High performance computing solutions

IMPLEMENTING LINEARIZED INVERSION

As discussed in Chapter **??**, and indeed throughout this thesis, practical implementation of these wave-equation based inversion schemes is not straightforward. Naively coding such a system would result in impractically long run times; for any significantly sized problem large computers much be used and care must be taken to use them to the best of their ability.

This final section will demonstrate how these operators can be optimally coded for both a system of CPUs and a system of nodes that contain both CPUs and Graphical Processing Units (GPUs).

Optimization

As a concept, optimization is an act or process that aims to make something as effective as possible. In applied mathematics, optimization is posed as a way to maximize productivity by finding the values and influences of controllable factors which determine the behavior of a system. This chapter will address computational, or programming, optimization. There are two major requirements to consider computational optimization: intimate knowledge of the computer (or network) that one is optimizing for, and intimate knowledge of the serial and parallel sections of the algorithm. The 'controllable factors' for this variety of optimization are the hardware and it's acceleration options (multi-core, vectorization etc), the algorithm (decomposition, parallel segments etc) and time.

For research code, time is a crucial factor for computational optimization - it is imperative that the additional time required to optimize the code, t_{opt}, is significantly less than the difference between the new runtime and the unoptimized runtime, say $t_{opt} \ll t_{orig} - t_{new}$. Put into words: time saved through optimization must be less than the time spent to optimise. For production code, that will be run many times, then optimization can save vast quantities of future compute time.

Furthermore, the abstract conception of optimization implies a final solution. When it comes to computational considerations 100% optimization is never achieved, as one could never balance $t_{opt} \ll t_{orig} - t_{new}$. From hereon the term optimization will denote an effort to accelerate computational runtime by any increment. This will be in particular reference to two systems, multi-core CPUs and linked GPUs. The focus will be primarily on the latter, nonetheless CPU techniques will be discussed in brief in order to construct a valid comparison.

CPU based linearized inversion

For optimisation, it is imperative to understand the architecture of the given card(s). Figure 1 shows a simplified diagram of the separate memory levels for a typical four-core CPU node (modern nodes may contain eight or sixteen cores, this is for illustrative purposes.) Each core features an FPU (Floating Point Unit), which performs the arithmetic, three individual memory levels (local to each FPU and only accessible from that FPU) and two larger memory levels, accessible to all FPUs on the node. As 'distance' from the FPU increases so does both size and latency of the memory level; latency is a measure of the time taken to transfer a byte from a given memory location to the FPU.

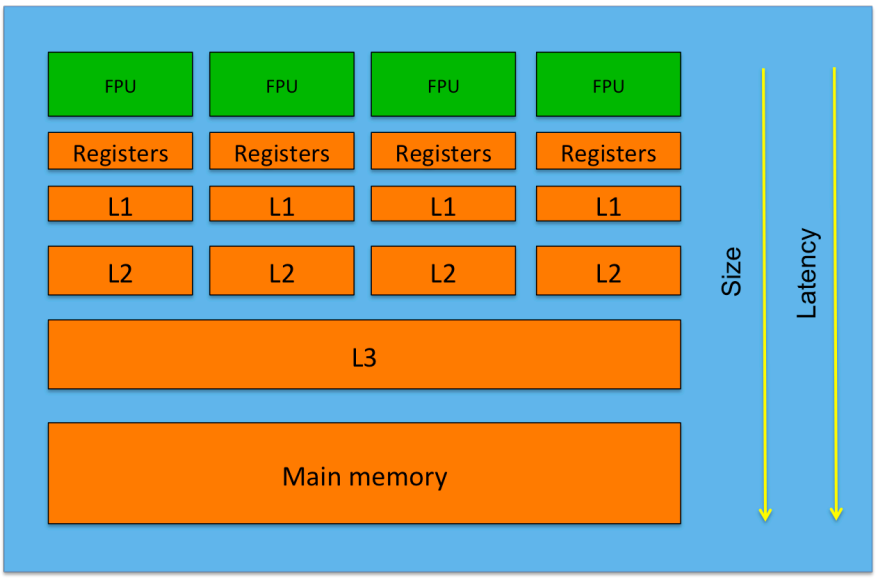

Figure 1: Diagram of the memory layout of a typical multi-core CPU node. As distance from the Floating Point Unit (FPU) increases, so does both the size and latency of the memory unit. [**NR**]

From this model there are already two options to consider - the fact there are multiple arithmetic units, and the fact there are memory levels of different latencies. Initially, making use of the additional FPUs will be addressed. There are many programming languages that can take advantage of multiple CPU cores on a single node - MPI, POSIX, Python etc, however this discussion will be limited to OpenMP (**?**). OpenMP is an advantageous choice due to the ease of its initial implementation; only a few additional directives are required in order to make a code run over all available cores. For high performance optimization this is also it's disadvantage, since these compute threads are not strictly independent and speed up saturates. However, the goal of this discussion is to provide a semi-optimized CPU code for reasonable GPU comparison, hence OpenMP plus some additional optimization is acceptable. Of course, when it comes to parallel programming one is fundamentally limited by the serial sections of the process, and speed up saturation is also seen due to this.

This phenomenon is known as Amdahl's Law (**?**).

Modern CPU nodes allow multi-threading, whereby a given core can run multiple threads. For some applications this can be an advantage, but due to the size of the fields allocated in seismic wave propagation and the relative operation-to-read count more often than not multi-threading results in a performance decrease. Due to this only one thread per core will be used, so during CPU discussions these terms can be considered interchangeable. OpenMP directives were added to the outer (slowest) loop of the propagation kernel (**?**). The node used comprised of an Intel Nehalem CPU running eight independent cores, tests were run on one, two, four and eight cores. The results of improved run times can be seen in Figure 2.

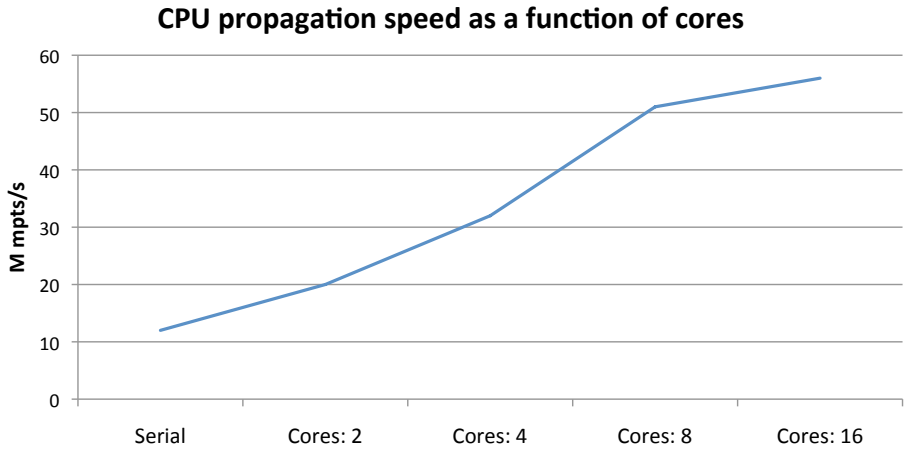

Figure 2: Compute speed (million models pts calculated per second) against implementation. [**NR**]

Noticeably, run times improved. However we see the relative speed up as a function of number-of-cores saturate very quickly; this is due to a number of reasons specific to CPU architecture. These explanations revolve around moving between shared memory spaces and the context switching necessary to do this. While the latter of these issues is unavoidable, the former can be addressed by further considering the memory levels shown in Figure 1.

To make full use of these low latency (cached) memories it is desirable to fill them with information that will either be re-used in the algorithm, or fill them with values that are aligned for computation (contiguous). This is called temporal and spatial locality (**?**). Spatial locality refers to values close to each other in physical memory, particularly those that may be read in a cache line; temporal locality refers to values that have been read into a cached memory already and can be efficiently reused before they are flushed.

For wave propagation, values that could be reused are local wavefield values (used for the derivative) and local velocity values (reused during adjoint propagation). When convolving with the finite-difference stencil run-times will be much faster if

the information needed resides on L1 or L2 memory. Once the code reaches for values in L3 or global memory then run times can increase dramatically. Whilst L1, L2 and register memory feature very low latency they are both very limited in size and for generic (non-machine specific) coding there is no explicit control over how the memories are used. It is possible to encourage the CPU to fill and re-use the cache levels in certain ways however it is very easy to pollute these cached memories.

A typical velocity field is of the order of 1000x1000x1000 model points in the three Cartesian directions, stored in 4-bit precision, giving a total size of 4 Gbytes. This is clearly vastly larger than even L3 memory so the problem must be divided into smaller blocks, this is called 'blocking' which is a type of 'domain decomposition.' The stencil used is a symmetric 8th order finite difference stencil, making it 25 pts (in 3D).

Initial estimates of how large to construct these blocks is to ensure they fit into L2 memory, thus individual FPUs do not have to access shared memory spaces in order to perform the key computations. Secondly, the shape of the blocks must be considered. The stencil is applied by looping over the three principle axes, with the loop over the inner axis being the 'fast' axis (the axis along which memory is contiguous.) This is the only axis for which it is possible to be confident over efficient memory reuse (spatial locality), so the most efficient cache coherency is observed by constructing long, thing, 'pencil' shaped blocks. In this working example the fastest axis is the depth (z) axis; by making the length of this axis substantially longer than the other two (x and y) then a speed up of around double is observed, purely through blocking.

Doubling the compute speed is a promising result, however it is far below the maximum possible speed up. A new plot detailing how blocking can assist in propagation speeds can be seen in Figure 3. It would be possible to further optimize the block size and the way the third axis is looped over; it would also be possible to extend the optimization attempt to take advantage of the vector units contained within the CPU cores. However the next step within this thesis will be to consider how GPU computing can assist the acceleration of acoustic wave propagation. The attempt described herein for CPU optimization creates a valid comparison to GPU run times, since both multi-threading and memory caches have been used to improve run times.

GPU BASED LINEARIZED INVERSION

Graphical Processing Units (GPUs) have been present inside computers for decades. They feature a highly parallel specialized electronic circuit which traditionally is used to output graphics to a display. The inception of General Purpose GPUs (GPGPUs) in the early 2000s allowed practical computing on these devices by virtue of manipulatable floating point support (**?**), however it wasn't until the advent of CUDA (Compute Unified Device Architecture), by NVIDIA in 2007, that the potential of this new programming paradigm could be easily harnessed by general developers (**?**).

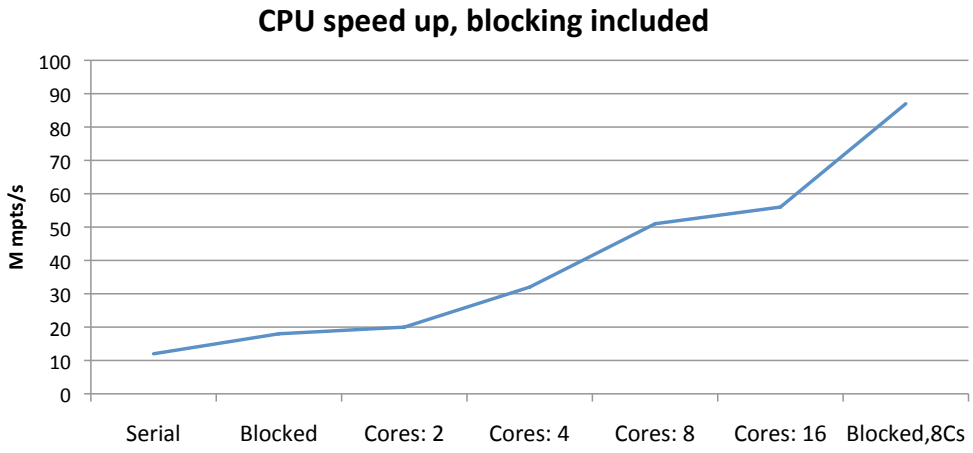

Figure 3: Compute speed (million models pts calculated per second) against implementation. [**NR**]

Many of the world's largest supercomputers now feature CPU-GPU hybrid capability. CUDA is very similar to C, but also allows GPU specific routines (kernels) to be written and run. Generally the CPU is referred to as the 'host,' and the GPU as the 'device.' For efficient GPU implementation an algorithm must feature two key characteristics: it must be scalable and parallelizable. The former of these simply means that as the problem size is reduced or increased the relative performance should be roughly preserved; this requirement is needed because the GPU primarily relies on fine-grain parallelization. The latter means that the problem must be able to broken into smaller sections that can operate, at least for a period, independently. Any algorithm that fits these criteria should be accelerated via GPU computation ((**?**); (**?**)). Certain methodologies, such as some classes of triangulation, can be slowed down when ported to a GPU, but the vast majority of algorithms can be adapted to harness the parallel power of GPU computing.

Figure 4 demonstrates the layout for a simplified GPU, and this can be compared to Figure 1, which shows the layout for a simplified multi-core CPU (the corollary, in this case). The GPU layout features many more Streaming Multi-Processors (SMPs) than that CPU; whilst a typical CPU will have around 8 cores (without multi-threading), a given GPU can have upwards of 1536. The exact cards used for the majority of this study were the Nvidia Fermi M2090 cards, which each feature 512 SMPs. However the true parallel power of the GPU is not exhibited solely due to these increased numbers, but by the fact that these individual SMPs (or thread-blocks) can operate 100% independently, without the need for context switching. Whereas the individual FPUs on a CPU node can operate simultaneously, the layout requires constant communication between memory levels and FPUs, which (especially for a naive implementation) severely inhibits performance (**?**).

Furthermore, each of these thread-blocks is divided into a two-dimensional grid

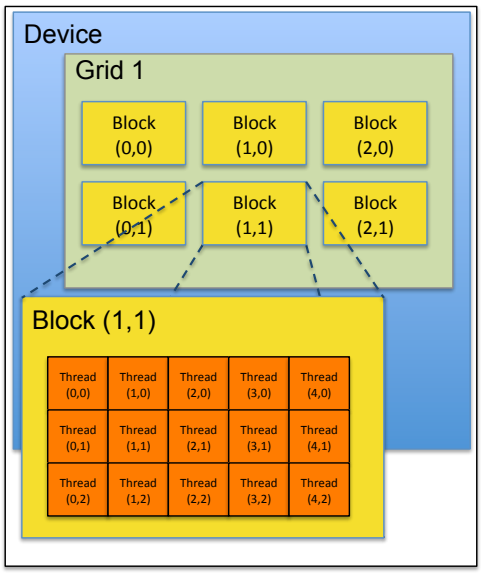

Figure 4: Schematic for a given GPU architecture, demonstrating how the grid is broken into two-dimensional thread blocks, which are broken up into individual threads. [**NR**]

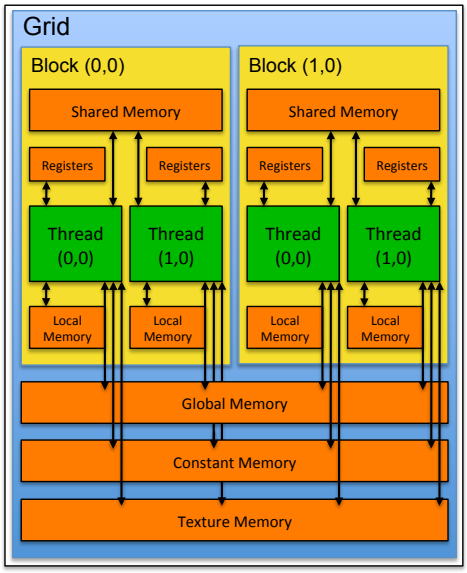

Figure 5: A visual representation for the different memory levels within a GPU. Each block has a unique shared memory and each thread has it's individual local memory and set of registers. Global, constant and texture memories are shared between all thread blocks. [**NR**]

of threads (figure 4). These 'threads' are the individual processors that perform the work; similarly to the thread-blocks, many threads can run independently and simultaneously. The caveat is that each thread must be performing exactly the same task (both within and across SMPs) on a reduced portion of the problem - hence, the algorithm must be scalable and parallelizable. By executing thousands of these individual threads simultaneously all latency is effectively hidden, and very high levels of multiple parallelism are possible. This latency hiding through mass thread execution is what allows the GPU to operate at such a high bandwidth.

The different memory levels of the GPU are shown in figure 5, again this can be compared to figure 1. Each thread-block features an individual and unique shared memory level, values stored here can be different between thread-blocks. This is analogous to the L1 cache in the CPU - only around 16kb or 32kb in size, and very low latency. Additionally, each thread has a separate local memory and set of registers, but these have to same for every thread across the whole GPU. The memory levels of significant size are global, texture, and constant memory, and these all share the 6 Gbytes of DRAM on the card (for the M2090, other vintages vary in DRAM size). These are similar to the global memory on the CPU; they are shared between all SMPs and have a much larger latency. In fact the latency for a thread to access data from the GPU global memory can be up to four times as large as that for a given FPU to access data in the CPU global memory. Thus for the GPU to out-perform the CPU intelligent use of shared memory is imperative, in addition to concurrent thread execution.

Single card GPU optimization

The global memory on a single GPU card is drastically smaller than a CPU. A Fermi M2090 features 6 Gbytes of addressable RAM, whereas a typical CPU node might feature 16x, 32x, or even more times this. Concordantly, a solution that uses multiple GPUs must be used, such that large 3D problems can be solved. However, developing an optimized single card solution for a modest problem size is the first step for both code testing and strong scaling performance evaluation.

To maximize single card performance it is important to understand how groups of threads are executed. When a kernel is written and called it gives each thread the same set of instructions to execute, and understanding which memory points these threads access is crucial. During execution threads are grouped into warps, which are simply a set of 32 aligned threads. A single warp is able to access 32 bytes (a GPU cache line), essentially for free. Any thread within this warp can access any of these bytes with negligible latency, however if a thread tries to access an address that belongs to a different warp then context switching must be used, which briefly stalls the bus. Figure 6 demonstrates this pictorially. Initially each thread is accessing a single memory point within it's warp, and bus utilization is 100%. Next, each thread is operating on scattered memory points within the warp, but because it does not reach outside the given 32 bytes bus utilization is still 100%. In the third case the

address that the threads are instructed to access do not all lie within the correct warp and there is a systematic shift, this creates a stall and utilization is reduced to 80%.

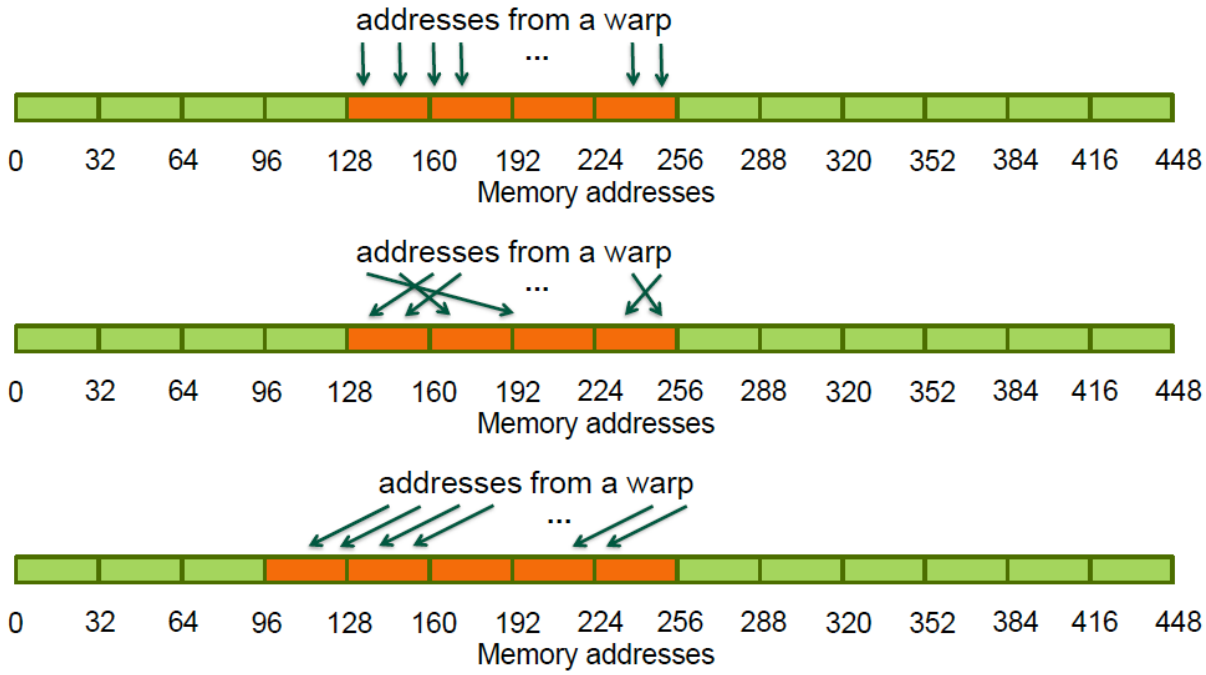

Figure 6: Diagram of warp alignment with memory requests. The above two cases feature 100% bus utilization, the bottom case has 80% due to misaligned memory accesses. [**NR**]

Fortunately, for finite-difference type operations warp alignment can be often be achieved by simple padding. In the case of applying an 8th order finite difference stencil out-of-warp access occurs due to the half-stencil width (halo) reaching outside of the warp, the case shown in figure 6. Padding the computational domain, at the beginning, by 28 points instigated a performance increase of almost 20% because this systematic shift was negated.

Whilst warp alignment is important for maximizing GPU performance potential, ensuring that the memory levels are used optimally is far more crucial. If, for example, an RTM implementation does not make use of shared memory then the GPU exhibits a worse performance than an eight core CPU node. This is entirely due to the aforementioned high latency of GPU global memory, despite the high level of parallelism. The implementation must aim to minimize any global memory access during thread execution. For the propagator the difference between CPU and GPU compute speeds can be seen in Figure 11. Note that the most naive GPU methodology, which does not use shared memory, is slower than the 8-core, blocked CPU code.

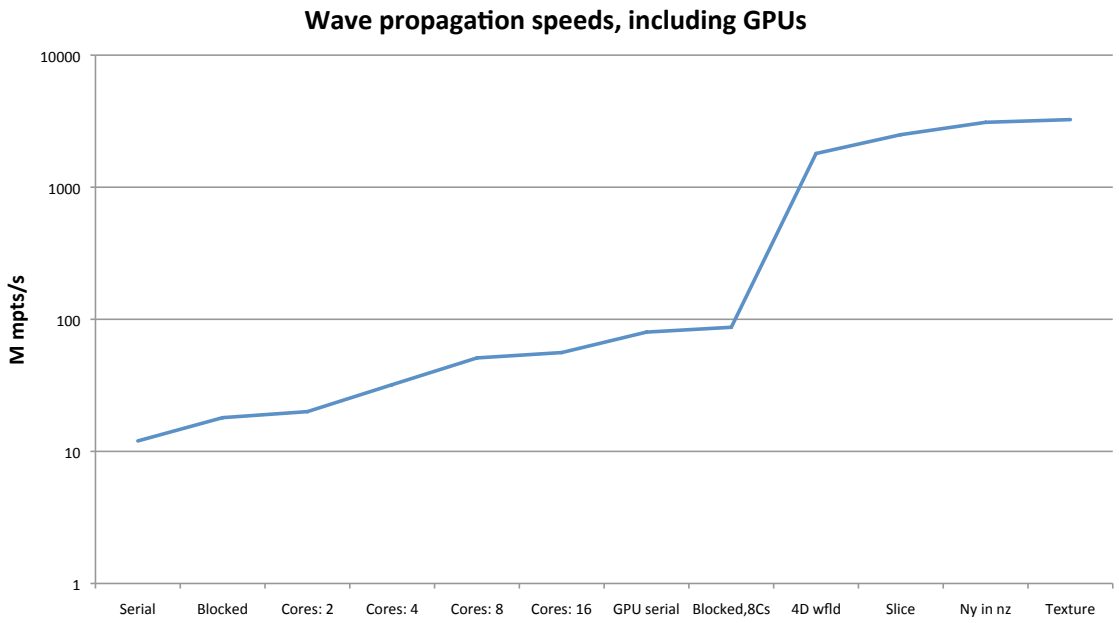

Figure 7: Compute speed (million models pts calculated per second) against implementation. [**NR**]

Random boundaries

A major bottleneck in RTM stems from the complex conjugate in equation **??**. The consequence of this is that the sense of time for the source wavefield is opposite to the receiver (data) wavefield, thus one of these must either be stored or propagated twice (**?**).

There have been multiple solutions to this source wavefield problem, such as source wavefield saving, checkpointing (**?**), compressive sensing and random boundaries (**?**). The most intuitive solution is the first of these - to simply save the entire source wavefield and read back the correct sections as necessary. This will often cause an I/O bottleneck as disk access can be very slow. This methodology when using a GPU is disadvantageous; the GPU can not read from the disk so all disk access must be first routed through the CPU. This creates even more of a bottleneck, especially since now propagation is accelerated, disk access appears as a larger time fraction. On the system used for these tests, disk access stagnates at around 200 Mbytes/second. Checkpointing is a way of favoring extra computation over disk access, and under certain circumstances can provide advantages for CPU based RTM. However it requires multiples source wavefield slices to be held in memory concurrently, and the global memory of the GPU cannot support this.

Random boundary based RTM favors extra computation over disk access; all disk based wavefield access can be mitigated. Generally, absorbing domain boundaries are used, this is to avoid high-amplitude, coherent and non-physical boundary reflections.

Simply put, any wavefield incident on the computational boundary is artificially removed before it can reflect. These conditions work well for artifact reduction, however the conservation of energy is violated and this propagation is not time reversible. If source propagation was time reversible then disk access during the adjoint process could be avoided. It is possible to instead pad the domain with an increasingly random velocity field. An incident wave will scatter incoherently as it approaches the boundary, and no energy is artificially removed.

Thus, by randomly padding the domain, propagating the wavefield to the final two time slices and saving these, it is possible to reverse the sense of time and exactly recover the initial wavefield (to within machine precision.) Practically, for RTM the source wavefield can first be propagated and the final slices saved, then when back propagating the receiver the source can be back propagated concurrently, and the imaging condition applied at each appropriate time step. This is summarized in algorithms 1 and 2.

Algorithm 1 RTM with source reading

```
while it < nt; it++ do
  Propagate source wavefield
  if imaging time step then
    Save source wavefield slice to disk
  end if
end while
while it > 0; it– do
  Propagate receiver wavefield
  if at imaging time step then
    Read source wavefield slice
    Correlate source and receiver
  end if
end while
```

Of course, this technique introduces significant quantities of random noise into the system. Fortunately, this noise is random and incoherent, and will stack out between time steps and between stacked shots. This noise reduction follows a random walk type reduction, and experience shows that by using ten seismic sources or more that this random noise has already been reduced to background noise levels.

A valid measure of how well a system is being used is to look at how much time is spent on computation and I/O as a function of the total time. Figure 8 shows such a breakdown for four separate methods (all GPU based). These are disk based RTM, asynchronous disk based RTM, CPU DRAM based RTM and finally random boundary based RTM. Only the last of these is computationally dominated. Given that the random boundary noise is rarely a problem, it can be concluded that this adaptation is the best suited for the given computers.

Algorithm 2 RTM with random boundaries

```
while it < nt; it++ do
  Propagate source wavefield
  if it=nt or it=nt-1 then
    Save source wavefield slice to disk
  end if
end while
while it > 0; it– do
  if it=0 then
    Read source wavefield slices
  end if
  Propagate receiver wavefield
  Propagate source wavefield
  if at imaging time step then
    Correlate source and receiver
  end if
end while
```

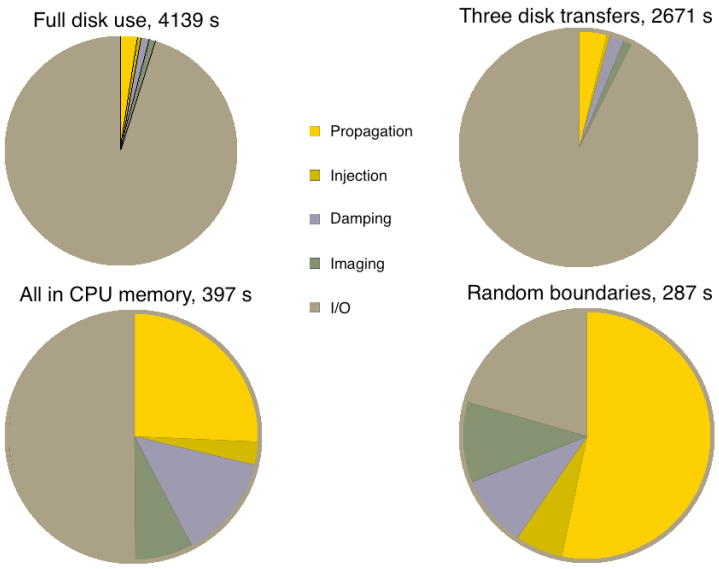

Figure 8: Visual representations of how I/O balances relative to other operations for four GPU based RTM scenarios: disk based wavefield access, asynchronous disk based wavefield access, CPU DRAM based wavefield access and GPU based random-boundary wavefield re-computation. [**NR**]

Extension to multiple GPUs

The bulk of this research was carried out in 2011 and 2012 when the Fermi M2090 cards and CUDA 4.0 were the state-of-the-art in GPU computing. For the following discussion the compute nodes used each contained 8 Fermi M2090 GPUs, each with 6 Gbytes of global memory (48 Gbytes combined), and 24 CPU cores, each with with 8 Gbytes global memory (192 Gbytes combined).

For the most simple RTM implementation seven fields of significant (3D) size must be allocated on the GPU - two source wavefields, two receiver wavefields, the velocity model, the image estimate, and the data. Since an individual Fermi M2090 card possesses 6 Gbytes of memory this limits the potential model size to just over $600pts^3$. Many contemporary imaging targets are far larger than this, so an approach that can accommodate larger models must be used.

The fact there are multiple GPUs per node can be taken advantage of. They have a combined global storage of 48 Gbytes, making the new potential image size $1900pts^3$. Through domain decomposition the full combined memory space can be utilized. Given a large 3D volume there are many ways to perform domain decomposition; for now decomposition into equally sized blocks is assumed. Due to the 'tree' layout of how the GPUs are linked it only makes sense to split the domain along a single axis, then the overlapping region for a given sub-domain lies on the GPUs 'left' and 'right' of the current GPU. Furthermore if this decomposition is performed along the slowest axis then this overlapping region (needed for communication) is contiguous in memory. Figure 9 shows this for the situation where the y-axis is the slow axis. The domain is split evenly along y and the overlapping regions for, say, GPU 1, lie on GPU 0 and GPU 2. These sub-domains overlap by the stencil width, so that the convolution can be applied correctly across these boundaries.

How these regions overlap is shown in Figure 10, the overlap is equal to the stencil width and the half stencil width region is often referred to as the 'halo'. To apply the convolution each GPU needs its halo region to be updated; it receives these values from its neighboring GPU(s).

An additional benefit of CUDA 4.0 is the ability to perform calculations and memory transfer operations simultaneously. Thus, by staggering some of the calculation it is possible to almost entirely hide the communication (halo update) part of the propagation. Initially, each GPU will run a kernel that only calculates the values of the region that it's neighbors will need (their halo region.) So GPU 0 will calculate the halo region of GPU 1, GPU 1 will calculate the right-side halo region of GPU 0 and the left-side halo region of GPU 2 etc. (Figure 10). Once all these transfer regions have been calculated then a new set of kernels can start running to calculate the rest of the wavefield (internal regions). While the internal calculation is being performed a set of asynchronous memory copies can be initiated, which transfers the halo values between GPUs to their appropriate spots.

The connections between GPUs are known as PCIe links, and these are duplex.

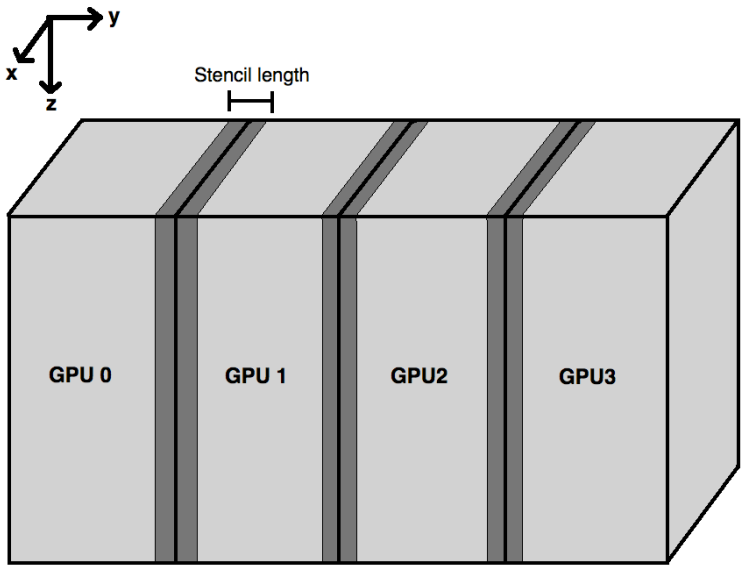

Figure 9: Breaking the computational domain across multiple GPUs, dark grey indicates the overlapped area between neighboring GPUs. [**NR**]

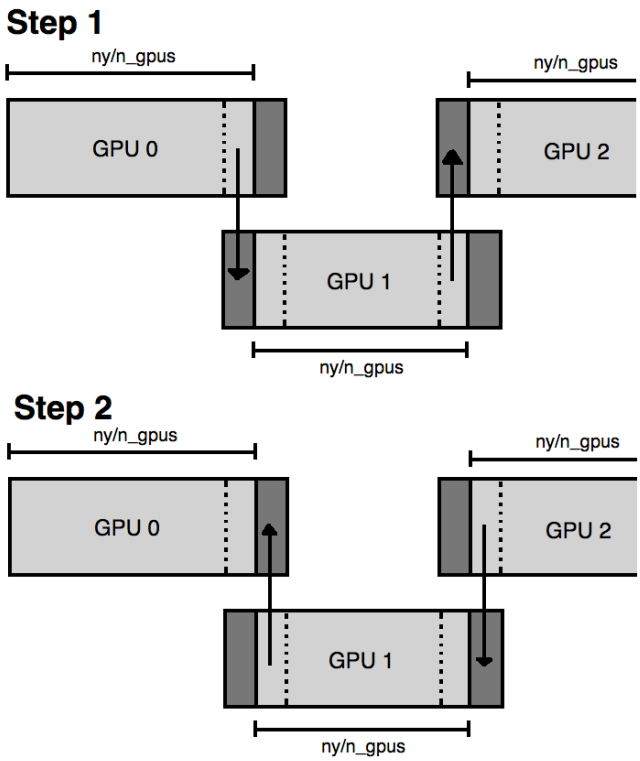

Figure 10: Staggered representation for halo region transfer. Initially all GPUs send to the 'right' and receive from the 'left,' then the order is reversed. [**NR**]

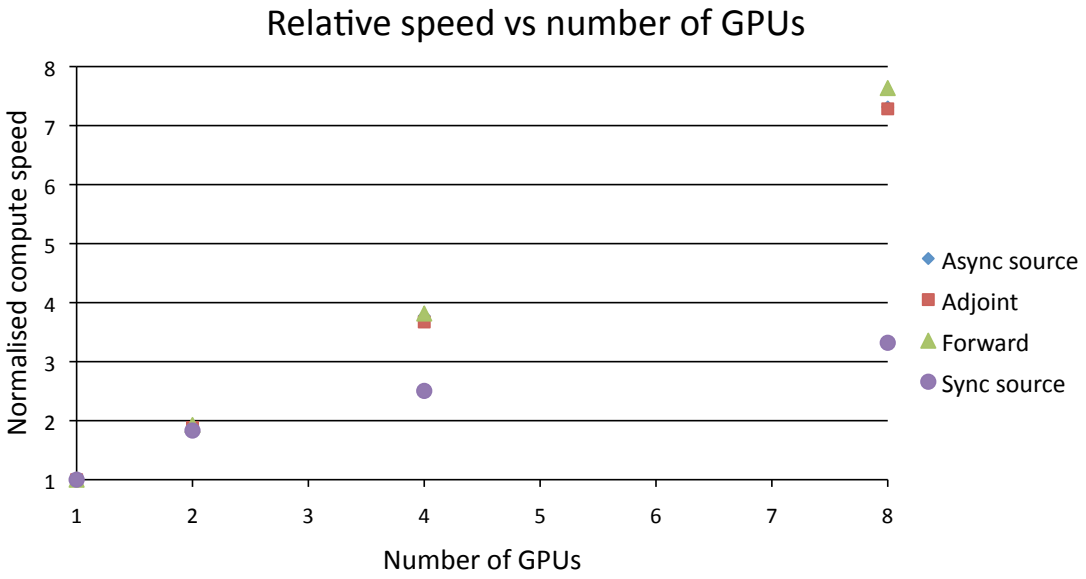

Figure 11: How the individual linearized inversion operations scale with the number of GPUs used, relative to normalized computational speeds. [**NR**]

Thus they can send and receive at the same time, as long as the directions are opposite. GPU 1 can receive information from GPU 0 and send information to GPU 2 at the same time, but could not send and receive to GPU 0 at the same time. Consequently, to complete the full set of transfers initially each GPU will send to the right (higher GPU index) and receive from the left. Then, send left and receive right. After this has completed each GPU will have its halo regions updated to the correct values. If this communication time is less than the time to complete the internal calculation then the communication is effectively hidden - there is no waiting time involved with the decomposition scheme. There is a small overhead cost associated with splitting the halo and internal calculations, but this is just a few percent. The sequence in abridged CUDA can be seen 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();
}
```

To judge how well the communication-hidden domain decomposition works one can begin with a model size that fits on a single GPU. The same imaging procedure can then be used for two, four and eight GPUs and their normalized compute speeds can be plotted as function of the number of GPUs used. This is known as a 'stong scaling' test. Figure 11 shows the results. In purple the speed-ups observed without communication hiding are plotted. The speed up quickly saturates, similar to the naive OpenMP efforts discussed earlier. However, for both the propagation (blue) the full forward process (green) and the full adjoint process (red) the speed up is very close to linear. This means the communication has been effectively hidden and decomposing a problem across 8 GPUs will give close to 8x the computational speed.

BEYOND ACOUSTIC GPU PROPAGATION

The majority of this thesis study used acoustic wave propagation, to describe how energy moves through the Earth. As discussed in Chapter 5, the field dataset used was acquired over a section of the Earth where these acoustic assumptions were too inaccurate. The provided models described the target area as Tilted Transverse Isotropic (TTI), which results in a more computationally intensive propagation kernel, and many more significant allocations necessary.

The full mathematical description of TTI based propagation involved many details unnecessary to this thesis discussion, but an approach as described by **?** was used, and a kernel written by Paulius Micikevicius was implemented.

The major issue with adapting this provided kernel was that of memory allocation. The test example was over a very small model, where many 3D field allocations was not a concern. For acoustic propagation, seven large fields must be allocation - two receiver wavefield snapshots, two source wavefield snapshots, the image, the velocity model, and the recorded data. TTI propagation decomposes both sets of wavefields into two components, requires an intermediate derivative calucation for both wavefields, and uses four additional Earth model components. This results in eight allocations per wavefield, and a total of 23 significant allocations.

This many allocations over 8 GPUs, in terms of memory requirements, is analogous to four times the memory needed for acoustic propagation. Fortunately, a few tricks could be used to make 3D GPU based TTI propagation viable for such a large dataset.

Firstly, these aforementioned intermediate derivative fields are not updated between propagation steps, they depend only on the inputs for a given time. This

means the computation for the source wavefield and the receiver wavefield can be performed sequentially, and the intermediate fields zeroed and re-used. This saves two allocations, and only increases propagation time by a fraction of a percent.

Secondly, the four anisotropic parameters do not require a large dynamic range for storage. The Thomsen parameters (**?**), ϵ and δ, vary between 0 and 0.3, with a required precision of two decimal places. The angular parameters, ϕ and θ, very between 0 and π, with a similar required precision. CUDA provides support for half-integer storage, and by storing these parameters in unsigned-short arrays, all four fields take up the equivalent space of two fields - saving another two allocations.

Thirdly, the full shot does not need to be allocated on the GPU at all times, only the necessary time slices for injection and propagation. This means the shot can be streamed as it is propagated, requiring much less GPU space. To account for the fact that the data is more coarsely sampled than the necessary propagation interval, an eight-point since interpolator is used. Thus, at the minimum, only eight time slices of the data need to be resident on the GPU. However, if only eight slices are used then there must be CPU-GPU transfers at every imaging time step, as the next data time-slice will need to be transferred. Instead, a block of, say, 100 slices can be allocated on the GPU. Once these slices have all been propagated into the model, this array can be flushed and re-filled with the next hundred slices. This method increases overhead by a few percent, but makes the data allocation essentially negligible.

For the given dataset, these three techniques reduced the necessary allocations to the point where imaging was now viable.

EXTENDED BORN MODELING

Chapter 4 referenced that extended Born modeling posed an additional computation issue, when compared to extended RTM. This is, again, due to the asymmetry of the CPU-GPU compute nodes.

During extended imaging, there is some flexibility when it comes to implementation. Each shot is independent, as are the source and receiver wavefields, and the output image is stacked over time, t. This means, given a CPU-GPU node, it is possible to overlap the propagation of one shot with the imaging of another. The first shot, S_i, is propagated on the GPU, and the wavefields are saved (either to CPU DRAM or disk, depending on size.) The next shot, S_{i+1}, can begin propagation on the GPU. Concurrently, a series of CPU threads can perform the extended imaging for shot S_i. This overlapping of imaging and propagation can result in significant savings.

For extended Born modeling, the source and receiver wavefields are not independent. Propagating the receiver wavefield is contingent on the source wavefield, after convolution with the image. As a result it is not possible to overlap the convolution of one source with the propagation of another.

CONCLUSIONS

There are many options for accelerating linearized optimization, and the best approach depends strongly on the computational system at one's disposal. For a set of multi-core CPU nodes then a straight OpenMP solution to accelerating the computation quickly saturates as a function of cores. By breaking the domain into sets of blocks to attempt a better cache performance a performance increase is seen, however it is far from the theoretical performance.

GPUs are a clear option since they are designed to perform with algorithms capable of fine-grain parallelization, such as time-domain finite-difference wave propagation. For single card performance fast speeds are observed for the propagator alone, but for use with RTM the source wavefield I/O problem must be addressed. Disk access during propagation can be mitigated by using a random boundary condition and favoring computation over communication. Such a scheme is the only implementation that is dominated by compute rather than I/O, giving excellent performance compared to the parallel CPU scheme.

The memory size of a single GPU is restrictive. By attaching several GPUs to a single node it is possible to break the computational domain across multiple GPUs, allowing larger models to be used. Such a method requires communication between regions between time steps, which can result in performance saturation with additional GPUs. By breaking the computation into multiple steps and using asynchronous memory transfers it is possible to entirely hide the communication, resulting in a close to linear speed up with additional GPUs.

ACKNOWLEDGMENTS

Special thanks to Paulius Micikevicius, formerly of NVIDIA, for sharing example codes and partaking in many insightful multi-GPU discussions.