Linearised inversion with GPUs

Chris Leader* and Robert Clapp

SEP147 - p139

Tuesday May 22nd

Presentation goals

Last year:

- Reverse Time Migration on GPUs
    - Random boundaries to remove I/O
    - Single card solution

Today:

- Brief recap
- Extending to linearised inversion
- Extending to mutli-GPU solutions

Table of contents

Last year

We discussed approaches to GPU based Reverse Time Migration

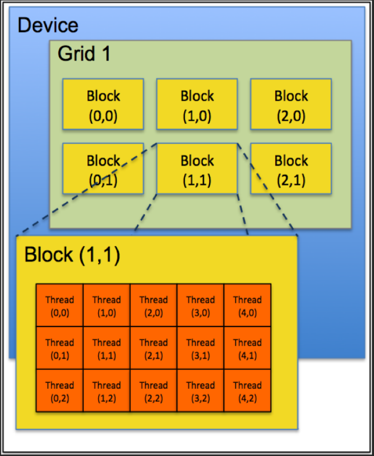
In particular, trying to solve:

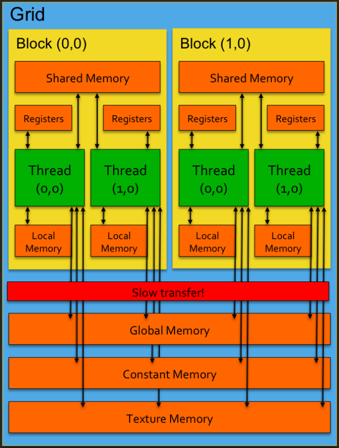
- Computational bottleneck
- I/O bottleneck

We did this by:

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

Memory heirarchy - GPU





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

Conventional algorithm

Forward model the source wavefield

- Save this to disk

Back propagate recorded data

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

Computational bottleneck

Conventional algorithm

Forward model the source wavefield

- Save this to disk

Back propagate recorded data

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

Computational bottleneck

IO bottleneck

GPU wave propagation

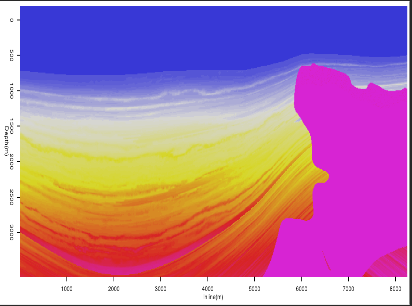
Follow Micikevicius, 2009

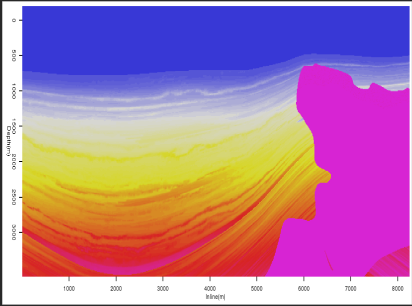
- Minimise global memory read redundancy
- Break wavefield into blocks, store in shared memory

Use texture memory for velocity array

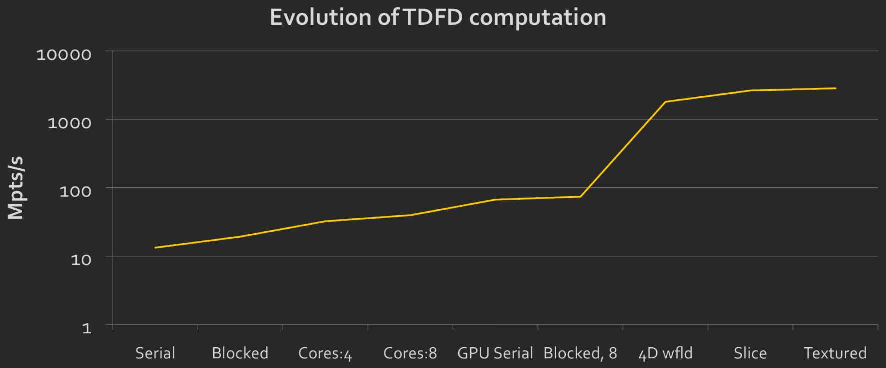
- Cached (useful for adjoint propagation)
- Normalised indexing option
- Out of boundary clamping $\Longrightarrow$ reduce boundary allocation

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

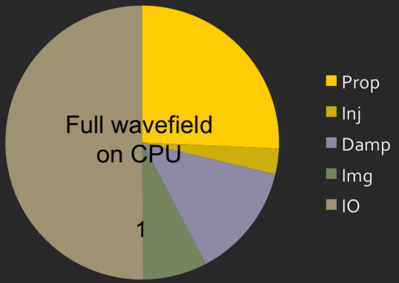
- Save this to disk (z, x, y, t)

Back propagate recorded data

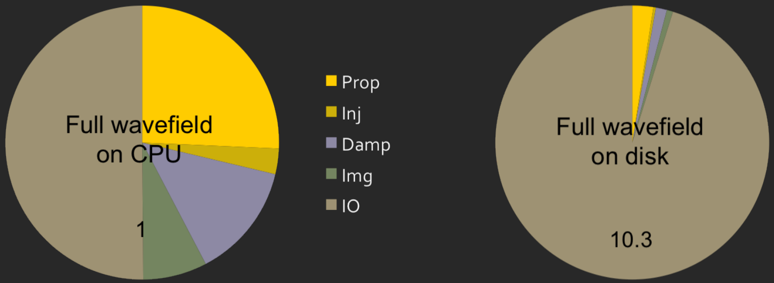
- At imaging time step?
  - Read the relevant source wavefield snapshot
  - Multiply source and receiver wavefields
  - Sum result to image estimate

IO bottleneck

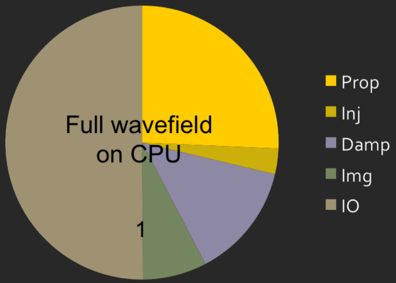
GPU performance



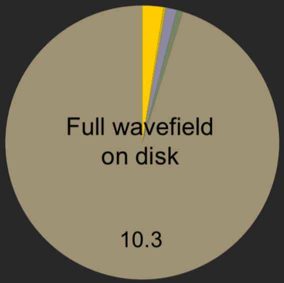
GPU performance



GPU performance

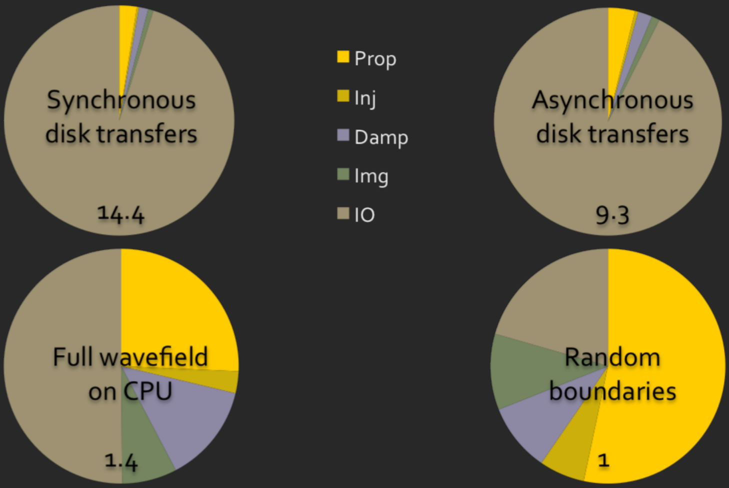


PCIe: ~ 2 Gb/s

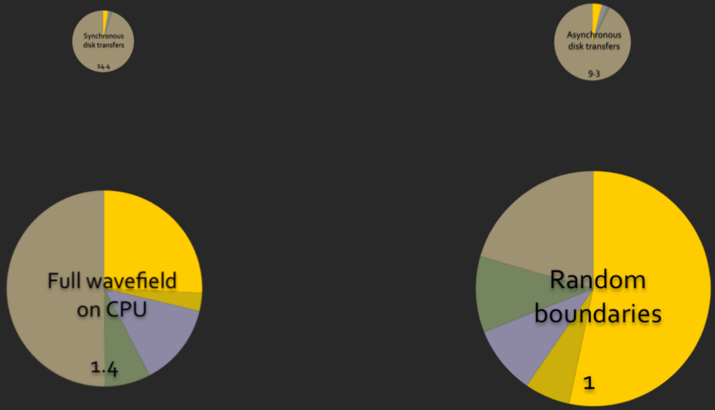


Disk: ~ 200 Mb/s

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

Recap summary

We can accelerate wave propagation by at least an order of magnitude

We can remove I/O during the RTM main loop by using random boundaries

- Stacking more than 50 shots $\implies$ no artifacts

We have to remain very aware of memory limitations

- Especially for more complicated propagation

Table of contents

Domain decomposition

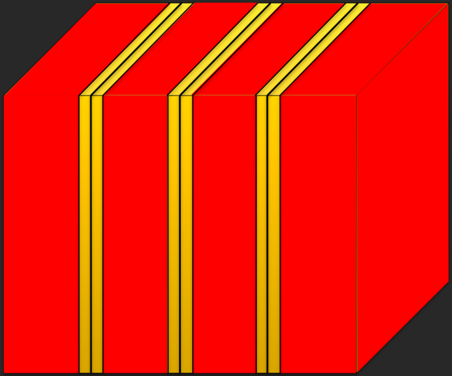
In 1D:

- Each block has to overlap

In 3D, break domain along slowest axis

More allocation, but easier communication

GPU Implementation



CUDA 4.0

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

- Peer to Peer (P2P) GPU communication
- CPU and GPU use a Unified Virtual Address space (UVA)
- The GPU can derefence a pointer:
    - On itself
    - On another GPU
    - On the host

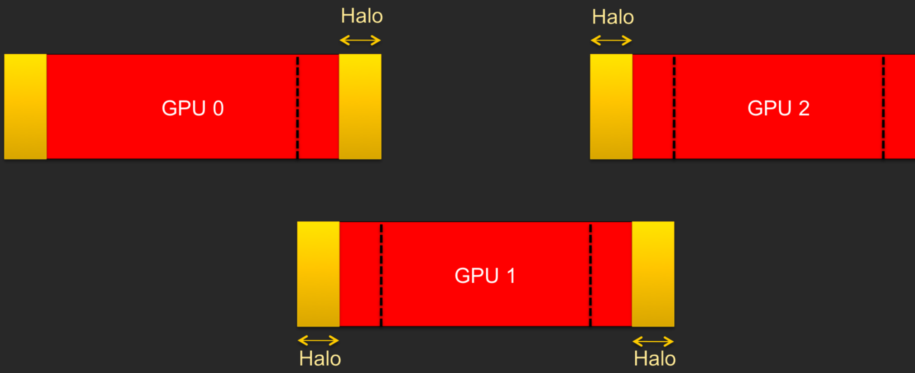
Multi-GPU programming

Main points:

- Faster/more convenient device-to-device transfer
  - Transferred along shortest PCIe path
  - Copies can overlap
- PCIe links are duplex
  - Send/receive can be done simultaneously
  - ...providing paths are in opposite directions
- Communication can be hidden by overlapping with kernels

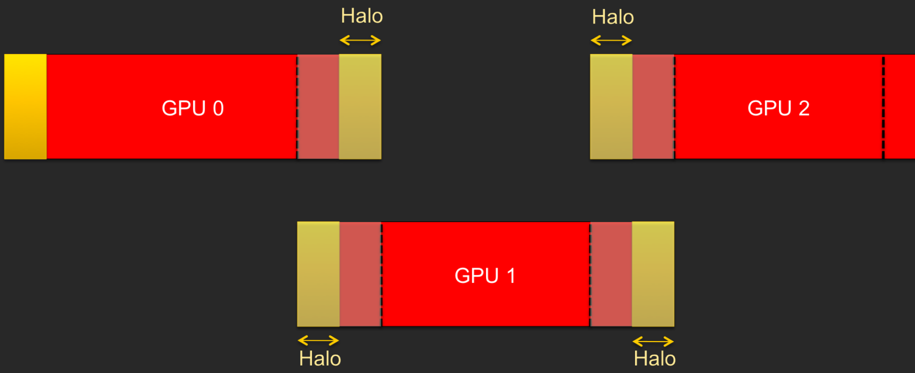
Visualising halo exchange

Computation order:



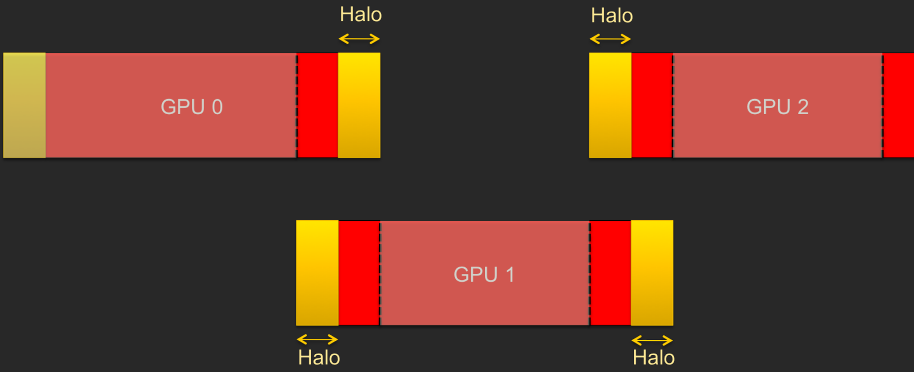
Visualising halo exchange

Calculate halo region, set to halo_stream[i]



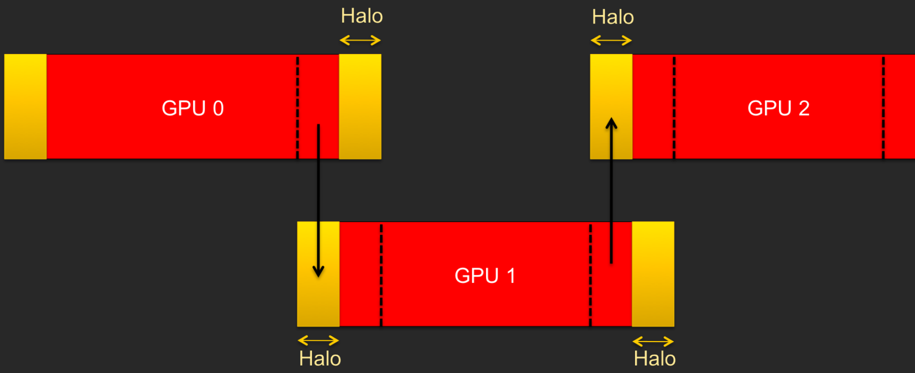
Visualising halo exchange

Calculate internal region, set to internal_stream[i]



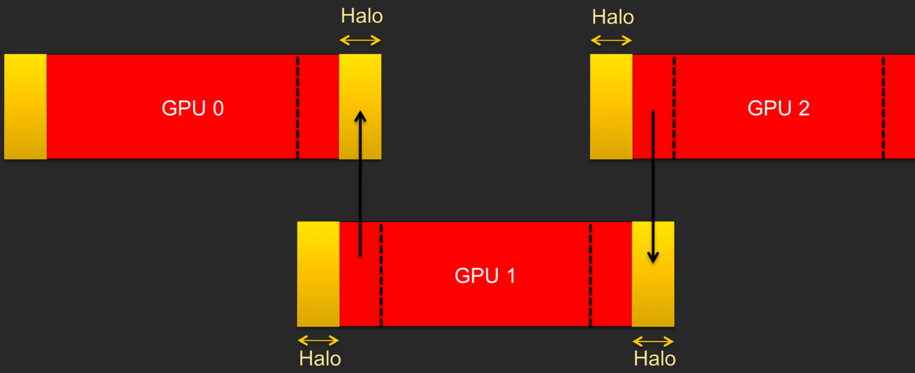
Visualising halo exchange

During internal computation, send halo to the right



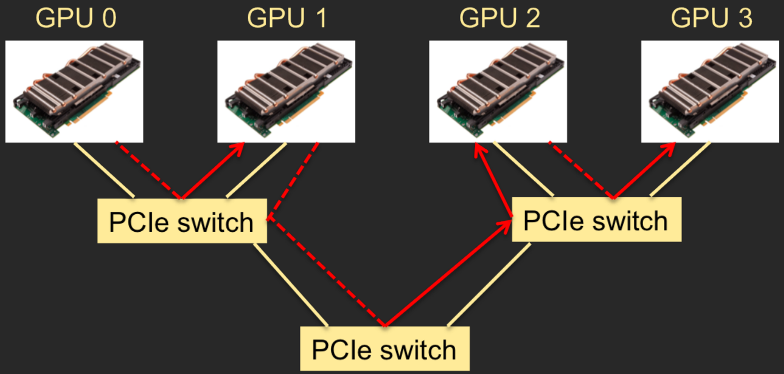
Visualising halo exchange

Then, send to the left



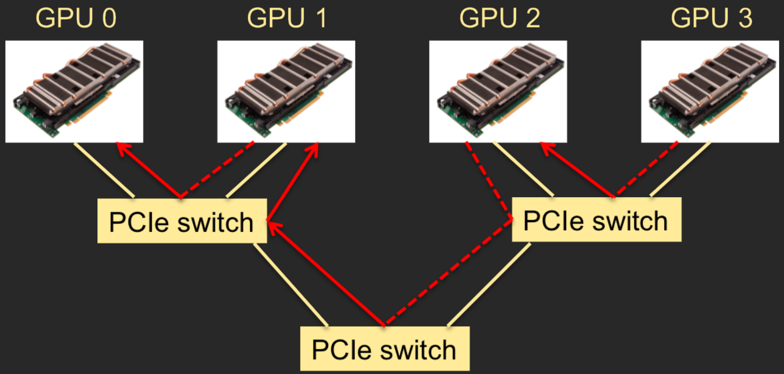
Visualising halo exchange

Send halo to the right, receive from the left



Visualising halo exchange

Send halo to the left, receive from the right



Pseudo-code

Loop through time

- Loop through GPUs
  - kernel(...,halo_stream[gpu_id]);
  - kernel(...,internal_stream[gpu_id]);
- Loop through GPUs
  - cudaMemcpyPeerAsync(...,halo_stream[gpu_id]);
- Loop through GPUs
  - cudaStreamSynchronize(halo_stream[gpu_id]);
- Loop through GPUs
  - cudaMemcpyPeerAsync(...,halo_stream[gpu_id]);
- Loop through GPUs
  - cudaDeviceSynchronize();

Pseudo-code

Loop through time

- Loop through GPUs
  - kernel(...,halo_stream[gpu_id]);
  - kernel(...,internal_stream[gpu_id]);
- Loop through GPUs
  - cudaMemcpyPeerAsync(...,halo_stream[gpu_id]);
- Loop through GPUs
  - cudaStreamSynchronize(halo_stream[gpu_id]);
- Loop through GPUs
  - cudaMemcpyPeerAsync(...,halo_stream[gpu_id]);
- Loop through GPUs
  - cudaDeviceSynchronize();

Pseudo-code

Loop through time

- Loop through GPUs
  - kernel(...,halo_stream[gpu_id]);
  - kernel(...,internal_stream[gpu_id]);
- Loop through GPUs
  - cudaMemcpyPeerAsync(...,halo_stream[gpu_id]);
- Loop through GPUs
  - cudaStreamSynchronize(halo_stream[gpu_id]);
- Loop through GPUs
  - cudaMemcpyPeerAsync(...,halo_stream[gpu_id]);
- Loop through GPUs
  - cudaDeviceSynchronize();

Pseudo-code

Loop through time

- Loop through GPUs
  - kernel(...,halo_stream[gpu_id]);
  - kernel(...,internal_stream[gpu_id]);
- Loop through GPUs
  - cudaMemcpyPeerAsync(...,halo_stream[gpu_id]);
- Loop through GPUs
  - cudaStreamSynchronize(halo_stream[gpu_id]);
- Loop through GPUs
  - cudaMemcpyPeerAsync(...,halo_stream[gpu_id]);
- Loop through GPUs
  - cudaDeviceSynchronize();

Pseudo-code

Loop through time

- Loop through GPUs
  - kernel(...,halo_stream[gpu_id]);
  - kernel(...,internal_stream[gpu_id]);
- Loop through GPUs
  - cudaMemcpyPeerAsync(...,halo_stream[gpu_id]);
- Loop through GPUs
  - cudaStreamSynchronize(halo_stream[gpu_id]);
- Loop through GPUs
  - cudaMemcpyPeerAsync(...,halo_stream[gpu_id]);
- Loop through GPUs
  - cudaDeviceSynchronize();

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 around 96% linear speed up

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:

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

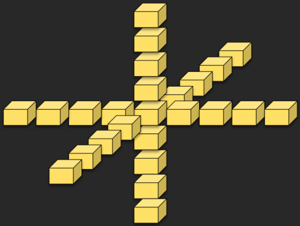
Both wavefields have the same sense of time

Adjoint propagation

We need an adjoint to our propagator

We now require velocity values along the length of our stencil

- 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

Conclusions

Extending GPU RTM to linearised inversion is fairly straightforward

- Allocate velocity for adjoint propagation in shared memory
- We can create an exact adjoint pair

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

- Asynchronous calls can overlap
- We can overlap internal computation with halo communication
- Close to linear speed up achieved

Conclusions

We can now perform large scale, GPU based linearised inversion

Acknowledgments

Robert Clapp - continuous coding assistance

Paulius Micikevicius - 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.