Linearised inversion with GPUs

Chris Leader and Robert Clapp

ABSTRACT

Graphical Processing Units (GPUs) can provide considerable computational advantages over multi-core CPU nodes or distributed networks by locally accelerating certain types of floating point operations. However, when processing and inverting exploration scale seismic datasets we encounter two key problems - compounded disk IO (explicit routing through the host is necessary) and the relatively small memory provided by the GPU ($\leq$ 6 Gbytes, restricting model sizes that can be allocated). As shown in an earlier discussion the IO bottleneck on the adjoint side can be somewhat circumvented by using random domain boundaries. Herein will be discussed how the forward modelling routine must be adapted to create an adjoint pair such that least-squares iterative inversion can be performed. We will then analyse how domain decomposition and P2P communication can be used to propagate over larger model sizes in such a way that communication can be effectively hidden and subsequently we can observe linear scaling.

INTRODUCTION

For smaller-scale research institutions, which may not have access to high performance computing facilities, processing terabytes of seismic data can be a significant challenge if attainable at all. As discussed in Ohmer et al. (2005) and Foltinek et al. (2009), GPUs can greatly assist any operation that can be considered as Single Instruction Multiple Data (SIMD) by running thousands of independent threads concurrently across the domain; two-way wave propagation can be considered as a SIMD operation as we are convolving a set stencil many times. However such a set up has disadvantages; the GPU can not read directly from disk, thus any disk based IO must be explicitly routed through a host CPU, compounding any such memory access. Furthermore the dynamic memory available on a GPU is 6 Gbytes or less, meaning that for propagation we are limited to a model size of 793 pt^3 for modelling and 600 pts^3 for imaging, assuming we are using acoustic, isotropic propagators. These numbers are significantly reduced when performing anisotropic and/or elastic propagation. Clapp (2009) and Leader and Clapp (2011) discuss how Reverse Time Migration (RTM) can be adapted to minimise disk access during propagation and hence better harness the computational power of the GPU without sacrificing significant performance for data movement. This paper will look at extending this system to inversion and also at how larger model sizes can be used. From here on a basic familiarity of GPU memory hierarchies and their uses will be assumed, one can to refer to Micikevicius (2009) or Leader and Clapp (2011) for more in depth discussions of these attributes.

GPU BASED RTM

Reverse Time Migration (Baysal et al., 1983) is increasingly becoming the industry and academic standard for seismic imaging. The full treatment of the wave equation provides us with accurate kinematic and amplitude information. Very few assumptions about the data (maximum dips, single scattering etc.) are necessary, relative to one-way wave equation or Kirchhoff imaging techniques. However, this process is computationally demanding and images still exhibit many artifacts due to the relatively simple nature of the imaging condition (Sun and Zhang, 2009).

We can describe an idealised modelling procedure as in equation 1, which is based on the first approximation of the Born scattering series. RTM is then the adjoint of this process, equation 2. Here d is the data, f the source function, G_0 are the respective Green's functions, m the model, $\mathbf{x}$ the 3D model coordinates, $\mathbf{x}_{r,s}$ the 3D source and receiver coordinates and $*$ denotes the complex conjugate. Despite this mathematical treatment assuming a single scattering, our propagator makes no such assumption and multiple scattering / diving waves are still positioned correctly.

$$d(\mathbf{x}_r, \mathbf{x}_s, \omega) = \sum_{\mathbf{x}_s, \omega} f(\omega) G_0(\mathbf{x}, \mathbf{x}_s, \omega) m(\mathbf{x}) \sum_{\mathbf{x}_r} G_0(\mathbf{x}, \mathbf{x}_r, \omega) \tag{1}$$

$$m(\mathbf{x}) = \sum_{\mathbf{x}_s, \omega} f(\omega) G_0(\mathbf{x}, \mathbf{x}_s, \omega) \sum_{\mathbf{x}_r} G_0(\mathbf{x}, \mathbf{x}_r, \omega) d^*(\mathbf{x}_r, \mathbf{x}_s, \omega) \tag{2}$$

Wave propagation can be performed very efficiently on GPUs by taking advantage of shared memory and read redundancy along the y axis (Micikevicius, 2009). The main challenge for GPU based RTM comes from the complex conjugate in equation 2. Since equation 2 is expressed in frequency, this conjugate reverses the sense of time of the data d relative to the source function f. As such, the source wavefield (4D, in this case) is forward modelled and saved, and then read back to the GPU at each imaging time step while back propagating the recorded data for correlation. Typically this source wavefield is several hundred gigabytes, meaning disk storage is unavoidable (without compression). In the case where we must transfer the source wavefield from disk our GPU based RTM scheme is far from optimal, as all computational advantages are now being counteracted by compounded disk access.

The reason we cannot simply back propagate the source wavefield is due to the fact that we artificially remove boundary reflections during forward modelling. This violates the conservation of energy, and these source wavefields are consequently not time reversible. A solution for this is to pad our domain with random boundaries (Clapp (2009); Fletcher and Robertsson (2011); Shen and Clapp (2011)); boundary reflections are now incoherently scattered, and since we are not removing any energy it is time reversible, to within computational precision. Our multiplicative imaging condition and subsequent stacking across the shot axes reduce any residual boundary noise to imperctible levels after around 50 or so shots / realisations. For a detailed

discussion of how to set up these boundaries and their scattering properties one can refer to Clapp (2009) or Leader and Clapp (2011).

Relative to source saving RTM we need to perform an extra propagation - the reversal and back propagation of the source wavefield, whereas previously we would just read back the appropriate wavefield slice. Fortunately, this extra computation is far faster than the wavefield reading scheme. In the case whereby the entire source wavefield is held by the CPU, and only CPU-GPU transfers are needed, the random boundary method is about 25% faster (Figure 1). For this to be the case the problem has to either be very small or the wavefield must be compressed or decimated, which will take extra computation. Once we have to use the disk we see the random boundary scheme is several times faster.

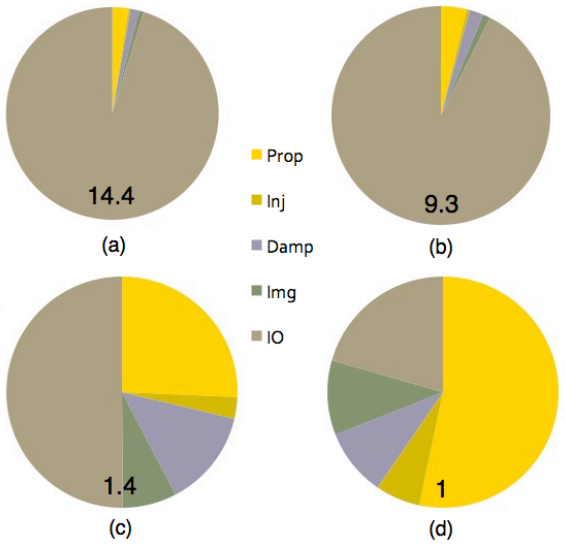

Figure 1: Relative speeds for RTM implementations for propagation, imaging, damping, injecting and IO. The bold numbers are normalised elapsed times. (a) shows the most naive disk set up, (b) an optimised disk set-up with asynchronous transfer, (c) the case where the 4D wavefield can be held by the CPU memory and (d) random boundaries. [**NR**]

GPU BASED LINEARISED INVERSION

GPU propagation augmented with random boundaries means no IO is required during propagation of either wavefield, making this a very computational effective scheme (relative to source saving.) However RTM images are often still artifact laden - low frequency artifacts can be prevalent near salt, acquisition footprints are noticeable, resolution is decreased (since we are squaring the source function) and random boundary noise may remain. All of these imperfections can be reduced by extending this scheme to least-squares inversion.

The inverse of our data modelling system can be approximated by constructing the adjoint pair of such a system and iterating in a least-squares sense. This is known as linearised inversion since we are assuming we know the kinematic model (the background velocity) and are trying to retrieve the high frequency (perturbation) part of the Earth model. This amounts to describing our slowness function as $s^2(\mathbf{x},\mathbf{y},\mathbf{z}) = b(\mathbf{x},\mathbf{y},\mathbf{z}) + m(\mathbf{x},\mathbf{y},\mathbf{z})$. As an algorithm it can be described by Algorithm 1. Here F is our operator, F' its adjoint, r the data-space residual, m the current model estimate,

d_{obs} the input data, gg the gradient and rr the back-projected residual. The stepper then updates the current model estimate and data-space residual. Since we are not changing our boundaries between iterations, we first forward our source wavefields and save the final slices necessary for back propagation. These can then be back propagated along with the receiver wavefield in the $gg = F'r$ step of the algorithm.

Algorithm 1 Linearised inversion with random boundaries

```
Initialise
Create random boundaries
Propagate source wavefield, save final slices
r = Fm − d_obs
while iter < niter; iter++ do
  gg = F'r
  rr = Fgg
  (m, r) =stepper(m, r, gg, rr)
end while
Output m
```

The adjoint process, F', is identical to the aforementioned RTM scheme with random boundaries. Now we must try and implement the forward process, F, in the most computationally efficient way. Initially we must think about our stencil. To take the spatial derivative of the wavefield we convolve the data with a star-shaped stencil, for an 8th order scheme in 3D this stencil is 25 points in total, as each direction shares the centre value. Previously we only needed to define the velocity value at the centre of our stencil, as the forward procedure was spraying the information to neighbouring grid cells. The adjoint of this propagation requires velocity values to be defined along the entirety of the stencil, as we now need the information from all points being calculated to carry to all new points. This means that in addition to allocating our wavefield values in the GPU kernel, we must also allocate the same number of velocity points.

Previously we were using texture memory to store our velocity values to take advantage of caching and automatic boundary control features (Leader and Clapp, 2011). What we can now do is save the relevant velocity values to shared memory, taking care that we do not saturate the memory. For large stencil sizes this can require the use of smaller blocks. By defining our local velocity array in the same way that we do our wavefield values, we can still take advantage of shared memory. This extra memory allocation and computation causes this adjoint propagator to be about 81% the speed of forward propagation. By only using texture memory we see this drop to around 43%, and by using global memory this decreases to 38%. On a Tesla GPU this final speed would be slower again, since Fermi cards provide some global L1 and L2 cache options. In order to make the system fully adjoint the source wavefield must be propagated through the same random boundary as in the corresponding RTM, but we can damp the data wavefield. Since we are now using equation 1 we have no time reversal problems, because the sense of time for both the source wavefield and the

data wavefield are the same.

The stepper is performed on the CPU. The reasons for this are twofold - firstly we want to transfer our data and image back to the CPU for writing purposes, so doing the model update on the CPU creates no unnecessary data transfer. Additionally, our solver is just a series of vector operations (dot products and subtractions) and such operations can not be accelerated by the GPU. They no longer have a SIMD corollary and global memory operations on the CPU are faster than on the GPU, often by a factor of 2 or 3.

Results from this linearised inverse scheme can be seen in Figure 3, in which we are attempting to recover the reflectivity model shown in Figure 2. Here we have 60 inline shots at a spacing of 100m and two crossline shots at a spacing of 1km, receivers are in a dense 825x200 grid. We can clearly see the inverse scheme improving the resolution of the top of the salt and mitigating the associated low-frequency artefacts.

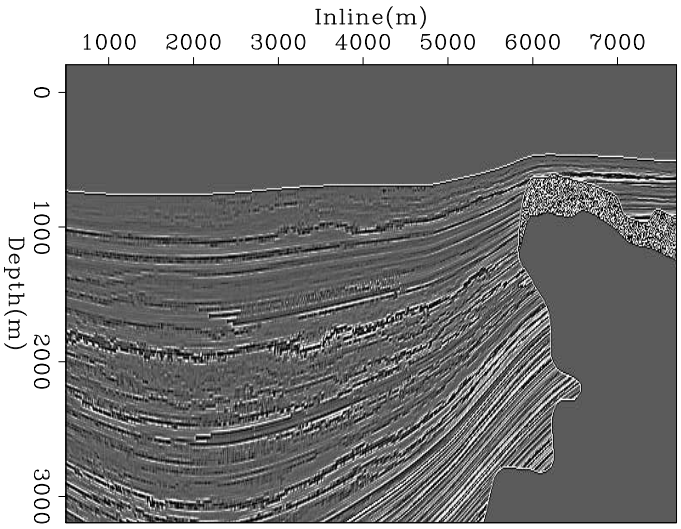

Figure 2: A 2D slice from the reflectivity model that we are attempting to recover. [**ER**]

DYNAMIC BOUNDARIES

As in Figures 2 and 3, we see inversion improve the images considerably, particularly when dealing with low frequency artifacts. However the remaining random boundary artifacts still seem to stack out fairly slowly, at a rate of about $\sqrt{N}$, where N is the iteration number. One option we have is to change these boundaries as a function of iteration, now we would expect to see a quicker reduction in boundary artifacts between iterations. This can be done by seeding our random boundary by iteration number, as well as shot position.

Our algorithm becomes Algorithm 2, giving us an additional propagation per iteration (recalculating the random source wavefield slices). For a typical model size this will increase computation time by around 14%. When doing phase encoded linearised inversion we can get this for free, since recalculation of the initial residual is needed during each iteration (Krebs et al., 2009). This system can be referred to as dynamic

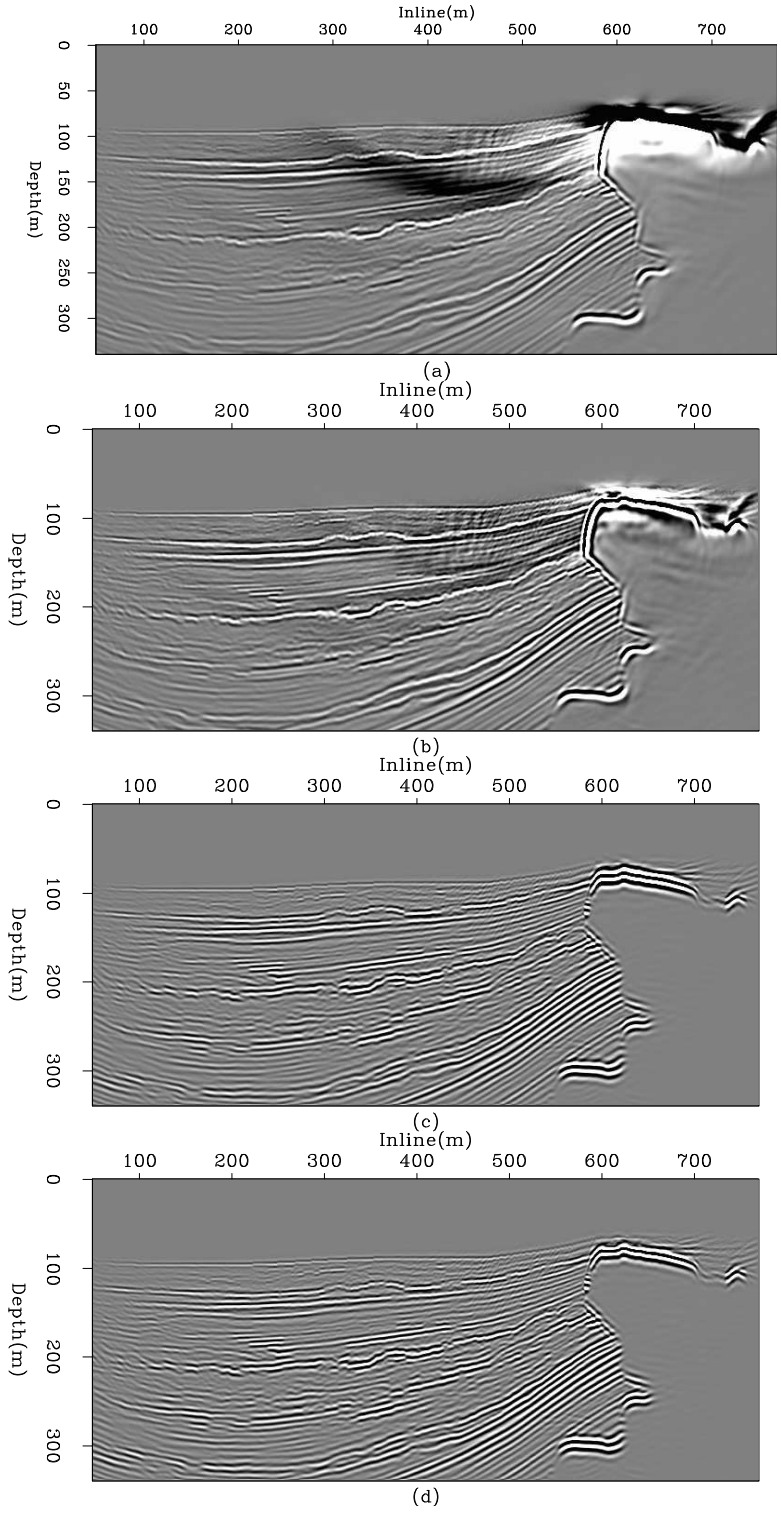

Figure 3: RTM and linearised inversion example 2D slices. (a) shows the raw RTM result, (b) raw inversion after 5 iterations, (c) RTM with lowcut bandpass filter and (d) inversion after 5 iterations and lowcut filter. [**CR**]

Algorithm 2 Linearised inversion with random boundaries

```
Initialise
r = Fm − d_obs
while iter < niter; iter++ do
  Create random boundaries
  Propagate source wavefield, save final slices
  gg = F'r
  rr = Fgg
  (m, r) =stepper(m, r, gg, rr)
end while
Output m
```

random boundaries, as opposed to static random boundaries. Furthermore our operator is now non-linear since we have altered our velocity function and hence our operator. This means theoretically we now have to use a non-linear solver. However since the operator difference is only manifested in the image noise there are some cases where a conjugate direction solver gives acceptable convergence characteristics. We see this system become useful in areas of poor shot sampling, where boundary artifact stacking-out can be slow, however in many cases the extra computation does not seem to outweigh improved convergence. As a function of iteration number we see slightly better residual performance (when using steepest descent for both), but when we look at this as a function of cost we see very little difference. Then, by comparing static boundaries with a conjugate direction solver to dynamic boundaries with a steepest descent solver, we get worse performance with dynamic boundaries as a function of cost.

It should be noted that comparing the l_2 norm of the residuals in this case is slightly misleading, since the high frequency noise we have slightly reduced is not well represented by this measure. When looking at the images more differences are notable than implied by this scalar fitting methodology. Additionally if we used a better non-linear solver we would fully expect to see more comparable performance.

DOMAIN DECOMPOSITION

In the case where our model size exceeds that of our computing system we must break up our domain. When using Fermi GPUs the global memory is 6 Gbytes, when using Tesla GPUs we are confined to 4 Gbytes. These limited memories severely restrict models that can be allocated. For propagation we have to allocate a minimum of three 3D fields - two wavefield slices and the velocity model (the third wavefield slice can replace the first) which confines us to a symmetric model of 793 pts^3 including padding regions. For RTM we need two slices for the source wavefield, two for the receiver wavefield, the velocity model, the image and the data. Assuming our time and depth axes are comparable in length, our restriction is now around 600 pts^3. We

are often concerned with models more at the scale of 1000 pts^3, and for wide azimuth data sets we may have dimensions larger than this. The solution is to decompose our domain, as in Figure 4. Performance does not strongly depend on the axis over which we decide to break our domain, but for current algorithm design cutting the domain along the y axis is the most effective. We want to only cut along one dimension, rather than say break our domain into cubes along two dimensions, as this minimises the quantity of halo transfer needed (Figure 4).

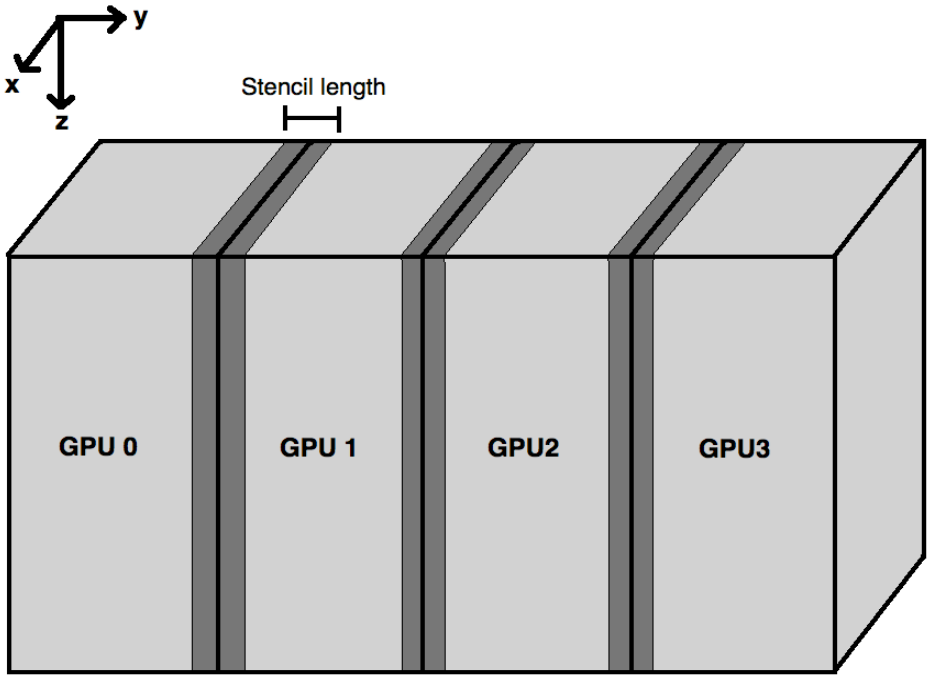

Figure 4: A diagram of how to decompose the model, dark grey regions are allocated on both neighbouring GPUs [**NR**]

We need our sub-domains to overlap along the y-axis by half the stencil length, else information would be lost between domains. Between time steps this halo region must be transferred and then synchronised before moving to the next time step. CUDA 4.0 and above used with Fermi cards allows for Peer to Peer (P2P) GPU communication and data transfer. Previously it was necessary to explicitly route all information transfer between GPUs first through the CPU, which was significantly slower. Additionally, neighbouring GPUs can now operate on a Unified Virtual Address space (UVA), meaning there is no risk of dereferencing a pointer that has the same address on a different GPU. In fact it is now possible to dereference pointers on other GPUs or on the host, by virture of this UVA. Arrays can not span GPUs, however.

The main technique of making such a method scale linearly is by using asynchronous memory copies and kernel calls. When doing this it is possible to overlap communication with computation, hiding halo communication time. This can be done by associating certain calls to separate GPU streams, where a stream can be

considered as a command pipeline. Within a stream, calls are serial, however different streams can execute concurrently. By restricting halo computation and communication to one such stream, and the other data (internal) computation to another, we can hide the halo communication. Some simplified CUDA code using streams is displayed below for reference.

```
for(i_gpu=0; i_gpu < n_gpus; i_gpu++){
   cudaSetDevice(i_gpu);
   kernel<<<...halo_region[],halo_stream[i_gpu]>>>(...);
   kernel<<<...internal_region[],internal_stream[i_gpu]>>>(...);
}
for(i_gpu=0; i_gpu < n_gpus; i_gpu++){
   cudaMemcpyPeerAsync(...halo_stream[i_gpu]);
}
for(i_gpu=0; i_gpu < n\_gpus; i_gpu++){
   cudaStreamSynchronise(...halo_stream[igpu]);
}
for(i_gpu=0; i_gpu < n_gpus; i_gpu++){
   cudaMemcpyPeerAsync(...halo_stream[i_gpu]);
}
for(i_gpu=0; i_gpu < n_gpus; i_gpu++){
   cudaDeviceSynchronise();
}
```

The key part to remember here is that kernel calls and async calls can overlap, providing they are defined to separate streams. So initially we loop through devices, then each one will calculate its halo region data. Once this is done, each GPU will move onto the internal data calculation. Whilst this is being calculated, each GPU moves onto the three communication loops. First off, each GPU sends its halo data to the right and receives from the left, then each device is synchronised, then each device sends to the left and receives from the right. Now each GPU has the relevant halo information for the next time step. Finally we synchronise to ensure all internal data computation is done before moving to the next time step.

The GPU devices are linked by PCIe switches and these switches are duplex, meaning each GPU can send and receive simultaneously - providing they are in different directions. This means a given device can send data to the GPU on its left and receive from the GPU on its right at the same time. We use the stream synchronise call because the moment a device tries to send and receive in the same direction the switch can stall, costing time. A pictorial demonstration of this is show in Figure 5.

Such a system assumes we have neighbouring GPUs on the active node in question, linked by PCIe switches. If one has GPUs on separate nodes (such as in a distributed network) MPI must be used for the halo communication. Unsurpisingly this is slower, albeit only by a factor of around 1.5x.

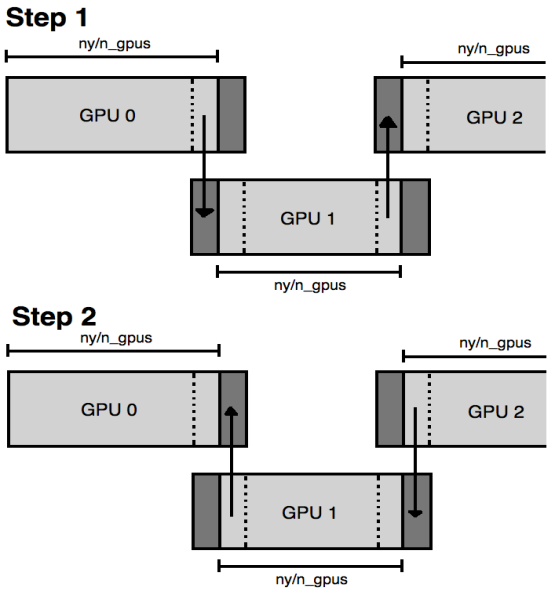

Figure 5: A diagram of the two stages of GPU halo communication, simplified to 1D [**NR**]

For acoustic propagation our internal computation time is always in far excess of halo transfer time (by an order of magnitude or so), meaning communication is always hidden and our domain decomposition scales linearly with model size, hence it is optimised. For TTI propagation this communication time approaches that of the internal data calculation, meaning for highly elastic and anisotropic media care must be taken to optimise the decomposition to ensure one is never waiting for communication. The moment we are waiting for information transfer we are not fully harnessing the potential of the GPU.

CONCLUSION

We conclude that GPUs can be effective when running linearised inversion. By storing a velocity stencil in shared memory, as well as the wavefield values, we can perform accelerated adjoint propagation. By augmenting this with a random boundary based RTM scheme and a CPU based model stepper we can perform least squares iterative linearised inversion very efficiently, with little time lost for data movement.

At the point when our domain size exceeds our GPU memory it is possible to decompose this domain across multiple devices, whether one has multiple GPUs per node, multiple nodes with single GPUs, or any combination thereof. By making halo communication calls overlap with internal data computation it is largely possible to hide all communication, meaning our time scaling with model size is still linear.

Combining these schemes gives us the potential to perform large scale inversion at a high level of fine grain parallelism.

ACKOWLEDGMENTS

Thanks to Paulius Micikivicius at NVIDIA for his invaluable help with GPU troubleshooting, his prompt response with domain decomposition queries and his willingness to share codes.

REFERENCES

Baysal, E., D. Kosloff, and J. Sherwood, 1983, Reverse time migration: Geophysics, **45**, 1514–1524.

Clapp, R. G., 2009, Reverse time migration with random boundaries: SEG Technical Program Expanded Abstracts, **28**, 2809–2813.

Fletcher, R. P. and J. O. A. Robertsson, 2011, Time-varying boundary conditions in simulation of seismic wave propagation: SEG Technical Program Expanded Abstracts, **30**, 2957–2961.

Foltinek, D., D. Eaton, J. Mahovsky, P. Moghaddam, and R. McGarry, 2009, Industry scale reverse time migration on GPU hardware: SEG Technical Program Expanded Abstracts, **28**, 2789–2793.

Krebs, J. R., J. E. Anderson, D. Hinkley, R. Neelamani, S. Lee, A. Baumstein, and M.-D. Lacasse, 2009, Fast full-wavefield seismic inversion using encoded sources: Geophysics, **74**, WCC177–WCC188.

Leader, C. and R. Clapp, 2011, Memory efficient reverse time migration: Stanford Exploration Project Report, **143**.

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

Ohmer, J., F. Maire, and R. Brown, 2005, Implementation of kernel methods on the GPU: DICTA'05 Expanded Abstracts, **78**.

Shen, X. and R. G. Clapp, 2011, Random boundary condition for low-frequency wave propagation: SEG Technical Program Expanded Abstracts, **30**, 2962–2965.

Sun, J. and Y. Zhang, 2009, Practical issues of reverse time migration: True amplitude gathers, noise removal and harmonic-source encoding: ASEG Expanded Abstracts.