Introduction

Acoustic wave propagation has been the preferred engine for geophysical exploration applications for the last few years due to the large cost involved in using better approximations, especially for 3D full-wave field modelling-based applications. Hence, simplified approaches have been used to generate images of the subsurface so that data processing can be finished in a reasonable time. The current trend in seismic imaging aims at using an improved physical model, considering that the Earth is not rigid but an elastic body. This new model takes simulations closer to the real physics of the problem, at the cost of raising the needed computational resources.
Moreover, to take the simulation representation a step closer to the real physics, some kind of anisotropy in the propagation medium should be considered. However, this again may rice the computational cost of the simulation.
On the hardware front, recently developed high-performing devices, called accelerators or co-processors, have shown that can outperform their general purpose counterparts by orders of magnitude in terms of performance per watt. These new alternatives may then provide the necessary resources for making possible to represent complex wave physics in a reasonable time.
There might be, however, a penalty associated to the usage of such devices, as some portion of the simulation code might need some re-writing or new optimization strategies explored and applied (Araya-Polo et al., 2011). In this work we will show some optimization strategies evaluated and applied to an elastic propagator based on a Fully Staggered Grid, running on the Intel® Xeon Phi™ coprocessor. It is important to remark, that the propagator is able to reproduce elastic wave propagation, even for an arbitrary anisotropy.

Full anisotropic elastic wave propagation

Seismic wave propagation is a phenomenon governed by Newton's dynamic laws and the theory of elasticity (see e.g. Aki and Richards, 2002). Neglecting the body rotation, a body can be subjected to six different stress components, three compressional stresses acting upon each Cartesian axis (σxx, σyy and σzz) and three shear stresses (σyz, σxz and σxy). Stresses, in turn, produce strains in the material which can be expressed in terms of displacement gradients. Differentiating in time the stress–strain relation leads us to the coupled equation system (Eqs. (1) and (2)) which describes wave propagation in elastic media where u, v, and w are the particle velocity components in the x-, y- and z-directions, respectively, and the stiffness matrix C_{6x6} determines the anisotropic behaviour of the material.

$$\begin{pmatrix} \partial_t\sigma_{xx} \\ \partial_t\sigma_{yy} \\ \partial_t\sigma_{zz} \\ \partial_t\sigma_{yz} \\ \partial_t\sigma_{xz} \\ \partial_t\sigma_{xy} \end{pmatrix} = \mathbf{C}_{6\times6} \begin{pmatrix} \partial_x u \\ \partial_y v \\ \partial_z w \\ \partial_z v + \partial_y w \\ \partial_z u + \partial_x w \\ \partial_x v + \partial_y u \end{pmatrix} \quad (1)$$

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

Although the C matrix can have up to 21 components in the most general anisotropic case, certain materials can be described with less parameters (the most frequent symmetry types are monoclinic, orthorhombic and transversely isotropic, with 13, 9 and 5 independent parameters). On the other end, the fully isotropic case requires only two material parameters to fully determine C.
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 (FSG, Davydychevaetal.,2003). Figure 1.c shows the structure of an FSG cell and is compared to more traditional Acoustic (Figure 1.a) and Elastic VTI (Figure 1.b) mesh cells.
On the software front, implementing an FD method over 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 grid. 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.

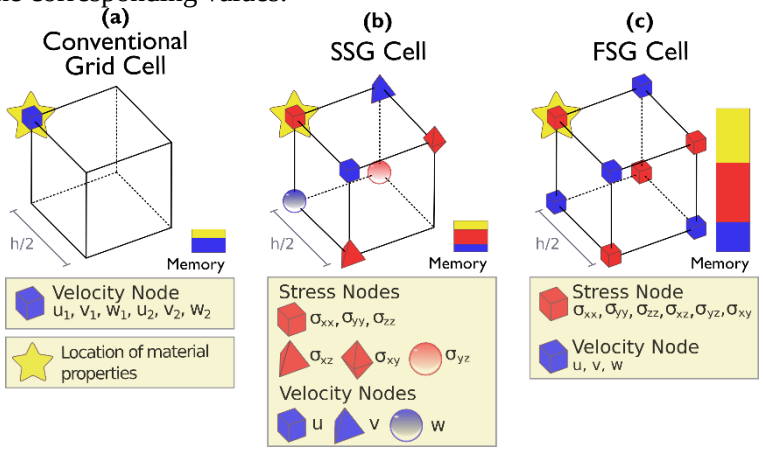

Figure 1 *Different grid cells for a conventional grid (a), traditionally used for acoustic propagation, Standard Staggered Grid (SSG) for elastic VTI media (b) and the FSG (c) used in this work. Notice, the lateral bar indicating the relative amount of memory needed for each grid cell.*

The Intel Xeon Phi coprocessor

The Intel Xeon Phi coprocessor is the first product based on the Intel® Many Integrated Core architecture. Each coprocessor features up to 61 cores. Each core is based on the x86 architecture but they are in-order cores that run at a much lower frequency than the standard Intel Xeon processors. Each core has four hardware threads that are scheduled in a round-robin fashion. Up to two instructions from a thread context can be issued per cycle. In addition, the cores come with a specially designed Vector Processing Unit that provides the architecture with 512-bit SIMD instructions. This SIMD instruction set also supports masked operations, gathers, scatters, and fused multiply-and-add operations. As a result of the combination of these SIMD instructions and the large number of threads the coprocessor yields up to 1.2 TFLOPS/s in double precision on a 300W TDP package.
The L1 cache (32KB for instructions, 32KB for data) is shared among the hardware threads of each core. The L2 cache is distributed across the different cores (with a 512KB slice in each core for both data and instructions). Both L1 and L2 caches are fully coherent. L2 caches can communicate between themselves to serve misses without going to the on-board memory. Cores and the L2 slices are connected through a bi-directional ring. The ring also connects the memory controllers and the I/O interface that allows the coprocessor to communicate with the host through the PCIe bus. The memory controller has up to 16 memory channels available to access the on-board GDDR5 memory.
The coprocessor supports a standard software stack with a Linux operating system and programming models such as OpenMP*, OpenCL*, MPI, or Intel® Threading Building Blocks. In this sense, applications written in one of these paradigms are readily available to run on the Intel Xeon Phi coprocessor.

Optimization strategies

Our initial implementation has the 3D spatial grid loop split into 12 parts for velocities and 4 for stresses. This way, we can reduce the main memory pressure by narrowing the number of components accessed in each spatial loop iteration. The outermost loop in space has been parallelized using OpenMP, leaving to each thread a number of consecutive memory planes to update. This version, named *OpenMP*, is our evaluation baseline. The following optimizations have been applied to the baseline version to improve its performance in the Intel Xeon Phi coprocessor:
Restrict. By using the *__restrict* qualifier, we can inform the compiler that the pointers in our code do not alias. This enables the compiler to remove false loop-carried dependences and auto-vectorize our code. This was by far the best optimization in terms of ROI.

OpenMP SIMD. We can slightly improve the aforementioned auto-vectorization by using the simd directives from OpenMP (Klemm et al., 2012). We annotated the innermost spatial loop with this directive and we also allocated memory properly aligned to the boundary of the vector length of the architecture (64 bytes). We also used the OpenMP *aligned* clause and the *__assume* built-in to pass this information to the compiler.
OpenMP Scheduling. The default loop scheduling algorithm in OpenMP is static, but it is not necessarily the best one. We evaluated the three most relevant loop scheduling strategies: *static*, *dynamic* and *guided*. The best performance in our case resulted from using dynamic scheduling.
Loop Blocking. To improve temporal data reuse and reduce the memory bandwidth requirements per updated grid point, we implemented a loop blocking strategy. In addition of splitting the Y dimension (the most significant one) to give a collection of full planes to each thread, the X dimension is also partitioned. Consequently, each thread processes a chunk of grid points contained in an XY block. We also evaluated to apply blocking in the Z dimension, but this did not yield any improvement. We performed a systematic benchmarking of block sizes to find out the best blocking configuration that best fits the characteristics of the memory hierarchy and the number of threads used.
Prefetching. The complex memory hierarchy of the Intel Xeon Phi coprocessor makes data software prefetching indispensable to get the best performance. We enabled aggressive software prefetching in the compiler and carried out an exhaustive exploration to find out the best prefetching distance in conjunction with the best block size, for both velocities and stresses calculation stages.
Code reordering. As compiler optimizations are not optimal over complex code sections, we reordered the code in the innermost loop of the *stresses* stage so that produced variables are consumed as soon as possible. This helps the compiler with its *liveness* analysis and improves the generated code.
Intrinsics. Although tedious and error-prone, users can use low-level vector intrinsics to provide highly tuned vector codes. In our case, we used aligned vector loads and vector shift instructions to minimize aligned and unaligned vector memory loads that overlap in memory (Caballero et al., 2015).
Reduced precision. We enabled the use of some fast but less precise floating point instructions using the *–fimf–domain–exclusion=31* compiler flag which guarantees that our application will not produce values from some special domains. The compiler can then generate more efficient code.

Compilers	The Mercurium C source-to-source Compiler and the Intel C Compiler 15.0.2
Systems	Intel Xeon Phi coprocessor C0PRQ-7120 (61 cores using 4 threads per core) ** 2S Intel Xeon E5-2697v3 (24 cores using 1 thread per core) **
CFLAGS	–O3 –std=gnu99 –ansi–alias –restrict –qoverride–limits –fopenmp –mmic
Grid Size	288x80x720

Table 1 *Evaluation environment specifications*

These proposed optimizations were implemented and evaluated for a computational grid built using the FSG cells from Figure 1. Table 1 shows the environment used for the evaluation. We used the Mercurium Source-to-source Compiler to parallelize and vectorize the application with OpenMP. The resulting source code was then compiled with the Intel® C Compiler for the Intel Xeon Phi coprocessor.
Figure 2 depicts the achieved performance on an Intel Xeon Phi processor for all the proposed optimizations. The optimizations were enabled one after the other, so the Figure shows the accumulative improvement of all previous optimizations. For reference, the performance of the same code and same optimizations on a system with two Intel® Xeon® E5-2697v3 processors based on the Haswell microarchitecture is also reported. Notice, that we overcome the performance of the Intel Xeon processor by almost 10% and we get 11 speed-up with respect to the baseline implementation. Taking advantage of the vectorization capabilities of the Intel Xeon Phi coprocessor produces the major speed-up, but implementing all the explained optimizations almost doubles that boost.

Conclusions

We have shown a set of optimizations, applied to a Finite Difference numerical method solving elastic wave propagation equations on the Intel Xeon Phi coprocessor. Moreover, the proposed scheme for solving the elastic equation supports arbitrary anisotropy at a higher computational cost when

compared to more traditional ways of solving elastic propagation. The evaluated set of optimizations ranges from memory to compute optimizations. Our results show that it is possible to obtain more than an order of magnitude of improvement when comparing the fully optimized code with a naïve version (which only had OpenMP parallelization included), while up to a 7x of improvement is possible with little investment on code optimization. A comparison with a system with two Intel Xeon E5-2697v3 processor shows that a single Intel Xeon Phi coprocessor is able to outperform such computational architecture.

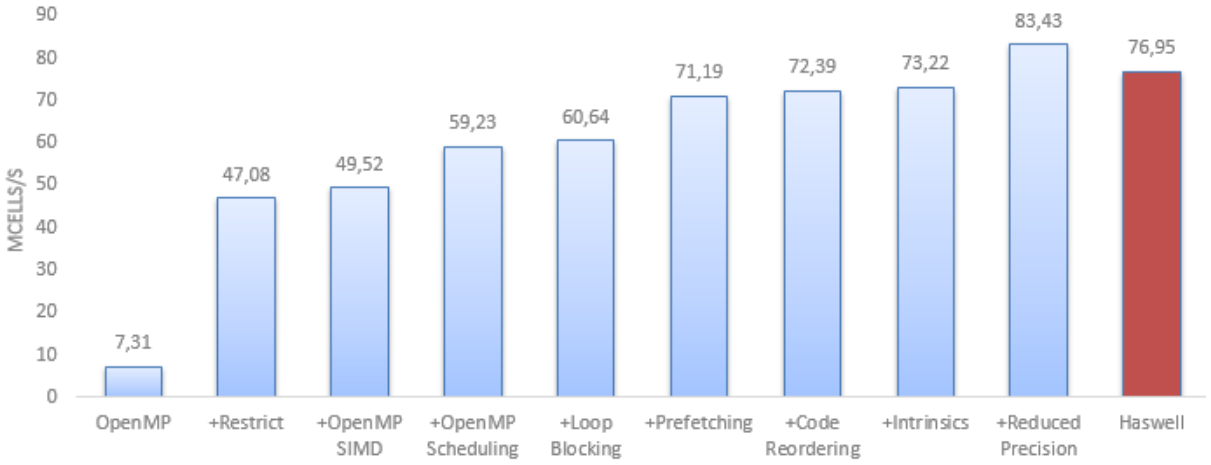

Figure 2 *Absolute performance of the elastic propagator optimizations on an Intel Xeon Phi (in blue) compared with an Intel Xeon E5-2697v3 processor (in red)*

Acknowledgements

We would like to acknowledge the support of the AGAUR (FI-DGR, 2014FLB2 00155), European Commission in the context of the DEEP-ER project (FP7-ICT-610476). Authors also thank Repsol for the permission to publish the present research, carried out at the Repsol-BSC Research Center.

** For more information about performance measurements, go to http://www.intel.com/performance.
System configurations used: Intel Xeon Phi coprocessor with C0 silicon and board version C0PRQ-7120 (61 cores at 1238095 Khz, 16 GB of GDDR Memory at 5.5 GT/sec, 300W TDP, MPSS v3.4.2). Intel Xeon E5-2697v3 processors at 2.6 GHz, SuSe Distribution 11 SP3, Linux Kernel 3.0.101-0.35, 128 GB.

References

Aki, K., Richards, P.G., 2002. Quantitative Seismology. University Science Books, Sausalito, California.
Araya-Polo, M., Cabezas, J., Hanzich, M., Pericas, M., Rubio, F., Gelado, I., Shafiq, M., Morancho, E., Navarro, N., Ayguade, E., CelaJosé, M., Valero, M., 2011. Assessing accelerator-based HPC reverse time migration. IEEE Trans. Parallel Distrib. Syst. 22 (1), 147–162.
Davydycheva, S., Druskin, V., Habashy, T., 2003. An efficient finite-difference scheme for electromagnetic logging in 3D anisotropic inhomogeneous media. Geophysics 68(5), 1525–1536.
Klemm, M., Duran, A., Tian, X., Saito, H., Caballero, D., Martorell, X., 2012. Extending OpenMP* with Vector Constructs for Modern Multicore SIMD Architectures. IWOMP'12.
Mercurium C/C++ Source-to-source Compiler. Accessed: 05/2015. http://pm.bsc.es/mcxx
Caballero, D., Royuela S., Ferrer R., Duran A., Martorell X., 2015. Optimizing Overlapped Memory Accesses in User-directed Vectorization. International Conference on Supercomputing.