ACCELERATING SEMBLANCE COMPUTATIONS ON HETEROGENEOUS DEVICES USING OPENCL

E. Borin, H. Cardoso da Silva, J. H. Faccipieri Jr., and M. Tygel

email: *edson@ic.unicamp.br, hercules.cardoso.silva1@gmail.com, jorge.faccipieri@gmail.com and tygel@ime.unicamp.br*

keywords: *Semblance, CRS, CMP, GPU, OpenCL, HPC*

ABSTRACT

The core of several seismic processing methods, such as the CRS and the CMP methods, is the computation of the traveltime and semblance functions. In this work we investigate the use of OpenCL to accelerate these computations on multicore CPUs, GPUs, and other hardware accelerators. Our experiments indicate that the OpenCL code is highly portable among different computing devices and the performance results suggests that GPUs are promising computing devices to accelerate the seismic processing methods that rely on high volumes of semblance computations.

INTRODUCTION

Several seismic processing techniques demand high amounts of data transfer, depending on the size of data to be considered, and also intensive computational power, depending on the operation complexity of the processes involved. In particular, imaging methods based on multiparametic traveltime stacking, such as the Common-Reflection-Surface (CRS) method, suffer from both difficulties. In fact, depending on three parameters in 2D case and eight in 3D, the computational cost associated with the CRS method renders its application unfeasible on large-scale seismic datasets routinely acquired by the oil industry.

The estimation of the CRS parameters lies on computation of traveltime surfaces and associated semblance functions, which represent almost 100% of the computing time. Since these methods require a large amount of computation when processing real data, they are typically coded to be executed in parallel on clusters with multiple machines.

Recent trends indicate that future computing systems will be composed by heterogeneous computing devices, including multicore Central Processing Units (CPUs), Graphical Processing Units (GPUs) and other hardware accelerators, such as the Intel Xeon Phi and Field Programmable Gate Arrays (FPGAs). However, in order to use the computing power available on these heterogeneous devices, existing programs will need to be adapted or, in some case, be completely rewritten using new programming frameworks.

Ni and Yang (2012) used CUDA to accelerate the so-called 3D Output Imaging Scheme (CRS-OIS) method on GPUs and report that the code running on the GPU can be 10 to 220 times faster than the CPU when processing a synthetic data. When processing a real data, the GPU (Model c1060) is roughly 35 times faster than a CPU processing with only one of the cores. Marchetti et al. (2011) accelerated the search for the eight parameters on the CRS method using OpenCL to run the semblance and traveltime operations on a GPU. The authors reported that the GPU (Radeon HD 5870) is roughly 60 times faster than the CPU processing when processing a 3GB seismic data. Marchetti et al. (2010) used the Maxeler MaxCompiler tool to accelerate the CRS method using FPGAs. The authors reported that their solution is 200 to 230 times faster than the CPU when processing a seismic data from a land survey.

In this work, we investigate how we can accelerate the computation of semblance operations using OpenCL, a parallel application program interface (API) designed to enable the same code to be executed on different parallel hardware accelerators, including GPUs from different vendors, multicore CPUs, and FPGAs. In particular, we implemented an OpenCL program to compute the semblance operations to estimate the Normal Moveout (NMO) velocity (see, e.g. Taner and Kohler, 1969) on the Common Midpoint (CMP) method (see, e.g. Neidell and Taner, 1971) and found that GPUs can execute it 6.4 to 43.3 times faster than CPUs. Since the core of the computation is semblance and traveltime evaluations, we expect to see similar performance benefits on other methods that rely on such computations, with the Common-reflection-surface (CRS) method as a prominent example.

The text is organized as follows: we first provide a brief overview of GPUs, hardware accelerators and OpenCL. Then, we discuss the implementation of the CMP method using both OpenMP and OpenCL. Finally, we present some experimental results and conclusions.

GPUS, HARDWARE ACCELERATORS AND OPENCL

Graphic Processing Units, or GPUs, are devices that were originally dedicated to accelerate graphics processing on computer systems. However, at the end of the last decade, new programming models and hardware advances allowed programmers to harness the computing power available on GPUs to perform general-purpose computation. A simplified hardware mechanism to control the execution flow allied with tens to hundreds of parallel computing elements render GPUs highly energy efficient and highly parallel computing devices. As a result, these devices became popular with the high-performance computing community and is currently present on the fastest supercomputers in the world (Top 500 supercomputer sites, 2015).

The use of GPUs for high-performance computing has influenced the microprocessor hardware industry and several new solutions, such as the Accelerated Processing Units, by AMD, and the Xeon PHI coprocessor, by Intel, were introduced. These new products and the supercomputer hardware usage trends (Top 500 supercomputer sites, 2015) indicate that these technologies are becoming a standard. Figure 1 illustrates the evolution of supercomputers over the years, including the computing devices and their respective programming technologies. Notice that during the nineties and the first half of the 2000's supercomputers were composed of multiple single-core nodes (several machines) and programmed mostly with Message Passing Interface (MPI). After the introduction of multicores, in 2005, the supercomputers started to employ this kind of processors in their nodes and new programming technologies were introduced to explore the parallelism available inside the node. Later, in 2009, when the supercomputers started to use hardware accelerators, new programming technologies, such as CUDA, OpenCL and OpenACC, were introduced. These new programming technologies were introduced because compilers were not capable of compiling legacy C/C++ programs, generated for traditional CPUs, into high performing GPU code. Despite recent progresses on code generation technologies, it is still very unlikely that compilers will be able to perform this task automatically. As a result, programs need to be modified to harness the computational power available on these accelerators.

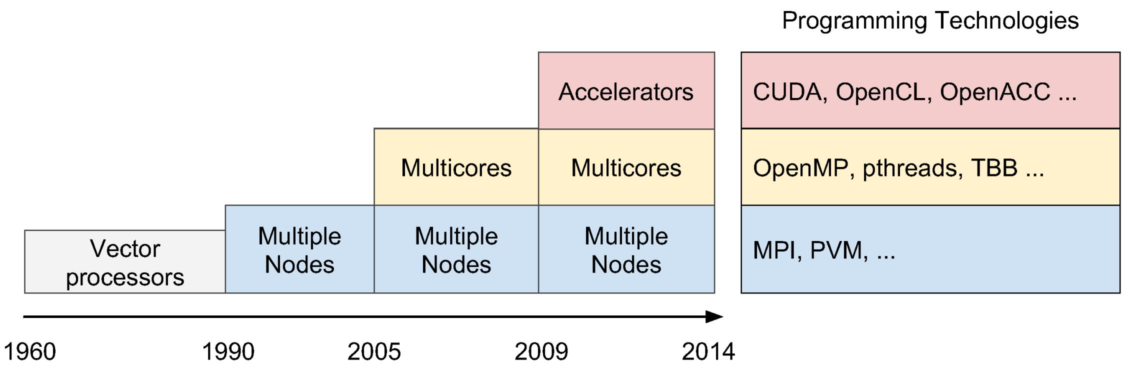

Figure 1: Evolution of supercomputers over the years, including their computing devices and respective programming technologies.

OpenCL

As soon as researchers demonstrated the benefits of using GPUs for general purpose computing, NVIDIA, one of the major manufacturers of GPUs, introduced a programming framework to enable users to execute general-purpose programs on their GPUs. This framework, called CUDA, was quickly adopted by the community and became a very popular framework for programming GPUs. However, since CUDA is proprietary, programs coded with CUDA are fated to be executed only on NVIDIA GPUs. After CUDA was introduced, the Khronos group (The Khronos Group – Connecting Software to Silicon, 2015) started developing OpenCL, a new parallel programming standard for heterogeneous computing systems that offers a common application programming interface (API) to enable programmers to implement a single program that runs on different types of computing devices, such as multicore CPUs, GPUs, FPGAs, and other hardware accelerators. The Khronos group is a non-profit organization dedicated to creating open standard APIs and promoted by several members of the microprocessor industry, including Intel, Apple, NVIDIA, AMD, ARM, Qualcomm, and Samsung. As a consequence, most of these companies implement OpenCL drivers so that users can run OpenCL code on their hardware accelerators. This makes OpenCL code highly portable across different devices, including devices from different vendors.

OpenCL programs are divided in two parts, one that runs on the host, or the system CPU, and another that runs on the OpenCL device, which can be a GPU, an FPGA or even a multicore CPU. The former is typically a C or C++ program and is compiled to run on traditional CPUs, such as Intel x86 processors. The later is a set of OpenCL language (C-like) functions, called OpenCL kernels, that are compiled to run on the hardware accelerator (Kaeli et al., 2015). OpenCL kernels are typically compiled by the OpenCL driver at runtime, as a result of an API that is called by the host program. This mechanism allows the same code to run on systems with different hardware accelerators (e.g. a cluster of machines in which each machine may have a different hardware accelerator).

The host program is responsible for invoking OpenCL APIs to allocate and transfer data to the OpenCL device, dispatching the execution of OpenCL kernels and retrieving the results from the device when the computation is done. The OpenCL device may execute several instances of the same OpenCL kernel in parallel, each instance on a different processing element. Each instance is known as a work item and, despite executing the same code, each work item has a different ID, which is typically used inside the kernel to ensure that each instance of the kernel works on a different part of the data.

CMP ON OPENCL

We now discuss how we accelerated the velocity analysis in the CMP method using OpenCL to parallelize the execution of semblance operations. Since the computational kernel of the CMP and CRS methods are very similar, the approach discussed here can be easily extended to accelerate CRS.

The CMP method searches for the Normal Moveout (NMO) velocity that yields the best semblance value for each time sample on each CMP gather. The search involves computing the semblance function for several different trial NMO velocities, which are defined by the search space. Listing 1 provides a pseudo-code that illustrates the CMP method. The `getmax_V` procedure computes the semblance for each value of V (the NMO velocity) in the search space and returns the one that generated the maximum semblance value. The `cmp` procedure, in turn, invokes the `getmax_V` procedure for every time sample (t0) on every gather to search for the velocity that provides the best semblance for the given time sample. Once the best NMO velocity is found, the traces are stacked, producing a single trace for each CMP gather.

As we can see in Listing 1, this computation is embarrassingly parallel. In fact, each gather, each t0 and even each semblance can be computed in parallel to accelerate the computation. In order to accelerate this kernel in multicore CPUs, we modified our original sequential C code to leverage OpenMP to process each CMP gather in parallel. We will refer to this program as `CMP-OpenMP`. Even though this program can only be executed on CPUs, it will serve as a basis to measure the quality of the OpenCL based code when running on multicore CPUs.

As discussed previously, OpenCL programs typically transfer the input data to the computing device(s), dispatch the execution of the computing kernel on the device(s) and retrieve the results once the computation is done. In case the input data is larger than the device memory, this computation may be performed in multiple steps, each one on a subset of the data. The computing kernel is executed in parallel by multiple

Listing 1 Pseudo-code for the CMP method

```
procedure: cmp ()
    for each CMP gather g; do
        for each t0 in g; do
            C = getmaxV (g, t0);
            stack (g, t0, V);
        end for
    end for
end procedure

procedure: getmaxV (g, t0)
    bestV = 0;
    maxSemb = 0;
    for each V in the search space; do
        semb = semblance (V, t0, g);
        if (semb > maxSemb); then
            maxSemb = semp;
            bestV = V;
        end if
    end for
    return bestV;
end procedure
```

work items, each one working on different slices of the data that is stored at the computing device memory. In this sense, we could transfer gathers to the computing device memory and have each work item processing a different t0, a different gather or even a different semblance. After reasoning about these options, we make the following observations:

- *One gather per work item*: assigning one gather per work item may require the computing device to store multiple (number of work items) gathers at the same time. However, if work items are executed in lock step, then the memory access pattern is very regular and the input data structure can be shapped to enable coalesced memory access, which is very important to improve memory bandwidth on several GPUs (Fauzia et al., 2015; Ryoo et al., 2008).
- *One t0 per work item*: assigning a different t0, of the same gather, per work item would allow the device to store only one (or a few) gather at the same time. Also, since several work items may access the same or nearby input data items, there may be an increased locality on spatial and temporal data access that could improve performance on computing devices that rely on cache memories.
- *One semblance per work item*: assigning a different semblance computation per work item would also allow the device to store only one (or a few) gather at a time and even increase performance on computing devices that rely on cache memories, however, after computing the semblance, the maximum semblance value would have to be computed, which would require different work items to synchronize. Since synchronization methods tend to impose significant overheads on GPUs, this options is likely to be worse than the previous ones.

Analyzing the typicall memory size of GPUs ($1 - 6$ GBytes), and the typicall CMP gather sizes ($\sim$500 KB), we concluded that several gathers can be stored at the GPU memory without a problem. Also, considering that several GPUs require a regular memory access pattern to maximize memory bandwidth, we decided to implement the first approach: one gather per work item.

Shaping the input data for coalesced memory accesses

The host program reads the seismic data from the input file and groups them into CMP gathers to transfer to the GPU memory. The gathers are transfered in sets of size NG. Since each gather have multiple traces and each trace has multiple samples, the data has three dimensions. In this sense, the host program builds cubes of data that are transfered to the OpenCL device to be computed. Figure 2 illustrates a cube of data with 6 gathers, each one containing 9 seismic traces with 9 samples each.

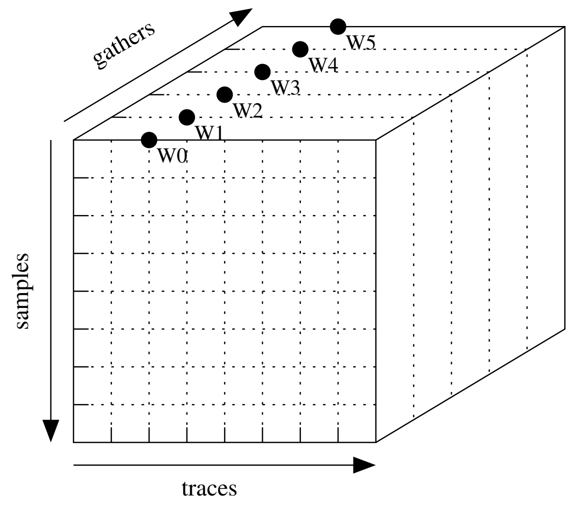

Figure 2: Cube of data where each axis represents the gathers, traces and time samples.

Work items may execute in lock step, which would cause different processing elements to access the memory at the same time. If these accesses are performed on consecutive addresses, then the GPU may issue a single, wide, memory access, which can improve the memory access bandwidth.

Since we assign a different gather for each work item, in order to promote the coalesced memory access, we interleaved the gathers data on the device memory. In this way, when accessing a given piece of data, say the first sample of the second trace of the gather, all the work items access consecutive elements on the GPU memory, each one belonging to a different gather. As a result, the data in our cube is organized so that the i^{th} sample from the j^{th} trace of all gathers are placed continuously on memory. The black circles in Figure 2 illustrate the different work items (W0, W1, ... W5) accessing the first sample of the third trace on different gathers in parallel.

The semblance computation

The semblance (e.g., Neidell and Taner, 1971) computation is performed over a traveltime curve that intersects seismic traces. This curve is defined by one parameter that corresponds to the NMO velocity in the CMP method. Since the traces are represented by discrete samples, the the point where the traveltime curve intersects a trace may not lie on an actual sample of the dataset. As a consequence, interpolation using the nearby samples is performed to estimate the seismic amplitude at that point. Figure 3 illustrates the traveltime curve on a CMP gather and an intersection point that lies between the seismic trace discrete samples, represented as black circles.

Listing 2 provides a pseudo-code that illustrates the semblance computation process. For a given gather g, a time sample t0, and a velocity V, the code computes for each trace (tr) the time (t) in which the curve intersects the trace and the amplitude (a) of the seismic trace in this time. We use a linear interpolation to compute the amplitude. The amplitude and its square are accumulated on the num and den variables in order to compute the semblance value.

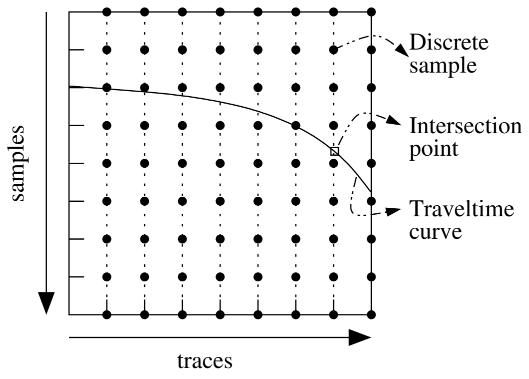

Figure 3: Traveltime curve and seismic trace intersections on a CMP gather.

Listing 2 Pseudo-code for the semblance computation

```
procedure: semblance (V, t0, g)
    num = 0;
    den = 0;
    for each trace tr in g; do
        t = time (t0, V, h(tr));
        a = interpol_linear (t, tr);
        num = num + a;
        den = den + (a×a);
    end for
    return (num×num) / den;
end procedure
```

In order to improve the signal-to-noise ratio, the semblance may be extended to also evaluate multiple amplitudes per trace. This approach involves the inspection of the amplitudes of neighboring time samples in each intersection point, which is defined by a window W. Listing 3 provides a pseudo-code for the semblance computation extended to analyze neighboring time samples. In this case, the program inspects W amplitudes per seismic trace when computing the semblance.

The amount of data read and computations performed by the semblance operation is highly dependent on F and W, where F is the gather fold and W is the semblance window size. For example, computing the semblance operation on a gather that contains 20 traces may be twice as expensive than computing the same operation on a gather with only 10 traces. In fact, we conjecture that the performance is highly dependent on the number of interpolations computed. In this sense, in order to factor these differences when comparing the performance of semblance operations on different datasets we propose using the following metric: $SemblanceTraces/s$. The $SemblanceTraces/s$ metric is computed as:

$$SemblanceTrace/s = \sum_{i=1}^{N} interpolations(S_i)/time \tag{1}$$

where:

- N is the total number of semblance operations computed;
- S_i is the i^{th} semblance operation being computed;

Listing 3 Pseudo-code for the extended semblance computation

```
procedure: semblance (V, t0, g)

    num[0..W] = 0;
    den[0..W] = 0;
    for each trace tr in g; do
        t = time (t0, V, h(tr));
        for i in 0..W-1; do
            a = interpol_linear (t + (i-W/2), tr);
            num[i] = num[i] + a;
            den[i] = den[i] + (a×a);
        end for
    end for
    NUM = 0;
    DEN = 0;
    for i in 0..W-1; do
        NUM = NUM + (num[i]×num[i]);
        DEN = DEN + den[i];
    end for
    return (NUM×NUM) / DEN;

end procedure
```

- $interpolations(S_i)$ is the number of interpolations performed by the i^{th} semblance operation. This is a function of F and W;
- $time$ is the execution time it took to compute the N semblance operations.

We expect this metric to provide similar values on the same system when processing different datasets, which is very helpfull to estimate the time it may take to process a different dataset. In fact, as we show in our experimental results, this metric provides similar values when processing different datasets.

EXPERIMENTAL RESULTS

We compare the performance of two different programs:

- `CMP-OpenCL`: CMP method parallelized with OpenCL – This program can be executed both on multicore CPUs and hardware accelerators, such as GPUs.
- `CMP-OpenMP`: CMP method parallelized with OpenMP – This program can only be executed on CPUs.

Two distinct seismic datasets were used in our experiments: one synthetic and the other one from a real data acquisition. The synthetic data has 428 gathers, most of them containing 15 seismic traces. Each trace contains 2502 samples and the search space for the NMO velocity contains 101 values. The semblance computations are performed on a window of size 3 and the total amount of $SemblanceTraces$ computed is 4 548 636 000. The real dataset is that of a single seismic line of the Jequitinhonha basin, in Brazil, and contains 201 gathers and an average fold of 29.5. Each trace contains 1701 samples and the search space for the NMO velocity contains also 101 values. The semblance computations are also performed on a window of size 3 and the total amount of semblance $SemblanceTraces$ computed is 6 261 208 290. Table 1 summarizes the properties of these seismic datasets.

Input name	# of CMPs	Search space sz.	Average fold	Samples per trace	Semblance window sz.	# semblance × trace
Simple-synthetic	428	101	14.01	2502	3	4 548 636 000
Jequitinhonha	201	101	29.5	1701	3	6 261 208 290

Table 1: Input datasets used in our experiments.

The performance experiments were conducted on four computing systems, containing six distinct OpenCL computing devices: two distinct multicore CPUs, three GPUs and a Intel Xeon PHI. Table 2 lists the processor model, memory characteristics, the operating system, the compiler and runtime systems used in each one of the computing system. It also lists the hardware accelerators that are installed on each one of these systems. When executing our OpenCL program, the host part is always executed on the system CPU. The OpenCL kernel may be executed on the hardware accelerator or on the host multicore CPU.

		System 1	System 2	System 3	System 4
Host	Processor(s)	2 x Intel Xeon E5-2670	1 x Intel Xeon E5-2630 v2		
	Memory	DDR3 64 GB	DDR3 32 GB		
	OS	Red Hat 4.4.7-16	Ubuntu 14.04 - LTS 64 bits		
	Compiler	gcc 4.4.7	gcc 4.8.4	nvcc V7.0.27	
	Runtime Drivers	Xeon Phi Driver 3.1.2-1	AMD OpenCL 2.0 Driver (14.41)	OpenCL 1.1 CUDA 7.0.28 Driver(346.46)	
Accelerator	Model	Xeon Phi 3120	Radeon R9 290x	GTX 770	GTX Titan
	Mem. Band.	240 GB/s	320 GB/s	224.3 GB/s	288.4 GB/s
	Cores	57 cores	2816 cores	1536 cores	2688 cores
	Frequency	1.1 GHz	1.04 GHz	1.05 GHz	0.84 GHz

Table 2: Computing systems used in our experiments.

Performance of `CMP-OpenMP` vs `CMP-OpenCL`

One of the factors that may affect a performance of an OpenCL program on multicore CPUs is the quality of the code generated by the runtime compiler. Hence, in order to establish a fair baseline for our comparisons, we check whether our `CMP-OpenCL` program is performing well on the multicore CPUs by comparing its performance with the performance of the `CMP-OpenMP` program, an OpenMP based version that was tuned on the host processor of the dual processor node.

Figure 4 shows the performance results of the `CMP-OpenMP` and the `CMP-OpenCL` programs when processing the Simple-synthetic and the Jequitinhonha datasets on two different computing systems: a single processor node and a dual processor node. The results indicate that the `CMP-OpenCL` program is 2.1 to 2.8 times faster than the `CMP-OpenMP` program.

Performance of `CMP-OpenCL` across devices

In this experiment we compare the performance of the `CMP-OpenCL` program across different OpenCL computing devices. Since we used OpenCL, there was no modification required to run the code on these devices. Figure 5 shows the performance of the program on six different computing devices: two multicore CPUs, three GPUs and one Xeon PHI.

Notice that the Radeon R9 290x GPU executes roughly 21 billion $SemblanceTraces$ per second and is 6.4 to 15.6 times faster than the CPU devices when running the `CMP-OpenCL` program. If we compare its performance with the one achieved by the `CMP-OpenMP` program on the CPUs the performance gains are even higher, between 14.3 to 43.3 times.

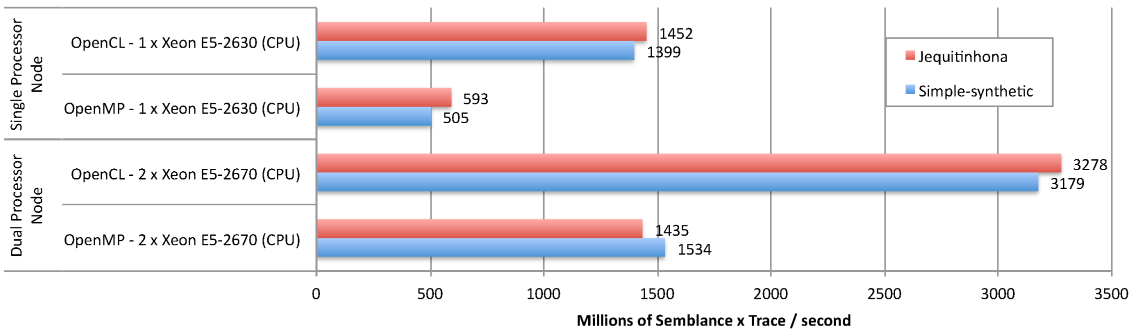

Figure 4: Performance of semblance compuations using OpenCL and OpenMP on multicore CPUs.

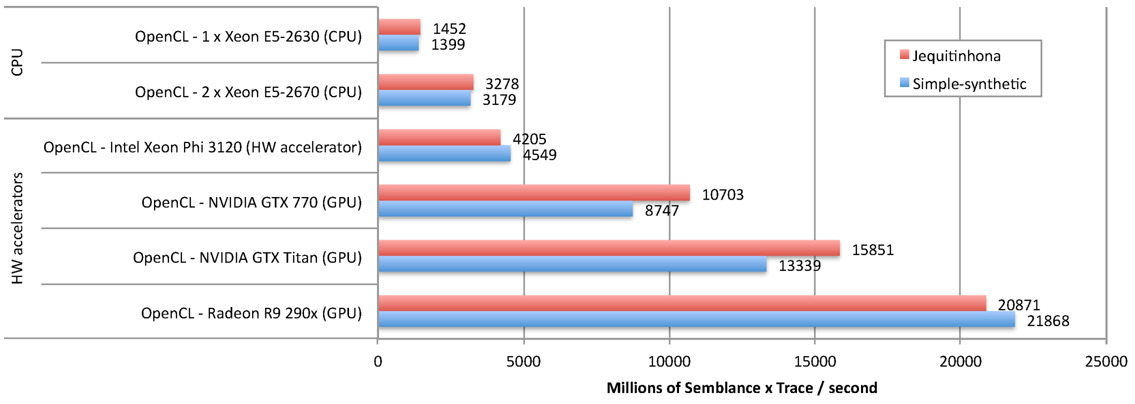

Figure 5: Performance of semblance computation using OpenCL on different CPUs and hardware accelerators.

Performance reporting discussion

Ni and Yang (2012) reported gains ranging from 10 to 220 times (35 times for real data) when accelerating the 3D-CRS-OIS method on a GPU. They compared the performance of the CUDA code running on a GPU against the performance of a sequential code running in only one of the cores of an 8-core CPU. Marchetti et al. (2011) reported that their GPU code is 60 times faster than their CPU code when executing the CRS method on a 3GB seismic data, however, it is not clear whether all the cores or just a single core of the CPU was used. Moreover, the CPU model was not reported. Marchetti et al. (2010) also reported very expressive performance gains (200-230 times) when using FPGAs to accelerate the semblance and traveltime computations, however, it is also not clear whether all the cores of the CPU were used measuring the baseline.

Clearly, the performance gains can be highly affected by the performance of the baseline system. For example, if Ni and Yang (2012) had optimized their CPU code to use the 8 CPU cores, the performance gain using the GPU could have dropped from 35 to 4.4 times when processing the real data. The same is true for Marchetti et al. (2011). In this sense, in order to allow an easier comparison, we propose the use of $SemblanceTraces/s$ metric when accelerating seismic processing methods that rely mostly on semblance computations. This metric will allow us to perform a direct comparison between acceleration techniques and will allow users to estimate how much it would take to process their own methods if they use the proposed acceleration technique.

CONCLUSIONS

In this work we investigated the potentials of using OpenCL to accelerate the computation of semblance operations on heterogeneous devices, including two multicore CPUs, three different GPUs and a Xeon PHI.

Our experiments indicate that the OpenCL program can be executed on these different devices without changes and the performance results indicate that GPUs are promising computing devices to accelerate seismic processing methods that rely on semblance computations.

Future work includes accelerating the CRS method using OpenCL, profiling and applying other optimizations to the OpenCL programs, such as using local storage (Ryoo et al., 2008), and experimenting with different OpenCL compatible hardware accelerators, such as FPGAs (Singh et al., 2013; Putnam et al., 2014).

ACKNOWLEDGMENTS

This work was kindly supported by the National Council for Scientific and Technological Development (CNPq-Brazil), the Coordination for the Improvement of Higher Education Personnel (CAPES), the National Institute of Science and Technology of Petroleum Geophysics (ICTP-GP-Brazil), the Center for Computational Engineering and Sciences (Fapesp/Cepid # 2013/08293-7-Brazil) and the Brazilian Oil Company - Petrobras (Cooperation term # 0050.0066919.11.9). We also thank Intel and AMD for providing the Intel Xeon PHI and GPUs. The authors finally acknowledge support of the sponsors of the Wave Inversion Technology (WIT) Consortium.

REFERENCES

Fauzia, N., Pouchet, L.-N., and Sadayappan, P. (2015). Characterizing and enhancing global memory data coalescing on gpus. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 12–22, Washington, DC, USA. IEEE Computer Society.

Kaeli, D. R., Mistry, P., Schaa, D., and Zhang, D. P. (2015). *Heterogeneous Computing with OpenCL 2.0*. Morgan Kaufmann, 3rd edition edition.

Marchetti, P., Oriato, D., Pell, O., Cristini, A., and Theis, D. (2010). Fast 3d zo crs stack–an fpga implementation of an optimization based on the simultaneous estimate of eight parameters. In *72nd EAGE Conference and Exhibition incorporating SPE EUROPEC 2010*.

Marchetti, P., Prandi, A., Stefanizzi, B., Chevanne, H., Bonomi, E., and Cristini, A. (2011). *OpenCL implementation of the 3D CRS optimization algorithm*, chapter 678, pages 3475–3479. Society of Exploration Geophysicists.

Neidell, N. S. and Taner, M. T. (1971). Semblance and other coherency measures for multichannel data. *Geophysics*, 36(3):482–497.

Ni, Y. and Yang, K. (2012). A GPU based 3D Common-Reflection-Surface stack algorithm with the output imaging scheme (3d-crs-ois). In *SEG Technical Program Expanded Abstracts*, pages 1–5.

Putnam, A., Caulfield, A., Chung, E., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., and Burger, D. (2014). A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)*.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA. ACM.

Singh, D. P., Czajkowski, T. S., and Ling, A. (2013). Harnessing the power of fpgas using altera's opencl compiler. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 5–6, New York, NY, USA. ACM.

Taner, M. T. and Kohler, F. (1969). Velocity spectra - Digital computer derivation and applications of velocity functions. *Geophysics*, 34(6):859–881.

The Khronos Group – Connecting Software to Silicon (Nov 2015). See https://www.khronos.org/.

Top 500 supercomputer sites (Nov 2015). See http://www.top500.org/.