Received: 20 April 2020 | Revised: 12 January 2021 | Accepted: 14 January 2021

DOI: 10.1002/cpe.6212

RESEARCH ARTICLE

WILEY

Energy efficiency and portability of oil and gas simulations on multicore and graphics processing unit architectures

Matheus S. Serpa[1] | Pablo J. Pavan[1] | Eduardo H. M. Cruz[2] | Rodrigo L. Machado[3] | Jairo Panetta[3] | Antônio Azambuja[4] | Alexandre S. Carissimi[1] | Philippe O. A. Navaux[1]

[1]Informatics Institute, Federal University of Rio Grande do Sul – UFRGS, Porto Alegre, Brazil

[2]Federal Institute of Paraná – IFPR, Paranavaí, Brazil

[3]Computer Science Division, Aeronautics Institute of Technology – ITA, São José dos Campos, Brazil

[4]Petrobras – Petróleo Brasileiro S.A., Rio de Janeiro, Brazil

Correspondence
Matheus S. Serpa, UFRGS, Bento Gonçalves 9500, Porto Alegre, RS, Brazil.
Email: msserpa@inf.ufrgs.br

Funding information
Conselho Nacional de Desenvolvimento Científico e Tecnológico; Coordenação de Aperfeiçoamento de Pessoal de Nível Superior; Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul, Grant/Award Number: 2016/2551-0000 488-9; Petrobras, Grant/Award Numbers: 2016/00133-9, 2018/00263-5

Summary

Reverse time migration (RTM) simulation is the basis of the seismic imaging tools used by the oil and gas industry. Developers have been porting their simulations to the new high-performance computing architectures, providing faster and more accurate results at each new generation. However, several challenges arrive when trying to achieve high performance on these new architectures. The first one is to choose the architecture that best fits the kind of simulation. After that, researchers should choose the API used to implement the simulation code. These two decisions are strongly related to the effort, performance, and energy efficiency of the simulations. In this article, we propose three optimizations for an oil and gas application, which reduce the floating-point operations by changing the equation derivatives. We evaluate these optimizations in different multicore and GPU architectures, investigating the impact of different APIs on the performance, energy efficiency, and portability of the code. Our experimental results show that the dedicated CUDA implementation running on the NVIDIA Volta architecture has the best performance and energy efficiency for RTM on GPUs, while the OpenMP version is the best for Intel Broadwell in the multicore. Also, the OpenACC version, which has a lower programming effort and executes on both architectures, has an up to 20% better performance and energy efficiency than the nonportable ones.

KEYWORDS

code portability, HPC, oil and gas simulation, performance optimization, reverse time migration

1 | INTRODUCTION

Exploration geophysics has been fundamental in the search for energy resources such as oil and gas. However, high drilling costs, with less than 50% accuracy per drill, limit its use.[1,2] Thus, the oil and gas industry relies on software focused on high-performance computing (HPC) as an economically viable way to reduce costs. The basis of many exploration geophysics software engines is wave propagation simulation. For example, in seismic imaging, modeling, migration, and inversion tools, this simulation can be used. These tools are built based on partial differential equation (PDE) solvers, where the solved PDE in each case defines the accuracy of the approximation to real physics.

The approximation of the acoustic wave propagation model is the current basis of seismic imaging tools. It has been extensively applied in recent years to oil and gas reservoirs with saline dome imaging potential. The industry continuously ported these applications to the latest HPC hardware available to remain competitive in the marketplace. Simultaneously, HPC architectures have evolved, and performance improvement just

by increasing the clock frequency is no longer possible. Because of that, oil and gas industries replace their old solutions by new ones based on multicore and GPU (graphics processing unit) architectures.[3,4]

The past decade has seen a trend toward building high-performance systems with dedicated devices and accelerators that produce a good return on energy efficiency (FLOPs/Watt).[5] Among the alternatives available, accelerator manufacturers have dedicated their efforts to provide tens to thousands of low-frequency processing units, such as GPUs and other accelerators.[6] Even more traditional multicore processors, such as the AMD EPYC and Intel Xeon, include dozens of cores in the processors, working at high frequencies.

Several challenges must be taken into consideration to achieve high performance on these architectures. One of the most important aspects is the memory subsystem's behavior, as memory access plays a crucial role in the performance.[7,8] Energy consumption and energy efficiency are also points to consider.[9-11] Also, heterogeneous computing has gained strength, increasing the complexity of using current architectures.[12,13] This increase in complexity occurs because these architectures have memories and cores with different latency characteristics, number of cores, and execution units. Also, data must be copied by the programmer from one memory to another, increasing the difficulty of programming. Several more generic and high-level programming models have emerged, which allow the writing of portable code that executes on multicore and GPU architectures without any change.[14]

In this article, we implement a baseline version and propose three optimizations to improve the performance and energy efficiency of a reverse time migration (RTM) based application, an seismic imaging method used by oil and gas companies. We evaluate and compare the computational performance, energy efficiency, and portability of the four application versions implemented using three parallel APIs (application programming interfaces). The contributions of this article are:

- A detailed description of four versions (baseline, Der1Der1, Der1Der1LM, and Der1Der1HM) of an oil and gas simulator based on established reverse time migration application as detailed by Fletcher.[15]
- The four versions implemented with OpenACC,[16] OpenMP,[17] and CUDA.[18]
- A comprehensive performance analysis to pinpoint the limitations of each implementation running on Intel Broadwell,[19] Intel Skylake,[20] NVIDIA Pascal,[21] and NVIDIA Volta[22] architectures.
- A discussion on the trade-off among performance, energy efficiency, and portability.

This article is an extension of our previous work,[23] published in the WSCAD 2019 conference. It includes the proposal and implementation of the Der1Der1LM and Der1Der1HM versions and evaluates newer hardware models such as Intel Skylake and NVIDIA Volta. We revised and increased the related work and discuss the trade-off among performance, energy efficiency, and portability.

We organized this article as follows. Section 2 discusses related work. Section 3 presents the reverse time migration application and details of its implementation. Section 4 presents the experimental design, including the architectures, metrics, and detailed information about the experiments. It also provides an assessment of performance, energy efficiency, and portability of different versions of the application and a discussion on each architecture and optimization's appropriateness. Finally, Section 5 presents concluding remarks and future work.

We make our code, data. and R scripts publicly available at https://gitlab.com/msserpa/oil-and-gas-ccpe, following a reproducible methodology.

2 | RELATED WORK AND MOTIVATION

Very few works optimize oil and gas models mathematically, implement them in different parallel APIs, and evaluate the trade-off between portability, performance, and energy efficiency on several computer architectures. Oil and gas models are much more commonly optimized for specific architectures than generically with mathematical optimizations.

This section presents related work focused on stencil optimization and investigation of the trade-off between portability and energy efficiency. Finally, we discuss and motivate the relevance of this work.

2.1 | Stencil optimization

Some works seek to improve the performance of stencil-based applications using different architectures and proposing optimizations. Abdelkhalek et al.[24] designed a GPU-based RTM simulator using the NVIDIA CUDA API. They showed that compared with a multicore implementation, the CUDA one is up to 10 times faster. Cabezas et al.[25] ported an RTM application to GPUs using CUDA. The GPU version presented a performance improvement of 9x compared with an Intel multicore and 14× when compared with an IBM processor.

Moussa et al.[26] reported the outline and preliminary results of their RTM simulator implemented for GPUs using CUDA. The preliminary results showed an execution time reduction of 10× approximately. Araya-Polo et al.[27] proposed RTM implementations for different architectures and

evaluated each version limitations. The results showed that accelerators like GPUs and FPGAs outperform multicore architectures by one order of magnitude.

Clapp et al.[3] discussed some algorithmic and optimization decisions used to implement an RTM kernel for different architectures. They explained the reasons for their choices. They conclude the paper with a discussion without showing measured results. Liu et al.[28] presented the implementation process of an RTM application for GPUs using CUDA, focusing on the analysis of numerical aspects. They mentioned that their GPU version was one order of magnitude faster than a CPU one.

Fu et al.[29] presented an FPGA-based implementation for the 3D RTM algorithm. They discussed some challenges faced when implementing the code. Their results showed that the FPGAs versions were up to 8× faster than an optimized CPU version. Fu et al.[30] implemented an RTM version for the Blue Gene/Q architecture using the MPI API. They discussed some details about their optimizations. The results showed a speedup of 14.9× over the Blue Gene/P architecture.

Clapp et al.[31] discussed how new computer architectures change the way RTM applications are implemented. The article focused on a discussion without showing any measured results. Nunes-do-Rosario et al.[32] implemented a parallel version of a RTM algorithm for AMD processors. They achieved a speedup of 17× on a 24-core processor.

Carrijo et al.[33] analyzed the performance of a 3D matrix-based synthetic application on GPUs, focusing on the proper use of the memory hierarchy. They conclude that the highest performing code combines the reuse of read-only cache by inserting the Z dimension loop into the kernel and reusing processor registers, which improves stencil performance by up to 3.3× compared with the classical stencil formulation.

Serpa et al.[34] proposed several optimization strategies for a wave propagation model for six CPU and GPU architectures. The work focused on improving cache memory usage, vectorization, load balancing, and memory hierarchy location. Nesi et al.[35] presented task-based parallel strategies for a CFD application that targets heterogeneous multi-CPU/multi-GPU computing resources. Their results indicated that the ghost cell strategy provides the best speedup considering the simulation time when the GPU resources still have available memory.

2.2 | Energy efficiency and portability

Some works focused on the energy efficiency of stencil-based applications, while others focused on metrics to measure the portability of codes. Qawasmeh et al.[36] developed an OpenACC implementation of seismic modeling and reverse time migration (RTM) algorithms for two NVIDIA GPUs. They reached performance improvements of up to 10×. Pavan et al.[37] proposed architecture-based optimizations to improve the energy efficiency of geophysics applications running on NVIDIA GPUs. Their results showed that a combination of shared memory and reuse of registers improve performance and energy efficiency by up to 54.1%.

Bittencourt et al.[38] compared the performance of FPGAs, GPUs, and CPUs for reverse time migration simulations. They showed that GPUs are up to 9% faster than FPGAs; however, FPGAs are 60% more energy efficient. Calore et al.[39] improved the lattice Boltzmann method's energy efficiency by up to 24%. Their approach changed the CPU core and uncore frequencies based on each kernel workload.

Daniel et al.[40] extended the work of Pennycook et al.[41] and proposed a metric for performance portability of codes written in OpenACC and Kokkos C++. The authors concluded that Pennycook's metric was sensitive to the problem size and then proposed a new one.

2.3 | Discussion and motivation

Table 1 summarizes the related work, showing some aspects of each article. The first column presents the authors and the year of the works. The second and third columns, when marked, indicate that the work implemented their RTM algorithms using different APIs and evaluated it in more than one architecture. The fourth and fifth columns indicate that the work investigates the impact of code portability and architectures' energy efficiency.

Nunes-do-Rosario et al.[32] and Nesi et al.[35] designed parallel versions of CFD simulations using different parallel programming paradigms. However, they do not take into account portability and energy efficiency. Andreolli et al.[42] and Serpa et al.[34] proposed several low-level optimizations to individual system resources, adding more complexity to programmers and not taking into account the portability of the code. Our work seeks to analyze and optimize a seismic image simulation widely used by oil and gas companies such as Brazilian Petroleum Corporation (Petrobras) and investigate the impact of portability in performance and energy efficiency.

Abdelkhalek et al.,[24] Cabezas et al.,[25] Moussa et al.,[26] Liu et al.,[28] and Carrijo et al.[33] implemented and propose optimizations for RTM applications on GPUs. However, they focused on CUDA programming, not taking into consideration the portability and energy efficiency effects.

Pavan et al.[37] and Calore et al.[39] proposed different GPU optimizations to a stencil code evaluating performance and energy efficiency. We go beyond by assessing the portability of a real-world application for different architectures and not only GPUs.

Araya-Polo et al.,[27] Fu et al.,[29] Fu et al.,[30] and Bittencourt et al.[38] implemented the RTM algorithm using different APIs in multiple computer architectures. However, their analysis ceases to investigate aspects like portability and energy efficiency.

TABLE 1 Comparison of our work with related work.

Proposal	Different APIs	Multiple architectures	Portability	Energy efficiency
Abdelkhalek et al.,[24] 2009		✓		
Cabezas et al.,[25] 2009		✓		
Moussa et al.,[26] 2009		✓		
Araya-Polo et al.,[27] 2010	✓	✓		
Clapp et al.,[3] 2010		✓		
Liu et al.,[28] 2010		✓		
Fu et al.,[29] 2011		✓		
Fu et al.,[30] 2013		✓		
Clapp et al.,[31] 2015		✓		
Nunes-do-Rosario et al.,[32] 2015	✓			
Qawasmeh et al.,[36] 2017	✓	✓	✓	
Carrijo et al.,[33] 2018				
Pavan et al.,[37] 2018				✓
Bittencourt et al.,[38] 2019	✓	✓		✓
Calore et al.,[39] 2019				✓
Daniel et al.,[40] 2019	✓		✓	
Serpa et al.,[34] 2019	✓	✓		
Nesi et al.,[35] 2020		✓		
Our work	✓	✓	✓	✓

Clapp et al.[3] and Clapp et al.[31] discussed important aspects about implementing and choosing the best computer architectures for RTM algorithms. However, their work only discusses those aspects without showing an experimental evaluation as we do. Also, they did not mention the programming efforts and portability of the code.

Qawasmeh et al.[36] developed two OpenACC applications. However, they evaluated their performance only on GPU architectures, neither comparing with CPUs nor evaluating its energy efficiency. Daniel et al.[40] proposed a metric for performance portability. However, they do not take energy efficiency into account, one of the requirements for exascale computing, as an important aspect of their evaluated scenarios.

In our work, we propose different optimizations for an oil and gas application implemented in OpenMP, OpenACC, and CUDA. Our goal is to verify the performance, energy efficiency and the relation between code portability and these two variables. The next section presents our oil and gas application with its baseline version, which is the same as implemented by most of the related work and the three proposed optimized versions.

3 | REVERSE TIME MIGRATION FOR OIL AND GAS SIMULATION

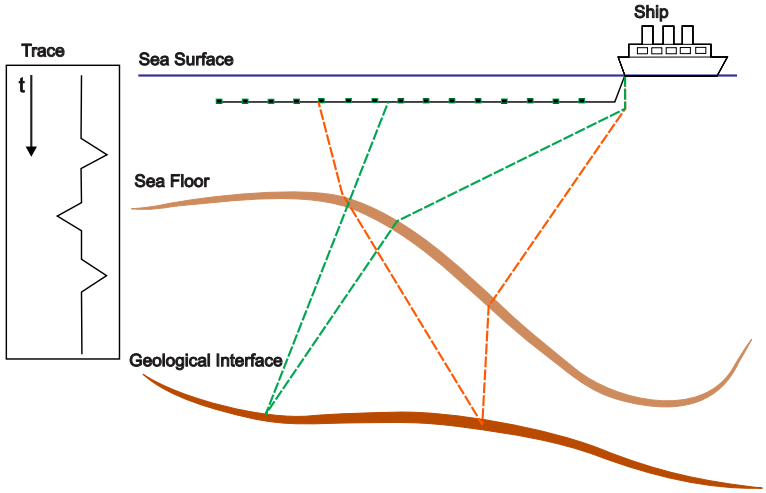
Reverse time migration (RTM) is a seismic imaging method to map the subsurface reflectivity using recorded seismic waveforms. It involves extrapolating seismic data from the recording receivers to subsurface reflectors, in reversed direction compared with the direction of wave propagation from the source.[43] People can find seismic anisotropy simulations in many exploration areas.[44-47]

Figure 1 represents a wave propagation and data collection in a seismic survey. From time to time, equipment attached to the ship emits waves that reflect and refract from underground medium changes. Eventually, these waves return to the sea surface and precise microphones coupled to cables towed by the ship collect them. The set of signals received in each microphone overtime constitutes a seismic trace. For each wave emission, we recorded the seismic traces of all cable earphones. The ship moves and emits signals over time.

Fletcher[15] developed a pseudo-acoustic wave propagation model in tilted transversely isotropic (TTI) media, one kind of weak anisotropy, feasible with RTM. Equation (1) represents his wave model.

$$\begin{aligned}\frac{\partial^2 p}{\partial t^2} &= v_{px}^2 H_2 p + \alpha v_{pz}^2 H_1 q + v_{sz}^2 H_1 (p - \alpha q),\\ \frac{\partial^2 q}{\partial t^2} &= \frac{v_{pn}^2}{\alpha} H_2 p + v_{pz}^2 H_1 q - v_{sz}^2 H_2 \left(\frac{1}{\alpha} p - q\right),\end{aligned} \tag{1}$$

FIGURE 1 Data collection in marine seismic survey



where $p(x, y, z)$ is the pressure of the wave, q is an auxiliary variable, v_{pz} is the P-wave velocity in the direction normal to the symmetry plane, v_{pn} is the P-wave normal moveout velocity, v_{px} is the P-wave velocity in the symmetry plane. All P-wave velocities are bound by equations $v_{pn} = v_{pz}\sqrt{1+2\delta}$ and $v_{px} = v_{pz}\sqrt{1+2\epsilon}$, where δ and ϵ are anisotropic parameters defined by Thomsen.[48] v_{sz} is the SV-wave velocity normal to the symmetry plane and α comes from the solution of the P-SV dispersion relation DPE.[49] Usually, α takes the value of 1.

H_1 and H_2 are linear operators defined as

$$H_1 = sin^2\theta cos^2\phi \frac{\partial^2}{\partial x^2} + sin^2\theta sin^2\phi \frac{\partial^2}{\partial y^2} + cos^2\theta \frac{\partial^2}{\partial z^2} + sin^2\theta sin2\phi \frac{\partial^2}{\partial x \partial y} + sin2\theta sin\phi \frac{\partial^2}{\partial y \partial z} + sin2\theta cos\phi \frac{\partial^2}{\partial x \partial z},$$
$$H_2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} - H_1, \quad (2)$$

where θ is the dip angle and ϕ the azimuth angle.

We wrote an anisotropic wave propagation modeling program in the C language for a project in cooperation with Petrobras, a Brazilian oil and gas company. The project aims to analyze the performance of different architectures running oil and gas simulations. We discretize the problem using finite differences and implement the application in different APIs as follows: an OpenMP version[17] to exploit only multicore processors, a CUDA version[18] for GPUs, and a single OpenACC version[16] for both architectures. We also considered OpenMP offloading. However, it was not possible due to nonavailability of a compiler that supports it.

In the following subsections, we describe the Baseline version and the optimized versions. We call the optimizations *Der1Der1*, Der1Der1 High Memory, also referenced as *Der1Der1HM*, and Der1Der1 Low Memory, referenced as *Der1Der1LM*.

3.1 | Baseline version

The baseline version is a direct implementation from Equations (1) and (2). We stored the data in single-dimensional arrays from direction x to z. The main kernel of the simulation nested three *for* loops over axis z, y, and x. The outermost loop is z, and the innermost is x. Inside these nested loops, we implement the Fletcher equations. Algorithm 1 shows the baseline version.

The first step is to calculate the second-order and cross-derivatives from vector p, and then, with the results, we calculate the operators H_1 and H_2 over p. We repeated this process for vector q. Afterward, with the results from operators H_1 and H_2 calculated for p and q, we can step forward in time using Equation (1).

All three loops are independent, which makes the parallelism easy. For the OpenMP version we create a parallel region encompassing all loops, parallelizing the outmost *for* with the *#pragma omp for* directive and the innermost with *#pragma omp simd* for vectorization. The OpenACC approach is similar, with a parallel region encompassing all loops but using the directive *#pragma acc loop independent* before each loop. The same strategies will be repeated in the optimizations.

For the CUDA version, we created one kernel that iterates over the z dimension and implements the Fletcher algorithm. We call the kernel creating one thread for each point in the xy plane. Due to GPGPU's differences and language specifications, more days were required to write the code and validate the CUDA versions compared with the multicore ones.

Algorithm 1. Baseline version

```
for (int z=bord; z<zSize-bord; z++) {
    for (int y=bord; i<ySize-bord; y++) {
        for (int x=bord; x<xSize-bord; x++) {
            int i = ind(x,y,z);

            // Calculate derivatives on p
            float pxx = DerXX(pc, i);
            ...

            // Calculate operators H1 and H2 for p
            float h1p = kxx[i]*pxx + ... + kyz[i]*pyz;
            ...

            // Repeat process for q

            // Step forward in time, C depends on operators H1 and H2
            pp[i]=2.0f*pc[i] - pp[i] + Cp*dt*dt;
            qp[i]=2.0f*qc[i] - qp[i] + Cq*dt*dt;
        }
    }
}
```

3.2 | Der1Der1 optimization

We profile the Baseline version using the PAPI (performance application programming interface)[50] for multicore architectures and NVIDIA Visual Profiler[51] for GPUs. We found that the calculation of cross-derivatives demanded 76% of the floating-point operations. The results are similar for the sequential, OpenMP and CUDA versions.

Further investigation showed a similarity of 82% in the arithmetic operations of the cross-derivatives. We can understand this problem by analyzing two consecutive derivatives. Suppose we are calculating $\partial^2 p/\partial x \partial y$ in all points of the plane xy for a fixed z. In particular, to calculate the derivatives on the point (x_0, y_0), first we calculate the derivative in y over the points $\{(x_{-4}, y_0), \ldots, (x_4, y_0)\}$ and then use the result to calculate the cross-derivatives. Moving to the next point, (x_1, y_0), we are going to calculate the first derivative over the points $\{(x_{-3}, y_0), \ldots, (x_5, y_0)\}$. Comparing both sets, we can see that the derivative in y were repeated for the points $\{(x_{-3}, y_0), (x_{-2}, y_0), (x_{-1}, y_0), (x_2, y_0), (x_3, y_0), (x_4, y_0)\}$. Another way to see this problem, is that $\partial p(x_0, y_0)/\partial y$ is necessary for the cross-derivative of the points $\{(x_{-4}, y_0), \ldots, (x_4, y_0)\}$, which means, one proper calculation and seven repetitions.

In this sense, we seek to reduce the number of operations to calculate cross-derivatives. The first optimization, which we call *Der1Der1*, focuses on that issue. It calculates the first derivatives on x and y over the whole cube, saving the results in auxiliary vectors and using that to calculate the cross-derivatives $\partial^2/\partial z \partial x$, $\partial^2/\partial z \partial y$ and $\partial^2/\partial y \partial x$, avoiding the repetitions. After that, the Fletcher[15] algorithm follows unchanged. Algorithm 2 shows the implementation.

This approach requires four new vectors to store the derivatives' partial results from vectors p and q. Therefore, it generates more data transfers instead of repeating the calculations. Ultimately, we decrease the number of floating-point operations while increasing the number of memory operations. The following two optimizations focused on mitigating the impact on the performance of these new memory operations.

We split the algorithm into two parts in the implementation: the calculus of the first derivatives and the Fletcher equations. Each part consists of sets of nested loops, which we parallelized using the same strategies as the baseline version for OpenMP and OpenACC.

3.3 | Der1Der1HM optimization

The second optimization, called *Der1Der1HM*, does not reduce the amount of memory used, but it makes better reuse of data aiming for a better cache hit. It is also a middle step between the first optimization, *Der1Der1*, and the last one *Der1Der1LM*. The main difference from the previous implementation is the first-order derivative calculation.

Der1Der1 calculates the first-order derivative over the whole cube before it calculates the cross-derivatives. Der1Der1HM calculates each plane xy at a specific time. Fletcher algorithm[15] needs only the cross-derivatives over the plane xy to move it one step forward in time. For that, it required $\partial/\partial x$ and $\partial/\partial y$ from all points in four planes below, four planes above and the current one. The derivatives in the planes above and below are required for $\partial^2/\partial z \partial x$ and $\partial^2/\partial z \partial y$, while the derivatives in the current plane for $\partial^2/\partial y \partial x$. For each iteration in index z_i, *Der1Der1HM* uses the following loop invariant: all $\partial/\partial x$ and $\partial/\partial y$ from planes z_{i+3} and below are saved and precalculated. It calculates all $\partial/\partial x$ and $\partial/\partial y$ in the plane z_{i+4} to move the plane in index z_i one step forward in time, fulfilling the requirements for the cross-derivatives, and following with the Fletcher[15] algorithm. When it moves to the next plane, z_{i+1}, the loop invariant is already satisfied. Algorithm 3 shows the implementation of *Der1Der1HM* optimization.

Algorithm 2. Der1Der1 optimization

```
// Calculates x derivative on p
for (int z=0; z<zSize; z++) {
    for (int y=0; i<ySize; y++) {
        for (int i=ind(bord,y,z); i<ind(xSize-bord,y,z); i++) {
            px[i] = DerX(pc,i);
        }
    }
}

// Repeat process for other required first-order derivatives

// Starts Fletcher calculation
for (int z=bord; z<zSize-bord; z++) {
    for (int y=bord; i<ySize-bord; y++) {
        for (int x=bord; x<xSize-bord; x++) {
            int i = ind(x,y,z);

            // Calculate derivatives on p
            float pyx = DerY(px, i);
            float pzx = DerZ(px, i);
            ...

            // Algorithm follows unchanged
        }
    }
}
```

Algorithm 3. Der1Der1HM optimization

```
// Satisfying preconditions
for (int z=0; z<2r; z++) {
    for (int y=0; i<ySize; y++) {
        for (int i=ind(bord,y,z); i<ind(xSize-bord,y,z); i++) {
            px[i] = DerX(pc,i);
        }
    }
}
...

// Starts main loop
for (int z=bord; z<zSize-bord; z++) {
    // Calculates first order derivatives required for current z
    for (int y=0; i<ySize; y++) {
        for (int x=bord; x<xSize-bord; x++) {
            px[ind(x,y,z+r)] = DerX(pc,ind(x,y,z+r));
        }
    }
    ...

    // Follows Fletcher algorithm using first-order derivatives
}
```

The implementation is quite similar to the previous optimization, except that we do not parallelize the zindex on the main loop because it requires synchronization. OpenMP and OpenACC keep the same parallelism strategies while CUDA moves the *z* loop out of the kernels and for each *z* calls the derivative kernel and the Fletcher kernel.

3.4 | Der1Der1LM optimization

Der1Der1LM is an optimization from Der1Der1HM that uses less memory. We aim to reduce the burden in the bandwidth introduced by the first optimization, Der1Der1.

As we mentioned before, the calculation of the cross-derivatives from the plane *xy* requires $\partial/\partial x$ and $\partial/\partial y$ from four planes below, four planes above, and the current point. Therefore, we do not need to store the first-order derivatives below the plane z_{i-4}. *Der1Der1LM* changes the loop

TABLE 2 Compiler flags information of the different implementations.

	Parameter	Value
System	OS	Debian 10
	Kernel	Linux 4.19.118
OpenMP	Compiler	Intel C
		19.1.0.166
	Flags	-O3 -qopenmp
OpenACC	Compiler	PGI
		19.10-0
	Flags	-O3 -acc -ta=tesla:cc60,cc70,multicore
CUDA	Compiler	NVIDIA CUDA
		10.2
	Driver	440.33.01
	Flags	-O3 -gpu-architecture sm_60,sm_70

invariant to: all $\partial/\partial x$ and $\partial/\partial y$ from planes z_{i+3} and z_{i-4} are stored and precalculated. This change reduces the extra memory from the order of an entire cube to a plane. Besides the difference in the loop invariant, Der1Der1LM is equal to Der1Der1HM.

The main difference between the Der1Der1HM (Algorithm 3) and this optimization is that this one uses a circular buffer data structure to keep the nine planes necessary for the calculations. In each iteration, the planes z_{i-4} and z_{i+4} are stored in the positions b_0 and b_8 of the buffer, respectively. At the end of a loop iteration, we shift the buffer to reuse data for the next iterations. We implement the parallelism in the same way as in the previous version.

4 | EXPERIMENTS AND PERFORMANCE RESULTS

This section presents the performance analysis of an oil and gas application over different multicore and GPU architectures. First, we present our experiment design, following a performance and energy efficiency analysis of different implementations and optimizations.

4.1 | Experimental design

The OpenMP, OpenACC, and CUDA implementations were executed on 12 increasing size grids, such that the grid size in either direction is a multiple of 32. We choose the value 32 because it is the size of the CUDA warps. Since the results were similar to each other, we show, in this work, the results for a $472 \times 472 \times 472$ cube, which is the largest size that fits in GPU's main memory. The experiments were randomly executed at least 30 times with the number of threads of each architecture. The result graphs show the mean values and the 95% confidence interval according to the Student's t-distribution.[52] Table 2 shows further information about the compiler and flags of each implementation.

We performed experiments in the Broadwell and Pascal environments from our research group cluster[1]. Broadwell has two 22-core Intel Xeon E5-2699 v4 processors. Each core supports a two-way simultaneous multithreading (SMT),[53] allowing up to 88 threads to run. The cores of each processor share an L3 cache while having private L1 and L2 caches. Pascal is an NVIDIA P100 GPU with 3584 CUDA Cores. We gathered information about the hardware topology using hwloc.[54]

We also run experiments in the `chifflot` cluster from Grid 5K.[55] Each node of `chifflot` is composed of two Intel Xeon Gold 6126 processors (Skylake), where each processor consists of 12 physical cores, allowing the execution of 48 threads with SMT. `chifflot` also employs NVIDIA V100 GPUs (Volta) with 5120 CUDA cores. Table 3 has detailed information on each computer architecture we used.

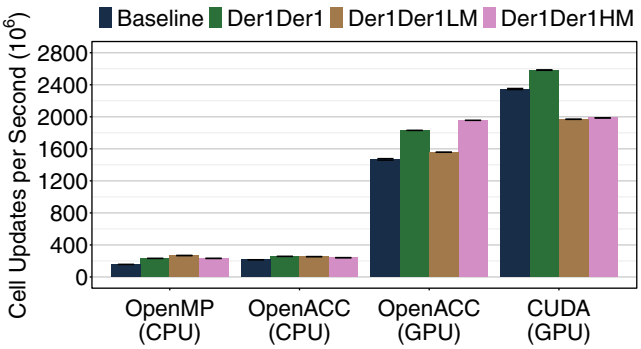
We show the results using the cell updates per second (CUPS) metric for performance and energy efficiency in cell updates per Joule (CUPJ). Performance is given by dividing the number of grid points calculated by the application execution time. We obtained the power consumption using intelligent platform management interface (IPMI),[56] which measures the power consumed by the motherboard and GPU. Finally, we obtained the energy efficiency by dividing the number of calculated grid points by the energy consumed. Despite the power measurements, we read it from IPMI every 200 ms.

[1] http://gppd-hpc.inf.ufrgs.br/.

TABLE 3 Configuration of the evaluated systems

System	Parameter	Value
Intel Broadwell	Processor	2× Intel Xeon E5-2699 v4, 2 × 22 *cores*, 2-SMT, 88 threads
	Cache/processor	22 × 32 KB L1I, 22 × 32 KB L1D, 22 × 256 KB L2, 55MB L3
	Main memory	256GB DDR4-2400
Intel Skylake	Processor	2 × Intel Xeon Gold 6126, 2 × 12 *cores*, 2-SMT, 48 threads
	Cache/processor	12 × 32 KB L1I, 12 × 32 KB L1D, 12 × 1024 KB L2, 19MB L3
	Main memory	192GB DDR4-2666
NVIDIA Pascal	GPU	NVIDIA Tesla P100, 56 × 64 *cores*, 3584 CUDA cores
	Cache	56 × 64 KB *shared*, 4096 KB L2, 56 × 24 KB L1/*texture (read-only)*
	Registers	56 × 256 KB
	Device memory	16GB 4096-bit HBM2
NVIDIA Volta	GPU	NVIDIA Tesla V100, 80 × 64 *cores*, 5120 CUDA cores, 640 Tensor cores
	Cache	80 × 128 KB L1/*shared*, 6144 KB L2, 80 × 24 KB *texture (read-only)*
	Registers	80 × 512 KB
	Device memory	32GB 4096-bit HBM2

FIGURE 2 Performance in millions of CUPS



4.2 | Trade-off between performance and portability of different optimizations

We first present a performance analysis of the implemented versions using the Intel Skylake and NVIDIA Volta architectures. This analysis leads to determine the cost of portability in multicore and GPU architectures, contrasting the performance of a portable implementation (OpenACC) compared with two architecture-specific nonportable implementations (OpenMP on multicore and CUDA on GPU). The portability of OpenACC is such that a compiler flag generates code for both multicore and GPU and an environment variable defines which architecture to use.

Figure 2 depicts the CUPS metric in millions on the Y-axis and the API used for implementation on the X-axis. The APIs used were OpenMP and OpenACC running on CPU, CUDA, and OpenACC running on GPU. In this plot, each color represents an optimization: Baseline, Der1Der1, Der1DerLM, and Der1DerHM. The results show that for multicore architectures, the cost of portability depends on the optimization. For Baseline, Der1Der1 and Der1DerHM, the performance of the portable implementation (OpenACC) is 27.4%, 9.8%, and 3.7% faster than the architecture-specific implementation (OpenMP). However, for the Der1Der1LM, which is the faster optimization in the Skylake CPU, OpenMP's performance is 5.2% better than OpenACC.

For GPUs, the performance of the different optimizations is even more notable than in multicore architectures. The CUDA versions are faster than the OpenACC ones, showing that, for GPUs, the cost for portability is higher than for multicore architectures. In case of the Baseline version, it costs 37.4% of the performance. For Der1Der1 and Der1Der1LM, the difference in performance was 29.2% and 20.9%. Finally, for Der1Der1HM, the optimization that stores the entire derivative fields in memory, the difference is only 1.6%. These results happen because the memory hierarchy of GPUs is very different from the multicores. In the Pascal architecture, the L1 and the texture cache shared the same resource, while in Volta, the L1 shares the same resource as the shared memory. Moreover, our optimizations change the arithmetic intensity in both amounts of floating-point operations and memory accesses, which lead to different performance results in these architectures with distinct memory hierarchies.

The second analysis we present is related to the performance of different optimizations in a specific API. The performance of the optimized versions using OpenMP was up to 41.6% better than the baseline one. Also, the performance of Der1Der1 and Der1Der1HM were almost the same, showing that our cache optimization was not too significant for OpenMP. On the other hand, optimizations that reduce memory usage, such as Der1Der1LM, shows the best performance improvements for OpenMP. The optimized versions were up to 16.2% faster than the baseline version implemented in OpenACC running in the multicore. The most effective optimization was Der1Der1, with a difference of 3% of the performance compared with Der1Der1LM. It shows that using reduced memory usage and cache optimizations together imply better performance in OpenACC than using them alone.

Our optimizations affect GPU architectures differently than in multicore architectures. OpenACC-GPU benefited most from Der1DerHM optimization with a performance improvement of 24.9%. The performance of Der1Der1LM was 5.7% better than the baseline, showing that cache and reduce memory optimizations together are not as better as only cache optimizations. It happens due to the GPU threads and memory subsystems, which are organized in a way that coalesced accesses from all threads are more efficient than a few accesses from a group of threads. For the CUDA version, the best performance improvements are provided by the Der1Der1 version. Furthermore, results show that our cross-derivative optimizations are beneficial for both architectures.

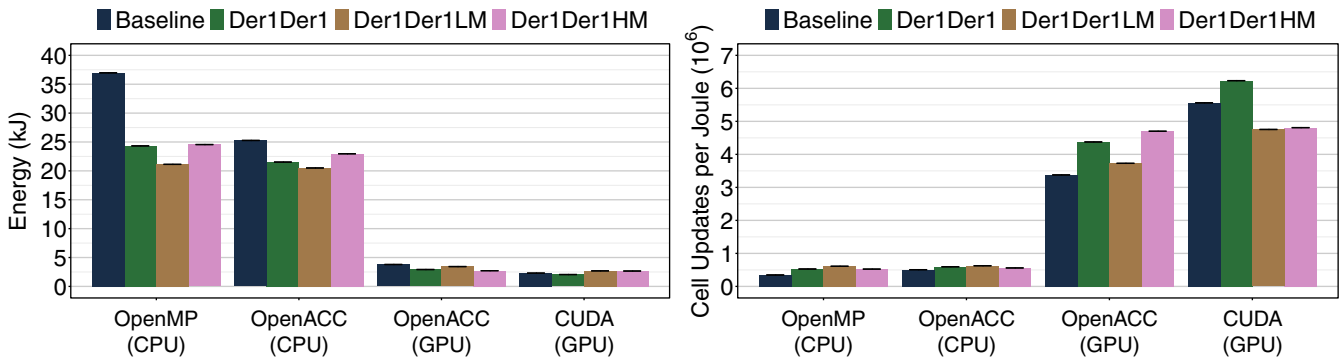
4.3 | Energy consumption and energy efficiency of different optimizations

In the second analysis, we focus on determining the energy consumption (measured in Joules) and the energy efficiency (measured in cells updated per Joule) for each of our implementations of the RTM method and analyzing the differences between the portable version (OpenACC) and the specific nonportable versions (OpenMP and CUDA). We used the Intel Skylake and NVIDIA Volta architectures in this analysis. Figure 3(A) shows the average energy consumption of the entire machine in Kilojoules on the Y-axis and the API in the X-axis. Most of the optimized versions present an energy consumption reduction compared with the Baseline version. The reductions were up to 75%, 23.2%, 39.3%, and 12.1% for OpenMP, OpenACC-CPU, OpenACC-GPU, and CUDA, respectively. Differences also happen when comparing multicore with GPU results. For instance, Der1Der1LM shows the higher reductions for the multicore architectures, while Der1Der1 and Der1Der1HM are better for GPUs.

The energy consumption of the portable OpenACC version is better than the OpenMP dedicated version in all scenarios evaluated. For instance, it consumes 25 kJ for the Baseline implementation against 37 kJ for OpenMP. Also, the optimizations applied to OpenACC reduce its consumption in up to 5 kJ. On the other hand, comparing with the CUDA version, the OpenACC consumes up to 1.4 kJ more energy. Moreover, we observe that, in general, GPUs have lower energy consumption than multicore architectures. Also, the difference in energy consumption between the GPU implementations are small, showing that OpenACC may be a better choice for researchers. When considering multicore implementations, we can note that the portable OpenACC implementations present lower energy consumption than the dedicated OpenMP ones.

Several authors explain energy efficiency as the efficient use of energy to obtain a specific result. For large-scale systems, such as data centers, energy efficiency has proven to be the key to reducing all types of costs related to capital, operating expenses and environmental impact.[57] In this work, we measure energy efficiency using the number of cell updates per Joule.

Furthermore, energy consumption is not enough to characterize the efficiency of different versions and architectures. For instance, an architecture that consumes 10 kJ with a performance of 20 MCUPS is less efficient than an architecture that consumes 15 kJ with a performance of 50 MCUPS. In this way, we also analyze which implementation has the best energy efficiency, measured by the ratio between energy and performance. Observing the general relationship between the baseline and the optimized versions, the optimized are up to 43% more efficient.



(A) Energy in Kilojoules

(B) Energy Efficiency in millions of CUPJ

FIGURE 3 Energy and energy efficiency of the proposed optimizations on different APIs

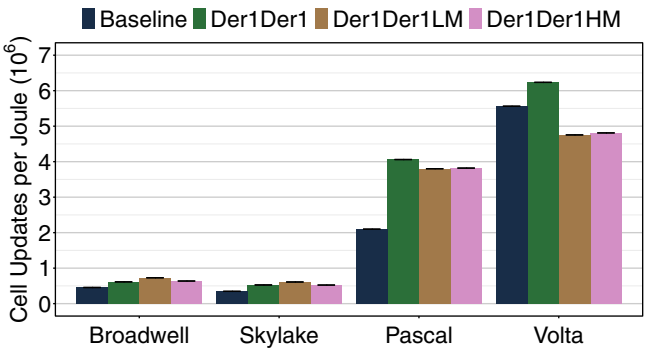


FIGURE 4 Energy efficiency of the OpenACC versions in different architectures in millions of CUPS

The energy efficiency of different APIs is also important for our analysis. Figure 3(B) differs only in the Y-axis, which is the CUPJ metric in millions. The best results for OpenMP and OpenACC-CPU are achieved using the Der1Der1LM optimization, with 0.61 and 0.63 MCUPJ, respectively. In CUDA and OpenACC-GPU, the best versions are Der1Der1HM for OpenACC-GPU and Der1Der1 for CUDA. It is the same behavior cited before in the performance results. The energy efficiency of both implementations is 4.7 and 6.2 MCUPS, respectively.

4.4 | Comparison between different architectures

We analyze the performance and energy efficiency of different multicore and GPU architectures running the portable OpenACC version. This analysis leads to determine which architecture performs better for our oil and gas application and the impact of each optimization in the performance and energy efficiency. Figure 4 depicts the CUPJ metric in millions on the Y-axis and the architecture used in the X-axis. The architectures used are Intel Broadwell, Intel Skylake, NVIDIA Pascal, and NVIDIA Volta. In this plot, each color represents an optimization: Baseline, Der1Der1, Der1DerLM, and Der1DerHM. The results show that, for GPUs, the Volta architecture has the best energy efficiency, while for multicore architectures, the Intel Broadwell is better.

Moreover, the impact of each optimization differs from architecture to architecture, even from the same chipmaker. For instance, the most energy-efficient optimization for Pascal and Volta was Der1Der1. This version achieved 4.1 and 6.2 MCUPJ, almost 35% of the difference in energy efficiency for GPUs that differ in one generation. This phenomenon happens due to the differences in the memory hierarchy of Pascal and Volta. In Volta, the register file was doubled compared with Pascal. They increased the L1 cache four times. Also, the number of CUDA threads changed from 3584 to 5120, increasing the computation throughput and the memory bandwidth. Due to these changes, the performance of Volta is higher than Pascal, and thus, the energy efficiency.

For multicore architectures, the Broadwell CPUs achieved the best energy efficiency with 0.7 MCUPJ, compared with 0.6 MCUPJ for the Skylake architecture. One of the reasons for this difference in performance is the number of cores of these architectures. Broadwell has 44 cores while Skylake has 32. Also, they changed the interconnect architecture from ring to mesh and the cache sizes.

4.5 | Discussion and guidelines

In our experiments, we analyzed performance, energy consumption and energy efficiency of different optimizations and architectures. We also investigated the feasibility of using a portable version of the RTM application. The results showed that the cost of writing a portable version is 5% for multicore CPUs and 20.9% for GPUs.

However, performance is not the only metric to be considered. Energy consumption is also concerned for the exascale supercomputer. However, energy consumption should not be considered alone because an application can consume more energy but run faster, achieving better energy efficiency. In this sense, we show that, when looking at energy efficiency, it is possible to note that GPUs are more efficient than multicore architectures even when the energy consumption is high. In this aspect, the use of GPU is more interesting than the use of multicore architectures for running oil and gas simulations.

Moreover, programming in OpenACC is easier than CUDA, as shown by several works such as Memeti et al.[58] OpenACC code is portable, meaning that a code written in OpenACC may execute in different architectures without any change. The point is that it exists a trade-off between performance, portability and energy efficiency that each user should consider. That is, how much lower performance we accept when using a portable code?

Finally, when comparing multicores and GPUs in terms of performance and energy efficiency, for oil and gas simulations, it is clear and known that GPU implementations are efficient. Moreover, when making a comparison considering the generality of the programming interfaces within the

same architecture, we note that, for the GPU, the less generic programming interface (CUDA) has higher energy efficiency, regardless of the input size. One of our hypotheses for that is the compiler's optimizations. The OpenACC compiler decides several parameters, such as the number of threads per block and blocks per grid, while for the CUDA implementation, all decisions are made by the developers. These parameters can also be tuned by developers using OpenACC clauses. However, it requires a more in-depth investigation. Also, for implementations considering the CPU architecture, the two implementations' efficiency is practically identical.

5 | CONCLUSION AND FUTURE WORKS

Today's multicore and GPU architectures introduce several challenges, such as the need to code parallel applications to properly make the most of them. In this article, we optimized an oil and gas application, reverse time migration (RTM), for the Intel Broadwell and Intel Skylake multicores, and NVIDIA Pascal, and Volta GPUs. We showed that the calculation of cross-derivatives required 76% of floating-point operations, and, because of this, we propose three optimizations to reduce this bottleneck. Developers can also apply these optimizations to other similar applications and other architectures.

In our experiments, we also investigate the application performance, energy consumption and energy efficiency, besides doing a study on the feasibility of using a portable version of the code. Our results showed that the best performance results were using CUDA on the NVIDIA Volta GPU. The difference in performance between the CUDA version and the OpenACC-GPU version, which can run on multiple architectures, was up to 20%. It shows a cost in using a portable version, but it may be interesting, depending on the difficulty of parallelizing the application. Afterward, we show that energy consumption is not the only determining metric, because an application can consume more energy while finishing quickly and thus having the best energy efficiency. In this regard, we analyze energy efficiency and show that CUDA and OpenACC versions have the best efficiency.

As future work, we plan to evaluate architectures with OpenCL programmed integrated FPGA cards. We are also interested in verifying the efficiency of using multiple GPUs on the same node and comparing our results with state-of-art solutions.

ACKNOWLEDGMENTS

This work has been partially supported by Petrobras (2016/00133-9, 2018/00263-5) and Green Cloud project (2016/2551-0000 488-9), from FAPERGS and CNPq Brazil, program PRONEX 12/2014. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). Some experiments in this work used the PCAD infrastructure, http://gppd-hpc.inf.ufrgs.br, at INF/UFRGS.

ORCID

Matheus S. Serpa https://orcid.org/0000-0001-5178-1036
Pablo J. Pavan https://orcid.org/0000-0002-7588-0503
Eduardo H. M. Cruz https://orcid.org/0000-0003-4484-3256
Jairo Panetta https://orcid.org/0000-0002-7798-3979
Philippe O. A. Navaux https://orcid.org/0000-0002-9957-5861

REFERENCES

1. Qutob H. Underbalanced drilling; remedy for formation damage, lost circulation, & other related conventional drilling problems. Paper presented at: Proceedings of the Abu Dhabi International Conference and Exhibition. Society of Petroleum Engineers. Abu Dhabi, UAE; 2004.
2. Lukawski MZ, Anderson BJ, Augustine C, et al. Cost analysis of oil, gas, and geothermal well drilling. *J Pet Sci Eng*. 2014;118:1-14.
3. Clapp RG, Fu H, Lindtjorn O. Selecting the right hardware for reverse time migration. *Lead Edge*. 2010;29(1):48-58.
4. Clapp RG. Seismic processing and the computer revolution(s). In: Schneider RV, ed. *Society of Exploration Geophysicists (SEG) Technical Program Expanded Abstracts 2015*. *Society of Exploration Geophysicists*. Tulsa: Society of Exploration Geophysicists (SEG); 2015:4832-4837.
5. Dongarra J, Meuer H, Strohmaier E. Top500 supercomputer; December 2019. https://www.top500.org/lists/2019/11/. Accesed December 10, 2019.
6. Witten IH, Frank E, Hall MA, Pal CJ. *Data Mining: Practical Machine Learning Tools and Techniques*. Burlington, MA: Morgan Kaufmann; 2016.
7. Diener M, Cruz EH, Pilla LL, Dupros F, Navaux PO. Characterizing communication and page usage of parallel applications for thread and data mapping. *Perform Eval*. 2015;88:18-36.
8. Serpa MS, Moreira FB, Navaux POA, et al. Memory performance and bottlenecks in multicore and GPU architectures. In: D'Agostino D, ed. *Proceedings of the 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. New York City: IEEE; 2019:233-236.
9. Subramaniam B, Saunders W, Scogland T, Feng WC. Trends in energy-efficient computing: a perspective from the Green500. In: Dongarra J, Matsouka S, eds. *Proceedings of the 2013 International Green Computing Conference*. New York City: IEEE; 2013:1-8.
10. Castro M, Francesquini E, Dupros F, Aochi H, Navaux POA, Mehaut JF. Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Comput*. 2016;54:108–120.
11. Souza MA, Penna PH, Queiroz MM, et al. CAP bench: a benchmark suite for performance and energy evaluation of low-power many-core processors. *Concurr Comput Pract Exper*. 2017;29(4):e3892.

12. Schepke C, Lima JV, Serpa MS. *Challenges on Porting Lattice-Boltzmann Method on Accelerators: NVIDIA Graphic Processing Units and Intel Xeon Phi*. Hershey, PA: IGI Global; 2018:30-53.
13. Freytag G, Serpa MS, Lima JVF, Rech P, Navaux POA. Non-uniform partitioning for collaborative execution on heterogeneous architectures. In: Caceres E, Santos R, eds. *Proceedings of the 2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. New York City: IEEE; 2019:128-135.
14. Sabne A, Sakdhnagool P, Lee S, Vetter JS. Evaluating performance portability of OpenACC. In: Brodman J, Tu P, eds. *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. New York, NY: Springer; 2014:51-66.
15. Fletcher RP, Du X, Fowler PJ. Reverse time migration in tilted transversely isotropic (TTI) media. *Geophysics*. 2009;74(6):WCA179-WCA187.
16. OpenACC Working Group The OpenACC application programming interface; 2019. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf. Accesed September 23, 2020.
17. OpenMP Architecture Review Board OpenMP application programming interface; 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Accesed October 01, 2020.
18. NVIDIA Developer Zone CUDA Toolkit Documentation v11.1.0; 2020. https://docs.nvidia.com/cuda/. Accesed September 29, 2020.
19. Nalamalpu A, Kurd N, Deval A, et al. Broadwell: a family of IA 14nm processors. In: Inaba S, Motomura M, eds. *Proceedings of the 2015 Symposium on VLSI Circuits (VLSI Circuits)*. New York City: IEEE; 2015:C314-C315.
20. Doweck J, Kao WF, AKy L, et al. Inside 6th-generation intel core: new microarchitecture code-named Skylake. *IEEE Micro*. 2017;37(2):52-62.
21. Foley D, Danskin J. Ultra-performance pascal GPU and NVLink interconnect. *IEEE Micro*. 2017;37(2):7-17.
22. Choquette J, Giroux O, Foley D. Volta: performance and programmability. *IEEE Micro*. 2018;38(2):42-52.
23. Serpa MS, Pavan PJ, Panetta J, Azambuja A, Carissimi A, Navaux POA. Portabilidade e Eficiência do Método Fletcher de Aplicações Sísmicas em Arquiteturas Multicore e GPU. In: Camargo RY, Martins WS, eds. *Proceedings of the 2019 Symposium on High Performance Computing Systems (WSCAD)*. Porto Alegre, Brazil: SBC; 2019:169-180.
24. Abdelkhalek R, Calandra H, Coulaud O, Roman J, Latu G. Fast seismic modeling and reverse time migration on a GPU cluster. Paper presented at: Proceedings of the 2009 International Conference on High Performance Computing & SimulationIEEE. Leipzig, Germany; 2009:36-43.
25. Cabezas J, Araya-Polo M, Gelado I, Navarro N, Morancho E, Cela JM. High-performance reverse time migration on GPU cluster. Paper presented at: Proceedings of the 2009 International Conference on High Performance Computing & Simulation. Leipzig, Germany; 2009:77-86.
26. Moussa N. Seismic imaging using gpgpu accelerated reverse time migration. *CS 315A Parallel Computer Architecture and Programming*. Santa Clara, CA: Stanford Exploration Project; 2009.
27. Araya-Polo M, Cabezas J, Hanzich M, et al. Assessing accelerator-based HPC reverse time migration. *IEEE Trans Parall Distrib Syst*. 2010;22(1):147-162.
28. LIU HW, Li B, Liu H, TONG XL, Liu Q. The algorithm of high order finite difference pre-stack reverse time migration and GPU implementation. *Chin J Geophys*. 2010;53(4):600-610.
29. Fu H, Clapp RG. Eliminating the memory bottleneck: an FPGA-based solution for 3d reverse time migration. Paper presented at: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays ACM/SIGDA; 2011:65-74.
30. Lu L, Magerlein K. Multi-level parallel computing of reverse time migration for seismic imaging on blue Gene/Q. Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming. Shenzhen, China; 2013;48(8):291-292.
31. Clapp RG. Seismic processing and the computer revolution (s). *Society of Exploration Geophysicists*. Tulsa: Society of Exploration Geophysicists (SEG); 2015:4832-4837.
32. Nunes-do-Rosario DA, Xavier-de-Souza S, Maciel RC, Costa JC. Parallel scalability of a fine-grain prestack reverse time migration algorithm. *IEEE Geosci Remote Sens Lett*. 2015;12(12):2433-2437.
33. Carrijo Nasciutti T, Panetta J, Pais Lopes P. Evaluating optimizations that reduce global memory accesses of stencil computations in GPGPUs. *Concurr Comput Pract Exper*. 2018;31(18):e4929.
34. Serpa MS, Cruz EH, Diener M, et al. Optimization strategies for geophysics models on manycore systems. *Int J High Perform Comput Appl*. 2019;33(3):473-486.
35. Nesi LL, Serpa MS, Schnorr LM, Navaux POA. Task-based parallel strategies for CFD application in heterogeneous CPU/GPU resources. *Concurr Comput Pract Exper*. 2020.
36. Qawasmeh A, Hugues MR, Calandra H, Chapman BM. Performance portability in reverse time migration and seismic modelling via OpenACC. *Int J High Perform Comput Appl*. 2017;31(5):422-440.
37. Pavan PJ, Serpa MS, Carreño ED, et al. Improving performance and energy efficiency of geophysics applications on GPU architectures. In: Hernandez CJB, Barrera HC, eds. *Proceedings of the Latin American High Performance Computing Conference*. New York, NY: Springer; 2018:112-122.
38. Bittencourt JC, Oliveira WL, Nascimento A, et al. Performance and energy efficiency analysis of reverse time migration on a FPGA platform. In: Bakos J, Blott M, Cappello F, Hoefler T, Plessl C, eds. *Proceedings of the 2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. New York City: IEEE; 2019:50-58.
39. Calore E, Gabbana A, Schifano SF, Tripiccione R. Energy-efficiency tuning of a lattice Boltzmann simulation using MERIC. In: Wyrzykowski R, Deelman E, Dongarra J, Karczewski K, eds. *International Conference on Parallel Processing and Applied Mathematics*. New York, NY: Springer; 2019:169-180.
40. Daniel DF, Panetta J. On applying performance portability metrics. In: Neely R, Doerfler D, Pennycook J, Newburn CJ, eds. *Proceedings of the 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. New York City: Institute of Electrical and Electronics Engineers (IEEE); 2019:50-59.
41. Pennycook SJ, Sewall JD, Lee VW. Implications of a metric for performance portability. *Futur Gener Comput Syst*. 2019;92:947-958.
42. Andreolli C, Thierry P, Borges L, Skinner G, Yount C. Characterization and optimization methodology applied to stencil computations. In: Reinders J, Jeffers J, eds. *High Performance Parallelism Pearls*. Boston, MA: Morgan Kaufmann; 2015:377-396.
43. Zhou HW, Hu H, Zou Z, Wo Y, Youn O. Reverse time migration: a prospect of seismic imaging methodology. *Earth Sci Rev*. 2018;179:207-227.
44. Guo J, Rubino JG, Glubokovskikh S, Gurevich B. Effects of fracture intersections on seismic dispersion: theoretical predictions versus numerical simulations. *Geophys Prospect*. 2017;65(5):1264-1276.
45. Zhu T. Numerical simulation of seismic wave propagation in viscoelastic-anisotropic media using frequency-independent Q wave equation. *Geophysics*. 2017;82(4):WA1-WA10.

46. Confal JM, Faccenda M, Eken T, Taymaz T. Numerical simulation of 3-D mantle flow evolution in subduction zone environments in relation to seismic anisotropy beneath the eastern Mediterranean region. *Earth Planet Sci Lett*. 2018;497:50-61.
47. Guo J, Germán Rubino J, Barbosa ND, Glubokovskikh S, Gurevich B. Seismic dispersion and attenuation in saturated porous rocks with aligned fractures of finite thickness: theory and numerical simulations–Part 2: Frequency-dependent anisotropy. *Geophysics*. 2018;83(1):WA63-WA71.
48. Thomsen L. Weak elastic anisotropy. *Geophysics*. 1986;51:1954-1966. https://doi.org/10.1190/1.1442051.
49. Fowler PJ, Du X, Fletcher RP. Coupled equations for reverse time migration in transversely isotropic media. *Geophysics*. 2010;75(1):S11-S22.
50. Terpstra D, Jagode H, You H, Dongarra J. Collecting performance data with PAPI-C. *Tools for High Performance Computing*. New York, NY: Springer; 2010:157-173.
51. Nvidia Developer zone - CUDA toolkit documentation; 2016. http://docs.nvidia.com/cuda/profiler-users-guide/. Accesed August 23, 2019.
52. Ott RL, Longnecker MT. *An Introduction to Statistical Methods and Data Analysis*. Nashville, TN: Nelson Education; 2015.
53. Tullsen DM, Eggers SJ, Levy HM. Simultaneous multithreading: maximizing on-chip parallelism. In: Patterson DA, ed. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. New York, NY: ACM; 1995.
54. Broquedis F, Clet-Ortega J, Moreaud S, et al. hwloc: a generic framework for managing hardware affinities in HPC applications. In: Danelutto M, Gross T, Bourgeois J, eds. *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. Pisa, Italy: IEEE; 2010:180-186.
55. Bolze R, Cappello F, Caron E, et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *Int J High Perform Comput Appl*. 2006;20(4):481-494.
56. Slaight T. Platform management IPMI controllers, sensors, and tools. *Intel Developer Forum*. 2002. https://www.intel.com/content/dam/www/public/us/en/documents/productbriefs/controllers-sensors-and-tools.pdf.
57. Varrette S, Pinel F, Kieffer E, Danoy G, Bouvry P. Automatic software tuning of parallel programs for energy-aware executions. In: Wyrzykowski R, Deelman E, eds. *Proceedings of the International Conference on Parallel Processing and Applied Mathematics*. New York, NY: Springer; 2019:144-155.
58. Memeti S, Li L, Pllana S, Kołodziej J, Kessler C. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In: Vaidya N, Fraigniaud P, Newport C, et al., eds. *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. New York, NY: ACM; 2017:1-6.

How to cite this article: Serpa MS, Pavan PJ, Cruz EHM, et al. Energy efficiency and portability of oil and gas simulations on multicore and graphics processing unit architectures. *Concurrency Computat Pract Exper*. 2021;e6212. https://doi.org/10.1002/cpe.6212