Strategies to Improve the Performance of a Geophysics Model for Different Manycore Systems

Matheus S. Serpa, Eduardo H. M. Cruz, Matthias Diener, Arthur M. Krause, Philippe O. A. Navaux
Informatics Institute, Federal University of Rio Grande do Sul (UFRGS), Brazil
{msserpa, ehmcruz, mdiener, amkrause, navaux}@inf.ufrgs.br

Albert Farrés, Claudia Rosas, Mauricio Hanzich
Barcelona Supercomputing Center (BSC), Spain
{albert.farres, claudia.rosas, mauricio.hanzich}@bsc.es

Jairo Panetta
Computer Science Division
Aeronautics Institute of Technology (ITA), Brazil
jairo.panetta@gmail.com

***Abstract*—Many software mechanisms for geophysics exploration in Oil & Gas industries are based on wave propagation simulation. To perform such simulations, state-of-art HPC architectures are employed, generating results faster and with more accuracy at each generation. The software must evolve to support the new features of each design to keep performance scaling. Furthermore, it is important to understand the impact of each change applied 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 five architectures: Intel Haswell, Intel Knights Corner, Intel Knights Landing, NVIDIA Kepler and NVIDIA Maxwell. We focus on improving the cache memory usage, vectorization, and locality in the memory hierarchy. We analyze the hardware impact of the optimizations, providing insights of how each strategy can improve the performance. The results show that NVIDIA Maxwell improves over Intel Haswell, Intel Knights Corner, Intel Knights Landing and NVIDIA Kepler performance by up to 17.9x.**

I. Introduction

Geophysics exploration remains fundamental to the modern world to keep up with the demand for 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 Partial Differential Equation (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 maintain 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 manycore 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, chip manufacturers have dedicated efforts to provide tens to hundreds of processing units working at low frequencies, such as Graphic Processing Units (GPUs), Intel Xeon Phi processors and coprocessors. Even more traditional multicore processors, such as the Xeon family, are including dozens of cores in the processors.

Several challenges must be addressed to better support these manycore systems and thereby achieve high performance. One of the most important aspects is the cache memory behavior, as the cache memory plays a key role in the performance. Likewise, the memory hierarchy, composed of several cache layers and memory controllers, has a significant impact in the execution. Xeon and Xeon Phi support vectorization, which allows several operations per instruction, and can boost the performance by several times. Furthermore, such manycore systems are heavily dependent on load balancing, due to the large number of cores. An application must address these challenges to take advantage of the new architectures.

In this paper, we optimize an acoustic wave propagator for two NVIDIA GPUs, Intel Xeon, Xeon Phi processor and coprocessor. We focus 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 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. The Laplace Operator is discretized by a 12^{th} order finite differences approximation on each spatial dimension. The derivatives are approximated by a 2^{nd} finite differences operator.

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

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

To the best of our knowledge, this work presents the first approximation to correlate the hardware impact of optimization performed on a largely used seismic imaging simulator running on new architectures, by applying these optimization techniques: (1) loop interchange to improve cache memory usage; (2) vectorization to increase the performance of floating point computations; (3) thread and data mapping to better use the memory hierarchy. (4) usage of shared and read-only GPU memories together to reduce the memory access latency. Experiments running in the Intel Knights Landing processor and NVIDIA Maxwell GPU have the best performance.

II. Related work

Recent architectures, including accelerators and coprocessors, proved to be well suited for geophysics, magnetohydrodynamics and flow simulations, outperforming the general purpose processors in efficiency. To obtain maximum performance from these new devices, some re-engineering of regions of the code, if not the entire application, is necessary. 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 resource 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], Andreolli et al. focused on acoustic wave propagation equations, choosing the optimization techniques from systematically tuning the algorithm. The usage of collaborative thread blocking, cache blocking, register re-use, vectorization and loop redistribution resulted in significant performance improvements. 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 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. They show that the best results are obtained from a combination of the first two and also that the performance with the Xeon Phi is close to the GPU. 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 anisotropy on general-purpose processors, 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 Xeon and Xeon Phi. On the Xeon, 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.

III. Manycore Systems that were Optimized

We used five environments to analyze the application performance. (1) 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. We refer to this system as *Haswell*. (2) We used a 57-core Intel Xeon Phi 3120P from the Knights Corner architecture. It supports 4-way SMT, where each core has a private L1 and L2 cache. We refer to this system as *KNC*. (3) We used a 68-core Intel Xeon Phi 7250 from the Knights Landing architecture. It supports a 4-way SMT, where each core has a private L1 and shared L2 cache. We refer to this system as *KNL*. (4) We used a NVIDIA Tesla K80 from the Kepler architecture. We refer to this system as *Kepler*. (5) We used a NVIDIA GeForce GTX TITAN X from the Maxwell architecture. We refer to this system as *Maxwell*. Table I summarizes the environments.

We measured execution time, cache misses and interchip interconnection traffic of the applications. To measure cache misses, we used the Intel PCM tool. To measure interchip traffic, we used Intel VTune. Each experiment was executed 30 times, and we show average values as well as a 95% confidence interval calculated with Student's t-distribution.

TABLE I: Configuration of the evaluation systems.

System	Parameter	Value
Haswell	Architecture	Haswell
	Processor	2 × Intel Xeon E5-2650 v3, 10 *cores*, 2-SMT
	Memory	10 × 32KB L1, 10 × 256KB L2, 25MB L3
		128GB DDR4-2133
KNC	Architecture	Knights Corner
	Coprocessor	Intel Xeon Phi 3120P, 57 *cores*, 4-SMT
	Memory	57 × 32KB L1, 57 × 512KB L2, 6GB RAM
KNL	Architecture	Knights Landing
	Processor	Intel Xeon Phi 7250, 68 *cores*, 4-SMT
	Memory	68 × 32KB L1, 68 × 512KB L2, 96GB DDR4
Kepler	Architecture	Kepler GK210
	GPU	NVIDIA Tesla K80, 2496
	Registers	13 × 512KB
	Memory	13 × 128KB L1 / *shared*, 1280KB L2
		13 × 48KB *texture (read-only)*, 12GB GDDR5
Maxwell	Architecture	Maxwell GM200
	GPU	GeForce GTX TITAN X, 3072
	Registers	24 × 512KB
	Memory	24 × 96KB *shared*, 3072KB L2
		24 × 48KB L1 / *texture (read-only)*, 12GB GDDR5

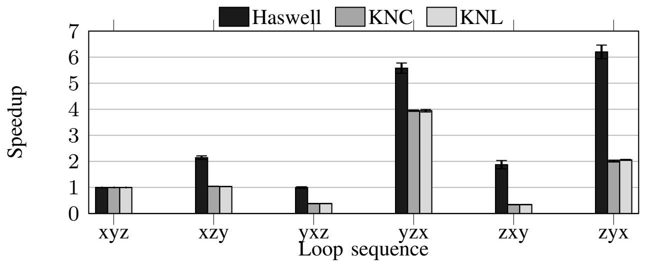


Fig. 1: Speedup over the xyz sequence.

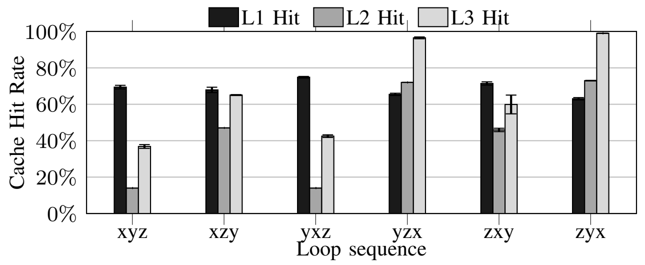


Fig. 2: Cache hit rate in Haswell.

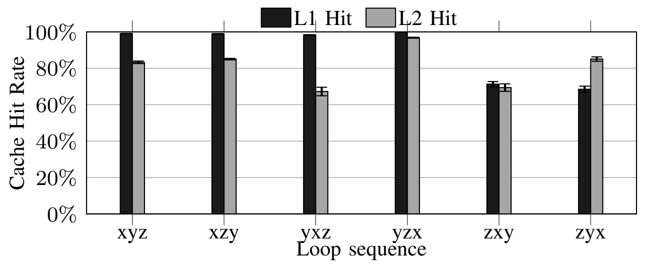


Fig. 3: Cache hit rate in KNC.

IV. Optimizing the Acoustic Wave Propagation Model in Manycore Systems

This sections presents the optimizations techniques we used to improve the performance in a real world application and the experiments 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 input stencil size was $1024 \times 256 \times 256$. We describe the optimizations and analyze how they address the challenges imposed by manycore systems. We also present the results obtained by each technique and the results of the optimizations, on the Intel's and NVIDIA's architectures.

A. Improving Cache Memory Usage

Current computer architectures provide caches and hardware prefetchers to help programmers manage data implicitly [10]. The 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

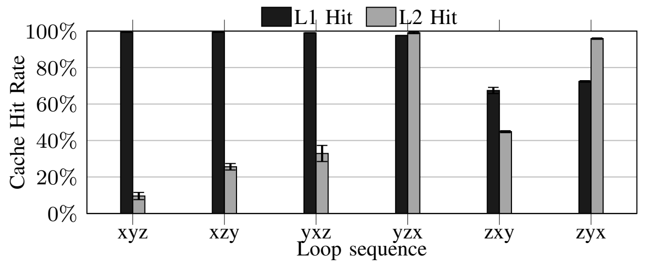


Fig. 4: Cache hit rate in KNL.

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 1, 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 sequence **zyx** has better results in Haswell. The speedup compared with the *xyz* version is $6.2\times$. This sequence is 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 2. 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%. In KNC and KNL, version **yzx** have better results. The speedups in these architectures are up to $2\times$ showing that this optimization impact less in the performance of Xeon Phi architectures than Haswell. The cache hit rates are showed in Figures 3 and 4. The L2 cache hit rates were improved from 83.3% and 9.6% to 85.1% and 95.9%. Although L1 cache hit rate decreases in both architectures, it shows that the best option aiming performance is to increase the last level cache (LLC) hit rates, even when the cache hit rate of any other level decreases.

The differences in cache misses happened because the data access stride becomes different when changing the loop sequence, influencing both spatial and temporal localities. Despite the reductions in the L1 hit rate, the increase of the LLC hit rates resulted in the highest performance improvement and is therefore the best choice for this application. The performance improvement in the KNC and KNL is lower than in Haswell because the amount of cache memory available per thread in the KNC and KNL is much lower.

B. Exploiting SIMD Vector Instructions

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 that perform, in one instruction,

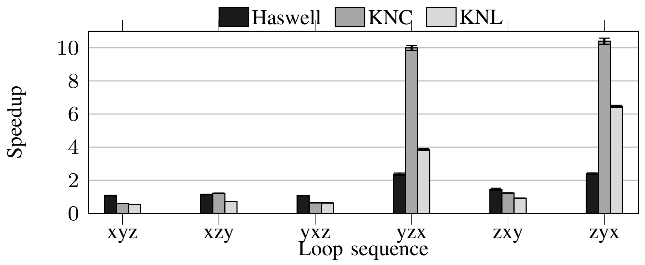


Fig. 5: Performance gain using vectorization.

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, wherever was possible, we modified the source code such that the memory addresses accessed by the same instruction were consecutive along loop iterations.

We used the Advanced Vector Extensions (AVX) instructions, 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 calculations 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 in Figure 5. The speedup shown is relative to the loop sequence without AVX. The sequences **yzx** and **zyx** have better results because they have more elements being accessed contiguously. The performance gain differs from architecture to architecture. In Haswell, the improvement was up to 2.38×. In the KNC and KNL architectures, the improvement was up to 10.4× and 6.5×, respectively. These differences are due to the size of each architecture's vector unit and the number of cores used.

C. 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 [12]. Other policies seek a uniform load distribution among the cores and memory controllers. In this work, we analyze five mapping strategies:

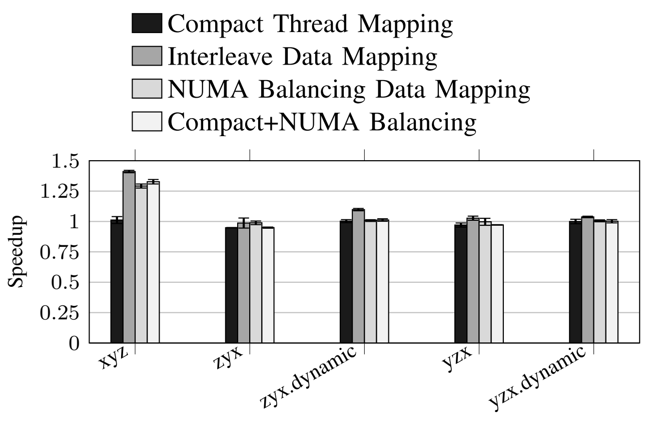
Baseline The default thread mapping of Linux, focused on load balancing, combined with a first-touch data mapping policy.

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

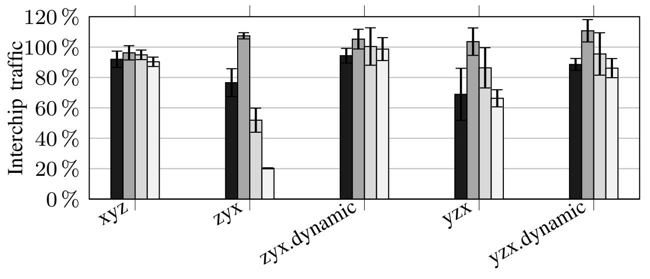
Interleave Data Mapping The default thread mapping of Linux combined with the interleave data mapping, which arranges consecutive pages to consecutive NUMA nodes.

NUMA Balancing Data Mapping The default thread mapping of Linux, combined with the NUMA Balancing data mapping [13], 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.



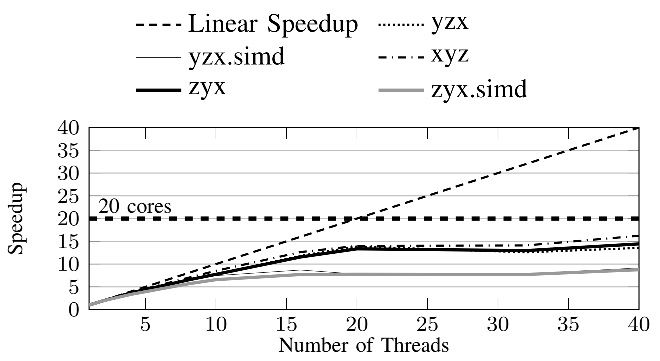
(a) Speedup from mapping.



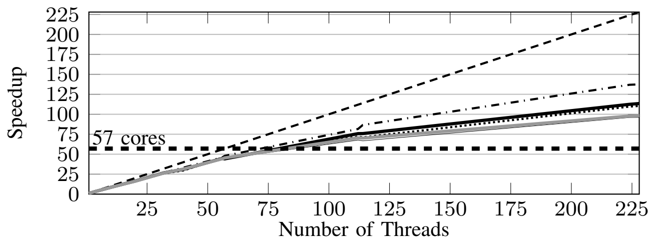
(b) Interchip interconnection traffic.

Fig. 6: Data and thread mapping results.

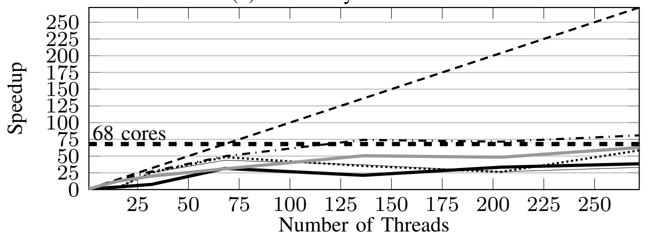
The results obtained from different mapping policies in the Xeon processor are shown in Figures 6a and 6b. We did not evaluate this on the Xeon Phi systems because they had a fixed mapping of memory addresses to memory controllers. The speedup and interchip interconnection traffic are normalized in relation 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 2, 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, but these benefits are mitigated if the cache hit rate is high. It can be observed that the **xyz**



(a) Scalability in Haswell.



(b) Scalability in KNC.



(c) Scalability in KNL.

Fig. 7: Scalability in different architectures.

variant, which benefited most from mapping, also had most cache misses. In the other configurations, since the L3 cache hit rate is very high, we have few accesses 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.

D. Scalability

Figures 7a, 7b and 7c show the speedup for different optimization algorithms in the Haswell, KNC and KNL architectures. The performance due to vectorization is better in KNC because its architecture provides wide SIMD vector units, 512 bits units as in KNL. Furthermore, the pipelines of KNL and Haswell are out-of-order, which are able to exploit a higher degree of instruction level parallelism (ILP) than the pipeline of KNC, which is in-order [14]. This higher ILP slightly mitigates the improvements from vectorization in Haswell and KNL compared to KNC.

The speedup decreases when the number of threads is greater than the number of physical cores (but lower than the number of virtual cores). The reason for this is because

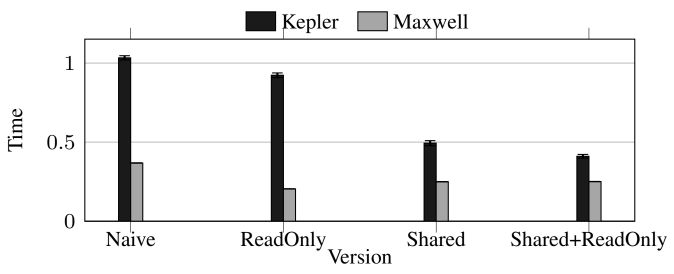


Fig. 8: Results obtained in Kepler and Maxwell architectures.

Haswell, KNC and KNL 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 [15]. In Haswell, the best speedup was 18.8× running 40 threads. In KNC, the best speedup was 137.5× running 228 threads. In KNL, the best speedup was 80.9× running 272 threads.

E. Shared and read-only memories to reduce global memory accesses

Kepler and Maxwell architectures cores share a 512 KB register file, composed of 128 K registers of 4 B. They also share access to three cache memories internal to the stream processor. The L1 memory cache is used to accesses the stack and register spill. This cache is organized in lines consisting of 128 B, generating accesses to the global L2 memory cache when lines are not found. The second cache memory is the shared cache memory, which stores data manually allocated by the programmer. The third cache is the read-only cache. Originally, it was used for textures, but in Kepler and Maxwell architectures, any data can be stored in this cache by using the intrinsic `lgd()`.

We developed four versions using different GPU memories aiming to understand the performance impact. Figure 8 shows the execution time running these different versions. The *naive* version uses only the memories that are automatically used by the compiler. The *ReadOnly* and *Shared* versions use the same memories as naive and use read-only and shared memory, respectively. The *Shared+ReadOnly* takes advantage of both read-only and shared memory. The performance improvement by using these memories was up to 2.5× in comparison with the naive version in the Kepler architecture and up to 1.8× in the Maxwell architecture. The Maxwell architecture was 2× faster than Kepler architecture.

F. Comparison between Haswell, KNC, KNL, Kepler and Maxwell

We optimize the wave propagation kernel for different manycore systems with very different architectural characteristics. Figure 9 shows the execution time of the best version for each architecture evaluated. In the Xeon Phi systems, the best version was using the *yzx* sequence. The Haswell architecture was better with the *zyx* sequence. In the Kepler architecture, the best version was *shared-readOnly*, which

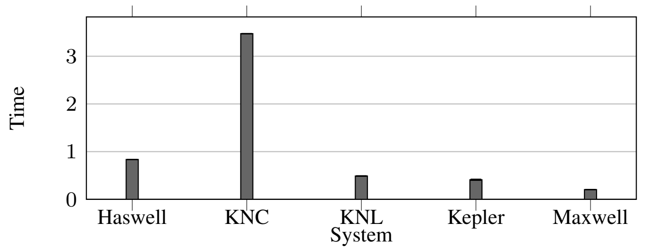


Fig. 9: Results obtained in the different architectures.

stores data in both shared and read-only memories. In the Maxwell architecture, the best version was *readOnly*, which stores data in the read-only memory.

Although KNC has hundreds of threads, its performance was the worst among the architectures. It has in-order cores, which limits the performance because it reduces the instruction level parallelism (ILP). The Haswell architecture has a high ILP degree, but its performance is limited by the low number of cores. The KNL architecture has both a high ILP degree and a large number of cores, resulting in low execution times. The Kepler architecture has a low execution time, but the Maxwell architecture is even better once its a newer GPU architecture. This architecture has thousand of cores that enables a high performance in this kind of application.

V. Conclusions and Future Work

Manycore systems introduce several challenges, such that parallel applications need to be coded properly to address them. In this paper, we applied and analyzed the performance of a set of optimization techniques on Intel Haswell, Intel Knights Corner (KNC), Intel Knights Landing (KNL), NVIDIA Kepler and NVIDIA Maxwell. 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 applied to other applications and architectures.

In our experiments, we show that loop interchange is a useful technique to improve performance of different cache memory levels, being able to improve the performance by up to 6.2×, 2.01× and 2.06× in Haswell, KNC and KNL, 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 2.39×, 10.4× and 6.5×. By modifying the scheduling, we were able to increase the performance by up to 2.19×, 1.29× and 1.04×, 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. Using these techniques, the speedup was up to 18.8×, 137.5× and 80.9× in Haswell, KNC and KNL, respectively. We also developed GPU versions aiming to understand which GPU architectures provide the best performance to the geophysics application. The results showed that using both shared and read-only memories the performance was improved by up to 2.5×. At the end, we compared the best version of each architecture and showed that NVIDIA Maxwell has the best execution time.

As future work, we will evaluate the energy consumption improvements of these optimizations. We also intend to evaluate newer architectures, such as the Coffee Lake and Pascal architectures.

Acknowledgment

This research received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E project, grant n.º 689772. It was also supported by Intel under the Modern Code Project, and Petrobras.

References

[1] R. G. Clapp, "Seismic Processing and the Computer Revolution(s)," in *SEG Technical Program Expanded Abstracts 2015*, 2015.

[2] R. G. Clapp, H. Fu, and O. Lindtjorn, "Selecting the right hardware for reverse time migration," *The Leading Edge*, vol. 29, no. 1, 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*, ser. WOLFHPC '16. 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*, vol. 7, no. 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 *High Performance Parallelism Pearls*, J. Reinders and J. Jeffers, Eds. Boston: Morgan Kaufmann, 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*, vol. 54, 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*, 2013.

[10] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, Jun. 2010.

[11] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE, 2012.

[12] 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.*, vol. 13, no. 3, Sep. 2016.

[13] J. Corbet, "Toward better NUMA scheduling," 2012. [Online]. Available: http://lwn.net/Articles/486858/

[14] G. Chrysos, "Intel Xeon Phi X100 Family Coprocessor - the Architecture," 2012. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[15] S. D. Casey, "How to Determine the Effectiveness of Hyper-Threading Technology with an Application," 2011. [Online]. Available: https://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-\/an-application/