Optimizing a Wave Propagation Model for Manycore Systems

Matheus S. Serpa[1], Eduardo H. M. Cruz[1], Matthias Diener[1], Albert Farrés[2], Claudia Rosas[2], Jairo Panetta[3], Mauricio Hanzich[2], and Philippe O. A. Navaux[1]

[1] Informatics Institute, Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil
{msserpa, ehmcruz, mdiener, navaux}@inf.ufrgs.br
[2] Barcelona Supercomputing Center (BSC), Barcelona, Spain
{albert.farres, claudia.rosas, mauricio.hanzich}@bsc.es
[3] Computer Science Division, Aeronautics Institute of Technology (ITA), São José dos Campos, Brazil
jairo.panetta@gmail.com

Abstract

Many software mechanisms for exploration geophysics in Oil & Gas industries are based on wave propagation simulation. To execute such simulations, state-of-art HPC architectures are employed, generating results faster and with more accuracy at each generation. To keep performance scaling, the software must evolve to support the new architectures features. Furthermore, it is important to understand the impact of such changes performed to the software, in order to improve the performance as most as possible. In this paper, we propose several optimization strategies for a wave propagation model for two manycore systems: the Intel Xeon and Intel Xeon Phi processors. We analyze the hardware impact of the optimizations, providing insights of how each strategy is able to improve the performance. Performance was improved by up to 22.6x in Xeon and 116x in Xeon Phi.

Keywords: Wave propagation, Geophysics, Vectorization, Manycore systems

1 Introduction

Geophysics exploration remains fundamental to the modern world to cope with the demand of energetic resources. This endeavor results in expensive drilling costs (100M$-200M$), with less than 50% of accuracy per drill. Thus, Oil & Gas industries rely on software focused on High-Performance Computing (HPC) as an economically viable way to reduce risks. The fundamentals of many software mechanisms for exploration geophysics are based on wave propagation simulation engines. For instance, on seismic imaging tools, modeling, migration and inversion use wave propagators at the core. These simulation engines are built as PDE solvers, where the PDE solved in each case defines the accuracy of the approximation to the real physics when a wave travels through the Earth's internals.

Acoustic wave propagation approximation is the current backbone for seismic imaging tools. It has been extensively applied for imaging potential oil and gas reservoirs beneath salt domes for the last five years. Such acoustic propagation engines should be continuously ported to the newest HPC hardware available to keep competitiveness. At the same time, on the HPC hardware front, the days of faster single core CPUs are over, and the solutions adopted are being replaced by many core technologies [1, 2]. The last decade has seen a trend of building systems with dedicated devices and accelerators, which produce a good return regarding FLOPs/Watt. Among the available HPC alternatives, Nvidia and Intel have dedicated efforts to provide tens to hundreds processing units working at low frequencies, being the Knights family from Intel, also known as Xeon Phi, one of the most promising coprocessors.

Systems are combining power-saving architectures that are readily available for simulations, such as Intel Xeon and Xeon Phi. The Intel Xeon Phi coprocessor is the first product based on the Intel Many Integrated Core architecture. Each coprocessor features up to 61 cores based on the x86 architecture but running at a much lower frequency than the standard Intel Xeon processors. The coprocessors support a standard software stack with a Linux operating system and programming models such as OpenMP, OpenCL, MPI or Intel TBB. Therefore, applications written in one of these paradigms are readily available to run on the Xeon Phi.

This paper evaluates several optimization techniques for an acoustic wave propagator and ports it to the Intel Xeon and Xeon Phi. Our evaluation focuses on improving the performance based on the hardware impact of each of the optimizations applied. Petrobras provided a standalone acoustic modeling program, with the same kernel used on Reverse Time Migration. The code was written in standard C and parallelized with OpenMP. The program simulates the propagation of a single wavelet over time by solving the isotropic acoustic wave propagation (Equation 1), and the isotropic acoustic wave propagation with variable density (Equation 2) under Dirichlet boundary conditions over a finite three-dimensional rectangular domain, prescribing $p = 0$ to all boundaries, where $p(x, y, z, t)$ is the acoustic pressure, $V(x, y, z)$ is the propagation speed and $\rho(x, y, z)$ is the media density.

$$\frac{1}{V^2}.\frac{\partial^2 p}{\partial t^2} = \nabla^2 p \qquad (1)$$

$$\frac{1}{V^2}.\frac{\partial^2 p}{\partial t^2} = \nabla^2 p - \frac{\nabla \rho}{\rho}.\nabla p \qquad (2)$$

The Laplace Operator is discretized by a 12th order finite differences approximation on each spatial dimension. The derivatives are approximated by a second-order finite differences operator.

This work presents a first approximation to correlate the hardware impact of optimization performed on a largely used seismic imaging simulator running on new architectures, by applying four optimization techniques: (1) loop interchange to improve data locality; (2) vectorization to increase the performance of floating point computations; (3) loop scheduling and collapse to improve load balancing; and (4) thread and data mapping, to have better resource usage; on a real-world application running in Xeon and Xeon Phi processors. Results showed an improvement of 22.6x in Xeon and 116x in Xeon Phi.

2 Related work

Recent architectures as accelerators and coprocessors proved to be well suited for geophysics, magneto-hydrodynamics and flow simulations, outperforming in efficiency the general purpose processors. To obtain maximum performance from these new devices is necessary some re-engineering of regions of the code, if not of the entire application. Thus, Krukeja et al. [3] automatically generate a highly optimized stencil code for multiple target architectures, while Niu et al. [4] suggest using run-time reconfiguration, and a performance model, to reduce re-

source consumption. Caballero et al. [5] studied the effect of different optimizations on elastic wave propagation equations, achieving more than an order of magnitude of improvement compared with the basic OpenMP parallel version.

In [6], they focused on acoustic wave propagation equations, choosing the right optimization technique from systematically tuning the algorithm. The use of collaborative thread blocking, cache blocking, register re-use, vectorization and loop redistribution, obtained a 30x of improvement using 244 threads. Our proposal chooses a largely used seismic imaging simulation based on the acoustic wave propagation, and provides a deeper evaluation of the hardware impact of the optimizations applied when porting it to the Xeon and Xeon Phi processors.

Research efforts such as the presented in Castro et al. [7] improved and evaluated the performance of the acoustic wave propagation equation on Intel Xeon Phi and compared it with MPPA-256, general-purpose processors and a GPU. The optimizations include cache blocking, memory alignment with pointer shifting and thread affinity. Where the best results are obtained from a combination of the first two and showed that performance with the Xeon Phi is close to the state-of-the-art solutions on GPUs. Our work goes one step further by understanding the effect of each optimization in the overall performance.

Rubio et al. [8] rewrote an elastic wave propagator for an arbitrary anisotropy on general-purpose processors, to GPUs and Xeon Phi, showing that the coprocessor provides good performance at reduced development cost. Our optimizations target only isotropic domains to reduce the complexity of the problem and restrict the number of variables playing in the analysis.

Zhebel et al. [9] compared scalability of unmodified codes for finite-differences and finite-element algorithms on Intel Sandy Bridge and Xeon Phi. On the Sandy Bridge, the scalability was similar and non-linear for all the methods, while on the Xeon Phi, only the finite difference showed less scalability, because of some idleness of the I/O and program control thread. Our proposal goes beyond a scalability analysis and looks for a greater understanding of the effect of optimizations on the expected scaling of a real-world application.

3 Manycore Systems that were Optimized

We used two environments to analyze the application performance. First, we used a 2-node Haswell architecture, where each node consists of a 10-core Intel Xeon E5-2640 v2 processor. Each core supports a 2-way Simultaneous Multithreading (SMT) and has private L1 and L2 caches, while the L3 cache is shared between all the cores of the processor. Second, we used a Knights Corner architecture, which is a 57-core Intel Xeon Phi 3120P, supports a 4-way SMT, where each core has a private L1 and L2 cache. Table 1 summarizes the environments used.

System	Parameter	Value
Xeon	Processor	2 × Intel Xeon E5-2650 v3, 10 *cores*, 2 SMT-*cores*
	Microarchitecture	Haswell
	Caches/proc.	10 × 32 KByte L1, 10 × 256 KByte L2, 25 MByte L3
	Main memory	128 GByte DDR4-2133
Xeon Phi	Coprocessor	Intel Xeon Phi 3120P, 57 *cores*, 4 SMT-*cores*
	Microarchitecture	Knights Corner
	Caches/proc.	57 × 32 KByte L1, 57 × 512 KByte L2
	Device memory	6 GByte

Table 1: Configuration of the evaluation systems.

We measured execution time, cache misses, interchip interconnection traffic and the load balance of the applications. To measure cache misses, we used the Intel PCM tool. To measure interchip traffic, we used Intel VTune. Regarding the load balance, we analyzed how many instructions were executed by each thread using the Linux Perf tool. Each experiment was executed 30 times, and we show average values as well as a 95% confidence interval calculated with Student's t-distribution.

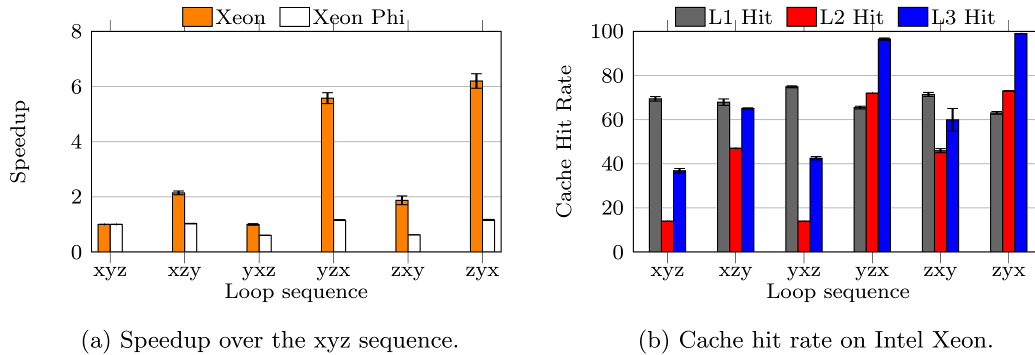
4 Optimizing the Acoustic Wave Propagation Model

This sections presents the optimizations techniques we used to improve the performance in a real world application and the experiments we performed to validate them. The application used as benchmark simulates the propagation of a single wavelet over time by solving the isotropic acoustic wave propagation with constant density under Dirichlet boundary conditions over a 3D domain. The program was wrote in C with OpenMP. The input stencil size was $1024 \times 256 \times 256$. We describe the optimizations and analyze the improvements obtained by each technique. Finally, we present the results of the optimizations combined, in both the Intel Xeon and Xeon Phi processors.

4.1 Improving Cache Memory Usage

Current computer architectures provide caches and hardware prefetchers to help programmers manage data implicitly [10]. Loop interchange technique can be used to improve the performance of both elements by exchanging the order of two or more loops. It also reduces memory bank conflicts, improves data locality and helps to reduce the stride of an array computation. In this way, more data that is fetched to the cache memories are effectively accessed, the data reuse in the caches is increased, and cache line prefetchers are able to fetch data from the main memory more accurately. In this application, we have three loops that are used to compute the stencil. The loops can be executed on any order without changing the results. The default loop sequence was **xyz**.

We propose to change the loop sequence from **xyz** to all possible combinations. The outermost loop is the one that was always parallelized using threads. In Figure 1a, we show in the X axis the sequences and in the Y axis the speedup versus the **xyz** sequence. The bars represent the architecture. Loop sequences **yzx** and **zyx** have better results in both Xeon and



(a) Speedup over the xyz sequence.

(b) Cache hit rate on Intel Xeon.

Figure 1: Results using different loop sequences.

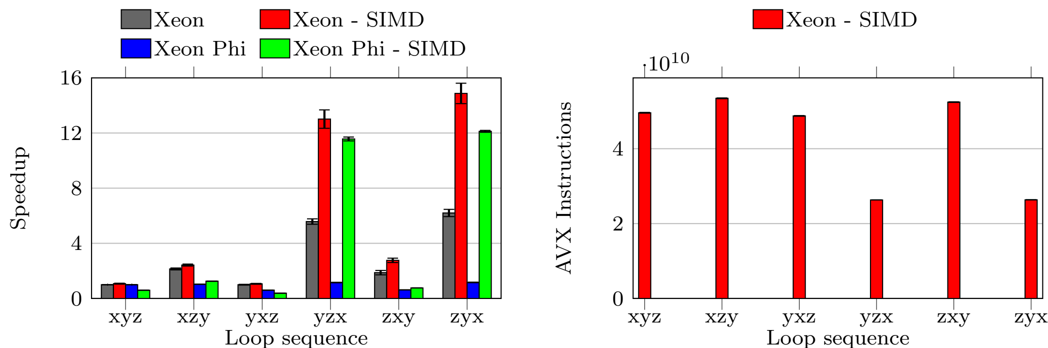
Xeon Phi. These sequences are better than others because the data is accessed in a way that benefits more from the caches, as can be observed in the cache hit rates shown in Figure 1b. The L2 and L3 cache hit rates were improved from 14% and 36.8% to 73% and 99% when the loop sequence was changed to **zyx**. However, this was not the case with the L1 cache, as its hit rate decreases from 69.4% to 63%.

The differences in cache misses happened because the data access stride becomes different when you change the loop sequence, influencing both spatial and temporal localities. Despite this small reduction in the L1 hit rate, the increase of the L2 and L3 hit rates in the **zyx** sequence resulted in the highest performance improvement and is therefore the best choice for this application. The performance improvement in the Xeon Phi is lower than in Xeon because the amount of cache memory available per thread in the Xeon Phi is much lower than in Xeon.

4.2 Exploiting Vector Instructions (SIMD)

Recent hardware approaches increase performance by integrating more cores with wider SIMD (Single instruction, multiple data) units [11]. This data processing technique, called Vectorization, has units which perform, in one instruction, the same operation on several operands. To maximize the effectiveness of vectorization, the memory addresses accessed by the same instruction on consecutive loop iterations must also be consecutive. In this way, the compiler can load and store the operands of consecutive iterations using a single load/store instructions, optimizing cache memory usage, since data is already fetched in blocks from the main memory anyway. More recent processors introduce the support for `gather` and `scatter` instructions, which reduce the overhead of loading/storing non-consecutive memory addresses. Nevertheless, the performance is still much higher when the addresses are consecutive. In this context, we modified the source code such that the memory addresses of the same instructions were the same in consecutive loop iterations.

We used Advanced Vector Extensions (AVX), which is a instruction set architecture extension to use SIMD units to increase the performance of the floating point computations. These instructions use specific floating point units that can load, store or perform clculations on several operands at once. As previously described, the efficiency of AVX is better when the elements are accessed in the memory contiguously, as they can be loaded and stored in blocks. We show the execution time speedup and the number of AVX instructions in Figures 2a and 2b.



(a) Performance gain using vectorization.

(b) Advanced Vector Extensions on Xeon.

Figure 2: Impact of vectorization on Xeon and Xeon Phi.

The speedup shown is relative to the **xyz** loop sequence without AVX. The sequences **yzx** and **zyx** have better results because they have more elements being accessed contiguously. The **yzx** sequence exploits more parallelism for vectorization, with a speedup of 14.9x in Xeon and 12.1x in Xeon Phi. It has less AVX instructions than other sequences such as **xyz**, because these other sequences need more loads and stores due to irregular memory accesses.

4.3 Improving Load Balancing

Some applications have regions with different computation load requirements, e.g. boundaries, potentially causing unevenness in the computing time among the threads. The time to execute a parallel application is determined by the task that takes more time to finish, and thereby by the amount of work of the core with most amount of work. Hence, by distributing the work more evenly among the cores, we can reduce the execution time of an application. Load Balancing techniques reduce these disparities and thereby improve resource usage and performance. In the context of manycore systems, load balancing is even more important due to the large number of cores.

OpenMP specification has a directive to indicate whether the scheduling is static, dynamic or guided. The *static* scheduling is the default value and it assigns chunks to threads in a round-robin fashion before the computation starts. The *dynamic* and *guided* approaches distribute the work during run time as thread request, but in *dynamic*, all the chunks have the same size, while *guided* assigns larger chunks first, and their size decreases along the iterations. In addition, loop collapse helps also to improve load balancing, because it increases the total number of iterations partitioned across the threads by collapsing two or more loops.

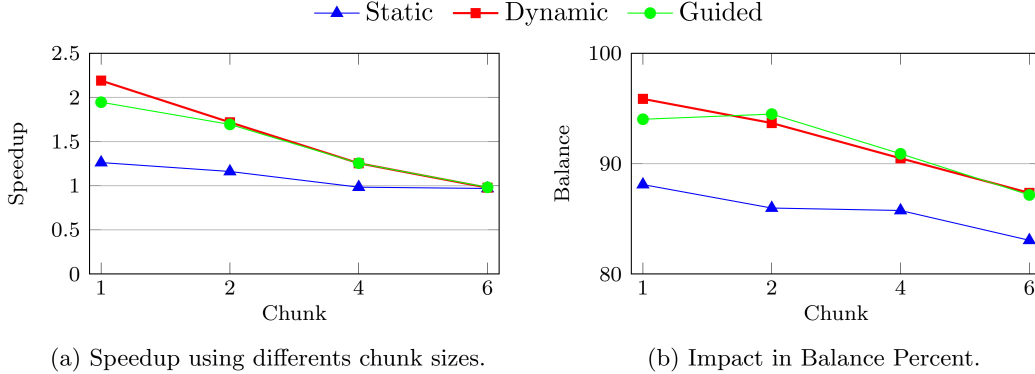
The metric *Balance*, shown in Equation 3 [12], helps to indicate imbalances in a code. Its output ranges from 0 to 100, where 100 means perfectly balanced threads.

$$Balance = 100 - \frac{max - avg}{max} \times 100 \quad (3)$$

We investigated the impact of different OpenMP scheduling policies and chunk sizes. Furthermore, we applied loop collapse in outer loops to increase the total number of iterations that will be partitioned across the threads. Figures 3a and 4a show the speedup of each combination of schedule policy and chunk size. We also measured the load balance of the applications, shown in Figures 3b and 4b, to help us understand the reasons for the performance improvements. The baseline used to calculate the speedup is the execution time of the default schedule policy, which is static with a chunk equals to the number of iterations divided by number of threads.

Figures 3a and 3b present results without collapse. In these experiments, only the outer loop is divided between the threads. We test all possible chunk sizes from 1 to the outer loop size divided by number of threads. Although a chunk size greater than this could be used, it would be a worse scenario, since some threads would not calculate anything. The best speedup without collapse is using the dynamic scheduling policy. It is up to $2.09\times$ faster than default scheduler. This scheduling policy is useful to applications whose workload changes in runtime or have some regions with unbalanced workloads. Results show a balance percent of 95.8% using the dynamic scheduler with 1 as chunk size. We evaluated different chunk sizes to reduce the overhead caused by the dynamic and guided schedulers. The experiments show that chunks greater than 1 have worse performance because the granularity becomes coarse grained, with a lower balance.

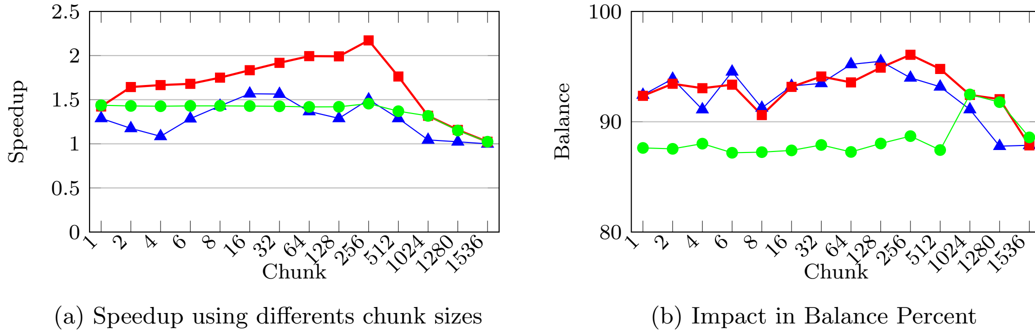
A way to improve the performance with larger chunks is collapsing the loops. The idea is that, with more work to be divided between the tasks, larger chunks can be used while keeping a good load balance. In order to investigate this, we collapsed the outer loops and



(a) Speedup using differents chunk sizes.

(b) Impact in Balance Percent.

Figure 3: Scheduling outer loop iterations.



(a) Speedup using differents chunk sizes

(b) Impact in Balance Percent

Figure 4: Effect of collapsing two loops and scheduling their iterations.

test different combinations of scheduler and chunk sizes. We show the speedup and balance percent in Figures 4a and 4b, respectively. Results show that the best policy is still the dynamic scheduler, with a speedup of 2.17×. As expected, we were able to use larger chunks without harming the load balance. The best chunk size was 256, which is the same size of the loop that was collapsed. The balance was improved to 96.1%.

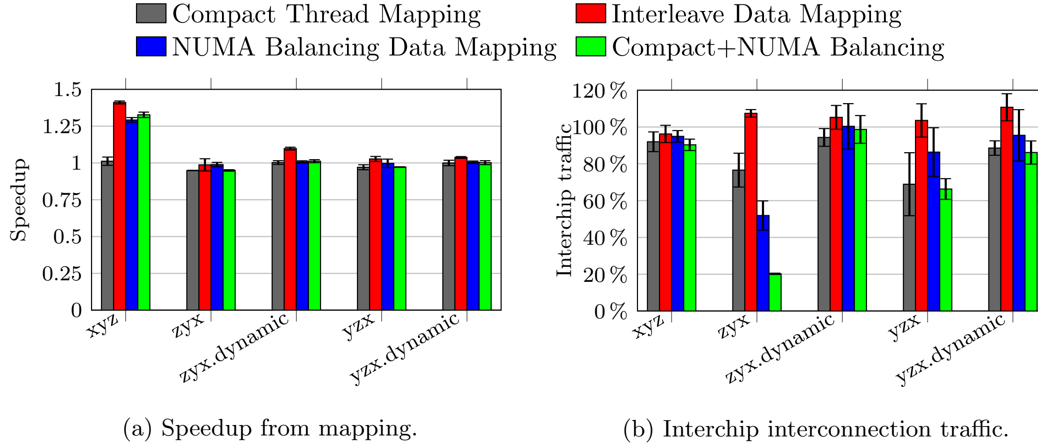
4.4 Thread and Data Mapping

The goal of mapping mechanisms is to improve resource usage by arranging threads and data according to a fixed policy, where each approach may target different aspects to enhance. For example, there are techniques focused on improving locality, to reduce cache misses and remote memory accesses, as well as traffic on inter-chip interconnections [13]. While others seek a uniform load distribution among the cores and memory controllers. In this work, we analyze five mapping strategies:

Baseline The default thread mapping scheduling of Linux, focused on load balancing, combined with a first-touch data mapping policy.

Compact Thread Mapping A compact thread mapping arranges neighbor threads to closer cores according to the memory hierarchy, coupled with a first-touch data mapping policy.

Interleave Data Mapping The default thread mapping scheduling of Linux combined with the interleave data mapping, which arranges consecutive pages to consecutive NUMA nodes.



(a) Speedup from mapping.

(b) Interchip interconnection traffic.

Figure 5: Results obtained evaluating the thread and data mapping policies.

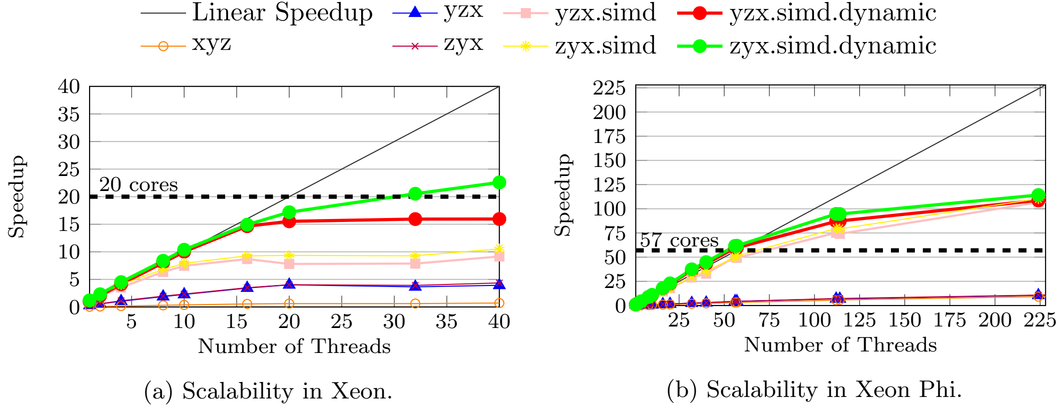
NUMA Balancing Data Mapping The default thread mapping scheduling of Linux, combined with the NUMA Balancing data mapping [14], which migrates pages along the execution to the NUMA node of the latest thread that accessed the page.

Compact+NUMA Balancing A compact thread mapping, combined with the NUMA Balancing data mapping.

The results obtained from different mapping policies in the Xeon processor are shown in Figure 5. We did not evaluate this on the Xeon Phi because it has a fixed mapping of memory addresses to memory controllers. The speedup and interchip interconnection traffic are normalized relative to the baseline mapping with the corresponding loop sequence, such that we can measure the benefits from mapping more precisely. The results of cache misses, previously shown in Figure 1b, can help us understand the behavior from different mapping policies. The reason for this is that most of the improvements from mapping are due to the reduction of accesses to the main memory, such that these benefits are mitigated if the cache hit rate is high. This can be observed in the results, where the **xyz** variant was the one that benefited most from mapping, was also the one with most cache misses. In the other configurations, since the L3 cache hit is very high, we have few access to the main memory, such that, as explained, the benefit from mapping is lower. The usage of an interleaved data mapping provided a better distribution of the load between the memory controllers, with the cost of additional interchip traffic. Despite the trade-off between the load and interchip traffic, the interleaved data mapping provided the best improvements overall.

4.5 Scalability

Figures 6a and 6b show the speedup for different optimization algorithms in the Xeon and Xeon Phi processors. In both cases, **yzx-simd** is our baseline, since previous versions were sequential optimizations. The **yzx-simd** and **zyx-simd** versions, which take advantage of SIMD units, have better performance and scalability than **xyz**, **yzx** and **zyx**. The performance due to vectorization is even better in Xeon Phi because its architecture provides wider SIMD vector units. Furthermore, the pipeline of Xeon is out-of-order, which is able to exploit a higher degree of instruction level parallelism (ILP) than the pipeline of Xeon Phi, which is in-order [15]. This higher ILP mitigates a bit the improvements from vectorization in Xeon compared to Xeon Phi.



(a) Scalability in Xeon.

(b) Scalability in Xeon Phi.

Figure 6: Scalability of different algorithm versions.

The performance and scalability of the versions that do not use a dynamic scheduler were limited by a high load imbalance. For the versions **yzx-simd-dynamic** and **zyx-simd-dynamic**, we analyzed different scheduling options, as explained in Section 4.3, and determined that the best configuration was the usage of a dynamic scheduler, with a chunk size of 256, and collapsing two loops. The improvement was better in Xeon because, in Xeon Phi, the runtime automatically sets a *scatter* affinity [16] from threads to cores, which in general improves the load balancing.

The speedup in both systems is not linear. The reason for this is because both Xeon and Xeon Phi use Hyper-Threading to allow executing several threads per core. Since threads running in the same core share several resources, the speedup is expected to be a little higher than the number of cores [17]. In Xeon, the best speedup was 22.6x for 20 cores. In Xeon Phi, the best speedup was 116x for 57 cores.

5 Conclusions and Future Work

In this paper, we applied and analyzed the performance of a set of optimization techniques on Intel Xeon and Xeon Phi. We showed that these techniques can improve the performance of a real world application in both processor and coprocessor. We also made use of hardware performance counters to analyze the impact of each optimization. The optimizations that we presented can also be used to optimize other applications and architectures.

In our experiments, we show that loop interchange is a useful technique to improve performance of different cache levels, being able to improve performance by up to 6.2x and 1.2x in Xeon and Xeon Phi, respectively. These improvements happened because we were able to increase the cache hit ratio by up to 99%. Furthermore, by changing the code such that elements are accessed contiguously between loop iterations, we were able to vectorize the code, which improved performance by up to 14.9x and 12.1x. By modifying the scheduling, we were able to increase the performance by 2.19x, due to a better load balance among the cores. Thread and data mapping techniques were also evaluated, but their performance improvements were mitigated by the high cache hit ratio that we were able to achieve. Combining all these techniques improved the performance up to 22.6x in Xeon and 116x in Xeon Phi.

As future work, we will evaluate these optimizations in the Knights Landing architecture.

Acknowledgments

Our research received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E project, grant agreement 689772, as well as from CNPq and Capes.

References

[1] R. G. Clapp. Seismic Processing and the Computer Revolution(s). In *SEG Technical Program Expanded Abstracts 2015*, pages 4832–4837, 2015.

[2] R. G. Clapp, H. Fu, and O. Lindtjorn. Selecting the right hardware for reverse time migration. *The Leading Edge*, 29(1):48–58, 2010.

[3] N. Kukreja, M. Louboutin, F. Vieira, F. Luporini, M. Lange, and G. Gorman. Devito: Automated fast finite difference computation. In *Procs. of the 6th Intl. Workshop on Domain-Spec. Lang. and High-Level Frameworks for HPC*, WOLFHPC '16, pages 11–19. IEEE Press, 2016.

[4] X. Niu, Q. Jin, W. Luk, and S. Weston. A Self-Aware Tuning and Self-Aware Evaluation Method for Finite-Difference Applications in Reconfigurable Systems. *ACM Trans. on Reconf. Technology and Systems*, 7(2), 2014.

[5] D. Caballero, A. Farrés, A. Duran, M. Hanzich, S. Fernández, and X. Martorell. Optimizing Fully Anisotropic Elastic Propagation on Intel Xeon Phi Coprocessors. In *2nd EAGE Workshop on HPC for Upstream*, 2015.

[6] C. Andreolli, P. Thierry, L. Borges, G. Skinner, and C. Yount. Chapter 23 - Characterization and Optimization Methodology Applied to Stencil Computations. In James Reinders and Jim Jeffers, editors, *High Performance Parallelism Pearls*, pages 377 – 396. Morgan Kaufmann, Boston, 2015.

[7] M. Castro, E. Francesquini, F. Dupros, H. Aochi, P. O.A. Navaux, and J.-F. Méhaut. Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, 54:108 – 120, 2016.

[8] F. Rubio, A. Farrés, M. Hanzich, J. de la Puente, and M. Ferrer. Optimizing Isotropic and Fully-anisotropic Elastic Modelling on Multi-GPU Platforms. In *75th EAGE Conference & Exhibition*. EAGE, 2013.

[9] E. Zhebel, S. Minisini, A. Kononov, and W. Mulder. Performance and scalability of finite-difference and finite-element wave-propagation modeling on Intel's Xeon Phi. In *SEG Technical Program Expanded Abstracts 2013*, pages 3386–3390, 2013.

[10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[11] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ACM SIGARCH Computer Architecture News*, volume 40, pages 440–451. IEEE Computer Society, 2012.

[12] L. DeRose, B. Homerl, and D. Johnson. Detecting application load imbalance on high end massively parallel systems. In *European Conf. on Parallel Processing*, pages 150–159. Springer, 2007.

[13] E. H. M. Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux. Hardware-Assisted Thread and Data Mapping in Hierarchical Multicore Architectures. *ACM Trans. Archit. Code Optim.*, 13(3):28:1–28:28, September 2016.

[14] Jonathan Corbet. Toward better NUMA scheduling, 2012.

[15] George Chrysos. Intel Xeon Phi X100 Family Coprocessor - the Architecture, 2012.

[16] Intel. OpenMP Thread Affinity Control, 2014.

[17] Shawn D. Casey. How to Determine the Effectiveness of Hyper-Threading Technology with an Application, 2011.