CORE
Metadata, citation and similar papers at core.ac.uk
Provided by UPCommons. Portal del coneixement obert de la UPC

Evaluating directive-based programming models on Wave Propagation Kernels

1 Introduction

In order to tackle the ever-increasing challenges imposed by new exploration areas around the world, HPC systems are mandatory. The need for more HPC resources increases with the complexity of the area under exploration. However, this HPC resources could not be provided without considering the involved energy consumption, as it may prevent the construction of bigger (e.g. Exascale) HPC systems.

To reduce overall energy consumption in the design of HPC facilities, leading technologies vendors introduced many-core devices and heterogeneous computing to the supercomputers. In the last decade, some of the fastest machines in the TOP500[1] increasingly include a mix of CPUs, and coprocessors or accelerators. Thus, forcing exploration codes to be ported to such new architectures. Considering that the Oil & Gas industry has more than 30 years of legacy code, the dedicated effort could be huge.

Also, porting codes became a more complex task because of the different types of parallelism and memory hierarchies available. As portability of parallel applications and the programmer productivity are recognized as major concerns in HPC field, architecture-specific programming, plus its corresponding rewrite phase, are not an option for performance porting [5].

To this extent, several programming models came into play to exploit the potential of the coprocessors and accelerators available. For example, high-level directive-based programming models (e.g. as OpenMP, OpenACC, and OmpSs) rely on specifying to the compiler the parallelism directives to release users from manually decomposing and processing the parallel regions.

In this work, we evaluate the costs and benefits (i.e. speedup) of porting a sequential Elastic Full-Waveform Inversion (FWI) propagation kernel to several parallel versions. Porting is achieved through high-level, directive-based programming models, such as OpenMP, OmpSs and OpenACC, to take advantage of modern architectures, such as NVidias's GPUs and Intel's Xeon Phi coprocessors. An important remark is that in this work no optimization will be applied to the ported kernel. With this, we will measure the benefit of porting and parallelizing at the minimum possible cost.

2 Wave propagation in anisotropic elastic media

To provide a complete FWI system, a kernel able to simulate the wave propagation phenomenon is required. In this work, we use an elastic wave propagator following Newton's second law and the linearly elastic stress-strain relationships [4]. Surface forces on an elastic body comprise six different stress components, three compressional stresses σ_{xx}, σ_{yy} and σ_{zz}, and three shear stresses σ_{yz}, σ_{xz} and σ_{xy}, where subscripts stand for the face and direction of force application. Stresses induce material strains that are quantified regarding displacement gradients. Time differentiation of the stress-strain relations (Eqs.1) coupled to the

[1] TOP500 Supercomputer Site, www.top500.org

Newton's equation of motion (2) allows describing wave propagation in an elastic medium. In this system, u, v, and w represent the particle velocity components in the $x-$, $y-$ and $z-$ directions, respectively, and the stiffness tensor C defines the anisotropic response of the material.

$$\begin{pmatrix} \partial_t \sigma_{xx} \\ \partial_t \sigma_{yy} \\ \partial_t \sigma_{zz} \\ \partial_t \sigma_{yz} \\ \partial_t \sigma_{xz} \\ \partial_t \sigma_{xy} \end{pmatrix} = C \begin{pmatrix} \partial_x u \\ \partial_y v \\ \partial_z w \\ \partial_z v + \partial_y w \\ \partial_z u + \partial_x w \\ \partial_x v + \partial_y u \end{pmatrix} \tag{1}$$

$$\begin{aligned} \rho \partial_t u &= \partial_x \sigma_{xx} + \partial_y \sigma_{xy} + \partial_z \sigma_{xz} \\ \rho \partial_t v &= \partial_x \sigma_{xy} + \partial_y \sigma_{yy} + \partial_z \sigma_{yz} \\ \rho \partial_t w &= \partial_x \sigma_{xz} + \partial_y \sigma_{yz} + \partial_z \sigma_{zz} \end{aligned} \tag{2}$$

The tensor C is always a symmetric matrix with 21 arbitrary components in the most general anisotropic case. However, certain materials can be described with fewer parameters, and the most frequent symmetry types are monoclinic, orthorhombic and transversely isotropic, with 13, 9 and 5 independent parameters, respectively. On the other end, only the two Lamé constants permit defining the non-zero C entries that model isotropy. In this work, we will focus on the complete C matrix to allow arbitrary anisotropy and nearly realistic topography [3].

In order to implement Eqs. 1 and 2 for supporting fully anisotropic scenarios, we have used a Finite Differences (FD) method over a Fully Staggered Grid [2]. Figure 1.c shows the structure of an FSG cell, compared to more traditional Acoustic (Figure 1.a) and Elastic VTI (Figure 1.b) mesh cells.

Using an FSG grid will lead us to a loop in time where velocities are updated based on stresses values in odd iterations and the other way around for even iterations. Velocities update involves the computation of 12 different 3D stencils plus 12 3D material interpolations for each point of the velocity grid. We store materials in a single vertex of the FSG cell for memory saving issues, trading storage per computation. On the other hand, stresses update consists of 28 3D stencils computation plus 84 3D interpolations for the material properties. Notice, that both velocities and stresses calculations are typically dominated by accesses to main memory to retrieve all the data needed to update the corresponding values.

3 Target Programming Models and Architectures

Our sequential version of the kernel of FWI was parallelized using the following directive-based programming models:

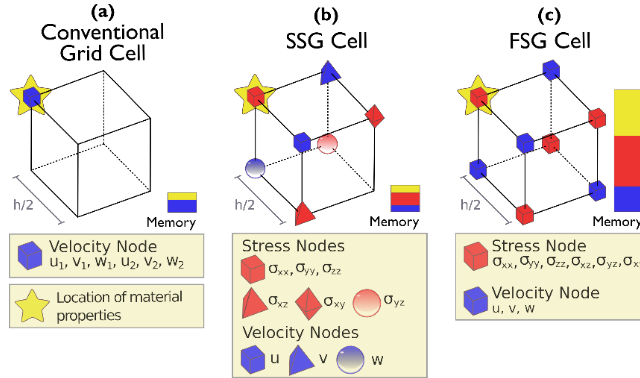

Figure 1: Different grid cells for a conventional grid (a), traditionally used for acoustic propagation, Standard Staggered Grid (SSG) for elastic VTI media (b) and the FSG (c) used in this work for arbitrary anisotropy and topography. Notice, the lateral bar indicating the relative amount of memory needed for each grid cell.

- **OpenMP** [7] is the leading open standard with a highly usable interface for parallel performance on heterogeneous devices. It supports a directive based API for C/C++ and Fortran on multiple shared memory systems. OpenMP is jointly defined by a group of vendors, laboratories, users, and academia.

- **OmpSs**, is an in-house programming model that supports irregular and asynchronous parallelism and heterogeneous architectures, including data dependences and data-flow concepts to automatically move data as necessary [1]. OmpSs is available for GPUs, FPGAs, and co-processors, such as Xeon Phi.

- **OpenACC** [6] directives provide portable high-level programming that is potentially less invasive to application code, but does not provide access to all features of the underlying hardware. OpenACC supports offloading to NVIDIA GPUs and, to x86 CPUs when using the PGI compiler suite.

The ported kernel was evaluated in different hardware architectures (as seen in Table 1), that are commodities at the time of writing. The general purpose architecture is based on a dual processor, Intel Xeon E5-2640 v3 with 8-core processors (Haswell) at 2.60GHz, and 256 GB of Main memory. The second architecture is a single socket Intel Knight Landing (KNL) Xeon Phi at 1.40 GHz, with 68 cores per socket (up to 4 threads per core), 6x16GB DDR4 and 8x2GB of MCDRAM. For GPUs we have two alternatives, the first is an Nvidia's K40 card with 12GB of GDDR55 memory and the second one is an Nvidia's Titan X with 12GB of G5X memory, based on the newest Pascal architecture from Nvidia.

4 Results

This work is concerned about key aspects of using current high-level programming models when aiming for portability. Hence, we have considered not only pure performance metrics (see Table 1), but also the effort and knowledge required to port the application (see Table 2).

It is noticeable that porting our code to selected programming models did not required profound modifications. A well-structured code and the amenable nature of finite differences scheme for parallel programming contributed to this fact. However, this may not happen for other applications.

Programming Model	Compiler (version)	General Purpose	KNL	K40	Titan X
Sequential	Intel 16.0.3	1.00			
OpenMP	Intel 17.1	3.13x	10.84x		
OmpSs	Intel 17.1	3.20x	9.51x		
OpenACC	PGI 17.1			15.91x	44.73x
OpenACC+CUDA	PGI 17.1			22.26x	55.01x

Table 1: Speedup for each programming model

	OpenMP	OmpSs	OpenACC		OpenACC extended	
			1 GPU	2 GPU	1 GPU	2 GPU
Lines of code	32	40	180		188 + CUDA kernels	
Impact on code structure	Low	Low	High		High	
Algorithm knowledge	Low	Medium	High		High	

Table 2: Development of ported version for FWI kernel

Regarding the amount of work required for porting the kernel for the evaluated architectures, OpenMP, OmpSs and OpenAcc versions took in the order of hours in contrary, the implementation based on OpenAcc+CUDA took a few days.

At a high level, OpenMP and OmpSs programming models are very similar from the code's point of view. However, writing our code based on OmpSs tasks exposed new levels of parallelism while done transparently to the programmer by the runtime. For instance, it was natural for OmpSs to overlap computation and I/O access, or calls to *free* and *malloc* functions. However, it came at the price of an in-depth knowledge of the algorithm at hand and careful attention to the granularity of tasks to target best performance.

OpenACC address one of the most complex issues of portability: heterogeneity. Also, and unlike the other two languages, it can generate machine-specific code for CUDA and general purpose devices, from sequential CPU code. It is important to notice that, although OmpSs can manage virtually any heterogeneous

setup it does not generate CUDA code automatically. Such a great feature, however, comes at the price of an extra effort on the programmer's side. Notions of CUDA programming model are critical for achieving good performance. PGI compiler needs to be informed about details of the micro-architecture, and architecture-dependent flags can be provided to control advanced parameters such as the number of registers used by the kernels. For instance, OpenACC involves double-buffering for overlapping memory transfers. It is interesting that OpenACC lets us write our low-level kernels using CUDA. It enables continuous optimization of applications that already run on GPUs. In this way, the programmer can focus on critical kernels while OpenACC takes care of managing data dependencies among multiple devices.

In summary, OpenMP is perhaps the most standardized and portable way of generating shared-memory CPU code. Starting at 4.0 version, the language is supporting some features of task-based programming paradigm and also including support for accelerators. However, and because it is a standard, these changes arrive slowly. Requiring the same amount of effort as OpenMP, OmpSs has the potential to increase the resulting performance. A careful comparison between OpenACC and OmpSs should be undertaken in the future, but always considering OpenACC capability for generating CUDA code.

5 Conclusions

In this work, we choose an elastic wave propagation kernel for porting to different HPC hardware architectures that are commodities today. This exercise was carried out for several pragma-based (i.e. code annotation) programming models, such as OpenMP, OmpSs, and OpenAcc. The results show that it is possible to obtain a parallel code for current HPC architectures investing a few hours or days. Moreover, the obtained speedup is up to an order of magnitude with respect to a sequential code. This conclusion is essential for evaluating the porting and parallelization of legacy code to current architectures. The shown results, however, provide parallelism inside a single computational node. A wider study should be carried out for evaluating the costs of porting and parallelizing the code across computational nodes.

6 Acknowledgments

Authors thank Repsol for the permission to publish the present research, carried out at the Repsol-BSC Research Center. This work has received funding from the European Union's Horizon 2020 Programme (2014-2020) and from the Brazilian Ministry of Science, Technology and Innovation through Rede Nacional de Pesquisa (RNP) under the HPC4E Project (www.hpc4e.eu), grant agreement n.° 689772.

References

[1] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc Gonzàlez, Francisco Igual, Daniel Jiménez-González, Jesús Labarta, Luis Martinell, Xavier Martorell, Rafael Mayo, Josep M. Pérez, Judit Planas, and Enrique S. Quintana-Ortí. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38(5):440–459, 2010.

[2] Sofia Davydycheva, Vladimir Druskin, and Tarek Habashy. An efficient finite-difference scheme for electromagnetic logging in 3D anisotropic inhomogeneous media. *Geophysics*, 68(5):1525–1536, 2003.

[3] Josep de la Puente, Miguel Ferrer, Mauricio Hanzich, José E Castillo, and José M Cela. Mimetic seismic wave modeling including topography on deformed staggered grids. *Geophysics*, 79(3):T125–T141, 2014.

[4] Aki Keiiti and Paul G. Richards. *Quantitative Seismology*. University Science Books, 2003.

[5] Verónica G. Vergara Larrea, Wayne Joubert, M. Graham Lopez, and Oscar Hernandez. Early Experiences Writing Performance Portable OpenMP 4 Codes . In *Cray User Group Proceedings*, 2016.

[6] OpenACC. OpenACC Standard, 2016.

[7] OpenMP. OpenMP, version 4.5, 2016.