Using graphics processing units on the cloud to accelerate and reduce processing cost of parameters estimation of seismic processing algorithm

Nicholas T. Okita* (CEPETRO/UNICAMP), Tiago A. Coimbra (CEPETRO/UNICAMP), José Ribeiro (CEPETRO/UNICAMP), and Martin Tygel (CCES/CEPID and CEPETRO/UNICAMP).

Abstract

A quiet revolution in high-performance computing (HPC), as carried out in academic and industry environments, is taking place in response to the ever-growing opportunities offered by cloud providers. More specifically, one has the choice of replacing physical, in-house computer infrastructure by virtual-machine (VM) clusters that are assembled and used for on-demand tasks. In principle, benefits of the cloud include unlimited computation power, no maintenance costs and cutting-edge technology for carrying out individual computational tasks. As significant price differences are attached to different VM specifications and their time allocations, a key issue of a cloud solution is to find the configuration solves a given application with optimal cost/benefit for the user. In this paper, a problem of parameter estimation of seismic reflection data is solved using five selected VM configurations and a performance analysis of the results is carried out. We hope that type of analysis may contribute to the best-possible use of the cloud in a variety of applications.

Introduction

For a long time clusters have been the primary option when executing seismic processing programs, many of them offering both multi-core processors and graphics processing unit (GPU) accelerators. However, the clusters are expensive to build and maintain, while having a limited lifespan regarding its specifications. Taking into consideration how fast technologies evolve, a cluster can be outdated a few years after it has been bought, requiring special care when selecting its components to guarantee good performance for the whole project. Recently, however, we have seen an alternative to this kind of hardware: cloud computing.

Public cloud providers, such as Microsoft Azure (MA) and Amazon Web Service (AWS), offer part of their data-centers resources to users as virtual machines (VMs), pricing these machines differently based on hardware configuration and, usually, in time contracts (usually advertised in price per hour and charged per second of use). A huge advantage to the cloud computing model is the opportunity for users to configure their VMs with any available hardware, enabling them to configure the infrastructure dynamically for the program being executed. However, the amount of VM options can lead to higher charges while offering lower processing power, therefore a good selection is important to optimize not only costs but also execution time (see, Okita et al., 2018).

Graphics processing units accelerators are highly parallel devices that offer from hundreds to even thousands of computing threads within a single chip while consuming less power when comparing to central processing units (CPUs) of similar performance. Due to a large number of threads, they can be used to decrease the execution time of parallel programs.

Some public cloud providers have GPU accelerated instances available, hence it is possible to use them to improve the execution time of workloads that can take advantage of these devices. Since they are priced similarly to CPU instances, their usage can come without additional costs. This work objective involves the usage of GPU accelerated instances to reduce the execution time and processing cost of a seismic processing algorithm in the Amazon Web Services Elastic Computing Cloud (AWS EC2).

Formulation of the problem

For a given 2D seismic data set and pose of estimating the kinematic attributes (or parameters) of traveltime, normal-moveout (NMO) velocities, slopes, and curvatures of reflection events in the post-stack and prestack domains. For simplicity of exposition, we assume that one-component (e.g, pressure or vertical-component elastic) data and that events of interest are non-converted primary reflections. In this case, data samples can be expressed as $u(m,h,t)$, in which u represents the observed amplitude, m is the midpoint location, h is the half-offset distance and t is the time sample, all these beings, for theoretical reasons, continuous variables. Under these circumstances, our problem consists of estimating the kinematic parameters of each sample at which arrival of a reflection or diffraction of interest is observed. As such arrivals are not to be a priori identified, the estimation procedure is carried out at all data samples, each of them supposed as a candidate location of the desired reflection. The above estimation problem is of great interest for advanced processing tasks, such as tomographic velocity-model building (e.g, Billette and Lambaré, 1998 and Dell et al., 2014) and data regularization (e.g, Zhang et al., 2001 and Coimbra et al., 2016).

Coherency

A key property of reflections (as well as other seismic events) is that they present themselves as *coherent* signals along their traveltime surfaces within the seismic data

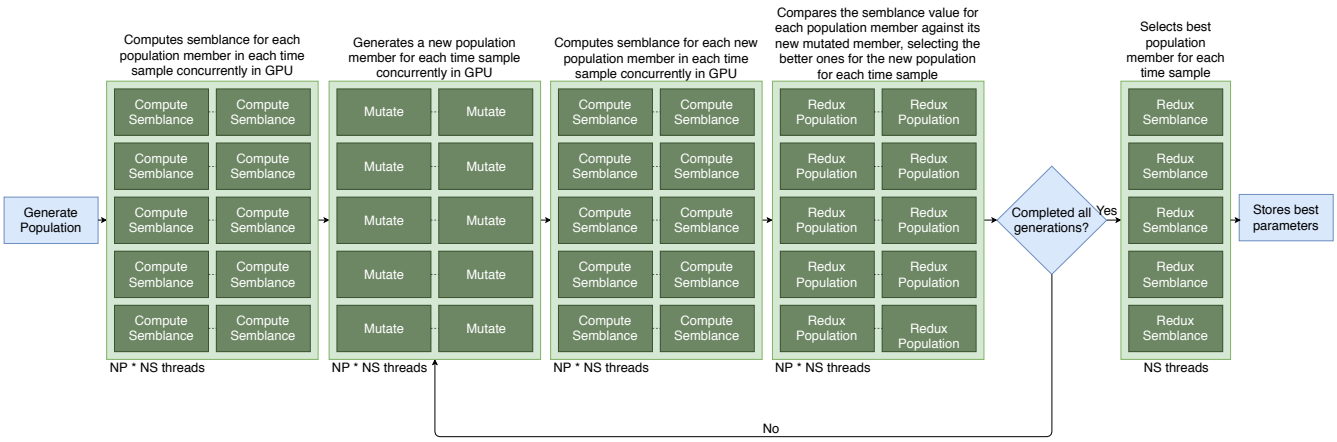

Figure 1: Flowchart of the Differential Evolution implementation in GPU, showing the algorithm steps of the Differential Evolution mutation ("Mutate"), objective function computation ("Compute Semblance") and selections ("Redux Population", "Redux Semblance")

set. By the consideration of a quantitative coherence measure, one can introduce an *objective function* so that our parameter estimating problem is transformed into an optimization problem. Namely, the desired parameters should be the ones that the coherence (objective function) is maximized. A second consequence is that we can also use that physical principle to build an initial image of the data by a stacking procedure. Stacking along the traveltime surface strip that refers to reflection point and under the use of optimal stacking parameters produces a high amplitude. For points not on a reflection event, one gets a small stacking result. These considerations form the basis of a number of stacking imaging schemes available in seismic processing.

The most popular coherence measure used in practice is *semblance* (see, Neidell and Taner, 1971). To define the coherence measure semblance, we suppose a given fixed, reference data point location (m_0, h_0, t_0) that belongs to an (unknown) traveltime surface $t = t(m,h)$ defined for midpoint and half-offset pairs (m,h) in the neighborhood of (m_0, h_0). In the same neighborhood, we now consider a given traveltime surface $T = T(m,h; p_1, \cdots, p_n)$ in which p_i are given kinematic parameters. In addition, in discrete form, the semblance between the two traveltime functions is given by

$$S(p_1, \cdots, p_n) = \frac{\sum_{k=-W}^{k=W} \left[\sum_{i=-I}^{i=I} \sum_{j=-J}^{j=J} u_{i,j,k}\right]^2}{N \sum_{k=-W}^{k=W} \left[\sum_{i=-I}^{i=I} \sum_{j=-J}^{j=J} u_{i,j,k}^2\right]} , \qquad (1)$$

where $N = (2I+1)(2J+1)$, $u_{i,j,k} = u(m_i, h_j, T_k)$ represents the amplitudes computed at the data points $(m_i, h_j, T_k(m_i, h_j; p_1, \cdots, p_n)$. Notations are as follows:

$$\begin{aligned} &m_i = m_0 + i\Delta m, \quad h_j = h_0 + j\Delta h, \\ &T_k(m_i, h_j; p_1, \cdots, p_n) = T(m_i, h_j; p_1, \cdots, p_n) + k\Delta t \ , \end{aligned} \qquad (2)$$

In the above equations, Δm, Δh, and Δt denote midpoint, half-offset, and time (uniform) sampling, respectively. Note that at these locations, actual data points may not exist. In this case, these are replaced by interpolation values from neighboring points.

Traveltime functions

It is now useful to say a few words about the traveltime functions that employed for our parameter-estimation experiments described below. The first two are the 2D hyperbolic zero-offset (ZO) and finite-offset (FO) common-reflection-surface (CRS) traveltimes (see, e.g, Jäger et al., 2001; Zhang et al., 2001; and Faccipieri et al., 2018). The former, denoted by ZO-CRS traveltime depends on three parameters, these being the slopes and curvatures that refer to all data points within the post-stack domain. The latter is denoted FO-CRS traveltime and depends on five parameters which refer to all data points within the prestack domain (see, e.g, Zhang et al., 2001). A final traveltime is a non-hyperbolic generalization of the ZO-CRS traveltime, being denoted as ZO-NCRS traveltime. Introduced in Fomel and Kazinnik (2012), ZO-NCRS depends on the same parameters of ZO-CRS, i.e, three parameters. Presentation and discussion on the the above traveltimes are outside the scope of the present paper, which is focused on the cloud-computational solutions of the parameter-estimation problems associated with the above traveltimes, for more details, see the related papers aforementioned.

Table 1: Machine configurations

Configuration	Processing Unit	RAM (GB)	Price (USD/h)
CPU1 (*c5.18xlarge*)	72 Xeon Platinum 8124 3GHz vCores	144	3.060
CPU2 (*m5.24xlarge*)	96 Xeon Platinum 8175 2.5GHz vCores	384	4.608
GPU1 (*p3.2xlarge*)	1 Nvidia Tesla V100 16GB	61	3.060
GPU2 (*g3s.xlarge*)	1 Nvidia Tesla M60 8GB	30.5	0.750
GPU3 (*p2.xlarge*)	1 Nvidia Tesla K80 12GB	61	0.900

Differential Evolution meta-heuristic

To find the best values for the objective function it is possible to make use of heuristics to iterate through the search space to find the best parameters combination.

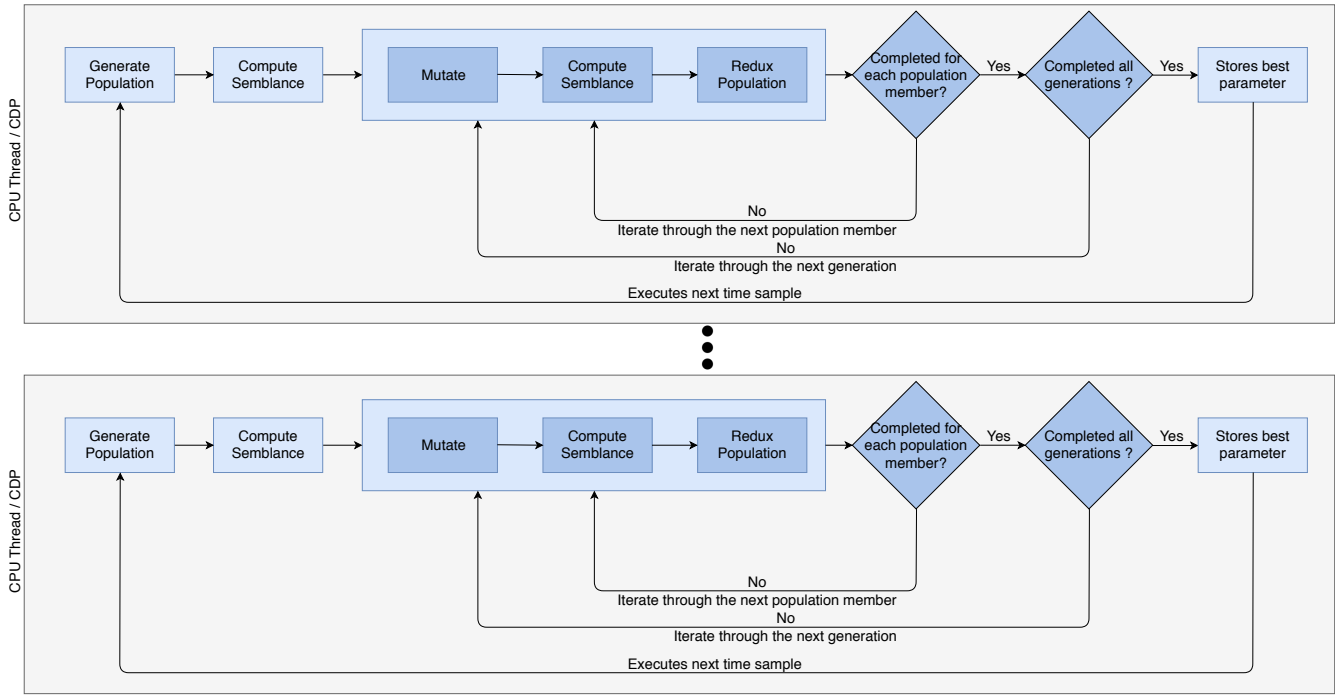

Figure 2: Flowchart of the Differential Evolution implementation in CPU, showing the algorithm steps in multiple CPU threads each for a different CDP, consisting of generating an initial population for the current time sample, then Differential Evolution mutation ("Mutate"), followed by computing the objective function ("Compute Semblance") and Differential Evolution selection ("Redux Population"), iterating through a defined number of generations and for each time sample

The Differential Evolution (DE) meta-heuristic is a global optimization technique that relies on the dynamics of a population (Storn and Price (1997)). It works by generating N_P individuals in a population, each characterized as a vector of parameters, initially distributed randomly through a search space. These individuals interact with each other in processes called mutation and cross over. The first involves a weighted difference between three population members to generate new parameters, while the second replaces the old parameters by the newly obtained ones. After generating a new individual, a process called selection verifies if that new parameter leads to a better value of the objective function. If so, that new parameter is kept, otherwise, it is not accepted. The processes of mutation, cross over and selection applied to the whole population is called a generation. The DE algorithm executes these processes to a predefined number of generations, choosing the parameters that achieve the best of the objective function.

Methodology and Experiments

The programs to compute ZO- and FO-CRS parameters using the DE algorithm are implemented in *C++* and compiled using the *gcc* compiler in its version *5.4.0* with the optimization flag *-O3*. The GPU version has its kernels is written in *OpenCL 1.1*. All implementations are performed using the scalable partially idempotent task system (SPITS) (see, Borin et al., 2016) programming model to support execution in clusters. To use the PYPITS runtime (see, Benedicto et al., 2017) to execute the program we use *python 3.5.2*. In the ZO configuration, parameter estimation is performed, in addition to CRS, also to its non-hyperbolic (NCRS) version (Fomel and Kazinnik, 2012).

Two data sets are used to execute the experiments. data set St is a synthetic seismic prestack file of around fourteen megabytes and around four thousand six hundred traces. Meanwhile, data set Re is a real seismic prestack file of around four hundred megabytes and around sixty thousand traces.

Concerning cloud hardware to be assembled, we selected a few CPU and GPU instances available on AWS EC2 On-Demand platform. The configurations used for the testing are shown in Table 1, the prices defined in this table were the ones practiced on February 25th, 2019. We selected every GPU instance available for the current instance generation while preferring two of the higher-end CPU instances. The reason to select higher-end CPU instances lies in the performance and price scaling in the platform. Within the same instance type, the price scales linearly according to the number of cores, which means, an instance with twice as many cores will cost twice as much. With our embarrassingly parallel algorithm, we achieve almost linear scalability with the number of cores, therefore our cost would remain relatively the same with smaller instances.

In our experiments, the DE parameter estimations are carried out for each hardware configuration displayed in Table 1.

It is important to highlight that there are differences between the GPU and CPU parallelism approaches, leading to different workflows as depicted in Figures 1 and 2, respectively.

The flowchart in Figure 1 shows how we implemented the DE algorithm in GPU. In this figure, NP represents the number of individuals in a population and NS the number of time samples being processed in the current trace. The first step consists of generating the initial population for each time sample, which is done in the CPU. We then create $NP*NS$ threads in GPU to compute the semblance for each population member in each time sample. Next, we create a new mutated population by mutating each population member in each time sample concurrently, computing their semblance and verifying if they are better than its original population member. After running the defined number of generations we verify the best population member for each time sample, selecting it as our parameters combination for that time sample.

Also, each dark green block represents a GPU kernel in Figure 1. "Compute Semblance" represents a kernel that receives parameters combination and trace samples to compute the semblance for these values. "Mutate" generates a new population member using the meta-heuristic Differential Evolution. "Redux Population" verifies if a new population member is better than its predecessor. At last, "Redux Semblance" selects the best member of each population to be returned as the best parameter combination for the current time samples.

Meanwhile, the flowchart in Figure 2 shows the CPU implementation. It's important to notice that we achieve parallelism by creating multiple threads to process multiple CDPs at the same time, differently from the GPU implementation which we achieve parallelism by processing multiple time samples at the same time. In each thread, we iterate through the time samples using the Differential Evolution heuristic to search for the best parameters combination. The heuristic works by iterating through each population member, computing its new mutated counterpart and verifying if it's better than the original using the semblance function, replacing it in the population before going to the next population member iteration. After repeating this for the defined number of generations we return the parameters combination which offered the best semblance value and iterate through the next time sample.

Similarly to the GPU flowchart shown in Figure 1, we have small blocks analogous to the GPU kernels. Blocks with the same name have the same logical functionality as the GPU kernel blocks. For example, the "Mutate" block generates a new population member using the Differential Evolution meta-heuristic, similarly to the GPU kernel "Mutate" block.

At last, we computed the parameter estimation cost as the time spent processing multiplied by the cost of the configuration shown in Table 1, disregarding additional cloud costs such as storage.

Results

We divide the results into three subsections, each of them corresponding to the use of a specific parametric traveltime: the first two subsections employ the CRS hyperbolic ZO and FO traveltimes, respectively. The third and last one uses the non-hyperbolic zero-offset CRS traveltime. The programs, data sets, and hardware configurations are the ones described in the Methodology and Experiments section. Our experiments show that, for each of the input data sets, all implementations (including both CPU and GPU) gives rise to estimation results with negligible discrepancies. As such, those results are not displayed and only the performance of the implementations are analyzed.

Zero-Offset Common-Reflection-Surface

The execution times and related costs for the configurations displayed in Table 1 applied to the two input data sets are shown in Table 2. The illustrative purposes, we show, for the real data set Re, the stacked and coherency sections that are obtained using the fastest configurations CPU1 and GPU1. Figure 3 for data set Re.

Table 2: Execution time and costs for ZO-CRS in data set St

Configuration	Data set	Execution Time (minutes)	Cost (USD)
CPU1	Data set St	0.62	0.032
CPU2	Data set St	0.55	0.042
GPU1	Data set St	0.11	0.006
GPU2	Data set St	0.32	0.004
GPU3	Data set St	0.39	0.006
CPU1	Data set Re	17.01	0.868
CPU2	Data set Re	14.19	1.090
GPU1	Data set Re	1.25	0.064
GPU2	Data set Re	10.44	0.130
GPU3	Data set Re	12.16	0.182

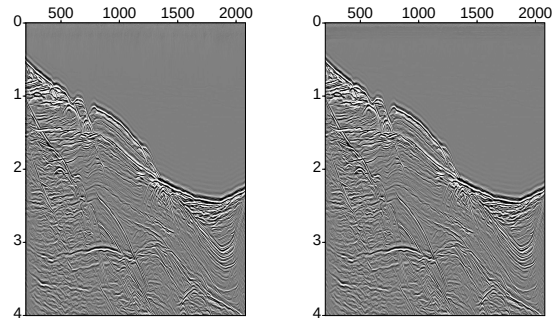

Figure 3: Comparison between CPU1 (left) and GPU1 (right) results for ZO-CRS in data set Re

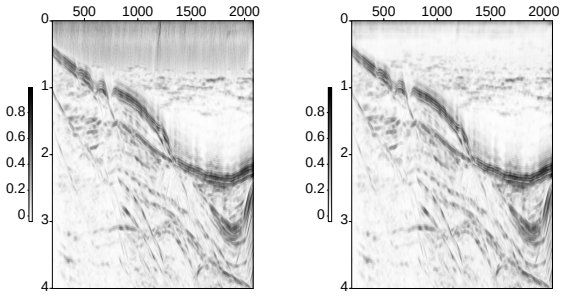

Figure 4: Comparison between CPU1 (left) and GPU1 (right) coherence for ZO-CRS in data set Re

It is to be noted that GPU instance implementations lead

consistently to lower costs and faster execution times, than the ones corresponding to their counterpart CPU instances.

Finite-Offset Common-Reflection-Surface

For the FO-CRS experiments, the better performance of GPU implementation as compared to CPU implementation, becomes much more evident. While having similar parameter estimation results. In fact, GPU costs turn out to be of the order of ten times less expensive than than their CPU counterparts, the main reason being the much shorter processing time. Table 3 shows the execution times and costs for each configuration for both data sets St and Re. A quick examination of Table 1) readily explains this situation. In spite of the fact that, in the AWS EC2 platform, both CPU1 and GPU1 have the same cost per hour, the GPU1 configuration offers much shorter execution times across the board, thus becoming much less expensive than the CPU1 configuration. In the same way as in the previous ZO-CRS case, stacked and coherency sections for the FO-CRS are shown in Figures 5 and 5, respectively. In those figures, results are shown for the few first (shorter) offsets only, as these are sufficient for our illustration purposes.

Table 3: Execution time and costs for FO-CRS

Configuration	Data set	Execution Time (minutes)	Cost (USD)
CPU1	Data set St	14.36	0.732
CPU2	Data set St	13.61	1.05
GPU1	Data set St	1.13	0.058
GPU2	Data set St	8.76	0.109
GPU3	Data set St	10.00	0.150
CPU1	Data set Re	843.59	44.27
CPU2	Data set Re	688.01	55.58
GPU1	Data set Re	69.27	3.53
GPU2	Data set Re	583.97	7.30
GPU3	Data set Re	709.72	10.65

As a useful technical remark, we observe that the DE implementation also used a queue system to keep the GPU processing traces for each offset. More specifically, the algorithm described by the flowchart in Figure 1 is executed asynchronously for each offset, so as to keep the device always busy and with reduced downtime, thus improving general performance. In the CPU processing, we kept the parallelism using multiple CPU threads to process each a CDP, each thread looping through the offsets.

Zero Offset Non-hyperbolic Common Reflection Surface

This last subsection presents the results obtained for the case of the zero-offset non-hyperbolic CRS. Table 4 shows the execution time and cost for each configuration and data set.

The stacked and coherency sections obtained for the real data set Re are shown in Figure 7 and 8, respectively. Consistently with the previous cases, we observe that GPU configurations offer a much better performance (i.e., lower costs and execution times) than their CPU counterparts.

Discussion

When comparing the CPU results for all three experiments, we can see that while CPU2 offered faster performance overall, with execution performance similar to the GPU2

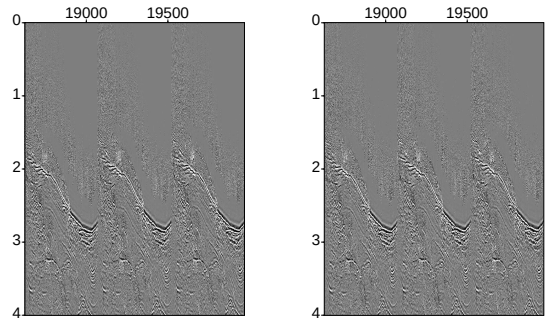

Figure 5: Comparison between CPU1 (left) and GPU1 (right) results for FO-CRS in data set Re

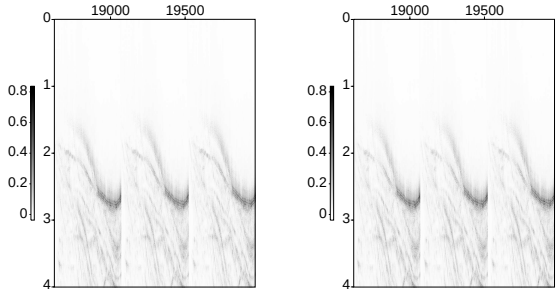

Figure 6: Comparison between CPU1 (left) and GPU1 (right) coherency for FO-CRS in data set Re

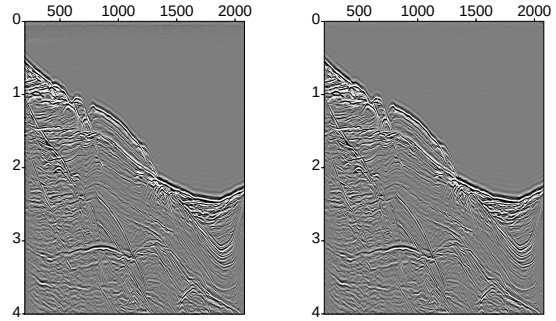

Figure 7: Comparison between CPU1 (left) and GPU1 (right) results for ZO-NCRS in data set Re

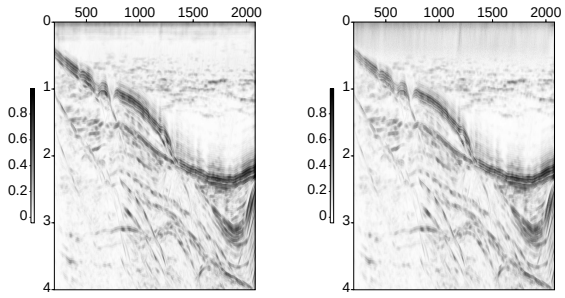

Figure 8: Comparison between CPU1 (left) and GPU1 (right) coherency for ZO-NCRS in data set Re

Table 4: Execution time and costs for ZO-NCRS

Configuration	Data set	Execution Time (minutes)	Cost (USD)
CPU1	Data set St	0.51	0.026
CPU2	Data set St	0.44	0.034
GPU1	Data set St	0.08	0.004
GPU2	Data set St	0.33	0.004
GPU3	Data set St	0.40	0.006
CPU1	Data set Re	14.93	0.761
CPU2	Data set Re	12.47	0.958
GPU1	Data set Re	1.31	0.067
GPU2	Data set Re	10.38	0.130
GPU3	Data set Re	12.79	0.192

and GPU3 configurations, standing usually closer to the GPU3 results. However, in all cases the CPU2 configuration was the most expensive and such good results do not justify its high cost. Furthermore, even though CPU2 offers similar performance as GPU2 and GPU3, its cost per hour is at least five times more expensive. As a consequence, to match the execution time of GPU2 and GPU3, the CPU2 will also cost five times more. Note that in Figures 4, 6 and 8 both GPU and CPU options had negligible differences in the coherence calculation for the parameters chosen.

The GPU1 configuration granted the best performance overall and usually the lowest cost, due to its shorter execution times. In all data set Re scenarios, GPU1 offered results at least ten times less expensive than both CPU2 and CPU1 or two to three times less expensive than even the other GPU configurations. Additionally, comparing CPU1 and GPU1 instances shows how even paying the same price per hour, there are still large performance gaps due to how the program can deal with parallelism.

In Table 5 the total cost for the best CPU and GPU selections are compiled. For all programs the best option for CPU was the CPU1 configuration, while the best GPU, and overall, option was the GPU1 configuration.

Table 5: Best cost for data set Re comparing CPU and GPU

	CPU		GPU	
Program	Price	Configuration	Price	Configuration
ZO-CRS	0.87	CPU1	0.06	GPU1
FO-CRS	44.27	CPU1	3.53	GPU1
ZO-NCRS	0.76	CPU1	0.07	GPU1

Conclusions

We implemented a GPU accelerated version of the ZO-CRS, ZO-NCRS and FO-CRS programs to explore what improvement we could get from embarrassingly parallel programs. While it is well documented that GPU acceleration can increase the performance of such programs, we took a step further and explored how the cost scaled with this performance increase.

Our results showed that we managed to complete our tasks with shorter time spans and up to ten times less money using GPU configurations than CPU configurations, all while retaining similar outputs. Also, we noticed in our experiments that even less powerful GPU configurations can provide us performance comparable to higher end CPU machines while costing only a fraction of the price.

In conclusion, the parameter estimation of seismic processing algorithms can easily be accelerated using GPUs, providing not only time reductions, but also cost reductions when processing in environments such as the cloud. Note that these cost results can vary depending on the cloud provider, further investigation of the comparison between cost-effectiveness of GPU versus CPU configurations are needed, however, it is clear that the GPU parallelism improves performance.

Acknowledgments

This work was possible thanks to the support of Petrobras and Fapesp (2013/08293-7). The authors also thank the High-Performance Geophysics team for technical support.

References

Benedicto, C., I. L. Rodrigues, M. Tygel, M. Breternitz, and E. Borin, 2017, Harvesting the computational power of heterogeneous clusters to accelerate seismic processing: Presented at the 15th International Congress of the Brazilian Geophysical Society & EXPOGEF, Rio de Janeiro, Brazil, 31 July-3 August 2017, Brazilian Geophysical Society.

Billette, F., and G. Lambaré, 1998, Velocity macro-model estimation from seismic reflection data by stereotomography: Geophysical Journal International, **135**, 671–690.

Borin, E., C. Benedicto, I. L. Rodrigues, F. Pisani, M. Tygel, and M. Breternitz, 2016, Py-pits: A scalable python runtime system for the computation of partially idempotent tasks: 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 7–12.

Coimbra, T. A., A. Novais, and J. Schleicher, 2016, Offset-continuation stacking: Theory and proof of concept: GEOPHYSICS, **81**, V387–V401.

Dell, S., D. Gajewski, and M. Tygel, 2014, Image-ray tomography: Geophysical Prospecting, **62**, 413–426.

Faccipieri, J. H., T. A. Coimbra, and R. Bloot, 2018, Stretch-free generalized normal moveout correction: Geophysical Prospecting, **67**, 52–68.

Fomel, S., and R. Kazinnik, 2012, Non-hyperbolic common reflection surface: Geophysical Prospecting, **61**, 21–27.

Jäger, R., J. Mann, G. Höcht, and P. Hubral, 2001, Common-reflection-surface stack: Image and attributes: GEOPHYSICS, **66**, 97–109.

Neidell, N. S., and M. T. Taner, 1971, SEMBLANCE AND OTHER COHERENCY MEASURES FOR MULTICHANNEL DATA: GEOPHYSICS, **36**, 482–497.

Okita, N., T. Coimbra, C. Rodamilans, M. Tygel, and E. Borin, 2018, Using spits to optimize the cost of high-performance geophysics processing on the cloud: Presented at the First EAGE Workshop on High Performance Computing for Upstream in Latin America, Santander, Colombia, 21-22 September 2018, EAGE.

Storn, R., and K. Price, 1997: Journal of Global Optimization, **11**, 341–359.

Zhang, Y., S. Bergler, and P. Hubral, 2001, Common-reflection-surface (CRS) stack for common offset: Geophysical Prospecting, **49**, 709–718.