Large scale linearised inversion with multiple GPUs

Chris Leader and Robert Clapp

ABSTRACT

As our computational power develops and evolves so does our desire to process more data with more advanced algorithms. Graphical Processing Units (GPUs) have been shown to accelerate algorithms that have a high computation to memory access ratio. However, their relatively small global memory (6 Gb) poses additional restrictions on how we must adapt our problem. Herein will be described how multiple GPUs can be used to accelerate the problem of wave-equation linearised inversion in such a way that poses no model or data size restrictions. Furthermore by splitting the internal and external parts of our domains we can achieve close to linear strong-scaling with the number of GPUs that we are using. Consequently we can design a method that outperforms CPU based inversion while overcoming the traditional restrictions of GPU based inversion.

INTRODUCTION

In today's world of heterogeneous computing systems it is increasingly important to design and tailor one's algorithm to the available hardware. For seismic imaging we are increasingly looking to numerically intensive inverse techniques to improve the quality, frequency content and fidelity of our images, and more and more often we are using heterogeneous computing systems to achieve this. This is a demanding process for both the computer and the programmer. Many of the largest supercomputers in the world are incorporating hybrid systems of CPUs and GPUs and the effective use of these two different processing units is paramount for system performance. Migrating legacy codes that worked and scaled well on clusters of CPUs to GPUs can easily run into problems; for seismic imaging these are generally related to compounded disk access and the limited global memory provided by a single GPU. The focus of this discussion will be on using a two-way wave-equation engine for recovering subsurface scattering potential, using a linearised inversion scheme.

Ohmer et al. (2005) and Foltinek et al. (2009) show how GPUs can greatly assist any operation that can be considered as Single Instruction Multiple Data (SIMD) by running thousands of independent threads concurrently across a given 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), Fletcher and Robertsson (2011) and Leader and Clapp (2011a) discuss how Reverse Time Migration (RTM) can be adapted to minimise disk access during propagation by using random boundaries, and hence better harness the computational power of the GPU without sacrificing significant performance for data movement.

LINEARISED INVERSION

We can describe an idealised modelling procedure as in equation 1, which is based on the first approximation of the Born scattering series. The adjoint of this process, equation 2, is often referred to as Reverse Time Migration (RTM). 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 single scattering, our propagator makes no such assumption and multiple scattering / diving waves are still positioned correctly.

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

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

By using these processes, along with an appropriate model updating scheme, we can perform linearised inversion. This is advantageous over RTM because it iteratively removes unphysical aspects of the recovered model - such as acquisition footprints, the effect of the wavelet and unbalanced amplitudes (Lambare et al., 1992). For each iteration we must run the forward and adjoint processes, making each update about twice as expensive as RTM. The fact that the GPU can perform wave propagation so fast makes such an imaging process possible.

GPU BASED INVERSION

Optimally adapting CPU based linearised inversion to a set of GPUs requires careful consideration of the architecture differences. While the GPU can accelerate floating point computation for comparatively less effort, it has a smaller global memory and cannot directly read or write from disk. This means disk access is compounded and must be avoided to observe acceptable performance. This fact, combined with our

accelerated computation, makes memory transfer and disk access appear a much larger problem than on a CPU.

As previously described in Clapp (2008a) and Leader and Clapp (2011b), the two wavefields in our adjoint process have an opposite sense of time, as denoted by the complex conjugate in equation 2, thus generally one is precomputed and stored on disk. We can circumnavigate this disk use by padding our computational domain with random boundaries making our source modelling time reversible. We must perform an extra propagation per time step under this regime, but this is much cheaper than accessing the disk-stored wavefield Clapp (2008b).

We can take two steps to further optimise our GPU based scheme (assuming our propagator is using shared memory efficiently (Nickolls et al., 2008).) These are aligning warps within the GPU and using shared memory to store the velocity array during adjoint propagation. The second of these was described in Leader and Clapp (2012).

A GPU kernel can be viewed as a Single-Instruction Multiple Threads (SIMT) process, and a given single instruction is issued to a warp (or thread-vector) which guarantees vectorisation. For a typical NVIDIA GPU, a warp is a vector of 32 contiguous threads. There are two ways we can use warps to increase performance: we can ensure each warp is only issued a single instruction (to avoid divergence within a warp) and we can encourage coalesced memory access (Micikevicius, 2011). The second of these is applicable to finite difference kernels.

Memory addresses from a warp are converted into line requests, these can be 32B or 128B depending on cache options. A single thread within a given warp can access a memory address from within that line at 100% bus utilisation; however, if a neighbouring thread requests a memory address from an adjoining warp, then additional bytes must be moved across the bus. Consistent misaligned memory access such as this can drastically decrease bus utilisation and hence overall kernel performance (as in Fig 1). By encouraging our domain to align along these line requests we can see an improvement of up to 20% (in this case) for minimal effort.

In our specific case we are using an 8th order 3D finite-difference stencil over our domain. Practically, this means that we pad our axes by four points on each side to account for the 'halo' (the width of the stencil arms) so that we do not try to access points outside of our domain. All we must do is to pad our fast axis by an additional 28 points so that our outer region now falls within a warp. The nature of our stencil means we cannot avoid some out-of-warp memory requests, but this additional padding, while marginally increasing our domain size, reduces our run time by about 20%.

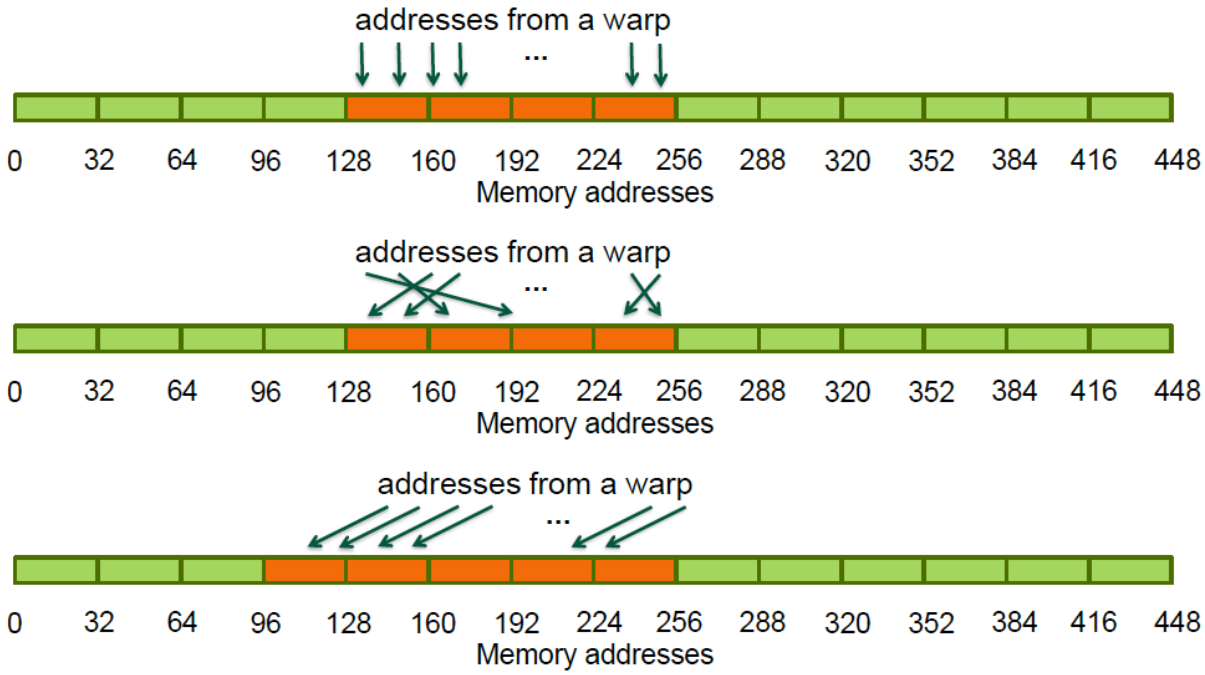

Figure 1: Schematic of warps alignment with memory requests. The above two cases feature 100% bus utilisation, the bottom case has 80% due to misaligned memory accesses. From Micikevicius (2011).

MULTIPLE GPU EXTENSION

There are three reasons why we might want to use multiples GPUs: if our domain exceeds the memory of a single unit, to accelerate computation for a smaller domain size, or to increase performance per watt (Micikevicius, 2012). To create our model size independent inversion scheme we must split our domain across multiple GPUs.

For propagation we have to allocate a minimum of three large 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 reverse time migration and for linearised modelling 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 free to break up our domain how we choose, however in this case we find it advantageous to break it up along the depth (z) axis, which is also the slowest axis. The reasons here are twofold - breaking up along one axis does require slightly more allocation, but the halo region is contiguous in memory (if broken along the slowest axis). This alleviates the need for a separate kernel that simply aligns the correct memory values. The second reason is that our receiver and source positions are in a plane near the surface; this means that only one GPU needs to have all the geometry

information and needs to deal with data/source injection/collection. Thus we do not need to loop over GPUs to perform these operations.

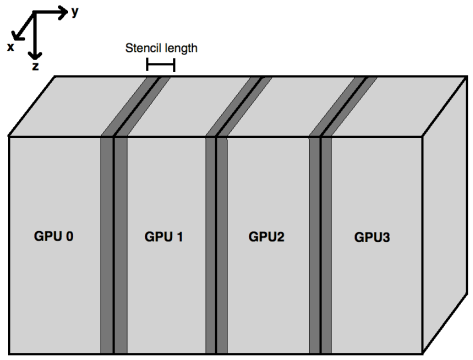

Figure 2: A diagram of how to decompose a model (in this case along the y-axis), dark grey regions are allocated on both neighbouring GPUs. [**NR**]

We need our sub-domains to overlap along the depth axis by half the stencil length, as this information is needed from neighbouring domains to continue to the next time step and to move the wave through the domain boundaries. Consequently, between time steps this halo region must be transferred and then synchronised before moving to the next time step, else we run the risk of spurious propagation. Recent NVIDIA releases have allowed Peer to Peer (P2P) GPU communication and data transfer, rather than explicitly routing all information transfer between GPUs first through a CPU buffer. 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. These two new additions make this GPU-GPU information transfer more efficient and easier to implement.

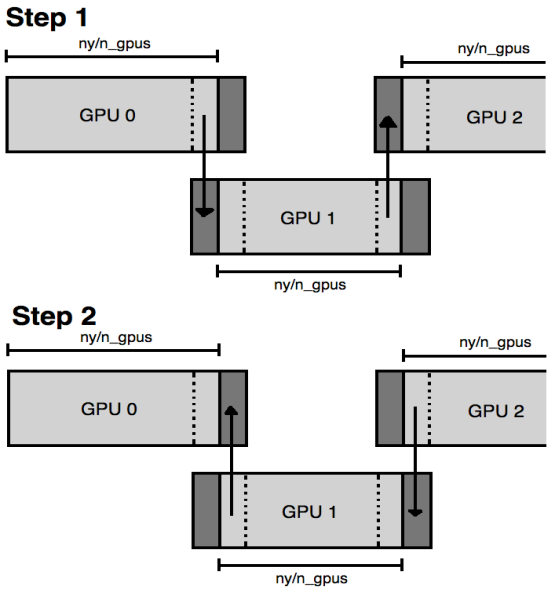

Figure 3: A diagram of the two stages of GPU halo communication, simplified to 1D. [**NR**]

We can make this process scale (very close to) linearly by using asynchronous

memory copies and kernel calls. When doing this it is possible to overlap (halo) communication with (internal) computation, as in figure 3, 'hiding' the communication time (Micikevicius, 2012). 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 (for certain operations). By restricting halo computation and communication to one such stream, and the other data (internal) computation to another, we can hide the halo communication by overlapping the memory transfer with the internal kernel. We can first calculate the wavefield values within these halos regions, then we can send the information to the 'right' (GPUs with higher IDs), then send to the 'left,' then synchronise. Again, this is shown in fig 3. The PCIe bus that links the GPUs is duplex, meaning it can send and receive at the same time in two different directions. However, if an application tries to send and receive along the same direction at the same time, the link can stall. So between the two send operations we add a synchronisation step. 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=1; 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-1; i_gpu++){
   cudaMemcpyPeerAsync(...halo_stream[i_gpu]);
}
for(i_gpu=0; i_gpu < n_gpus; i_gpu++){
   cudaDeviceSynchronise();
}
```

This discussion has been limited to the problem of propagation thus far. Performing linearised modelling and migration over multiple GPUs adds a level of complexity. For linearised modelling we must propagate two fields, apply a scattering condition, inject the source term and perform data collection. For RTM, we must propagate two fields, apply an imaging condition, and inject both the source term and the recorded data.

The corresponding adjoint routine code, grossly simplified, will look something like:

```
for(i_gpu=0; i_gpu < n_gpus; i_gpu++){
```

```
   cudaSetDevice(i_gpu);
   kernel<<<...data_halo_region[],halo_stream[i_gpu]>>>(...);
   kernel<<<...source_halo_region[],halo_stream[i_gpu]>>>(...);
   kernel<<<...data_internal_region[],internal_stream[i_gpu]>>>(...);
   kernel<<<...source_internal_region[],internal_stream[i_gpu]>>>(...);
}
for(i_gpu=1; i_gpu < n_gpus; i_gpu++){
   cudaMemcpyPeerAsync(data_halo_region[i_gpu]...halo_stream[i_gpu]);
   cudaMemcpyPeerAsync(source_halo_region[i_gpu]...halo_stream[i_gpu]);
}
for(i_gpu=0; i_gpu < n\_gpus; i_gpu++){
   cudaSetDevice(i_gpu);
   cudaStreamSynchronise(...halo_stream[igpu]);
}
for(i_gpu=0; i_gpu < n_gpus-1; i_gpu++){
   cudaMemcpyPeerAsync(data_halo_region[i_gpu]...halo_stream[i_gpu]);
   cudaMemcpyPeerAsync(source_halo_region[i_gpu]...halo_stream[i_gpu]);
}
cudaSetDevice(0);
src_inject_kernel<<<...source_internal_region[],internal_stream[i_gpu]>>>(...);
data_inject_kernel<<<...data_internal_region[],internal_stream[i_gpu]>>>(...);
for(i_gpu=0; i_gpu < n_gpus; i_gpu++){
   img_kernel<<<...,internal_stream[i_gpu]>>>(...);
}
for(i_gpu=0; i_gpu < n_gpus; i_gpu++){
   cudaDeviceSynchronise();
}
```

We assign the injection and imaging kernels to the internal stream since these steps all depend on the previous having completed and we are operating on the internal region for injection and collection. Each of these operations individually scale roughly linearly with problem size, so they require no unique treatment. The final normalised speed ups as a function of number of GPUs used can be seen in Figure 4. We can see that we achieve very close to linear scaling. The reason we do not get to 100% is that splitting the halo and internal computation requires a small amount of overhead, meaning we typically see around 92% to 96% of strong-scaling speed up with number of units used. If we do not use the communication overlap we see a much worse performance, with numbers typical of a naive OpenMP implementation on an eight-core CPU.

CONCLUSIONS

Previous reports have discussed how RTM and linearised inversion must be adapated to run efficiently on a single GPU. Descirbed herein was how we can futher improve

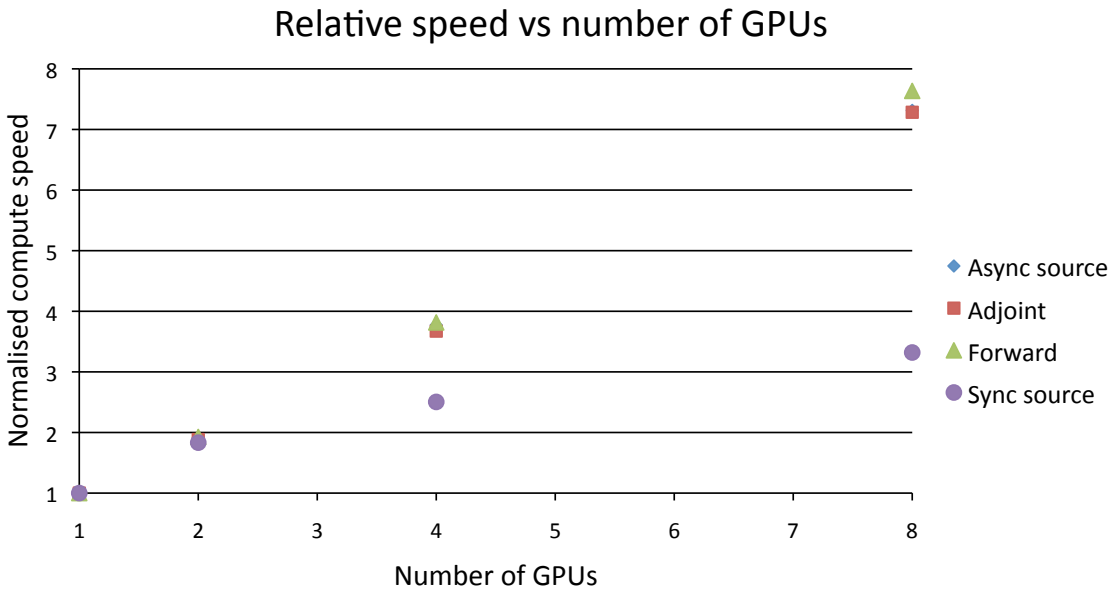

Figure 4: A graph of how synchronous source modelling, asynchronous source modelling, the forward process and the adjoint process scale with number of GPUs. [**NR**]

performance through warp alignment, and results on inversion across multiple GPUs. Through warp alignment we can accelerate the propagation part of our algorithm by 20% for minimal extra work. Then by splitting the computation and communication of different parts of our domain we can propagate our fields over multiple GPUs, and we can scale almost linearly with the number of GPUs that we use. When performing linearised inversion over 8 GPUs we get a relative speed up of 7.6.

ACKNOWLEDGMENTS

Thanks to Paulius Micikevicius at NVIDIA for invaluable GPU troubleshooting advice and code sharing. Further thanks to all sponsors of the Stanford Exploration Project for their continued financial and intellectual support.

REFERENCES

Clapp, R., 2008a, Reverse time migration: Saving the boundaries: SEP-136, **136**.

———, 2008b, Reverse time migration with random boundaries: SEP-138, **138**.

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.

Lambare, G., J. Virieux, R. Madariaga, and S. Jin, 1992, Iterative asymptotic inversion in the acoustic approximation: Geophysics, **57**, 1138–1154.

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

———, 2011b, Memory efficient reverse time migration: SEP-143, **143**.

———, 2012, Linearised inversion with GPUs: SEP-147, **147**.

Micikevicius, P., 2011, Performance optimization: Presentation at Supercomputing 2011.

———, 2012, Multi-GPU programming: Presentation at GPU Technology Conference 2012.

Nickolls, J., I. Buck, M. Garland, and K. Skadron, 2008, Scalable parallel programming with cuda: Queue, **6**, 40–53.

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