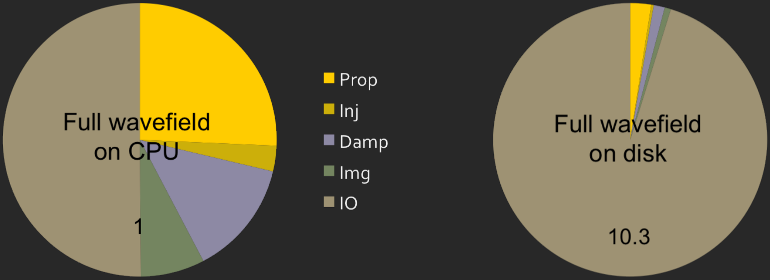
GPU performance



GPU performance



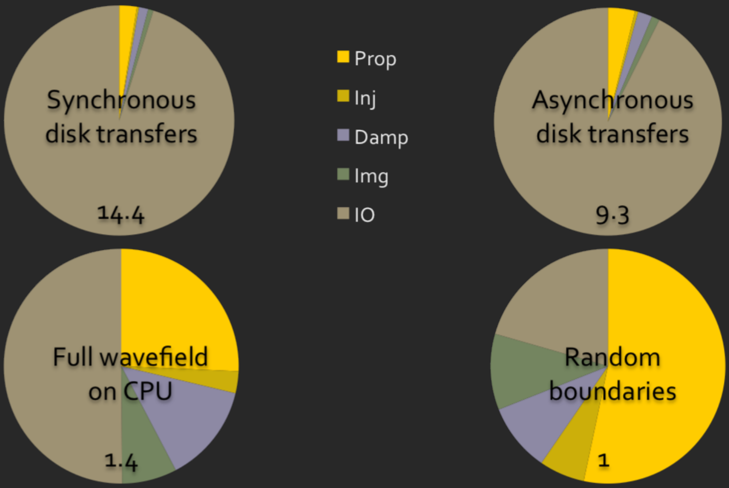
GPU performance



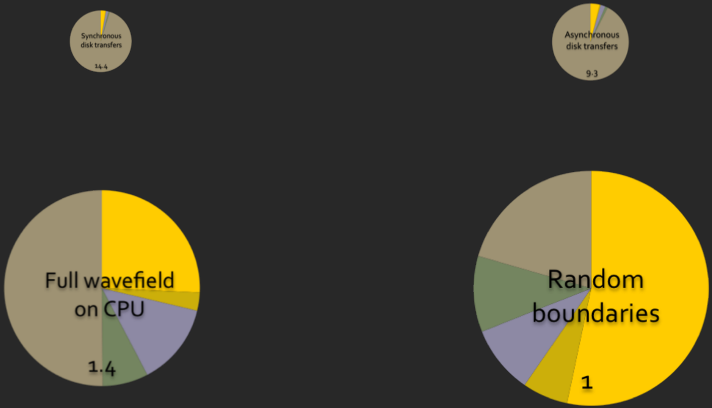
PCIe: ~ 2 Gb/s

Disk: ~ 200 Mb/s

IO and computation balancing



IO and computation balancing



Memory considerations

Fermi global memory: 6 GBytes

RTM objects that must be allocated:

- Four 3D wavefield snapshots
- Recorded data (one shot)
- Velocity model
- Image

If our domain is larger than 600^3:

- Decompose our propagation across multiple GPUs

Table of contents

Linearised inversion

We can extend RTM to linearised inversion

- Construct a forward modelling process
- Ensure RTM and forward are fully adjoint
- Use a conjugate direction solver for updates

The forward process

First order approximation to the Born scattering series

Adjoint process:

$$m(\mathbf{x}) = \sum_{\mathbf{x}_s,\omega} f(\omega) G_0(\mathbf{x},\mathbf{x}_s,\omega) \sum_{\mathbf{x}_r} G_0(\mathbf{x},\mathbf{x}_r,\omega) d^*(\mathbf{x}_r,\mathbf{x}_s,\omega)$$

Forward process:

$$d(\mathbf{x}_r,\mathbf{x}_s,\omega) = \sum_{\mathbf{x}} f(\omega) G_0(\mathbf{x},\mathbf{x}_s,\omega) m(\mathbf{x}) G_0(\mathbf{x},\mathbf{x}_r,\omega)$$

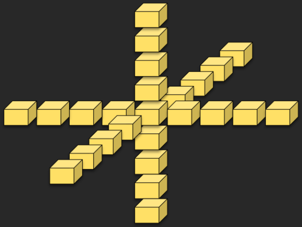
Both wavefields have the same sense of time

Adjoint propagation

We need an adjoint to our propagator

We now require as much velocity information as wavefield information

- Read from:
  - Global memory array
  - Textured velocity array
  - Copy values to shared memory



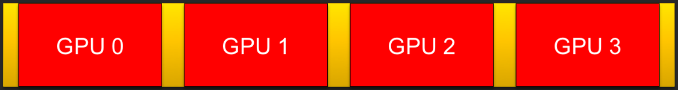
Get around a 2x speed up by using shared memory

Table of contents

Domain decomposition

In 1D:

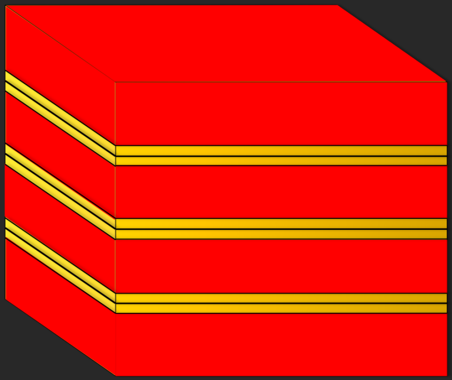
- Each block has to overlap



In 3D, break domain along slowest axis

More allocation, but easier communication (transfer regions contiguous in memory)

Visualising 3D decomposition



CUDA 4.0 (and later)

CUDA 4.0 and Fermi architectures have made several things easier / possible

- Peer to Peer (P2P) GPU communication
  - Direct GPU to GPU information transfer
- CPU and GPU use a Unified Virtual Address space (UVA)
  - Pointers can be dereferenced across host and devices

Multi-GPU programming

Main points:

- Faster/more convenient device-to-device transfer
- PCIe links are duplex
  - Send/receive can be done simultaneously
- Communication can be hidden by overlapping with computation

Overlapping operations

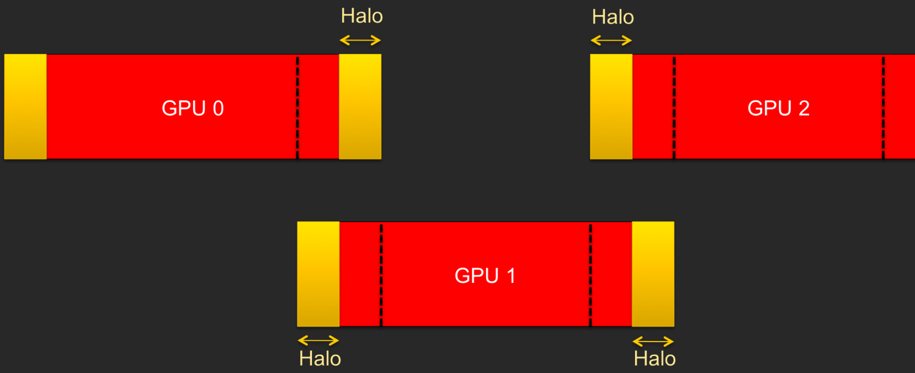
Kernels and asynchronous memcopies can be assigned to streams

- Can be considered as a command pipeline
- Kernels are queued
- Async memcopies can overlap with kernels

Successful overlap $\implies$ linear scaling

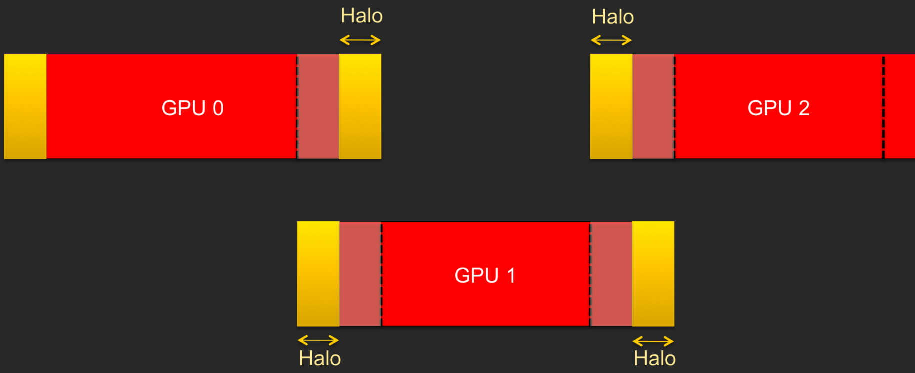
Visualising halo exchange

Computation order:



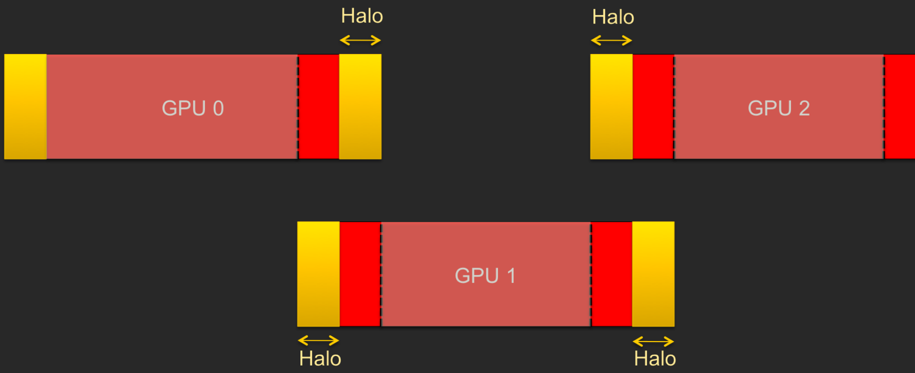
Visualising halo exchange

Calculate halo region, set to halo_stream[i]



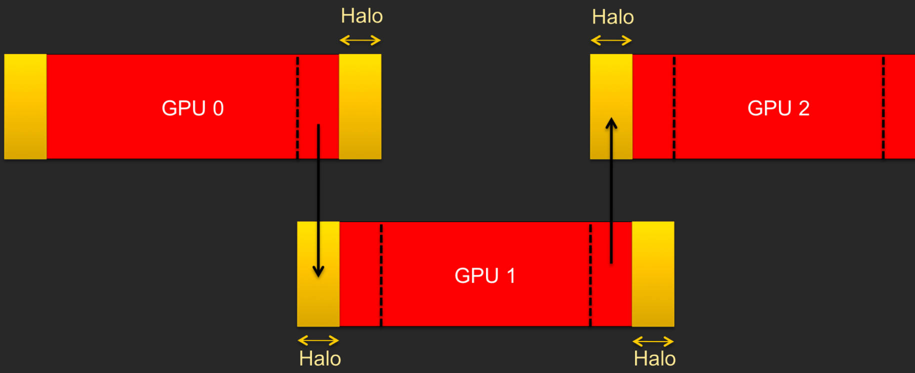
Visualising halo exchange

Calculate internal region, set to internal_stream[i]



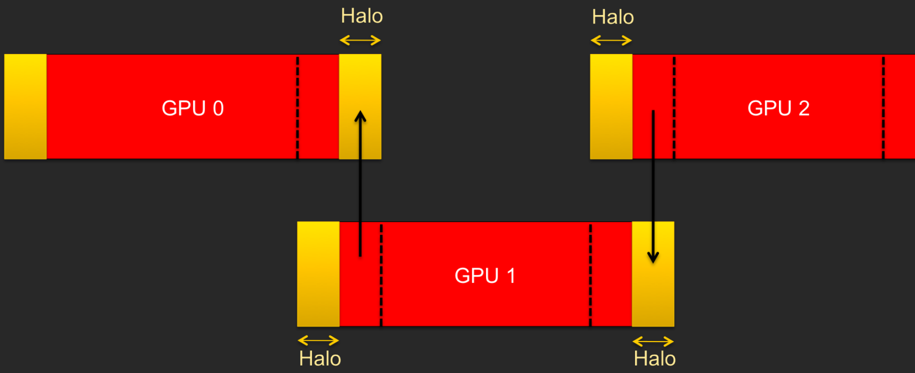
Visualising halo exchange

During internal computation, send halo to the right

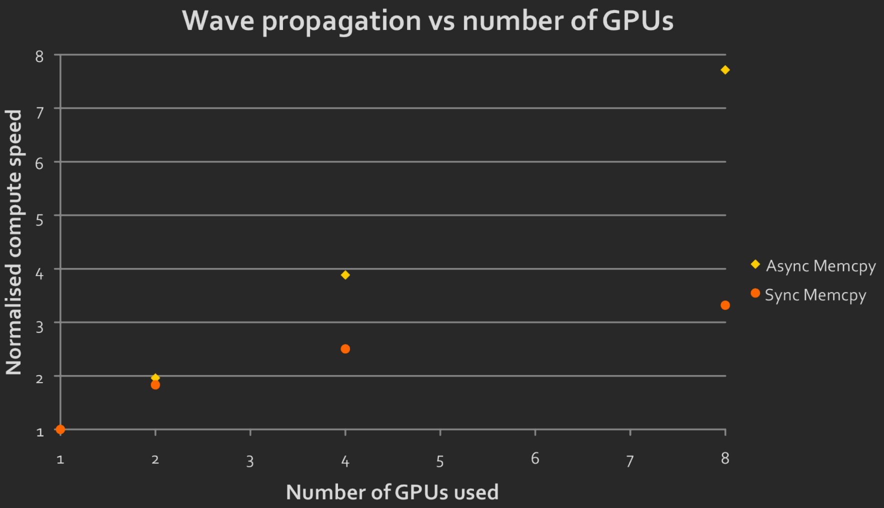


Visualising halo exchange

Then, send to the left



Do we overlap?



Do we overlap?

Even for TTI, we completely overlap communication (Micikevicius, 2012)

We get close to linear speed up, but not quite

- Splitting the computation requires some small overhead
- Get up to 96% linear speed up

How does this extend to inversion?

Forward linearised modelling

During each time step:

- Adjoint propagate data wavefield
- Propagate source wavefield
- Inject source
- Convolve source snapshot and image, sum to data snapshot
- Extract data at receiver positions

Forward linearised modelling

- Calculate data wavefield halos
- Calculate source wavefield halos
- Adjoint propagate data wavefield
- Propagate source wavefield
  - Transfer data wavefield halos
  - Transfer source wavefield halos
- Inject source
- Convolve source snapshot and image, sum to data snapshot
- Extract data at receiver positions

Forward linearised modelling

- Calculate data wavefield halos
- Calculate source wavefield halos
- Adjoint propagate data wavefield
- Propagate source wavefield
  - Transfer data wavefield halos
  - Transfer source wavefield halos
- Inject source
- Convolve source snapshot and image, sum to data snapshot
- Extract data at receiver positions

Adjoint linearised modelling

During each time step:

- Propagate data wavefield
- Propagate source wavefield
- Inject source
- Inject data
- Convolve source snapshot and data snapshot, sum to image

Adjoint linearised modelling

- Calculate data wavefield halos
- Calculate source wavefield halos
- Propagate data wavefield
- Propagate source wavefield
  - Transfer data wavefield halos
  - Transfer source wavefield halos
- Inject source
- Inject data
- Convolve source snapshot and data snapshot, sum to image

Adjoint linearised modelling

- Calculate data wavefield halos
- Calculate source wavefield halos
- Propagate data wavefield
- Propagate source wavefield
    - Transfer data wavefield halos
    - Transfer source wavefield halos
- Inject source
- Inject data
- Convolve source snapshot and data snapshot, sum to image

GPU implementation

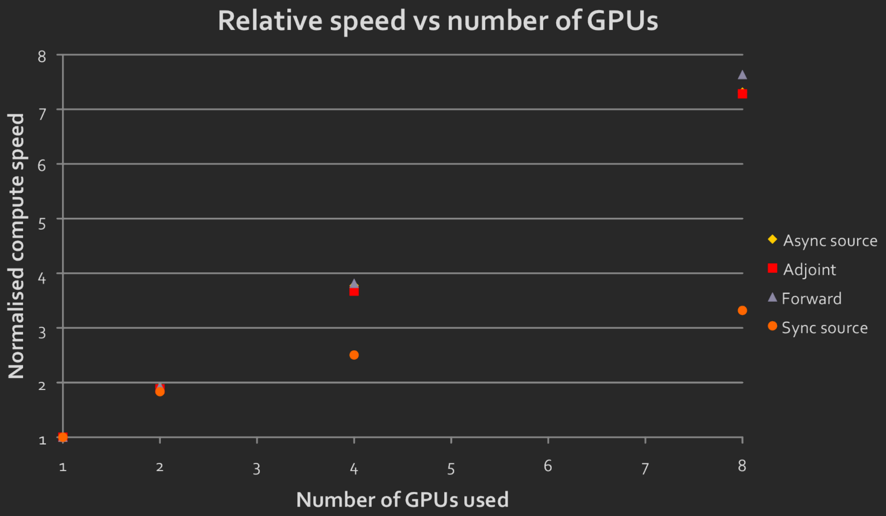


Table of contents

Conclusions

Extending GPU RTM to linearised inversion is fairly straightforward

- Use adjoint propagation

Once our domain exceeds 600^3, we must move to domain decomposition

- We can overlap internal computation with halo communication
- Close to linear speed up achieved for each stage of the inverse process

Acknowledgments

Robert Clapp (SEP) - continuous coding assistance

Paulius Micikevicius (NVIDIA) - GPU troubleshooting, code sharing and discussions

All SEP sponsors - continued financial, intellectual and moral support

References

Micikevicius, P., 2009, 3D finite difference computation on GPUs using CUDA: GPGPU, 2.

Micikevicius, P., 2012, Programming multiple GPUs: GPU Technology Conference, 2012.