geoconvention
Calgary • Canada • June 20-22 2022

GPU applications for modelling, imaging, inversion and machine learning

Daniel Trad
CREWES - University of Calgary

Summary

Modelling seismic data is a key part of research for acquisition design, imaging, full waveform inversion (FWI) and machine learning (ML). The Finite Difference (FD) method is one of the most used for structural or stratigraphic modelling because it provides a wide range of options, and its accuracy is sufficient for most applications. It is widely used for simulating surveys and also for forward and reverse modelling required in RTM and FWI. The applicability of these techniques depends strongly on the computational cost of FD because usually many modelling steps are required for iterative inversion. Large elastic 3D FWI is prohibitively slow, except when done on very large computer clusters. Although cloud computing provides a way to alleviate this issue, when it comes to research, development and initial testing, we still rely heavily on local resources. Therefore, decreasing computational time for finite-difference modelling has been a key research topic since its first use.

In addition, we are on the verge of a different and powerful technology that requires modelling even more than traditional techniques: machine learning (ML). This approach promises to be significantly more flexible for solving many problems than traditional physics-based approaches. While typical forward modelling requires us to write the specific rules that 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 modelling 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:

- Obtain more powerful computers, usually in the form of clusters or cloud computing. Large clusters require a large amount of energy for power supply and refrigeration.

- Design better algorithms for both modelling and inversion. 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.

- Use the advances of hardware in the domain of computer science by parallelization techniques, which typically require a major restructuring of the software. Therefore, it is reasonable to incorporate parallelization into early development.

- Use radically different new technologies, like ML and Quantum Computing (QC). This can well become extremely important for modelling in the 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 abstract, I will discuss the third option to explain 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) deals with 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.

The most common HPC parallelization approach is distributed computing for clusters. In this model, we have a modest number of nodes, where each node would act as an independent computer. The programmer makes them work simultaneously on the same task by using libraries like Message Passing Interface (MPI) (Gabriel et al., 2004) that create a main program controlling the dataflow, typically run on the head node (called master). The main overhead is passing the data to the nodes and collecting intermediate calculations. The communication across nodes and master takes 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 modelling. The combination of multi-core nodes is standard these days for any processes that require heavy computations.

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 (lightweight 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.

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 where data moves across the two models as it fits for optimized calculation.

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) but they also control them. GPUs programming relies on the programmer's skills to place the variables on the optimal memory components. For example, GPUs have a type of shared memory with a 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.

In addition to the memory hierarchy, GPUs can support also a special arrangement of threads in blocks (which can be 1D, 2D or 3D). Blocks are arranged in grids, which can also be 1D, 2D or 3D. 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 more challenging than other types of parallelization.

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. To get peak performance on GPU calculations, we need to use what is called "shared memory". This shared memory is visible to only threads in the same block and its size is small. Therefore, the working arrays have to be subdivided into small tiles, which are distributed across blocks. Since each block has its own shared memory, variables can be reused efficiently without the usual problems of cache coherence (updating cached variables across multiple 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. Being able to perform computations on variables stored in the shared memory is the main reason why we can achieve 100× speedups.

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. The wavefields, originally allocated in CPU RAM are transferred to the GPU and decomposed 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 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.

RTM and FWI Examples

Let us see some examples to illustrate the speedups achieved by using FD with GPUs in RTM and FWI. In Figure 1, we see the result for the Sigsbee salt model using 50 shots, each with 10000 samples, and migrating with a bandwidth of 0-25Hz. This model is in a grid with 1200x3200 cells and was calculated using a single desktop and with one GPU, RTX2070, in 20 minutes. The same result takes one hour 20 minutes, using CPUs and a hybrid MP-MPI distributed model in a 10-node cluster. The GPU implementation takes 1⁄3 of the time using a computer approximately 10 times smaller than for the CPU case, which also consumes 10 times less energy.

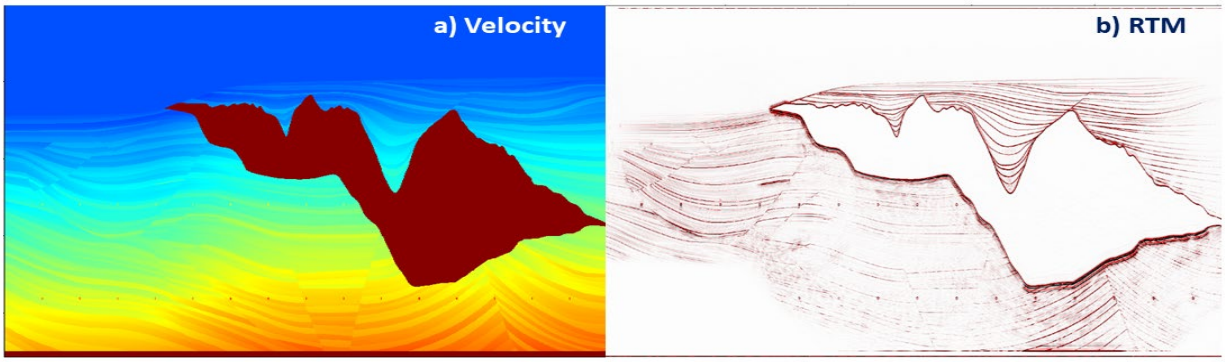

FIG. 1. GPU-RTM for 50 shots with frequency bandwidth between 0-25Hz. a) Initial model, b) RTM. Running time 20 minutes on a desktop with 1 GPU, 60 minutes in a 10 nodes cluster.

In Figure 2 we see the result from applying multi-stage FWI (Trad, 2021) for a Foothills model. The multigrid method requires applying the inversion with progressively larger frequencies, and therefore finer cell grids. Doubling the frequency in a 2D model like this requires approximately an increase of 8 times in computation time (2X doubling each space dimension and 2X reducing the time step). In a 3D case, the computation time increase is 16X.

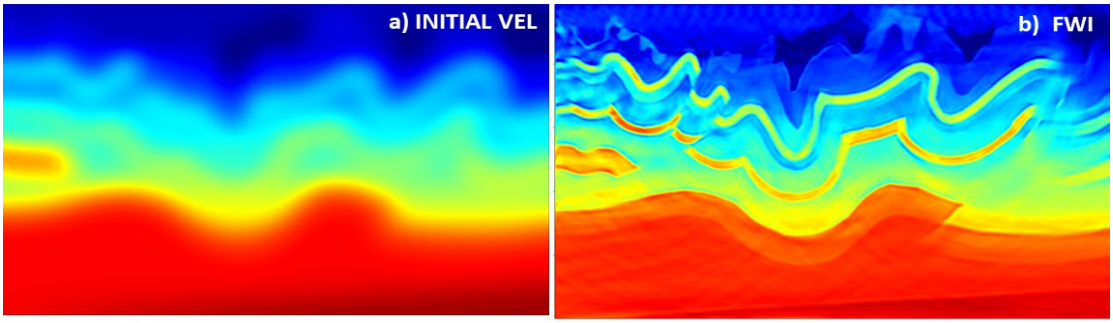

FIG. 2. Multigrid FWI for 50 shots with bandwidth between 0-25Hz. a) Initial model, b) Inverted model with multigrid FWI in 3 stages (32m cell, max Hz, 16m cell, 16Hz, 8m cell, 25Hz).

In Figure 3, we see the individual stages for the Marmousi model. The calculation times for these 3 stages (and 3 grids) were compared in both CPU (MPI in a 10-node cluster) and GPU (1 desktop with 1 GPU). Table 2 shows the expected time increase for the CPU case, but in Table 3 we don't see the same time increase. Since calculations in the GPU do not normally use all the available resources, usually we see that increasing the complexity of the problem or the number of calculations does not increase the runtime.

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

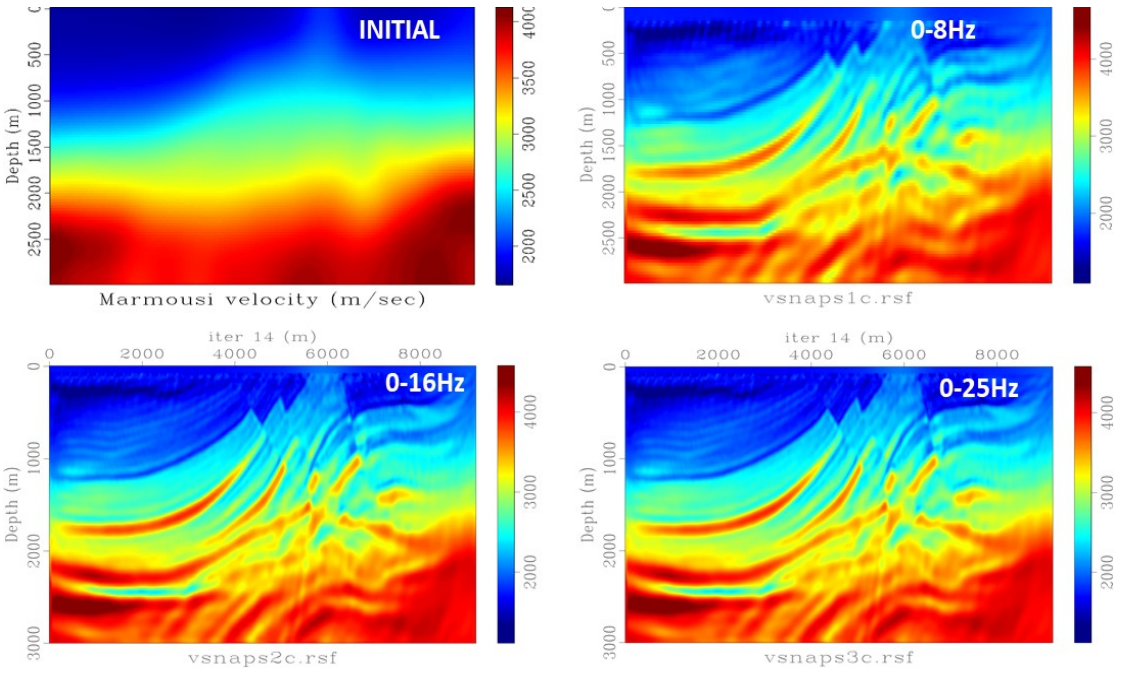

FIG. 3. GPU Multigrid FWI results (3 stages) for the Marmousi model. We transition from 8Hz maximum frequency to 16Hz and then to 25Hz by using 32m cell size, then 16m cell size and finally 8 m cell size.

Conclusions

This abstract compares FD implementations by using GPUs with both distributed (MPI) and shared memory (multithreaded) approaches. GPUs have tremendous potential for modelling, RTM and FWI because these methods are essentially bounded in efficiency by the FD algorithms. GPUs can accelerate FD about 30*X*-100*X* by using the convolutional pattern that 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

Acknowledgements

I thank Sam Gray, Torre Zuk, Penliang Yang, 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

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

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 UsersâA˘ Z Group´ Meeting, Springer, 97–104.

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.

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

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

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 fullwaveform inversion: Geophysics, 80, No. 3, F31–F39.