GPU applications for modelling, imaging, inversion and machine learning

Daniel Trad

ABSTRACT

There is a gap in best-practices between research in industry and academia. One of the weaknesses of research in academia is that tests are often carried out on very few data sets, missing the rich feedback provided by variety in testing. In industry, on the other hand, we are forced to work with different datasets. This data diversity brings a significant inflow of information that benefits research and development. In academia, it is hard to get new datasets and the necessary preprocessing makes it difficult and time-consuming to achieve testing diversity. Another difference is that industrial environments may have at least one or two orders of magnitude larger computer resources than academia. This report is about reducing these differences by creating an environment where students can, with limited resources, quickly model, migrate and invert seismic data, starting from known models or even simple pictures of models, that can easily be found in publications.

Modeling seismic data is a key part of research for acquisition design, imaging, full waveform inversion and machine learning. From the convolutional model to more sophisticated wave propagation methods with anisotropic visco-elastic 3D wave equations, there is a wide variety of approaches to simulate seismic data in different geophysical models. The more sophisticated the approximation, the more realistic the events we see on the simulated data. The same applies to inversion methods like migration and full waveform inversion (FWI). The efficiency and accuracy of modeling directly translate to better reverse time migration (RTM) and FWI. Although there are different options for wave propagation modeling, the finite difference method (FD) is most commonly applied because of its good trade-off between accuracy and efficiency. In this report, we discuss and illustrate how Graphics Processing Units (GPU) implementations for FD using a convolutional pattern can help in closing this research gap mentioned above.

INTRODUCTION

Modeling seismic data is a key part of research for acquisition design, imaging, full waveform inversion (FWI) and machine learning (ML). From the convolutional model, to more sophisticated wave propagation with anisotropic visco-elastic 3D wave equations, there is a wide variety of approaches to simulate seismic data in different velocity models. The more sophisticated the approximation, the more varied the types of events we see on the data, but also the higher the computational cost.

The FD method is one of the most commonly used for modeling, especially structural modeling but even stratigraphic modeling (SEAM models) because it provides a wide range of options and its accuracy is sufficient for almost all applications. We use it for simulating surveys and forward and reverse modeling for RTM and for FWI. The applicability of these techniques depends strongly on the computational cost of FD, because usually a large number of modeling steps are required for iterative inversion. In fact, we could say that the

computational cost of FD controls what we can do in research. Large elastic 3D FWI is prohibitive slow except when done on very large computer clusters. Although cloud computing provides a way to solve this issue in principle, still we rely heavily on local resources to do development and initial testing. Therefore efforts in decreasing computational time for finite difference modelling have been a key research topic since the first time it was used for seismic.

In addition, we are on the verge of a different and powerful technology that requires modeling even more than traditional techniques: machine learning (ML). This approach promises to be significantly more flexible for solving many problems than physics based approaches. While typical forward modeling requires us to write the specific rules nature follows, ML approaches have the potential to extract and implement those rules directly from the data. In the first case, we use physics to constrain the range of possible outputs that can be obtained; in the ML case, we allow every possible output to occur and we use pruning by training to eliminate non-physical possibilities. The ML approach, pruning by training, is highly dependent on the existence of abundant amounts of data in quantities never required before. Therefore, more than ever, we see that forward modeling to produce training data is the constraint on what can be achieved.

To address this issue, we can consider several approaches, which are combined in real applications:

1. Obtain more powerful computers, usually in the form of clusters. Cloud computing seems to have become a more affordable approach but is still not cheap. Large clusters require a large amount of energy for power supply and refrigeration.
2. Design better algorithms for both modeling and inversion, so that for example, we can decrease the number of inversion iterations by applying proper preconditioning and regularization. These often lead to a reduction of computation time in small percentages since the major advances in algorithmic design are usually already in place. For example, we could think of 10% or 20% reduction time by adding a preconditioner.
3. Use the advances of hardware in the domain of computer science by parallelization. This is often thought of by geophysicists as a different domain, and this approach is left to be dealt with later by a specialized developer. However, parallelization techniques typically require a major restructuring of the software. Therefore it is reasonable to incorporate parallelization into early development. In addition, research requires significant testing, and therefore the earlier we can accelerate the algorithms, the more this approach can benefit research.
4. Use radically different new technologies, like ML and Quantum Computing (QC). This can well become extremely important for modeling in short (ML) to long term (QC). Although ML is also considered as a faster radical new technology, when the need for training is taken into account the need for modelling grows combinatorially.

In this report, I will discuss option 3. The discussion will not be leading edge computer science, nor will it be too illuminating for true experts in High Performance Computing.

Instead, my intention is to explain to geophysicists some of the reasons why we can't ignore advanced parallelization techniques and leave them as an afterthought. I will show several applications where the speedup is between 50-100 times. We will summarize why and how that happens.

HIGH PERFORMANCE COMPUTING

High-Performance Computing (HPC) in the field of computer science deals with, among other things, the techniques for parallelization of algorithms. Originally a specialized discipline for software engineers and programmers acting in the background of major scientific achievements, now it has become a necessary skill for geophysical researchers. This is in part because parallel hardware is no longer a specialized tool available only to major processing companies but a common tool in regular desktop computers. Ignoring parallelization is restricting oneself to using a small fraction of computer resources.

Twenty years ago, our desktops had one CPU, and parallelization was only used in some cases to deploy expensive algorithms like migration or tomography onto clusters. This was only done for certain processes that justified the effort from a specialized team of programmers. After a code was developed by a researcher, HPC developers would re-factor this program to apply parallelization by distributed memory. This led often to changes in results, and there was a significant delay on prototyping + testing + industrialization + testing + deployment, a cycle typically done in an iterative fashion. Sometimes the researcher would write parallel code directly, but in general, there was a separation between these tasks. If a code was not deployed into clusters, researchers relied on Moore's law, which promised that the CPU speed would double every two years. Therefore the second approach mentioned above, better algorithm design, was extremely important. However, even Moore's law has to obey physics: quantum effects and heat dissipation put an end to this bonanza. HPC and parallelization became the key for speed and still is.

The most commonly used HPC parallelization approach is distributed computing for clusters. In this model, we have a modest number of nodes, typically around 30 or so, and each node would act as an independent computer. The programmer would make them work simultaneously on the same task by using libraries like Message Passing Interface (MPI) (Gabriel et al., 2004) that would create a main program controlling the dataflow, typically run on the head node (called master). The master reads the data, defines parameters and controls or commands the other nodes, named slaves. The main overhead is passing the data to the nodes and collecting intermediate calculations. An example of such a dataflow for a migration algorithm is shown in Figure 1. Advanced clusters, with a larger number of nodes and very fast communication across them, can perform these operations very efficiently but they are at least one order of magnitude more expensive than the common clusters used in production environments (known as Seawolf clusters). For regular production clusters, the communication across nodes and master take a significant toll on computation time so the effort is focused on decreasing this communication time. This is usually achieved by doing coarse parallelization, for example sending complete shot gathers to nodes and letting each node do a complete geophysical operation like migration or modeling. This is more efficient than dividing the model into small areas. Figure 1 illustrates this type of coarse-grained parallelization where each node works with a section of the seismic survey

(data space) instead of a section of the migrated section (model space). Splitting the data across nodes is also better than distributing the model because surveys are larger than the models.

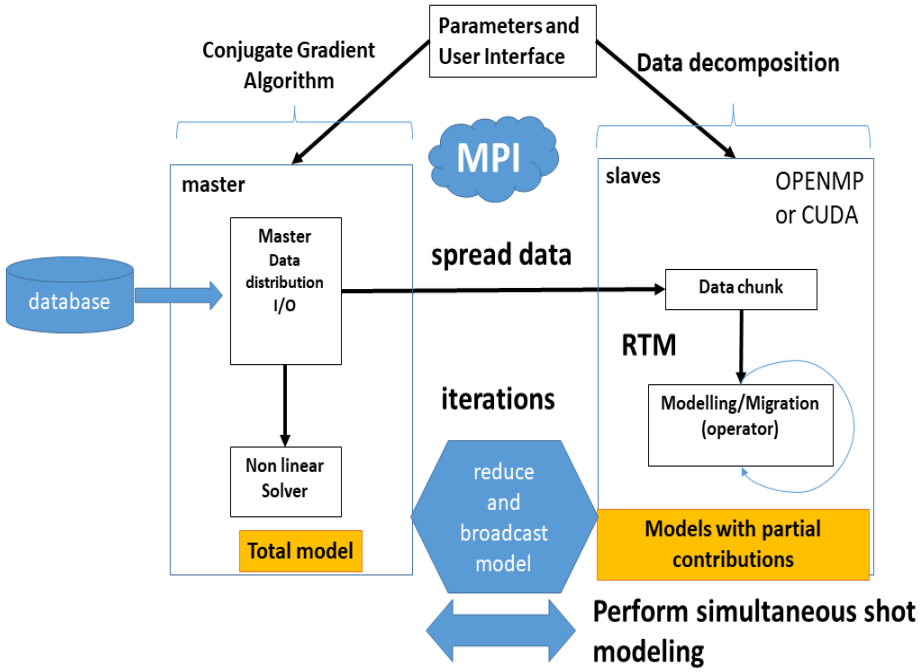

FIG. 1. A typical HPC dataflow with object oriented design for Least Squares migration and or Full Waveform Inversion.

Around 10-20 years ago, we saw a proliferation of multi-core CPUs. The number of CPUs on a single desktop went from one or two to a dozen or several dozens. This hardware is the norm now, and it is not possible to find a single-core CPU any longer. Tools like OpenMP (Dagum and Menon, 1998) make it fairly easy to use multi-core CPUs because these devices have shared memory and therefore all threads share a common computation space. Although this model is the easiest to use, it has a limitation on scalability as a consequence of relying on cache memory for fast access to the DRAM. Variables updated in one thread may have to be accessed by another and to do this efficiently with the cache implies constantly updating it. Shared memory parallel programs are not hard to write and cache memory achieves excellent speedups. However, this only happens for variables with spatial and temporal locality. It is up to the programmer to make sure this happens. In addition, the programmer is responsible for avoiding other common parallelization problems like race conditions and deadlocks. A big advantage of this type of parallelization is that it can often be achieved with minimal changes to existing code, but this can not be said about distributed memory CPU or GPU codes.

The combination of shared and distributed memory with multi-core nodes is standard these days for any processes that require heavy computations. They are sometimes referred to as the hybrid model. This parallelization gives flexibility when running parallel programs because, by changing the ratio thread/nodes we can adapt to different hardware to achieve optimal scalability.

This model heavily relies on:

1. Fast communication across nodes (for the distributed memory part).
2. Temporal and spatial locality of variables (for the shared memory part).
3. Large/fast cache memories and efficient mechanism for updating them.
4. Programmer skills to separate the calculations across nodes and threads.
5. Programmer skills to minimize overhead on memory transfers.

This trend is still dominant in our days, but around 10-15 years ago, thanks to the release of the "Compute unified device architecture" (CUDA) (Nvidia, 2007), a new philosophy for parallelization became widely available: the use of Graphics Processing Units (GPUs). These devices, originally designed for controlling the pixels on displays, hide latency (the delay in data arrival) by splitting calculations across thousands to millions of threads (light weight units of execution). They also have cache memories but that is not their main mechanism for latency hiding (Cheng et al., 2014; Han and Sharma, 2019). From the hardware point of view, this philosophy is possible because the capability demand for each thread is much smaller than for CPU threads. The threads on GPUs do not perform the heavy optimizations that CPU threads do because they were originally designed for simple operations involving pixels. However, as we well know from the evolution of computers, a large number of simple operations can perform very complex tasks. GPU programs rely on:

- A main dataflow controlled by the CPU. This is necessary because GPUs have limited capability for many operations, so a CPU is necessary to control the dataflow.
- The existence of a large number of parallel operations that also each support small vector parallelization in what is called Single Instruction Multiple Data (SIMD).
- Programmer skills to move information from/to GPU memory with minimum overhead. As in the other parallel models, this transference has to be minimized or all the benefits of parallelization will be lost.

Figure 2 (Han and Sharma, 2019) shows a comparison of these two different models. GPUs require fine-grained parallelization, meaning that operations are subdivided into fundamental units. CPUs on the other hand, are designed for coarse scaled parallelization. They can handle complicated optimizations on a larger scale than GPU threads can. The CPUs and GPUs can be combined in heterogeneous computing models (Figure 3) where data moves across the two models as it fits for optimized calculation.

Although in principle, this is not very difficult, the main complications of working with GPUs come from the hierarchy of memories that GPUs provide. CPUs also work with different memories (DRAM, L3, L2, L1 caches, registers in order of increasing speed and decreasing size) but they also control them. GPUs programming languagues more often instead rely on the programmer's skills to place the variables on the optimal memory components. Figure 4 shows the GPU memory model and the different bandwidth on each

(Han and Sharma, 2019). We can see that shared memory has a memory bandwidth of 20000 GB/sec, which is more than 100 times faster than CPU DRAM. If we can use shared memory instead of DRAM we can achieve a 100 times speed up. Which other processes do we know promise speedups of that order? Except for quantum computers in the future, near but not present, we can achieve only modest speed-ups by using faster algorithms or pre-conditioners on CPUs.

As illustrated in Figure 3, in addition to the memory hierarchy, GPUs can support also a special arrangement of threads in blocks (which can be 1D, 2D or 3D) (Cheng et al., 2014). Blocks are arranged in grids, which can also be 1D, 2D or 3D (Figure 3). Grids are arranged in streaming devices. Typically a working space is divided into blocks and grids, and different algorithmic sections can be sent to different streams (sequence of operations in the GPU). In addition, GPUs are also arranged in clusters, where each node can have multiple GPUs. The reason for all these hierarchies is to achieve an efficient distribution of information and memory for large computations. The proper use of all these memory and computation hierarchies is what makes programming for GPUs much more challenging than other types of parallelization.

The convolutional pattern

Differently from other parallelization techniques, GPU programming requires some knowledge about the hardware because much of the responsibility of data distribution is on the programmer. One way to simplify development is to search for similarities between our target problems and existing computing patterns developed by experts. The particular problem of explicit finite differences happens to be similar to the convolutional pattern, which is very common for example in the development of neural networks for computer vision. In fact neural network applications would not be practical without the efficiency of the convolutional pattern implementation for GPUs.

Although in principle, parallelization by simultaneous computations with the thousands or millions of threads a GPU has to offer can lead to significant speedup, in practice efficiency is decreased by overhead time of moving variables from CPUs to GPUs and back. This is the case when the ratio of computation/memory is close to one, meaning variables are used only once per transfer. To get peak performance on GPU calculations, we need to perform many calculations for each transfer. One way to achieve this is by reusing memory variables using a GPU memory called global memory, which is accessible to all threads in the GPU. However, this memory is not very fast. A much better alternative is to use what is called "shared memory", which is 30 times faster (see Figure 4). This shared memory is visible to only threads in the same block and its size is small. Therefore we can't use it directly to store an entire wavefield. Instead, the working arrays have to be subdivided into small tiles, which are distributed across blocks (Figure 5). Since each block has its own shared memory, variables can be reused in an efficient manner without the usual problems of cache coherence (updating cached variables across muliple copies). This subdivision of arrays into small tiles is similar to the type of model decomposition applied for distributed memory models (MPI). Each tile has to have an overlapping area with neighbour tiles to share boundary conditions (Figure 5). Being able to perform computations on variables stored in the shared memory is the main reason why we can achieve $100\times$ speedups.

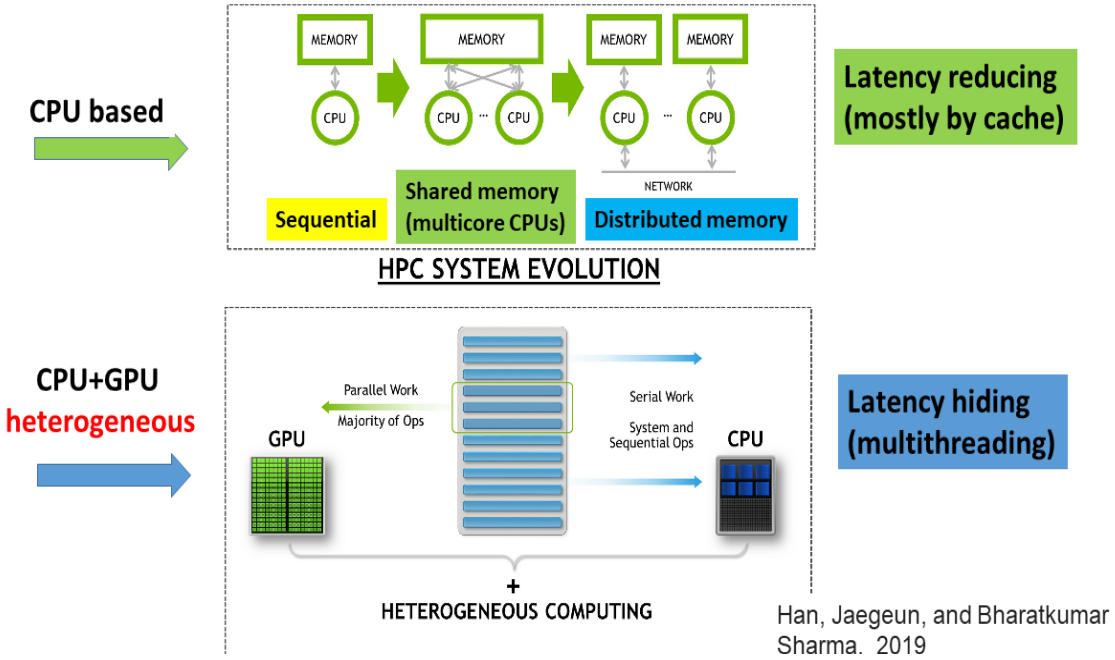

FIG. 2. The different philosophy underlying the CPU and GPU models. CPUs use cache memory to hide latency, GPUs use thousands of threads instead. (Adapted with permission from Han and Sharma (2019))

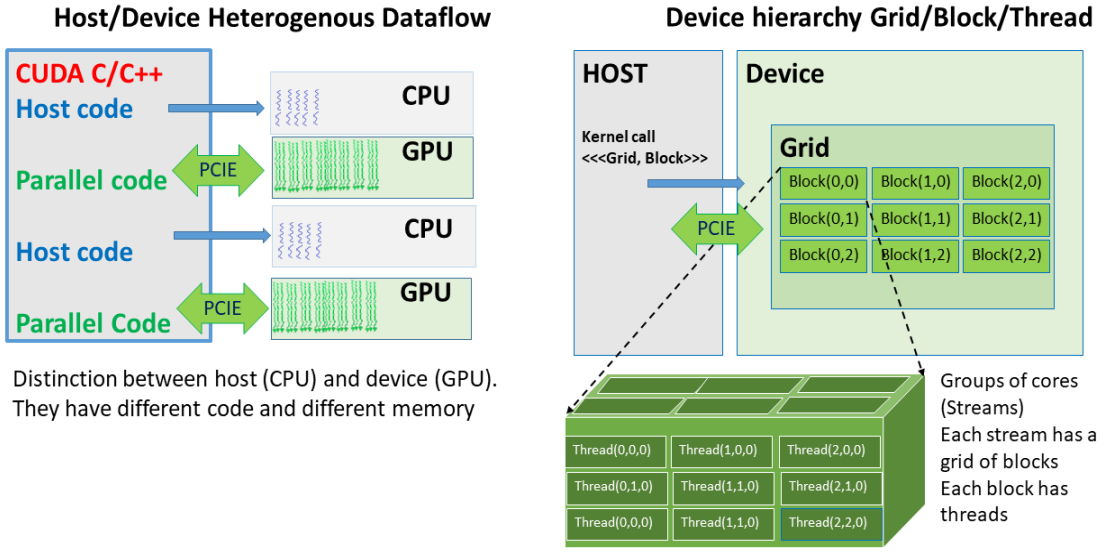

FIG. 3. GPU computational model. Left: CPU+GPU combined dataflow, Right: GPU organization in terms of grid of blocks and blocks of threads (based on Cheng et al. (2014)) .

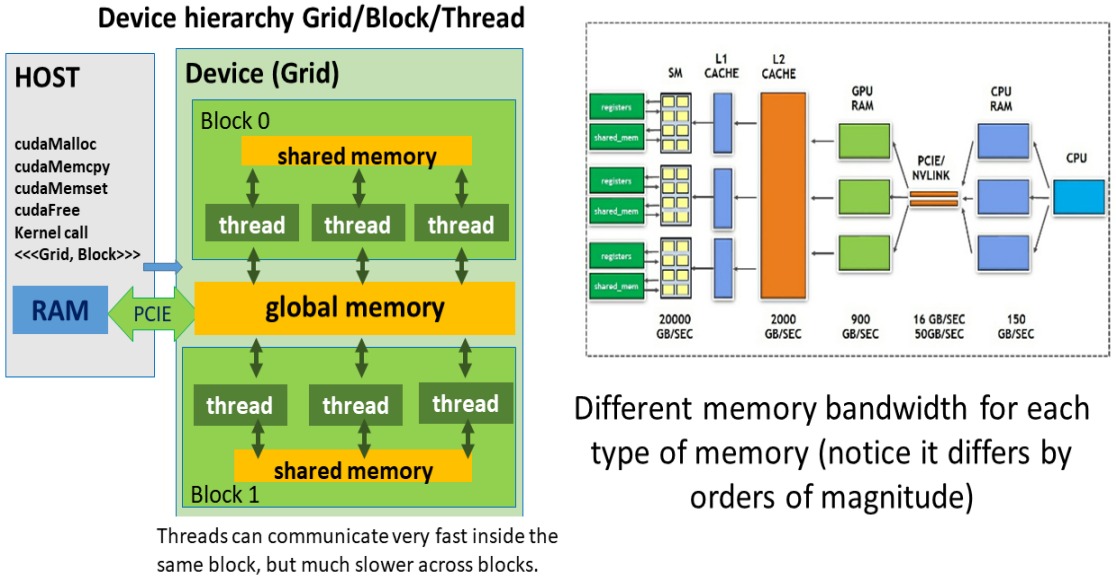

FIG. 4. The hierarchical memory model in GPUs. Left: differences between shared and global memory from the thread and blocks point of view (based on Nvidia website). Right: The travelling of information across memories and different bandwidths (reproduced with permission from Han and Sharma (2019)).

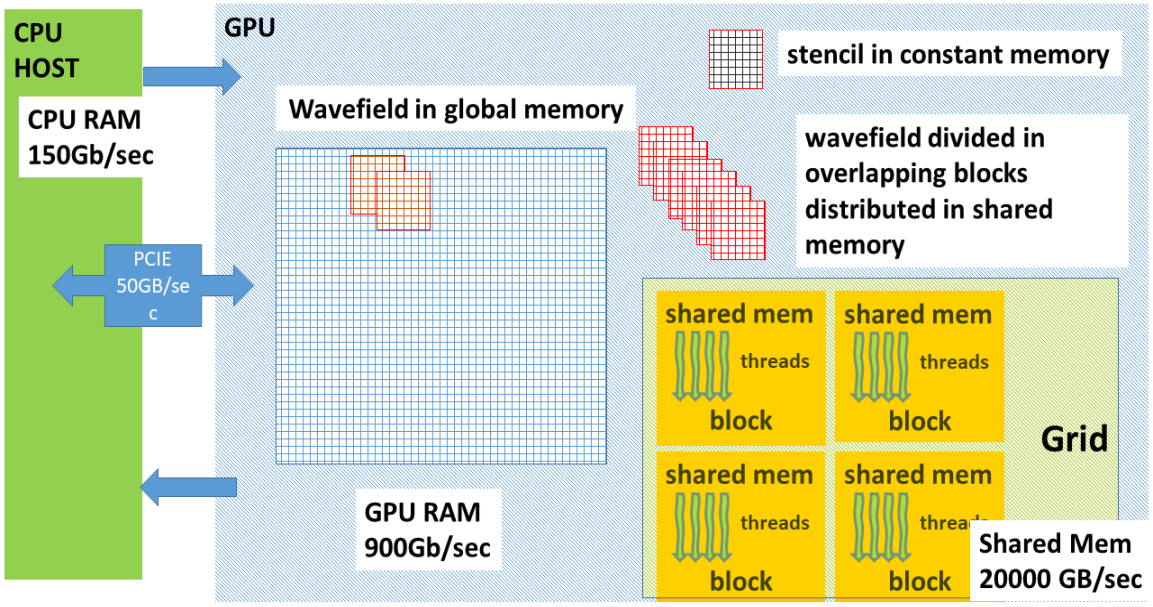

FIG. 5. The convolutional pattern for finite difference modeling. The wavefield is distributed across overlapping blocks where the shared memory is 100X faster than the CPU RAM.

In summary, the finite difference algorithm can be solved on GPUs two orders of magnitude faster than on regular CPUs by using the convolutional pattern (Figure 5). The wavefields, originally allocated in CPU RAM memory with a bandwidth of 150Gb/sec, are copied to the GPU RAM memory, which is significantly faster (900Gb/sec). That copy is done through the PCI-Express which is relatively slow (50Gb/sec). The GPU takes the wavefield and decomposes it into a series of blocks that lie in a grid. Each block has hundreds of threads, but most importantly all the threads inside a block have access to a very small but very fast memory called shared memory (with a parallel bandwidth of 20000Gb/sec, that is as fast as a CPU register). Although all blocks can access variables on the GPU global memory, they can't see each other's shared memory. Therefore to be able to calculate a finite difference wavefield that is continuous across blocks we need to have overlapping regions. The wavefield is therefore calculated very rapidly by splitting the computations across thousands of threads.

MODELING

Modeling is an essential part of a research environment, either as an alternative to real data or as a prediction tool for inversion methods and machine learning training. Among others, Finite-difference (FD) approximations are a favourite modeling tool when the geological complexity and the needs for multiple reflections and other complex wave phenomena make convolutional methods less desirable.

FD can be computationally very expensive in CPUs, even with proper parallelization and it becomes a bottleneck when considering more accurate models, like visco-elastic anisotropic 3D. We often have to settle with poor approximations of wave propagation because of the computational cost. One way to achieve fast modelling is to use GPUs which require CUDA (Nvidia, 2007) or other specialized languages (e.g. AMD HIP) and libraries. The achieved speedup justifies the extra effort, particularly when proper use of the fast GPU memories is possible (shared memory). Figures 6, 7, 8 and 9 show acoustic modeling examples for 4 different velocity models and the calculation times for each shot. From runtimes of 0.3 sec in Marmousi to 6 seconds in Sigsbee, we see that hundreds of shots can be modelled in a few minutes. Although these examples are acoustic 2D models, putting them on the low cost of the spectrum, the achieved GPU/CPU speedups of 100 times bring significant advantages for testing and research.

Figure 10 shows a computing time comparison for creating 10 shots on the Marmousi model using different orders and different parallelization strategies. All times were measured on the same desktop computer: a CPU hexacore i7, 2016 and a GPU RTX2070 (2019). The 3 strategies compared are:

- CPU using C++ OpenMP with shared memory parallelization with 12 threads.
- CPU with C++ OpenMP+OpenMPI (hybrid model) with each thread simulating a node.
- GPU with C++ CUDA using GPU shared memory (convolutional pattern).

The test shows computing times for 2nd, 4th, 8th orders in space. All the times are for

10 shots, to balance initialization overhead in GPU and to have enough workload for CPU distributed memory. Depending on the order, we see that the hybrid model is more efficient than the multithreaded model by 2-3 times. This is common because the hybrid model has fewer issues with cache coherency than purely multithreaded models (each process uses a different memory space). The GPU simulation was 20 to 100 times more efficient than the hybrid model (and therefore 40 to 300 times more efficient than the multi-threaded version.

Superlinear scaling for GPUs?

In Figure 10 we see that the computation cost for the GPU is constant as the approximation order increases. The reason for this is that the FD complexity in this test is well below the GPU capacity. Therefore increasing the complexity from 2nd to 4th and 8th order uses more GPU resources but not more time. This can be seen in other aspects of modeling. For example, Figure 11 shows modelling in CPU (4th order) and GPU (8th order) when changing from sponge boundary conditions (ABC) to Perfectly Matching Layers (PML). For the CPU, when changing from ABC to PML the computation time doubles because PML requires staggered grids with modelling for the pressure and the 2 velocity components (for 2D models). This is roughly double the amount of work; the runtime for the CPU multithreaded version almost doubles as expected, but for the GPU, the time increase is only around 18 percent. We see a phenomenon similar to what in HPC is called "superlinear scaling", meaning that the speedup achieved by parallelization is more than expected (Ristov et al., 2016). This suggests that we can increase the accuracy of our approximations, for example, TTI visco-elastic and so on before we see an increase in computing time. Going from 2D to 3D (work in progress) may be somewhat more expensive than just increasing complexity in the same dimensions because of the additional complexities of moving 3D wavefields from global to shared memory. However, switching to more modern or bigger industrial GPUs will allow us to keep increasing the quality of the approximations without major computing penalties. Industrial GPUs have far more memory and threads capacity than the GPU used in this report. We will see this phenomenon again in this report for multigrid FWI.

Strictly speaking, what we see here is not the same phenomenon as superlinear scaling, but it has a similar effect. Superlinear scaling is a complex phenomenon that depends on hardware details beyond the scope of this report. The spatial organization in memory of the variables used in the calculations and the temporal order of these calculations have far-reaching consequences for cache coherency in CPUs and a tremendous impact on computing times. In GPUs, the problem is different since they rely much less on cache memory and more in multithreading to hide latency (Figure 2). The effect is that with GPUs we can often improve our approximations without additional cost. This effect is similar to that of superlinear scaling, but for a different reason.

Still, I think I can justify the term superlinear speedup for GPUs for two reasons. First, since GPUs rely on multithreading and CPUs in cache coherence to hide latency, we can say that superlinear speedup achieved in CPUs by better cache coherence is similar to superlinear speed up in GPUs achieved by better utilization of threads. The second reason will be discussed later when we see that computing time for multigrid FWI is much less than linear with the number of operations for GPUs although is close to linear in CPUs.

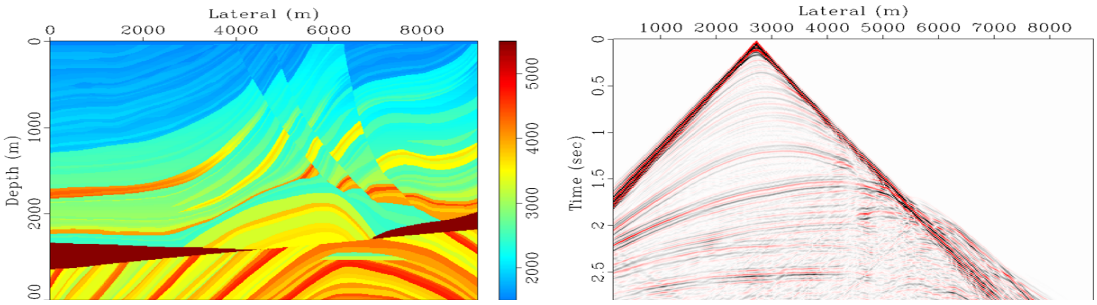

FIG. 6. Synthetic shot in Marmousi model, 376x1151, 8th order, time per shot 0.3 sec.

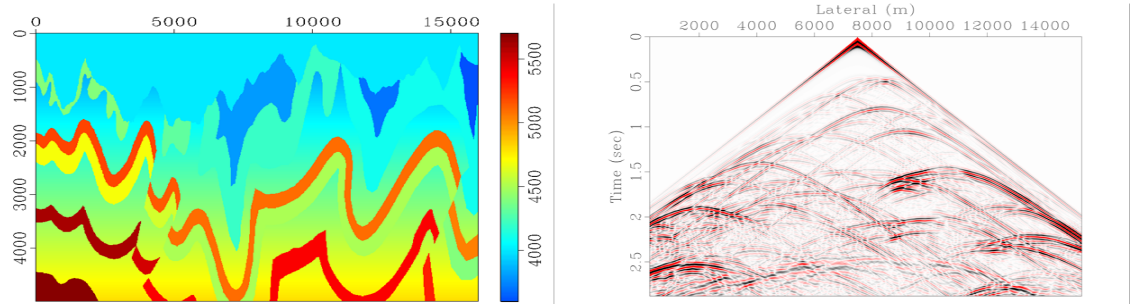

FIG. 7. Synthetic shot in Foothills model, 500x1600, 8th order in space, time per shot 1sec.

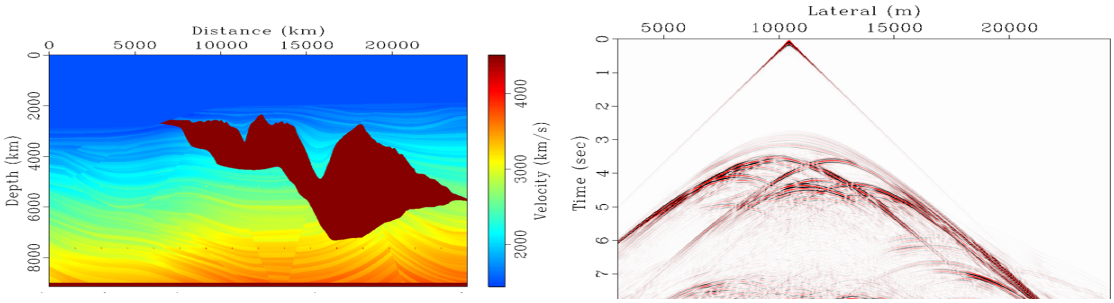

FIG. 8. Synthetic shot for Sigsbee model, 1200x3200, 8th order in space, time per shot 6sec.

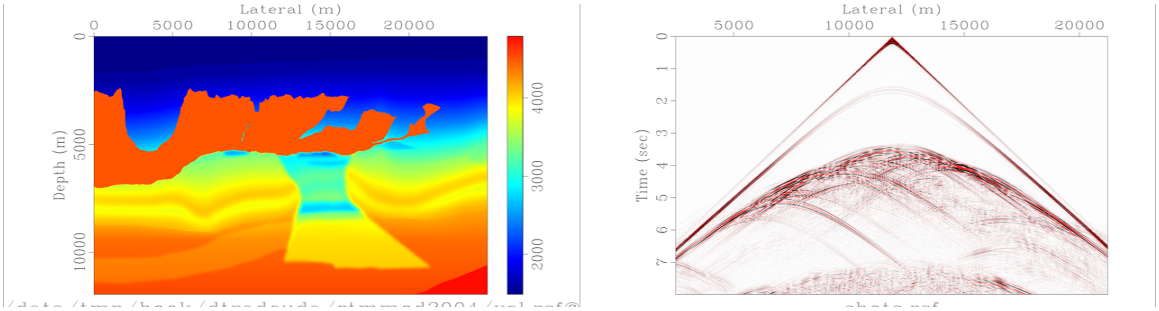

FIG. 9. Synthetic shot for BP2004 model, 956x2000, 8th order in space, time per shot 2sec.

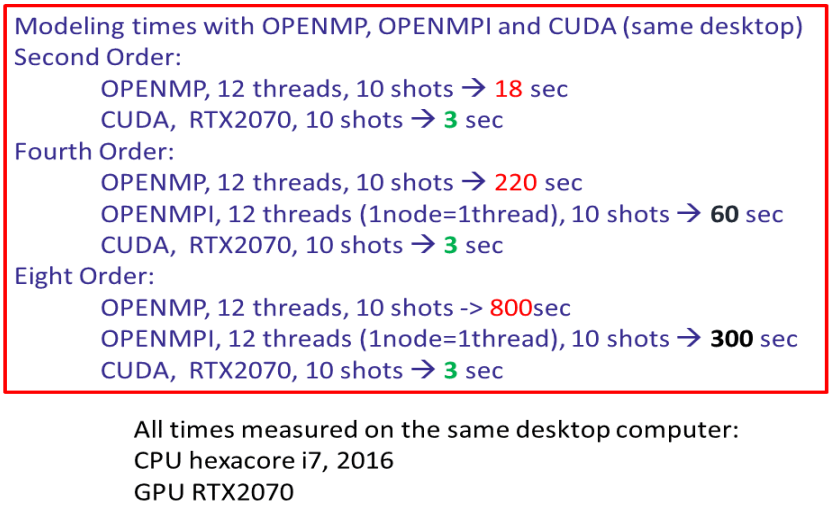

FIG. 10. Comparison times for modeling 10 shots in the Marmousi model using different HPC approaches.

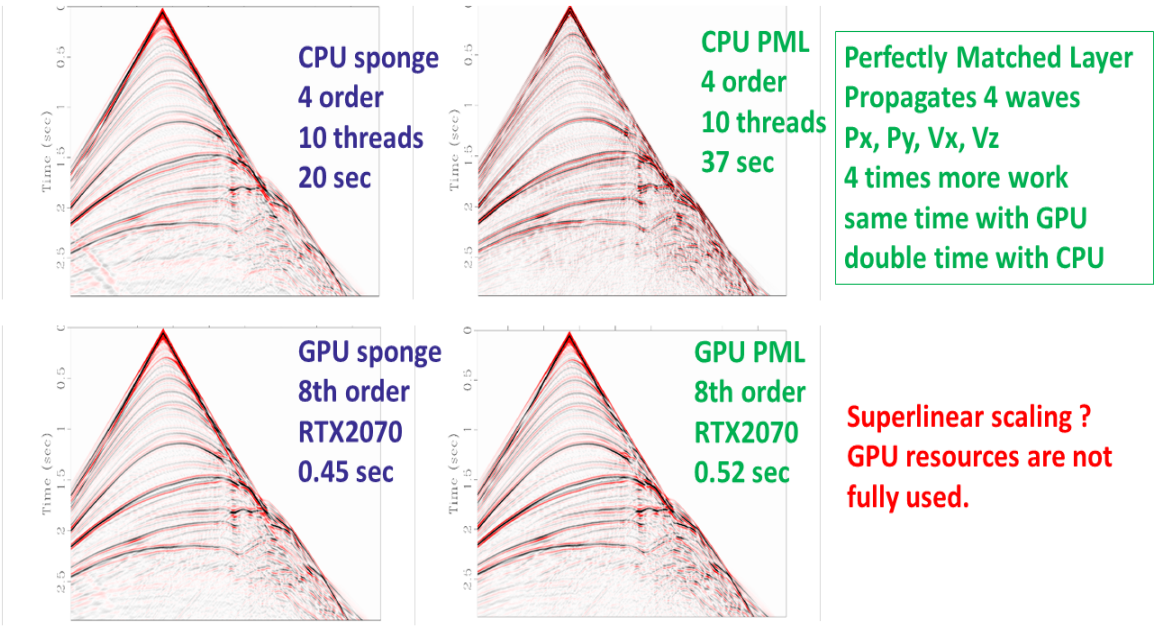

FIG. 11. Comparison times for modeling 10 shots in the Marmousi model using different boundary conditions.

APPLICATION TO REVERSE TIME MIGRATION

Reverse Time Migration (RTM) provides high-quality results for complex environments (Baysal et al., 1983). Its main drawback is the high computational cost, in particular when anisotropy and elasticity are taken into account. When creating 3D migrated angle gathers, this can become prohibited even for large industrial clusters. An RTM-GPU implementation (Yang et al., 2014) can help with this issue since its main computational cost is the FD algorithm.

For this report, I implement an acoustic constant density GPU-RTM using the convolutional pattern discussed in the previous section. Then, I illustrate the efficiency of this implementation by showing the results and computational cost for migration of 6 different velocity models. In Figures 12, 13 and 14, we see the RTM for three 2D structured models. The computational cost is roughly in the order of 1 second per shot for the smallest model (Marmousi), to around 3 seconds per shot in the largest size model (Foothill model). Each of these migrations finished in less than one minute on a single desktop. (Note: the actual Foothills model has varying topography and recording surface,but for convenience I used a version of it with a flat recording surface).

In Figures 15, 16 and 17, we see similar examples for salt models. These models tend to be larger, requiring also longer records and offsets to achieve penetration under the salt. For these tests, migration time was approximately 7-10 seconds per shot. Since normally we need at least a hundred shots, all these examples were performed between 15-20 minutes on a single desktop. For comparison, the Sigsbee example took longer than one hour to run on a 10 nodes cluster, where each node has a quadcore CPU using hyperthreading. This arrangement allows 80 threads to work simultaneously plus a head node for the dataflow. Using the hybrid model, the dataflow for the cluster was set to 20 nodes, each with 4 threads, which is more efficient than the default of 10 nodes with 8 threads each. It is significant that the GPU implementation took $1/4$ of the time using a computer approximately 10 times cheaper and consuming 10 times less energy.

Regarding RTM image quality in the examples, a few clarifications are in order. Since my RTM modeling code uses a constant density wave equation, in Figure 17 we don't see reflections caused by density variations that are visible in other tests of this dataset (BP2004). In all these examples, I did not smooth the velocity models because the quality of the RTM results will depend on the amount of smoothing and it would be difficult to evaluate migration quality without taking that parameter into consideration. In other words, I do not know the answer to the question: what is a fair smoothing value to match the limited resolution of tomographic velocities? Instead, here I produced the best possible results in the shortest possible time, with the understanding that in practice with real data results are never as good. Similarly, noise, limited aperture, multiples, density variations, irregular geometries, near-surface velocity variations, are all effects that will deteriorate these results. I ignore them because it would impossible to adopt reasonable values for all of them and explain the reasons for these choices without changing the focus of this report, which is High-Performance Computing.

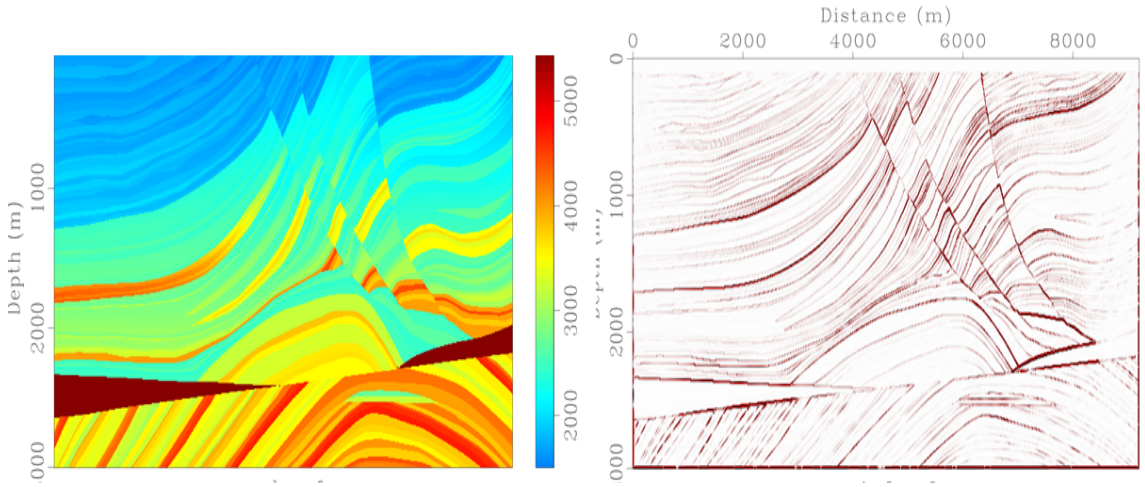

FIG. 12. RTM (20Hz)for Marmousi model, 376x1151 cells, 10 shots, total computing time 10 seconds (1 second per shot). The same test would take 1 minute in a 10 nodes cluster.

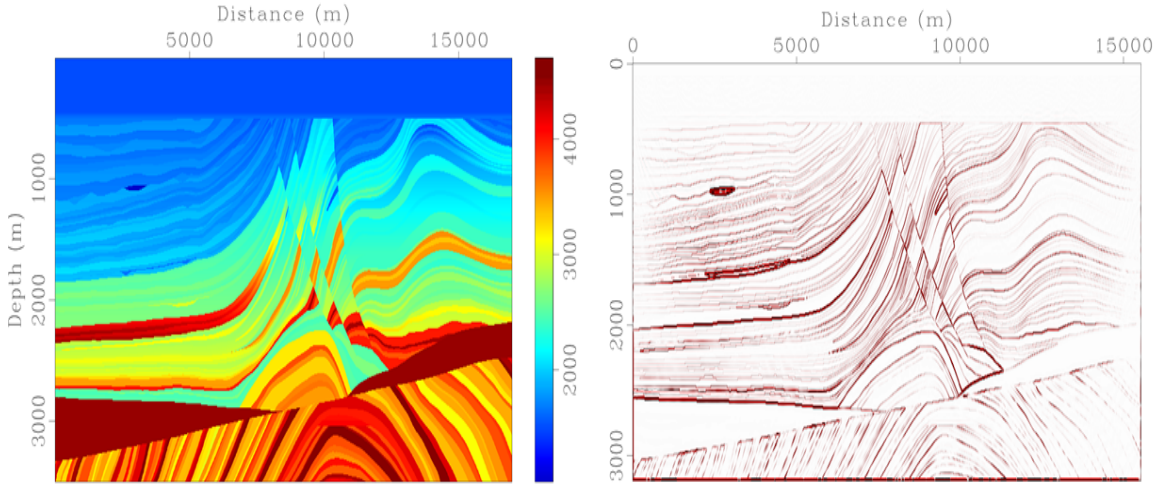

FIG. 13. RTM (20Hz) for Marmousi II model, 400x1942 cells, 25 shots, total computing time 46 seconds (2 seconds per shot).

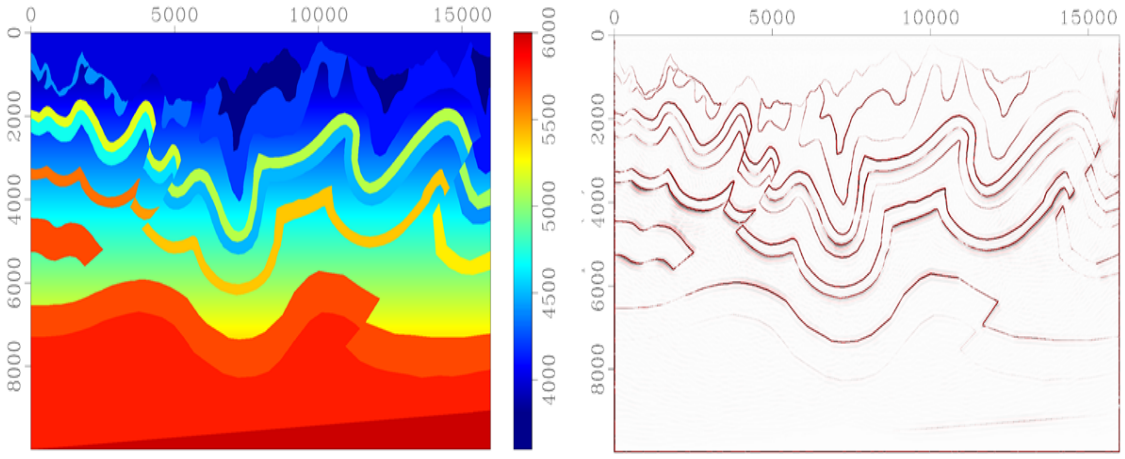

FIG. 14. RTM (20Hz) for Foothills model, 1000x1600 cells, 50 shots, computing time 3 minutes (3 seconds per shot).

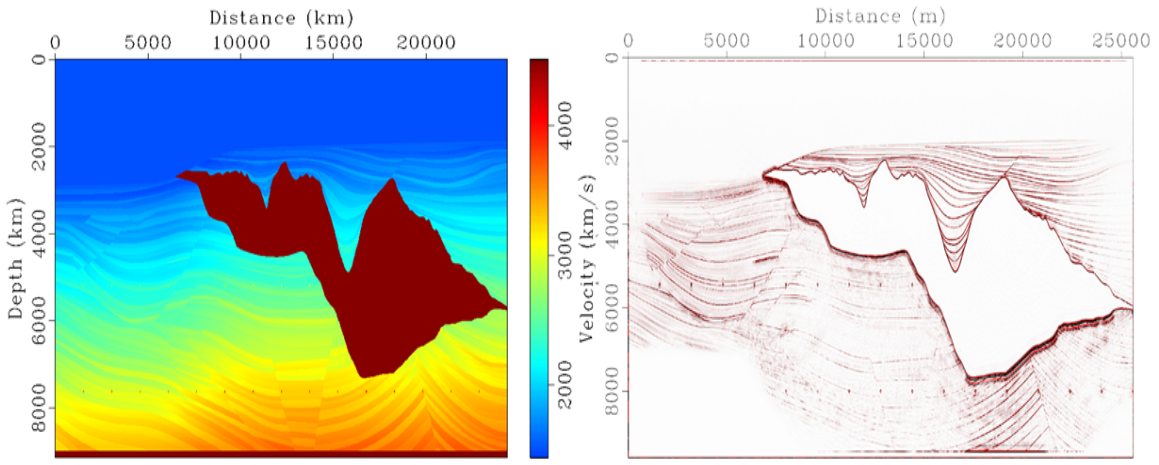

FIG. 15. RTM (20Hz) for Sigsbee model, 50 shots, computing time in one GPU 15 minutes (18 seconds per shot). In comparison, the same test took 60 minutes in a 10 nodes cluster with the hybrid model.

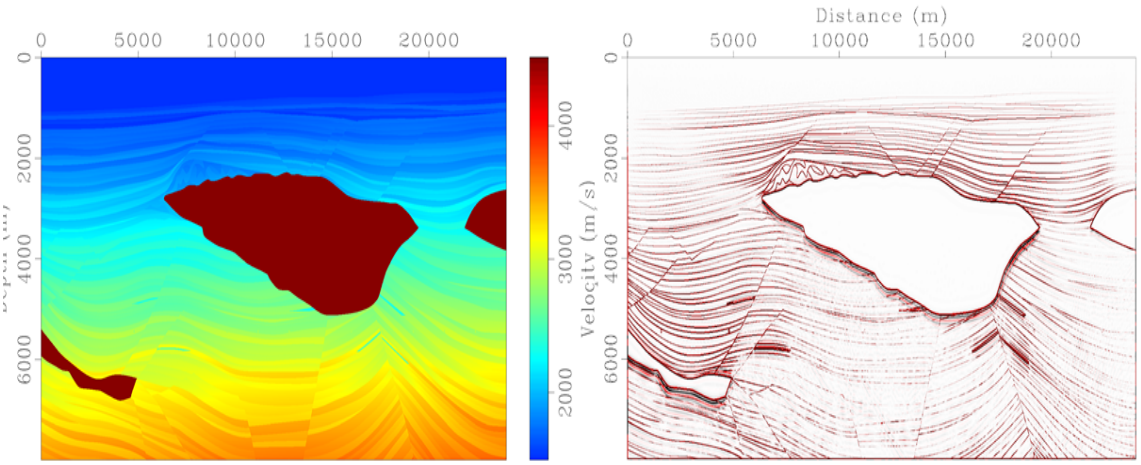

FIG. 16. RTM (20Hz) for Pluto salt model, 1000x3000 cells, 50 shots, computing time 16 minutes (20 seconds per shot).

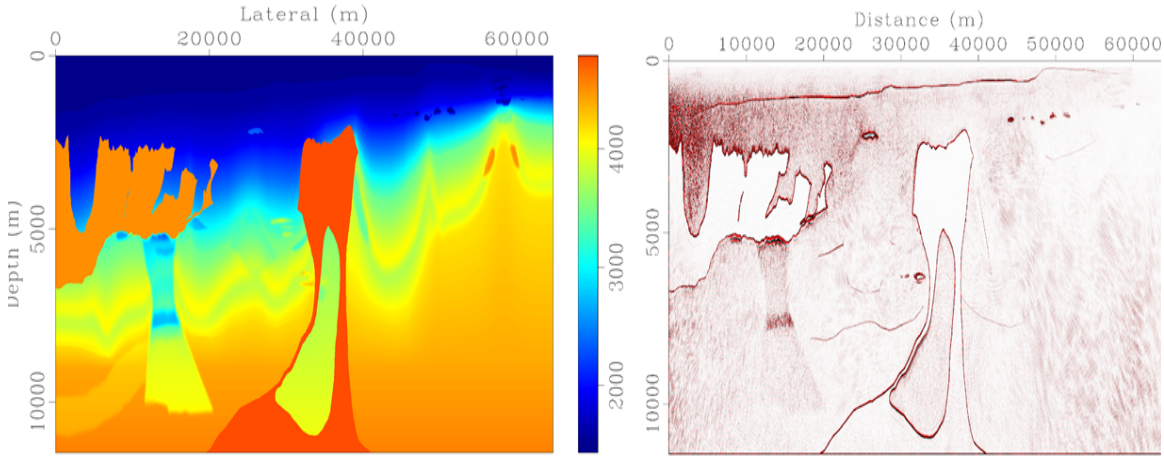

FIG. 17. RTM (20Hz) for BP2004, Salt model with 956x5396 cells, 160 shots, computing time 1 hour 40 minutes (37 seconds per shot).

APPLICATION TO FULL WAVEFORM INVERSION

Full Waveform Inversion (FWI) (Tarantola, 1984) is one of the most powerful but also difficult inversion algorithms to apply in real data processing. FWI can produce detailed velocity information about the subsurface by fitting modeled data to observations (Schuster, 2017). However, its high computational cost has been an obstacle for its application until GPU implementations (now the norm) appeared. For example, Yang et al. (2015), describe early work on a GPU-FWI implementation that I have used as starting point for this report. In Trad (2020) I discussed the need for Multigrid Full Waveform Inversion (Bunks et al., 1995), because of the difficulties related to cycle skipping (Virieux and Operto, 2009) and its benefits in addressing the inverse crime problem (Figure 18). In that report, I described also an adaptive time domain multigrid approach based on shaping filters and cross-correlation to help to mitigate that problem. To summarize the main ideas, the multigrid approach requires FWI to be solved in stages, each stage producing an initial solution for the next stage. These stages are solved with increasing frequency bandwidths and therefore with finer grids as required by FD stability constraints. Since the non-linearity of the inverse problem increases with higher frequencies, the initial models required at each stage need to be closer to the final solution (that is, to the global minimum) (Figure 19). In addition, a current leading edge of research on FWI is high-resolution FWI. If a high frequency velocity model can be achieved by FWI, this model can be used to estimate the reflectivity directly without migration. There are many advantages to this approach, in particular the possibility to bypass many processing steps like migration and multiple attenuation, but its drawback is the computational cost which can be much higher than the traditional approach of FWI followed by migration or even by least squares migration. In this section, I focus on the advantages and need for a GPU implementation for this multigrid time domain FWI described in Trad (2020).

An unintended consequence of working on stages with increasing frequency bandwidth is the need for a reduction of the cell size and time interval. In general, the cell size is constrained by the minimum number of points required for the finite difference approximation. This criterion is much more restrictive than Nyquist. For example, a general estimate requires 10 points per wavelength in FD vs. just 2 points per wavelength in Nyquist. The seismic wavelength can be calculated as $\lambda = \frac{velocity}{frequency}$. Therefore, the minimum velocity and maximum frequency will define the shortest wavelength. FD requires 5 to 10 points per wavelength, depending on the order of the approximation along space. The smaller the wavelength the smaller the cells should be. As a consequence, doubling the frequency requires reducing the cell size by half along each dimension. For a 3D data set, doubling the frequency requires an increase of the number of cells by $8\times$, which is roughly equal to 8 times more computation time.

In addition, to avoid dispersion, the time interval is related to the cell size through the Courant-Friedrichs-Lewy (CFL) condition (Gnedin et al., 2018) given by the following equation:

$$C = \Delta t \sum_{i=1}^{n} \frac{v_{max_i}}{\Delta x_i} \leq C_{max} \tag{1}$$

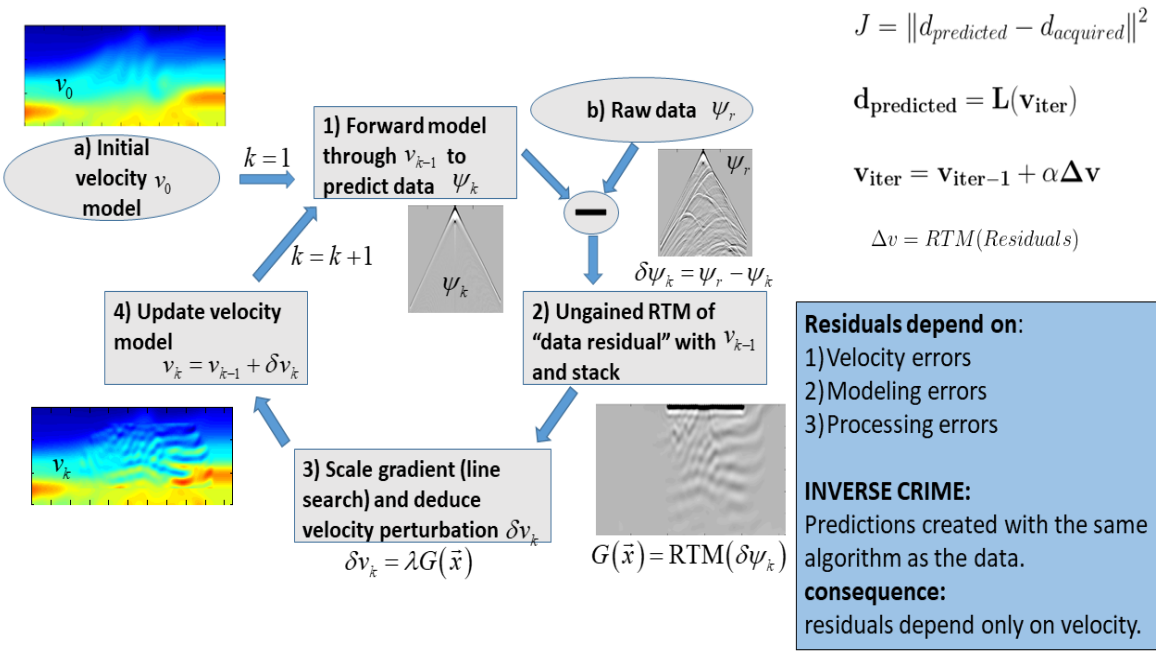

FIG. 18. FWI basics and the inverse crime problem. If data are synthetic and predicted in the same grid as they are inverted, then residuals contain information only about velocity and modeling artefacts contribute to the inversion making the problem much easier than it is in real data.

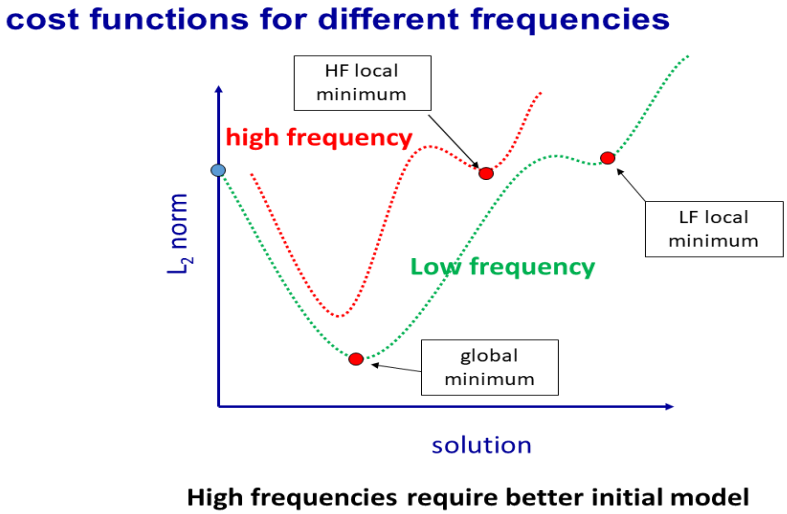

FIG. 19. The cycle-skipping problem and the need for the multigrid approach. The cost functions become narrower as we invert higher frequencies. Solving the problems in stages permits to start with better initial solutions as moving to higher frequencies.

where C_{max} depends on the method used to approximate the derivative ($C_{max} = 1$ for explicit methods as used here), and n is the number of spatial dimensions. We see that as the cell size decreases, the time interval should also decrease at the same rate. Therefore, doubling the frequency requires and additional reduction by half of the time interval, an additional $2\times$ factor in computation time. In summary, for 3D FD (and therefore FWI as well) there is an increase of $16\times$ in computation time every time we move from one stage to another by doubling the maximum frequency.

In simple terms, suppose your initial 3D FWI running to a maximum of 8Hz requires 24 hours, we need 16 days to achieve 16Hz, and 8 months to achieve 32Hz. We see that the concept of high resolution FWI can quickly become infeasible unless:

- The initial runtime is very low, for example one hour. Then we can expect to finish at 32 Hz in 256 hours, roughly 10 days.
- Advances in computer science allow us to reach superlinear scaling. This means the runtime is reduced more than linearly by adding new computer nodes.
- Using alternative inversion or algorithms. For example, frequency domain approaches could lead to some shortcuts by using only a few of the frequencies required, but in general, they are too expensive for 3D work and they scale poorly with the number of nodes because they require very fine grained parallelism.

Here I focus on achieving what I call superlinear scaling by taking advantage of GPUs.

Numerical experiments

Figure 20 shows a GPU multigrid FWI result for the Marmousi model. We see the initial velocity model on the top left. At the top right, we see the first stage calculated on a grid of 32x32m up to a maximum frequency of 8Hz (dominant frequency 4Hz) with 15 iterations and 40 shots. This result is used as the initial model for the second stage calculated on a 16x16m grid, which we see at the bottom left (maximum frequency 16 Hz). This result is interpolated and used as the initial model for the third stage (bottom right), calculated on a 8x8m grid up to a frequency of 25Hz. The computation times and parameters are shown in Table 1. In all cases, we started with synthetic data calculated on a fine grid (8x8m) with a maximum frequency of 25Hz. To change the data bandwidth for each stage, I used the dataflow explained in Trad (2020) with shaping and cross-correlation filters. This step is critical since the predictions for the inversion have to match the input data. An additional advantage of this approach is that the inverse crime problem is mostly (but not entirely) avoided since predictions and input data are calculated on different grids and therefore have different artefacts.

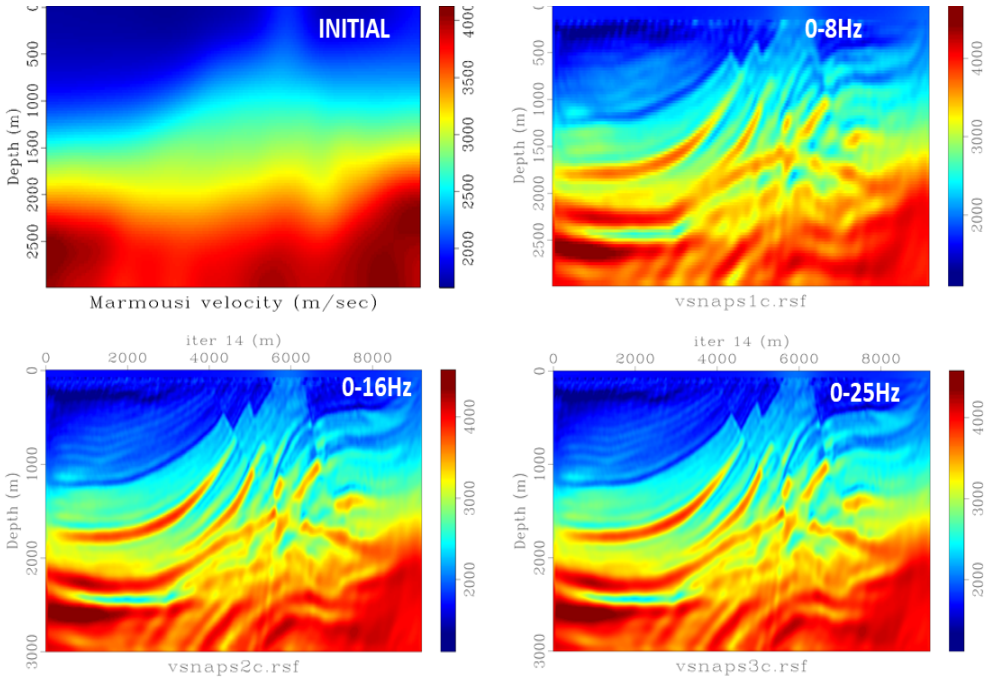

FIG. 20. GPU Multigrid FWI results (3 stages) for the Marmousi model. We transition from 8Hz maximum frequency to 16Hz and then to 25Hz by using 32m cell size, then 16m cell size and finally 8 m cell size.

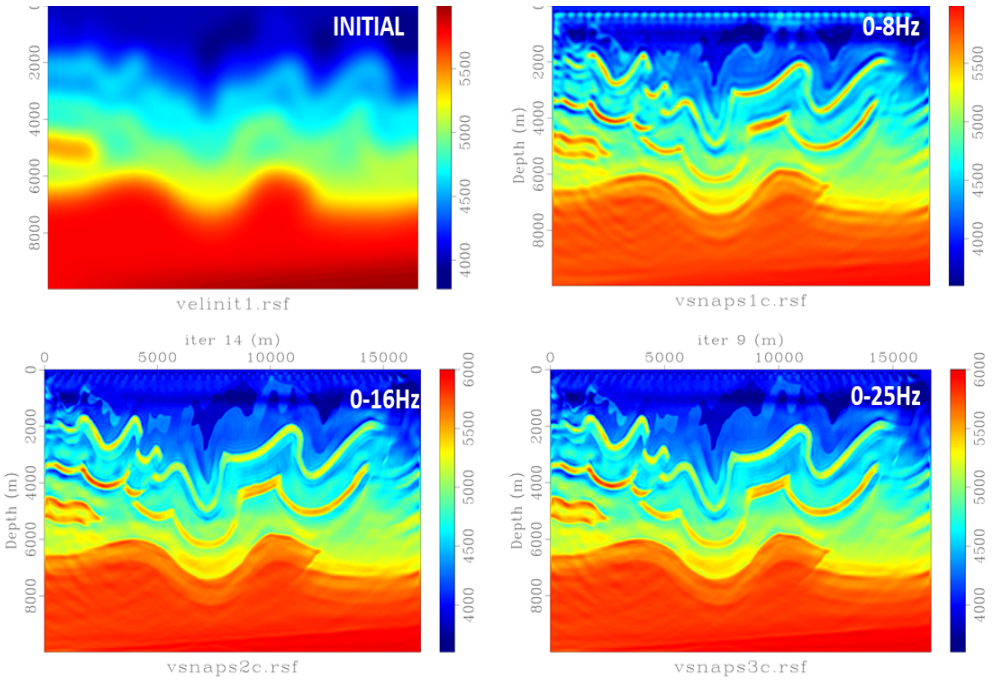

FIG. 21. GPU Multigrid FWI results (3 stages) a Foothills model. We transition from 8Hz maximum frequency to 16Hz and then to 25Hz by using 32m cell size, then 16m cell size and finally 8 m cell size.

Figure 21 shows the same dataflow for the Foothill velocity model. To simplify the dataflow, the data were generated on a flat datum, not on the original foothills elevations. A simplification in both of these tests is that the geometry was taken to be regular and at a convenient sampling interval to facilitate the data conversion across different grids. In practice, data need to be regularized before RTM so receiver intervals fit at the cell centres. Although this is a limitation, the same problem occurs for all RTM implementations. Finally, anisotropy and 3D effects have also been left out but they will be included in a future report (work in progress).

Table 1. GPU Computation times in a regular desktop (for the example in Figure 20)

model size	cell size	time steps	nshots	iterations	time
96 x 288	32, 32	4600	40	15	263 secs
188 x 576	16, 16	4600	40	15	410 secs
376 x 1151	8, 8	4600	40	5	366 secs

For these FWI results, we are still committing some inverse crime since data have been generated with the same engine (finite differences) as the input data. However, in this case several factors reduce the amount of inverse crime. When operating in the time domain, with different cell sizes and different source wavelets, many details change during the wavefield simulation. As mentioned above, the absorving boundary conditions have different thickness. The sources are different. Multiples are different. As a consequence, the finite difference artifacts are very different and they do not help the inversion as they do when keeping the grid and frequencies unchanged. Although we are still biasing our FWI towards perfection (inverse crime) we are taking a large step in the right direction.

Superlinear scaling for multigrid FWI

As an indication of how computation times increase as we go to finer scales, I show in Table 2 the runtimes (using CPUs) in a 10 node cluster, each node with 4 cores running with hyper-threading (8 threads). The implementation is a hybrid OPEMPI-OPENMP, which permits freedom on how to combine multiprocessing and multithreading in a cluster. This means that nodes can be subdivided in different processes, each with different threads. Multiprocessing is more efficient than multi-threading if processes can execute with minimum communication since they don't need to keep cache coherency. On the other hand, processes require more memory. Therefore, there is not a unique combination that fits all cases, since the trade-off depends on the computer cluster architecture and data size. In this particular test, I choose an optimal combination where each node runs as two slaves, and each slave executes the FD algorithm with 4 threads.

As we go to higher frequency bands the runtimes expand by several factors: the number of cells multiplies by 4 (2 in each direction), and the number of time steps is increased as a consequence of FD limitations demanding a smaller time step. Assuming doubling the dominant frequency we should roughly expect an increase of 8 in the computation time from band to band. On the other hand, we are doing a very simple implementation of the wave equation, order 4 along space and 2 along time. To account for variable density, we would need a staggered grid with 3 times more operations. Elastic propagation would require roughly around 5 times more operations. A 3D inversion would require around 2

to 3 orders of magnitude more computations since it would scale with the number of cells along y as well as x. Another factor, as we move to 3D surveys, would be the memory requirements. For 2D the RAM requirements are minimum so we can put as many shots as needed on each node. A 3D survey would limit how many shots we could run on each node.

As mentioned earlier, the cluster in which I am running these tests is very low cost because it does not have a very fast intra-node communication. This does not greatly affect the time domain performance, since communication across nodes is kept to a minimum. but a frequency domain version would require a more costly cluster to run efficiently.

Table 2. CPU Computation times (MPI-Multithread) in a 10 nodes cluster

model size	cell size	time steps	nshots	iterations	time
96 x 288	32, 32	2800	40	20	196 secs
188 x 576	16, 16	2800	40	20	576 secs
376 x 1151	8, 8	4600	40	20	4481 secs

Table 3. GPU Computation times in a regular desktop with a RTX2070

model size	cell size	time steps	nshots	iterations	time
96 x 288	32, 32	2800	40	20	206 secs
188 x 576	16, 16	2800	40	20	327 secs
376 x 1151	8, 8	4600	40	20	1466 secs

Now let us compare with the runtimes using a RTX2070 GPU. Table 3 shows the GPU runtimes with the same parameters in Table 2. Although for the first stage times are similar (10 CPU nodes vs 1 GPU), for the following stages the GPU times do not follow the same increasing rate as in the CPU. Tables 4 and 5 show the expected and real times and ratios with respect to stage 1 for the CPUs and GPU respectively. Overall, we see for the CPU a very close match between the computing times and the number of cells/operations. For example, referring to the first stage the expected time increase is 26X and we observed 23.5X, very close to linear. For the GPU, the time increase was only 3.6, a factor of 7.2 faster than linear. We see that superlinear scaling for the GPU provides us with a way to achieve high resolution FWI.

It is also important to keep in mind that for the GPU test we are using a 10 times smaller hardware than for the CPU test, which consumes 10 times less power and costs 10 times less money than the cluster. Perhaps, we can say it took 10 times more time to program, but the program is done only once, and the tests are done hundreds of times.

APPLICATIONS TO MACHINE LEARNING

Machine learning (ML) has been proposed for performing many of the calculations typically done with standard algorithmic techniques. While standard approaches use physics to constraint the type of possible outcomes for a given algorithm, for example modelling, ML uses a flexible approach where all outcomes are in principle possible, at least in terms

Table 4. CPU scaling, expected (from stage 1) and real (based on parameters from Table 1).

stage	time expected	time real	ratio expected	ratio real
1	–	196	–	–
2	784	576	4	2.9
3	5100	4600	26	23.5

Table 5. GPU scaling, expected (from stage 1) and real (based on parameters from Table 2).

stage	time expected	time real	ratio expected	ratio real
1	–	206	–	–
2	824	327	4	1.6
3	5100	1466	26	3.6

of the algorithmic architecture. However, this flexibility has to be trimmed or pruned to produce useful outcomes. This pruning is done in several ways but most commonly by training. In physics guided ML, we use physics to constrain these outcomes. For example, let us consider wavefield simulation by using a Convolutional Neural Network (CNN). The CNN is hard wired to produce any kind of outcomes in a wavefield, but most of these outcomes do not comply with physics. Nature demands that wavefields have particular continuity, causal principles and limitations. That is, they live in a subspace of the full space of posibilities, a manifold (Chollet, 2021). To prune those non-physical outputs we need a training process that, by calculating the values of the weights or parameters (filters) for each layer, will map flexibility to feasibility. Training is the major tool by which physics is injecting into the network.

It is often claimed that neural networks are more efficient than regular algorithms using physics modeling. However, this claim assumes the network has already been trained, and training is by far the most expensive part of the process. If a network is properly trained to be completely general, we can indeed obtain results very fast. But "faster" is perhaps not the correct word since it assumes we are comparing the network with the best possible outcome of physics based methods, and that has never been consistently shown. Neural networks use the very fast CuDNN library implemented by NVIDIA experts. This library uses for convolution, a similar pattern as described in this report. For fair comparison betwen the network and purely physics-based models, a CNN should be compared with a physics based implementation done also with CuDNN and run on a similar hardware.

The role of GPUs for training CNNs can not be ignored. Neural networks high computational cost would make them impractical without GPUs. However, in this report I want to focus instead on the use of GPUs for injecting physics into the networks by creating training samples for typical problems like migration.

Computer vision with supervised training is perhaps the most successful application of ML to geophysics, for example recognizing salt boundaries and faults. This process requires several components:

- Hundreds to thousands of "samples", where each sample could be an image or part of an image (not the same as what we call "samples" in Geophysics).
- Labels for each of these samples, meaning an example of what we intend to obtain (typically, the result of manual interpretation).

In many applications, samples and labels are abundant (for example from Internet pictures). When they are not sufficient, people apply the technique of data augmentation and/or transfer learning. Data augmentation means to change the samples by rotation, cropping translation, and other simple manipulations. Since in geophysical applications we have to respect the vertical ordering this approach is often limited or impossible. Transfer learning is possible when we have already networks trained for a similar task. That is common in other sciences but less common in Geophysics.

Although seismic data is abundant, processed, migrated and interpreted data is not so abundant. These are required for creating labels for training. That leaves ML to rely on modeling synthetic data to obtain samples and labels. This modeling may be a reflectivity convolved with a wavelet, combined perhaps with ray tracing, or wave propagation for better fidelity. If the modeling method is too simple, the training data won't be realistic, and ML will fail. This problem is similar to the inverse crime issue, so we need to improve realism by better (more computationally expensive) modeling. Thus, we see that the application of HPC techniques, and in particular GPUs is essential for ML applications, not only during training but also to create data for training.

In Figure 22, taken from Huang and Trad (2021), we use the power of ML to apply Hessian filtering for Least Squares Migration in the image space. We train a U-Net network to learn the transformation from adjoint imaging (regular RTM), to final reflectivity. In addition, we are trying to use this network to distinguish the contribution from multiples and primaries. On the left, we see the result of using primaries only, and on the right we use multiples and primaries. For this training we need hundreds of examples, generated by using the GPU implementations of this report, both for modeling and for RTM. The inputs to the network are the RTM result from a synthetic velocity model. The labels are the reflectivity obtained by converting velocity to reflectivity, converting to time, convolving with a seismic wavelet, and then back to depth. The network learns how to convert from RTM images, affected by sampling and illumination artifacts, to the best possible reflectivity. Effectively, the U-Net weights calculate the filters that remove the Hessian. Reducing the computation time by 10x-100x clearly makes the difference between tens and thousands of training images. In ML, that is the difference between success and failure. This speedup opens the door for the on-the-fly simulation for training. This means the network can be set to be constantly learning without the need to store large amounts of data.

CONCLUSIONS

This report compares Finite difference implementations by using GPUs with both distributed (MPI) and shared memory (multithreaded) approaches. The report contains three main messages, summarized below:

GPUs have tremendous potential for modelling, RTM and FWI because these meth-

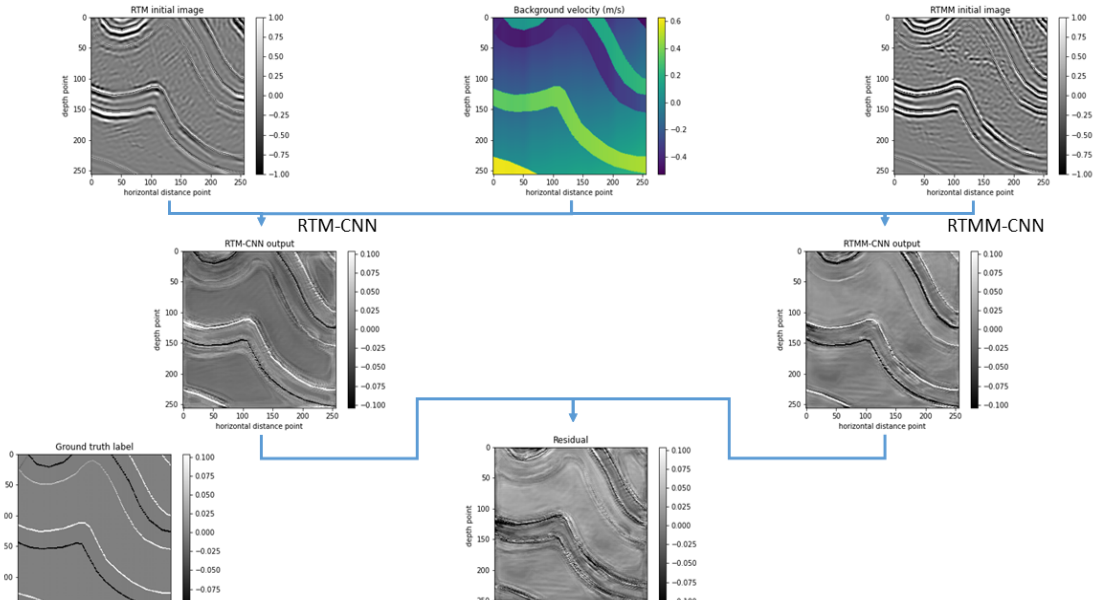

FIG. 22. Least Squares RTM in the image space using U-Net. Models generated with GPU modelling.

ods are essentially bounded in efficiency by the FD algorithms. GPUs can accelerate FD about $30X$-$100X$ by using the convolutional pattern the distributes calculations across the different memory hierarchies.

For the data sizes we checked so far, GPU resources are under-used and the computation time is not linear with the number of computations (superlinear), so modelling algorithms can be made more precise and grids can be made finer for higher resolution FWI without significant penalty in computational time. This also introduces the possibility of on-the-fly synthetic generation during neural network training, which makes many ML approaches very attractive since store space for training data is not required. Although, we need to explore 3D datasets to verify if this happens in a larger scale, also we can use much larger and modern GPUs than used in this report.

Parallelization often has a much larger effect on computational time than algorithmic approaches like preconditioners. Of course, we should pursue any technique that helps in computational cost reduction, but we should not leave parallelization as a final process because more complicated algorithms often temper parallelization gains. Therefore, a more sophisticated algorithmic design, for example reducing the number of iterations by 10%, should not be applied if it eliminates a gain of 1000% speed up by parallelization.

ACKNOWLEDGMENTS

I thank Sam Gray and Torre Zuk for many fruitful discussions, and review of this report. Penliang Yang for his enormous contribution with Madagascar examples, and CREWES sponsors for contributing to this seismic research. I also gratefully acknowledge support from NSERC (Natural Science and Engineering Research Council of Canada) through the grants CRDPJ 461179-13, CRDPJ 543578-19 and NSERC Discovery Grant.

REFERENCES

Baysal, E., Kosloff, D. D., and Sherwood, J. W., 1983, Reverse time migration: Geophysics, **48**, No. 11, 1514–1524.

Bunks, C., Saleck, F. M., Zaleski, S., and Chavent, G., 1995, Multiscale seismic waveform inversion: Geophysics, **60**, No. 5, 1457–1473.

Cheng, J., Grossman, M., and McKercher, T., 2014, Professional CUDA C Programming: Wrox Press Ltd., GBR, 1st edn.

Chollet, F., 2021, Deep learning with Python: Simon and Schuster.

Dagum, L., and Menon, R., 1998, Openmp: an industry standard api for shared-memory programming: IEEE computational science and engineering, **5**, No. 1, 46–55.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A. et al., 2004, Open mpi: Goals, concept, and design of a next generation mpi implementation, *in* European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting, Springer, 97–104.

Gnedin, N. Y., Semenov, V. A., and Kravtsov, A. V., 2018, Enforcing the courantâĂŞfriedrichsâĂŞlewy condition in explicitly conservative local time stepping schemes: Journal of Computational Physics, **359**, 93–105.
URL `https://www.sciencedirect.com/science/article/pii/S0021999118300184`

Han, J., and Sharma, B., 2019, Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10. x and C/C++: Packt Publishing Ltd.

Huang, S., and Trad, D. O., 2021, Least squares migration with Neural Networks: CREWES Research Report, **33**, 54.1–54.20.

Nvidia, C., 2007, Compute unified device architecture programming guide.

Ristov, S., Prodan, R., Gusev, M., and Skala, K., 2016, Superlinear speedup in hpc systems: Why and when?, *in* 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), 889–898.

Schuster, G. T., 2017, Seismic inversion: Society of Exploration Geophysicists.

Tarantola, A., 1984, Inversion of seismic reflection data in the acoustic approximation: Geophysics, **49**, No. 8, 1259–1266.

Trad, D. O., 2020, A multigrid approach for time domain FWI: CREWES Research Report, **32**, 54.1–54.20.

Virieux, J., and Operto, S., 2009, An overview of full-waveform inversion in exploration geophysics: Geophysics, **74**, No. 6, WCC1–WCC26.

Yang, P., Gao, J., and Wang, B., 2014, Rtm using effective boundary saving: A staggered grid gpu implementation: Computers & Geosciences, **68**, 64–72.

Yang, P., Gao, J., and Wang, B., 2015, A graphics processing unit implementation of time-domain full-waveform inversion: Geophysics, **80**, No. 3, F31–F39.