UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MATHEUS DA SILVA SERPA

Source Code Optimizations to Reduce Multi-core and Many-core Performance Bottlenecks

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Philippe O. A. Navaux

Porto Alegre
July 2018

CIP — CATALOGING-IN-PUBLICATION

Serpa, Matheus da Silva

Source Code Optimizations to Reduce Multi-core and Many-core
Performance Bottlenecks / Matheus da Silva Serpa. – Porto Alegre: PPGC da UFRGS, 2018.

74 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: Philippe O. A. Navaux.

1. Performance evaluation. 2. Source code optimizations. 3. Many-core. 4. HPC. I. Navaux, Philippe O. A.. II. Título.

“Success is the ability to move
from one failure to another
without loss of enthusiasm.”
— SIR WINSTON CHURCHILL

AGRADECIMENTOS

Em primeiro lugar, gostaria de agradecer ao meu orientador Prof. Philippe Olivier Alexandre Navaux pelo apoio durante os últimos dois anos. Sou muito grato por todos ensinamentos e oportunidades que contribuiram para o meu crescimento profissional e pessoal. Obrigado pela confiança no Mestrado e agora no Doutorado.

Ao Prof. Claudio Schepke, meu orientador no curso de Bacharelado em Ciência da Computação. Obrigado por apostar em mim quando ainda estava no terceiro semestre do curso. Também agradeço pelos conselhos pessoais e profissionais, pelas várias viagens juntos à conferências e pela amizade mantida até hoje.

À Profa. Márcia Cristina Cera (*in memoriam*), que foi muito importante para minha formação. Obrigado por todos ensinamentos. Obrigado por me fazer um profissional melhor. Lembro, como se fosse hoje, a última vez que tive o prazer de ouvir seus conselhos durante o WSCAD 2016 em Aracaju.

Aos Professores Marcelo Pasin e Pascal Felber, pela oportunidade e pelas contribuições durante os três meses em que fui pesquisador visitante na Université de Neuchâtel. Também à senhora Pepita e ao senhor Georges que me receberam em sua casa em Neuchâtel. E, finalmentente, aos amigos que fiz na Suíça: Rafael, Sébastien, Dorian, Veronica e Amanda.

Ao corpo docente e quadro de funcionários do Instituto de Informática da UFRGS, em especial: Prof. Carissimi, Prof. Lucas, Prof. Antonio Beck, Prof. Luigi, Claudia e Rafael. Também gostaria de agradecer as empresas HPE, Intel, Petrobras e agências financiadoras CAPES e CNPq, pelo apoio financeiro.

Aos colegas do Grupo de Processamento Paralelo e Distribuído (GPPD). Em especial: Eduardo Cruz, Emmanuell, Francis, Eduardo Roloff, Matthias, Edson Padoin, Pablo, Jéssica, Víctor, Vinicius, Lucas, Jean e Otávio. Estendo aos amigos do LSE-UFRGS.

À minha namorada, Luana, por todo apoio, amor, carinho e dedicação. Aos amigos da "Sala 303", "EAD Paizinho", "Guarita", "GAMA", "GESEP", "Engenharias" e ERADs. Agradeço de forma especial as pessoas mais próximas a mim em Porto Alegre: Arthur, Gabriel, Thaíse e Uillian. Também agradeço ao Dalvan e ao Prof. João.

Por fim, o mais importante, meu Pai Airton e minha Mãe Maria. Obrigado por todo apoio, ensinamentos e confiança. Dedico este trabalho a vocês. Peço desculpas por não estar presente em vários momentos nos últimos seis anos.

Aos familiares e amigos aqui não mencionados, obrigado por tudo.

ABSTRACT

Nowadays, there are several different architectures available not only for the industry but also for final consumers. Traditional multi-core processors, GPUs, accelerators such as the Xeon Phi, or even energy efficiency-driven processors such as the ARM family, present very different architectural characteristics. This wide range of characteristics presents a challenge for the developers of applications. Developers must deal with different instruction sets, memory hierarchies, or even different programming paradigms when programming for these architectures. To optimize an application, it is important to have a deep understanding of how it behaves on different architectures. Related work proved to have a wide variety of solutions. Most of then focused on improving only memory performance. Others focus on load balancing, vectorization, and thread and data mapping, but perform them separately, losing optimization opportunities.

In this master thesis, we propose several optimization techniques to improve the performance of a real-world seismic exploration application provided by Petrobras, a multinational corporation in the petroleum industry. In our experiments, we show that loop interchange is a useful technique to improve the performance of different cache memory levels, improving the performance by up to 5.3× and 3.9× on the Intel Broadwell and Intel Knights Landing architectures, respectively. By changing the code to enable vectorization, performance was increased by up to 1.4× and 6.5×. Load Balancing improved the performance by up to 1.1× on Knights Landing. Thread and data mapping techniques were also evaluated, with a performance improvement of up to 1.6× and 4.4×. We also compared the best version of each architecture and showed that we were able to improve the performance of Broadwell by 22.7× and Knights Landing by 56.7× compared to a naive version, but, in the end, Broadwell was 1.2× faster than Knights Landing.

Keywords: Performance evaluation. source code optimizations. many-core. HPC.

Otimizações de Código Fonte para Reduzir Gargalos de Desempenho em Multi-core e Many-core

RESUMO

Atualmente, existe uma variedade de arquiteturas disponíveis não apenas para a indústria, mas também para consumidores finais. Processadores multi-core tradicionais, GPUs, aceleradores, como o Xeon Phi, ou até mesmo processadores orientados para eficiência energética, como a família ARM, apresentam características arquiteturais muito diferentes. Essa ampla gama de características representa um desafio para os desenvolvedores de aplicações. Os desenvolvedores devem lidar com diferentes conjuntos de instruções, hierarquias de memória, ou até mesmo diferentes paradigmas de programação ao programar para essas arquiteturas. Para otimizar uma aplicação, é importante ter uma compreensão profunda de como ela se comporta em diferentes arquiteturas. Os trabalhos relacionados provaram ter uma ampla variedade de soluções. A maioria deles se concentrou em melhorar apenas o desempenho da memória. Outros se concentram no balanceamento de carga, na vetorização e no mapeamento de *threads* e dados, mas os realizam separadamente, perdendo oportunidades de otimização.

Nesta dissertação de mestrado, foram propostas várias técnicas de otimização para melhorar o desempenho de uma aplicação de exploração sísmica real fornecida pela Petrobras, uma empresa multinacional do setor de petróleo. Os experimentos mostram que *loop interchange* é uma técnica útil para melhorar o desempenho de diferentes níveis de memória *cache*, melhorando o desempenho em até 5,3× e 3,9× nas arquiteturas Intel Broadwell e Intel Knights Landing, respectivamente. Ao alterar o código para ativar a vetorização, o desempenho foi aumentado em até 1,4× e 6,5×. O balanceamento de carga melhorou o desempenho em até 1,1× no Knights Landing. Técnicas de mapeamento de *threads* e dados também foram avaliadas, com uma melhora de desempenho de até 1,6× e 4,4×. O ganho de desempenho do Broadwell foi de 22,7× e do Knights Landing de 56,7× em comparação com uma versão sem otimizações, mas, no final, o Broadwell foi 1,2× mais rápido que o Knights Landing.

Palavras-chave: avaliação de desempenho, otimizações de código fonte, many-core, HPC.

LIST OF ABBREVIATIONS AND ACRONYMS

2D	Two-Dimensional
3D	Three-Dimensional
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
API	Application Programming Interface
AVI	Audio Video Interleave
AVX	Advanced Vector Extensions
B+	B+ tree
BFS	Breadth-First Search
BP	Back Propagation
CFD	CFD Solver
CPU	Central Processing Unit
DDR	Double Data Rate
GB	Gigabyte
GPU	Graphics Processing Unit
HPC	High-Performance Computing
HS2D	HotSpot 2D
HS3D	HotSpot 3D
HW	Heart Wall Tracking
I/O	Input/output
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
KM	K-means
KNC	Knights Corner

KNL	Knights Landing
KNN	k-nearest neighbors
LC	Leukocyte Tracking
LLC	Last Level Cache
LRU	Least Recently Used
LUD	LU Decomposition
MCDRAM	Multi-Channel DRAM
MD	LavaMD
MIC	Many Integrated Core
MIMD	Multiple Instruction Multiple Data
MPPA	Massively Parallel Processor Array
MS	Myocyte Simulation
NUMA	Non-Uniform Memory Access
NW	Needleman-Wunsch
ODE	Ordinary Differential Equation
OpenMP	Open Multi-Processing
PATH	Pathfinder
PCM	Performance Counter Monitor
PDE	Partial Differential Equation
PF	Particle Filter
QPI	QuickPath Interconnect
RTM	Reverse Time Migration
SC	Stream Cluster
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multithreading
SRAD	Speckle Reducing Anisotropic Diffusion

SSE	Streaming SIMD Extensions
TLB	Translation Lookaside Buffer)
TLP	Thread-Level Parallelism

LIST OF SYMBOLS

$p(x, y, z, t)$	Acoustic Pressure
∇^2	Laplace Operator
$\rho(x, y, z)$	Media Density
∂	Partial Derivative
$V(x, y, z)$	Propagation Speed

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

High-performance computing (HPC) has been responsible for a scientific revolution. The evolution of computer architectures improved the computational power, increasing the range of problems and quality of solutions that could be solved in the required time, e.g., weather forecast. The introduction of integrated circuits, pipelines, increased frequency of operation, out-of-order execution and branch prediction are an important part of the technologies introduced up to the end of the 20th century. Recently, the concern about energy consumption has grown, with the goal of achieving computation at the Exascale level in a sustainable way (HSU, 2016). The technologies so far developed do not enable Exascale computing, due to the high energy cost of increasing the frequency and pipeline stages, as well as the fact that we are at the limits of exploration the instruction-level parallelism (ILP) (BORKAR; CHIEN, 2011; COTEUS et al., 2011).

In order to increase computational power, the industry has shifted its focus to parallel and heterogeneous architectures in recent years. The main feature of parallel architectures is the presence of several processing cores operating concurrently. To use such an architecture, an application must be programmed by separating it into several tasks that communicate with each other. Heterogeneous architectures, on the other hand, have different environments in the same system, each one with its specialized architecture for task type. The usage of accelerators is one of the main forms of heterogeneous architectures, in which a generic processor is mostly responsible for managing the system, and several accelerators present in the system perform the computation of specific tasks to which they are tuned and expected to perform well.

Several challenges must be addressed to support these architectures better and thereby achieve high performance (MITTAL; VETTER, 2015). Applications need to be coded considering the particularities of each environment, as well as considering their distinct architectural characteristics (GROPP; SNIR, 2013). For example, in the memory hierarchy, the presence of several cache memory levels, some shared and others private introduces non-uniform access times, which impact applications' performance (CRUZ et al., 2016a). It is even more critical in heterogeneous architectures since each accelerator can have its own, distinct, memory hierarchy. Also, in heterogeneous architectures, the number of functional units may vary between different accelerators, and the instruction set itself may not be the same. In this context, it is essential to analyze the behavior of architectures, in order to provide better support for optimizing applications performance.

1.1 Contributions of this research

The main objective of our research is to evaluate multi-core and many-core architectures and reduce the performance bottlenecks using source code optimization techniques. Considering these goals, our main contributions are the following:

- We analyze a set of performance metrics on several applications with distinct parallel execution characteristics aiming to find a correlation between the metric and the application performance (IPC).
- We addressed a set of performance optimization strategies, aiming to increase the performance of a real-world seismic exploration application. The techniques employed were: loop interchange to improve cache memory usage; vectorization to increase the performance of floating point computations; loop scheduling and collapse to improve load balancing; and thread and data mapping to better use the memory hierarchy.

1.2 Document Organization

The document is organized as follows. Chapter 2 presents a background on the topics of this dissertation and discusses related work in performance optimization. Chapter 3 presents the results of our evaluation of the performance bottlenecks. In chapter 4, we addressed some performance optimization strategies and optimized a real-world application. Finally, chapter 5 draws conclusions based on our findings and presents some future work insights.

2 MULTI-CORE AND MANY-CORE: OVERVIEW AND RELATED WORK

The following sections explain some concepts that serve as a base for this dissertation. Two state-of-art architectures for High-Performance Computing is presented. Furthermore, this chapter also details related work on performance optimization for these architectures.

2.1 Multi-core and Many-core Architectures

Technological innovations brought up powerful single-core processors with high clock frequencies. However, because of technological and power limitations, multicore and many-core processors emerged as new computer architectures (KIRK; WEN-MEI, 2016). These architectures rely on both instruction-level parallelism (ILP) and thread-level parallelism (TLP) to achieve high performance. Today's multicore and many-core processors differ in the number of cores, the memory hierarchy, and their interconnection.

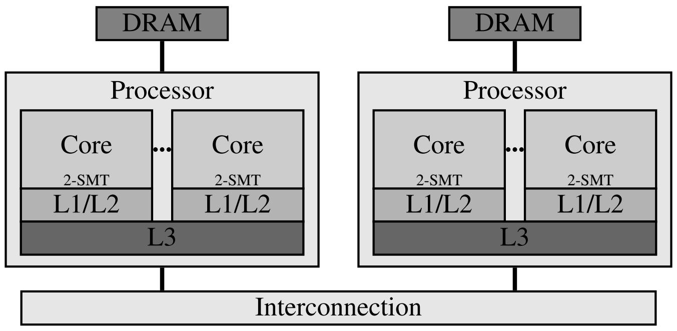
The design of multicore and many-core architectures is different to the point that depending on the application, the performance can be high in one architecture and low in the other (COOK, 2012). The multicore architecture uses sophisticated control logic to allow single-threaded statements to run in parallel. Large cache memories are provided to reduce access latencies to instructions and application data. Finally, the operations of the arithmetic logic units (ALUs) are also designed to optimize latency.

Many-core architecture takes advantage of a large number of execution threads. Small cache memories are provided to prevent multiple threads accessing the same data from having to go to main memory. Besides, most of the chip is dedicated to floating-point units. Such architectures are designed as floating-point computing mechanisms and not for conventional operations, which are better executed by multicore architectures.

2.1.1 The Broadwell Architecture

The Broadwell architecture (NALAMALPU et al., 2015) is a non-uniform memory access architecture. In such architectures, the latency in the access to the main memory will change, as it depends on which memory bank is being accessed (CRUZ et al., 2016a). Each processor in the system contains a memory controller, composing one

Figure 2.1: Example of the memory subsystem in the Broadwell architecture. In this figure there are 2 processors, each constituting a NUMA node. Each processor is composed of 22 physical processing cores, each executing 2 logical threads. There are 3 levels of cache memory.



Source: The Author.

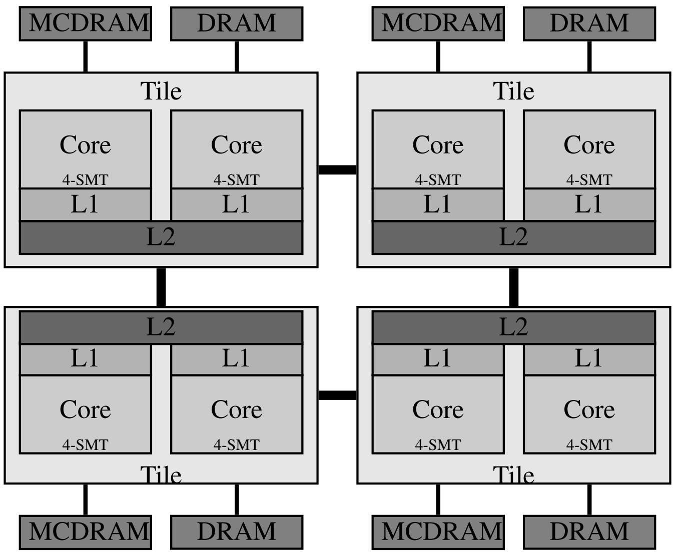
NUMA node. Each physical core executes 2 logical threads by employing simultaneous multi-threading (SMT). The connection between different processors is made through Intel's *QuickPath Interconnect* (QPI) (ZIAKAS et al., 2010). The memory hierarchy of this system is illustrated in Figure 2.1.

To exemplify how the memory hierarchy can impact a memory access latency, Figure 2.1 shows an example of a system where there are different possibilities for accesses to memory. Threads might access memory by obtaining their data from the private L1 or L2 caches in each core, obtaining high-speed access. A significant change in the Broadwell architecture, when compared to the previous Intel processors, is the duplication of available bandwidth in the private cache memories. The L1 cache memory has 2 load ports and 1 store port, thus it supports 2 concurrent load requests given no bank conflict between the accesses. Threads might also access the L3 cache memory, which is shared between all cores, taking 3 to 4 times longer than accessing the L2 cache, and up to 10 times longer than accessing the L1 cache. If the requested data is not found in the local caches, the request is snooped by remote caches in the other NUMA nodes, which might have the data. If the data is still not found, then the main memory of the NUMA node responsible for the address is accessed.

2.1.2 The Knights Landing Architecture

The Knights Landing (KNL) architecture (SODANI et al., 2016) is shown in Figure 2.2. The architecture is organized in tiles and has a distributed tag directory, and has a mesh interconnection. Each tile contains two cores, with private L1 caches, a shared

Figure 2.2: Example of the memory subsystem in the Knights Landing architecture. In this figure there are 4 tiles, each with 2 cores. The processor is composed of 57 physical processing cores, each executing 4 logical threads. There are 2 levels of cache memory.



Source: The Author.

L2 cache and a tag directory (omitted in the figure). The architecture, besides memory controllers to access external DDR4 memory, includes an MCDRAM memory, which can work as a cache to the DDR4 memory (cache mode). We used this mode in the experiments, or as a separate memory in the same address space (flat mode). The cores in KNL implement an out-of-order pipeline and can execute 4 threads in parallel using simultaneous multithreading (SMT).

In KNL, in case of a cache miss, the corresponding tag directory is checked, and the data is forwarded from another cache if present there. If no cache of any tile has the data, two behaviors are possible. First, in cache mode, the MCDRAM works as an L3 cache, such that the processor first checks if the data is cached in it, and if not, the memory controller is accessed to fetch the data from the DDR4 memory. Second, in flat mode, the memory address determines if the data is stored in the MCDRAM or the DDR4 memory. The distribution of memory addresses between the tag directories and memory controllers can also be configured, which is called clustering mode. The two main configurations for clustering modes are: quadrant, where the tiles are split to 4 quadrants, and the addresses are divided between the quadrants by the hardware; and sub-NUMA clustering, where each memory controller and MCDRAM form a NUMA node, and the operating system is responsible for selecting the node that stores each page.

2.2 Related Work

In this section, the most representative works that evaluate and optimize applications performance are discussed. They are listed by property in chronological order.

2.2.1 Memory Optimization

Falch and Elster (FALCH; ELSTER, 2014) proposed the usage of a manually managed cache to combine the memory from multiple threads. Using their technique, they achieved a speedup of up to 2.04 in a synthetic stencil. They concluded that manual caching is an effective approach to improve memory access and that applications with regular access patterns are suitable to implement their technique.

A mechanism is proposed by Jia, Shaw and Martonosi (JIA; SHAW; MARTONOSI, 2014) to balance memory accesses. The authors' motivation is that the design of cache memories used in GPU architectures are the same as those designed for CPU architectures, which is unfit for their operation. The massive use of threads through block-threading in warps means that normal caches provide few bytes per thread, and when all threads need the cache, the data is thrown away without being reused (thrashing). When threads from multiple warps share the cache, there is contention for the request queue itself, which cannot timely serve requests from such a large number of threads. The authors propose two solutions: queue request reordering and cache bypassing. Through queues that use block identifiers, it is possible to separate the requests and prioritize all requests of 1 block, to make use of the spatial and temporal localities of this block. To avoid starving and contention, additional policies were developed to balance use cases, such as prioritizing a full queue that receives a new request. Because caches are also not designed for so many threads, some warps have all their accesses redirected directly to main memory, effectively avoiding cache accesses. It improves the access of all the requests to the cache memory, since the waiting delays in the queue of the accesses that go to the cache are reduced, and the accesses that do not have access to the cache because they do not have priority would probably be missed in the cache. By avoiding waiting and useless accesses, requests are serviced faster by being forwarded directly to main memory.

Maruyama and Aoky (MARUYAMA; AOKI, 2014) present a method for stencil computations on the NVIDIA Kepler architecture that uses shared memory for better data locality combined with warp specialization for higher instruction throughput. Their method achieves approximately 80% of the value from roofline model estimation.

In Ausavarungnirun et al. (AUSAVARUNGNIRUN et al., 2015), the authors proposed a mechanism to balance the memory accesses. The main observation of the article is that a warp with several hits in the L2 cache, which also generates misses in the cache L2, has as bottleneck these same misses, despite all the hits, which become unutilized. Hits on warps with lots of misses are also useless since the worst access time always defines the execution of warp. Through changes in the memory subsystem, prioritizing warps accesses with most hits and redirecting warps accesses with most misses directly to the main memory, the technique can improve application performance, on average, by 21%. The work exploits an inherent architectural feature of the stream processor model, mitigating the problem of divergence of memory accesses in individual warps. The research considered in this article deals with issues related to the pressure of several threads of scalable benchmarks at a system level, although they are related problems when considering the level of the stream processor.

Sao et al. (SAO et al., 2015) presented a sparse direct solver for distributed memory subsystems comprising hybrid multi-core CPU and Intel Xeon Phi coprocessors, which combines the use of asynchrony with the accelerated offload, lazy updates, and data shadowing to hide and reduce communication costs.

Heinecke et al. (HEINECKE et al., 2016) optimized seismic simulations in the Knights Landing architecture, exploiting its two-level memory subsystem and 2D mesh interconnect.

Nasciutti and Panetta (NASCIUTTI; PANETTA, 2016) did a performance analysis of 3D stencils on GPUs focusing on the proper use of the memory hierarchy. They conclude that the preferred code is the combination of read-only cache reuse, inserting the Z loop into the kernel and register reuse.

2.2.2 Vectorization

Wang et al. (WANG et al., 2016) introduced a fast tridiagonal algorithm for the Intel MIC architecture, achieving the best utilization of vectorization and registers.

Hasib et al. (HASIB et al., 2017) investigate the effects on performance and energy from a data reuse methodology combined with parallelization and vectorization in multi-core and many-core processors. They achieve a speedup of 17× using AVX2 and 35× using AVX512.

2.2.3 Load Balancing

The work of Lastovetsky, Szustak and Wyrzykowski (LASTOVETSKY; SZUSTAK; WYRZYKOWSKI, 2017) have used load imbalance (different work partition sizes) as a way to improve performance in a KNC platform. As their application of interest, MPDATA, sometimes shows a decrease in execution time for larger domain (data) partitions, they use a self-adaptable implementation to benchmark some balanced (same size of subdomain among groups of threads) and imbalanced (half of the threads have a slightly larger subdomain) partitions, and then choose the best partition among the measurements, achieving performance improvements of 15% over the balanced distribution. Nevertheless, this work distribution algorithm is limited to well-behaved, iterative, static applications.

2.2.4 Thread and Data Mapping

Tousimojarad and Vanderbauwhede (TOUSIMOJARAD; VANDERBAUWHEDE, 2014) show that the default thread mapping of Linux is inefficient when the number of threads is as large as on a many-core processor and presents a new thread mapping policy that uses the amount of time that each core does useful work to find the best target core for each thread.

Liu et al. (LIU et al., 2015) propose an approach based on profiling to determine thread-to-core mapping on the Knights Corner architecture that depends on the location of the distributed tag directory, achieving significant reductions on communication latency.

On Cruz et al. (CRUZ et al., 2016b), a method that uses the time that an entry stays in the TLB and the threads that access that page as a metric to perform thread and data mappings better is presented, achieving great performance improvements with low overhead.

Diener et al. (DIENER et al., 2016b) proposes *kmaf*, a kernel-based framework that uses page faults of parallel applications to profile their memory access pattern and improve the thread and data mapping online (DIENER et al., 2016b).

He, Chen and Tang (HE; CHEN; TANG, 2016) introduces NestedMP, an extension to OpenMP that allows the programmer to give information about the structure of the tasks tree to the runtime, which then performs a locality-aware thread mapping.

Cruz et al. (CRUZ et al., 2018) improve state of the art by performing a very detailed analysis of the impact of thread mapping on communication and load balancing in two many-core systems from Intel, namely Knights Corner and Knights Landing. They observed that the widely used metric of CPU time provides very inaccurate information for load balancing. They also evaluated the usage of thread mapping based on the communication and load information of the applications to improve the performance of many-core systems.

Serpa et al. (SERPA et al., 2018) focus on Intel's multi-core Xeon and many-core accelerator Xeon Phi Knights Landing, which can host several hundreds of threads on the same CPU. They study the impact of mapping strategies, revealing that, with smart mapping policies, one can indeed significantly speed up machine learning applications on many-core architectures. Execution time was reduced by up to 25.2% and 18.5% on Intel Xeon and Xeon Phi Knights Landing, respectively.

2.2.5 Combined Different Properties

A memory model to analyze algorithms for many-core systems is presented in Ma, Agrawal and Chamberlain (MA; AGRAWAL; CHAMBERLAIN, 2014). The model considers several architectural parameters, such as the latency for accessing the memory, the number of cores, the number of words that can be read from memory, the size of cache memory and the number of threads per core. It also considers the total number of operations that the running application must perform, as well as the total number of memory operations, cache memory usage and the number of threads. The authors conclude that applications with similar characteristics can have different performance using different architectural parameters.

In Andreolli et al. (ANDREOLLI et al., 2015), the authors focused on acoustic wave propagation equations, choosing the optimization techniques from systematically tuning the algorithm. The usage of collaborative thread blocking, cache blocking, register reuse, vectorization and loop redistribution resulted in significant performance improvements.

Mei and Chu (MEI; CHU, 2015) analyzed the characteristics of the memory subsystem in 3 different GPU architectures: Fermi, Kepler, and Maxwell. They used a pointer-chasing benchmark and observed the memory access latencies to define the characteristics of all memories within each GPU. In doing so, they were able to identify interesting features, such as a line replacement policy different from the expected Least

Recently Used (LRU) in the L2 cache memory. The authors conclude that the Kepler architecture planning was aggressive in its memory bandwidth, which has often been underused, and that, in the Maxwell architecture, more resources were invested in shared memory, generating a more efficient and balanced system.

Slota, Rajamanickam and Madduri (SLOTA; RAJAMANICKAM; MADDURI, 2015) presented a methodology for graph algorithm design on many-core architectures, such as NVIDIA and AMD GPUs and the Intel Xeon Phi MIC coprocessor, considering thread synchronization and access to global and shared memory, as well as load balancing.

Research efforts such as the presented in Castro et al. (CASTRO et al., 2016) 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.

Serpa et al. (SERPA et al., 2017) propose several optimization strategies for a wave propagation model for six architectures: Intel Broadwell, Intel Haswell, Intel Knights Landing, Intel Knights Corner, NVIDIA Pascal and NVIDIA Kepler. The results show that current GPU NVIDIA Pascal improves over Intel Broadwell, Intel Haswell, Intel Knights Landing, Intel Knights Corner, and NVIDIA Kepler performance by up to 8.5$\times$.

Deng et al. (DENG et al., 2018) analysis the performance difference between Sandy Bridge, MIC, and Kepler. They also proposed some memory optimizations that improve the performance of an ADI solver by up to 5.5 on a Kepler GPU in contrast to two Sandy Bridge CPUs.

2.3 Summary

In this chapter, we analyzed the related work on memory optimization, vectorization, load balancing and mapping. The related work proved to have a wide variety of solutions, with very different characteristics, in which we summarize in Table 2.1.

With the analysis of the related work, we can conclude that deep knowledge of the application behavior at the architectural level allows developing techniques to gain performance. Our work goes beyond analysis and looks for a greater understanding of the performance of different applications on different multi-core and many-core systems. We decide for using multi-core and many-core architectures because their programming

Table 2.1: Summary of related work. Each line represents a related work. Each column represents a desired property.

	Multi-core	Many-core	Memory	Vectorization	Load Balancing	Mapping
Falch e Elster (2014)	✓		✓			
Jia, Shaw e Martonosi (2014)			✓			
Ma, Agrawal e Chamberlain (2014)		✓	✓			
Maruyama e Aoki (2014)			✓			
Tousimojarad e Vanderbauwhede (2014)						✓
Andreolli et al. (2015)	✓	✓		✓	✓	
Ausavarungnirun et al. (2015)			✓			
Liu et al. (2015)		✓				✓
Mei e Chu (2015)			✓			
Sao et al. (2015)	✓	✓	✓			
Slota, Rajamanickam e Madduri (2015)		✓	✓		✓	
Castro et al. (2016)		✓	✓			✓
Cruz et al. (2016b)	✓					✓
Diener et al. (2016b)	✓					✓
He, Chen e Tang (2016)	✓					✓
Heinecke et al. (2016)		✓	✓			
Nasciutti e Panetta (2016)			✓			
Wang et al. (2016)		✓		✓		
Hasib et al. (2017)	✓	✓		✓		
Lastovetsky, Szustak e Wyrzykowski (2017)		✓			✓	
Serpa et al. (2017)	✓	✓	✓	✓		✓
Cruz et al. (2018)		✓				✓
Deng et al. (2018)	✓	✓	✓			
Serpa et al. (2018)	✓	✓				✓
This Thesis	✓	✓	✓	✓	✓	✓

Source: The Author.

style is similar, different from a GPU architecture that has a very different and unusual programming style.

The related work also shows that most of the work is focused on memory optimizations, but several works aimed at vectorization, load balancing and mapping. In this way, we decide to perform the four optimizations together on a real-world application. The next chapters describe our proposals in detail.

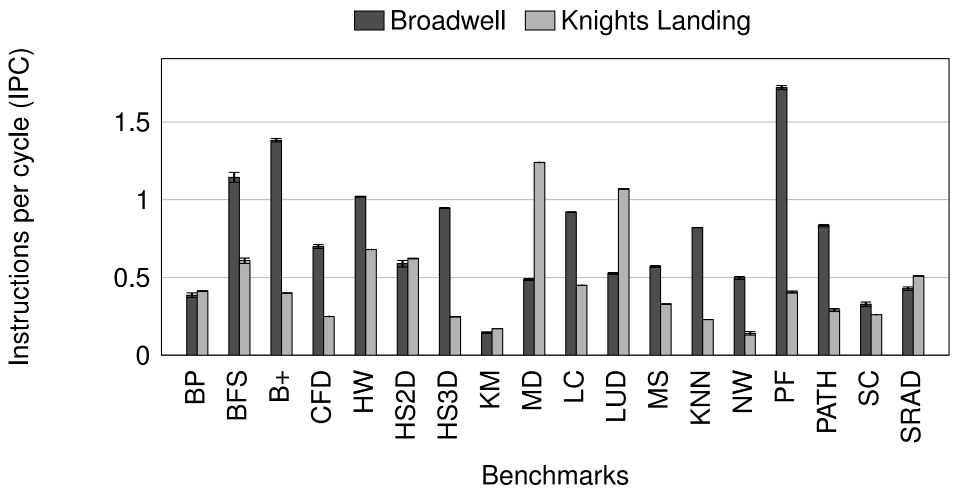
3 ANALYSIS OF PERFORMANCE BOTTLENECKS

Nowadays, there are several different architectures available not only for the industry but also for final consumers. Traditional multi-core processors and many-core accelerators such as the Xeon Phi present very different architectural characteristics. This wide range of characteristics present a challenge for the developers of applications because the same application can perform well when executing on one architecture, but poorly on another architecture.

To better explain our motivation, we show how parallel applications perform on these architectures. Figure 3.1 shows the IPC (instructions per cycle) metric, which indicates the average number of instructions executed per cycle, for 18 benchmarks from the Rodinia suite (CHE et al., 2010). As expected, the performance of each application depends on the architecture. There are three groups of applications: better on Broadwell; better on Knights Landing; and almost the same performance in both architectures.

It motivates the study of applications and architectural characteristics, aimed to understand what limit applications performance and how to improve that. We used hardware performance counters to gather accurate measurements of the actual impact of different factors that influence the performance. By doing so, we arrive the conclusion that some metrics help in understand applications performance, but there are cases where a metric alone is not representative. Such study served as a basis for the next chapter where we optimize a real-world seismic exploration application.

Figure 3.1: Performance of different architectures (higher IPC is better).



Source: The Author.

Table 3.1: Execution Environment.

System	Parameter	Value
Broadwell	Processor	2 × Intel Xeon E5-2699 v4, 2 x 22 cores, 2-SMT cores
multi-core	Threads	88
	Microarchitecture	Broadwell-EP
	Caches/processor	22 × 32 KByte L1, 22 × 256 KByte L2, 55 MByte L3
	Memory	256 GByte DDR4-2400
	Environment	Linux 4.4, Intel Compiler 18.0.1
Knights Landing	Processor	Intel Xeon Phi 7250, 68 cores, 4-SMT cores
many-core	Threads	272
	Microarchitecture	Knights Landing
	Caches	68 × 32 KByte L1, 68 × 512 KByte L2
	Memory	96 GByte DDR4, 16 GByte MCDRAM
	Environment	Linux 4.4, Intel Compiler 17.0.4

Source: The Author.

The remainder of this Chapter is organized as follows. First, we discuss the methodology, introducing the architectures used to perform the experiments of this dissertation and the benchmark suite applications. Then, the results on Broadwell and Knights Landing are presented. Finally, we discuss and summarize the conclusions of this study.

3.1 Methodology

The experiments were performed in the Broadwell and Knights Landing system environments. The Broadwell system is composed of two Intel Xeon E5-2699 v4 processors, where each processor consists of 22 physical cores, allowing execution of 44 threads with 2-SMT Hyper-Threading. The Knights Landing system is an Intel Xeon Phi 7250 processor with 68 physical cores and 272 threads with 4-SMT Hyper-Threading. It also has a 16 GB MCDRAM memory which is almost four times faster than the DRAM. Table 3.1 exhibits the details of each system.

The experiments shown in the next sections present the average of 30 random executions. The standard deviation presented is given by the t-Student distribution with a 95% confidence interval. Moreover, we also investigate other metrics such as bandwidth and cache hit ratio in the memory subsystem. The Intel PCM (INTEL, 2012) and Intel VTune (INTEL, 2016) tools were used to obtain data for the Broadwell and Knights Landing.

3.1.1 Workloads

As workloads, we used the OpenMP implementation of the Rodinia Benchmark, v3.1 (CHE et al., 2010). The benchmark suite Rodinia was chosen since it implements a set of applications with distinct parallel execution characteristics. We configured the benchmarks to run with the number of virtual cores of the architecture. The applications used were the following:

Back Propagation (BP): BP is an iterative algorithm used to train the weights of the connections between neurons in a multi-layer neural network. The training is performed in two phases: the Forward Phase, in which the activations are propagated from the input to the output layer, and the Backward Phase, in which the error between the observed and requested values in the output layer is propagated backward to adjust the weights and bias values. In each layer, the processing of all the nodes can be done in parallel.

Breadth-First Search (BFS): BFS implements breadth-first search traversal of an arbitrary graph. The application read a graph specification from a file. The result of the traversal is an array of integers where each element is the distance from the source node to the node with the element's index. The traversal starting node is picked randomly by the application. The host performs traversal in a loop until every node is visited.

B+ Tree (B+): B+tree implements queries on large n-ary trees. The benchmark constructs the tree as a dynamic data structure of heap allocated nodes. As the size of the tree grows, the cost of conversion increases substantially. If the tree is modified, the entire tree must be converted again. The application reads the query commands from a file, one command at a time, and processes them immediately.

CFD Solver (CFD): CFD solves the 3D Euler equations for a zero density, compressible fluid. The algorithm is implemented as an unstructured grid finite volume solver, where each thread operates on a block of the 3D space.

Heart Wall Tracking (HW): HW implements the tracking stage of the Heart Wall application. The application comes with a single input AVI file containing the sequence of ultrasound images. The kernel is invoked once for each frame in the video sequence. Thus the application is a wrapper for the kernel implementing the tracking stage of the Heart Wall application.

HotSpot 2D (HS2D): HS2D estimates the temperature of a logic circuit given the initial temperature and the power dissipation of each logic cell. The temperature value of all cells is calculated in parallel and stops after a predetermined number of timesteps. This benchmark can be classified as Structured Grid.

HotSpot 3D (HS3D): HS3D performs the same calculations as HS2D but for a tridimensional integrated circuit.

K-Means (KM): A set of k initial centers is generated in the same multidimensional space as the data set one wishes to cluster. The algorithm assigns each point to the center that is closer to it. After that, the center of these clusters is calculated, and the process is repeated with the new centers for a predetermined number of iterations. The main computation performed is calculating the Euclidean distance between the points, and that is done in parallel. The problem is classified as Dense Linear Algebra.

LavaMD (MD): MD calculates the movement of particles due to the forces from other particles in a large tridimensional space. The 3D space is first divided into boxes. Then, for every particle in a box, the loop processes the interactions with other particles in this box and then with particles in the neighbor boxes. The processing of each particle consists of a single stage of calculation that is enclosed in the innermost loop. The nested loops in the application were parallelized in such a way that at any point of time wavefront accesses adjacent memory locations.

Leukocyte Tracking (LC): Detect and track rolling leukocytes in video microscopy of blood vessels, useful for medical imaging. The cells are detected on the first frame by computing the Gradient Inverse Coefficient of Variation score for every pixel across a range of possible ellipses that could be the contour of the cell. The application then computes the Motion Gradient Vector Flow matrix in the area around each cell to track the flow of the blood. The dwarf classification is Structured Grid.

LU Decomposition (LUD): LUD performs the LU Decomposition of a matrix, that means decomposing an input matrix into two triangular matrices and a diagonal matrix. The factorization is a sum of products and is made in parallel. It is a Dense Linear Algebra application.

Myocyte Simulation (MS): MS is a simplified implementation of the simulation that models the heart muscle cell. The simulation is based on ordinary differential equations (ODE) describing the activity in the cell. It uses the Runge-Kutta-Fehlberg method to find approximate solutions for the ODE. The simulation is performed for a number of steps in the time interval specified as a parameter on the command line. The number of ODEs is 91, and the parameters and the initial data that specify the ODEs are read from the input files.

K-Nearest Neighbors (KNN): The KNN is a simple clustering algorithm that attributes to each point the label of its nearest neighbor. The main computational task performed is calculating the distance between the points, which is done in parallel and behaves as a Dense Linear Algebra dwarf.

Needleman-Wunsch (NW): The NW algorithm is an algorithm for sequence alignments, which obtains the best alignment by using optimal alignments of smaller subsequences. It is often used in bioinformatics to align protein or nucleotide sequences. It consists of three steps: initialization of the score matrix, calculation of scores, and deducing the alignment from the score matrix. The second step is parallelized. Because NW has diagonal stride memory access pattern, it is hard to exploit data locality to improve performance.

Particle Filter (PF): PF is a probabilistic model for tracking objects in a noisy environment using a given set of particle samples. The application has several parallel stages, and implicit synchronization between stages is required. It is a Structured Grid dwarf application.

Pathfinder (PATH): Pathfinder finds the shortest path in a 2D grid using dynamic programming. It computes row by row, achieving the smallest sum of weights between the beginning and the end of the path. In each iteration, the shortest path calculation is parallelized.

Stream Cluster (SC): Given a stream of input points, SC calculates median points and assigns each point of the input stream to the cluster created from the median that is closer to it. The main computational work in the benchmark is calculating the cost of opening new centers and the distance of the points from those centers, and so these calculations are made in parallel. In both, the distance between two points is the most time-consuming task. It is a Dense Linear Algebra application.

Speckle Reducing Anisotropic Diffusion (SRAD): SRAD performs a Speckle Reducing Anisotropic Diffusion on an image, frequently used on radar and ultrasonic imaging applications. In each iteration of the method, the whole image is updated. Each thread computes on a slice of the input matrix, providing high thread locality, and thus good scalability. The dwarf classification of this benchmark is Structured Grid.

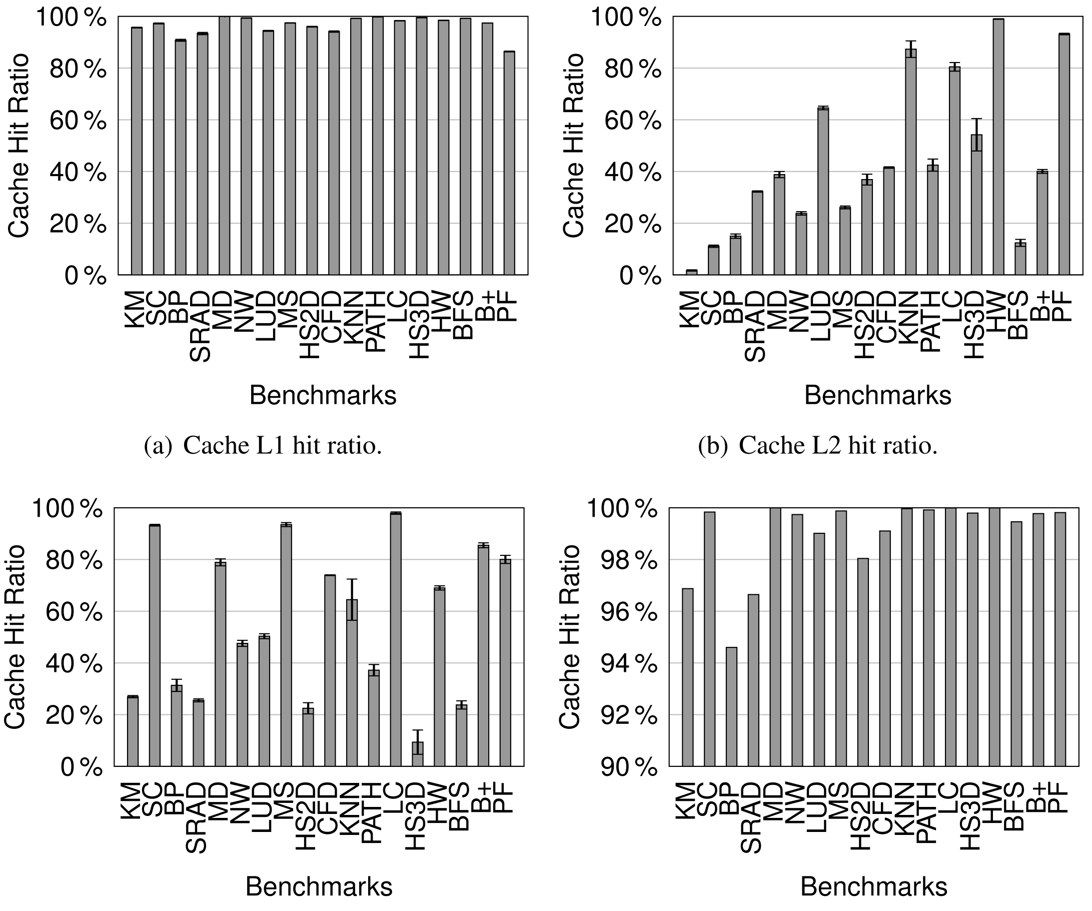
3.2 Results on Broadwell

The Broadwell architecture obtains data by accessing the private L1 and L2 caches of each core, the L3 cache shared by all cores of the same processor, or the main memory. Figures 3.2(a), 3.2(b) and 3.2(c) show the respective hit ratios of each cache level. Figure 3.2(d) shows the combined hit rate, which is the number of memory transactions serviced by any cache. The x-axis presents the name of each application ordered by the IPC (smaller shown first), and the y-axis indicates the hit ratio in percentage. The L1 cache is the fastest and smallest memory and is private to each core. The average L1 hit ratio was 96.5%. Even with an average close to 100%, small variations in the L1 hit ratio implies in significant variations in performance, and therefore it is not a good predictor of IPC by itself. Applications such as *MD*, *NW*, *KNN*, *PATH*, *HS3D*, and *BFS* have L1 hit rates up to 99.9% due to their data accesses pattern. On the other hand, applications such as *BP*, *SRAD*, *LUD*, *CFD*, and *PF*, have lower hit rates (86.4% in *PF*), affecting their performance because they access slower memories more frequently. The L2 cache is also private to each core. *LUD*, *KNN*, *LC*, *HW*, and *PF* are applications that take advantage of the L2 cache and thus have better performance results. The L2 hit ratios in these applications were up to 98.9%.

The L3 cache, the last-level cache in the Broadwell architecture, is shared between all cores of the same processor. It helps several applications by reducing the accesses in the main memory. The applications *SC*, *MS*, and *LC*, have an L3 hit ratio close to 100%, which means that their entire data fit in the L3. On the other hand, *HS2D* has an L3 hit equals to 22.5%, which is low compared to the average (56.2%), and yet the application performs above average when compared to the other applications. Most likely, the majority of its accesses were already filtered by the higher level caches.

We can observe, in the combined hit ratio, that most of the applications with the highest hit ratios are the ones with the highest IPCs, which indicates that the cache memories have a significant impact on the performance. For instance, *PF* has the best IPC even without the highest L1 hit ratio, but it has a very high combined hit ratio. To improve the performance, techniques such as loop interchange and loop tiling can be used. Using these techniques, more data is fetched to the cache memories, the data reuse in the caches increase, and cache line prefetchers can fetch data from the main memory more accurately.

Figure 3.2: Cache hit ratio in Broadwell architecture.



(a) Cache L1 hit ratio.

(b) Cache L2 hit ratio.

(c) Cache L3 hit ratio.

(d) Combined hit ratio.

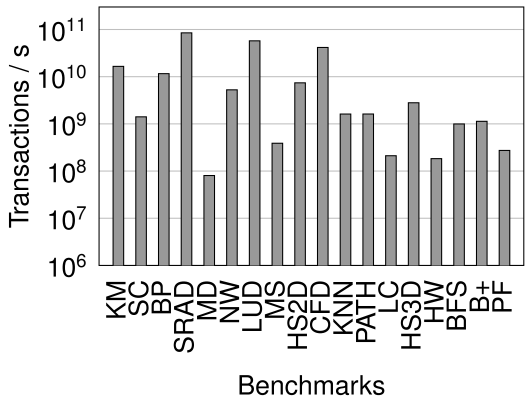
Source: The Author.

Figures 3.3(a) and 3.3(b) show results of dynamic random-access memory (DRAM) transactions per second and Quickpath Interconnect (QPI) transactions per second. Applications with low cache hit ratios have a large number of DRAM and QPI transactions, which reduce their performance by accesses to local and remote memories. The DRAM transactions per second were in average 13.1×10^9 which is 6.5 times less than the maximum value that is 84.9×10^9 (in *SRAD*). The applications *SRAD*, *LUD*, and *CFD*, have the highest values of transactions per second, which limit their performance. Transactions across the QPI also reduce the performance of applications.

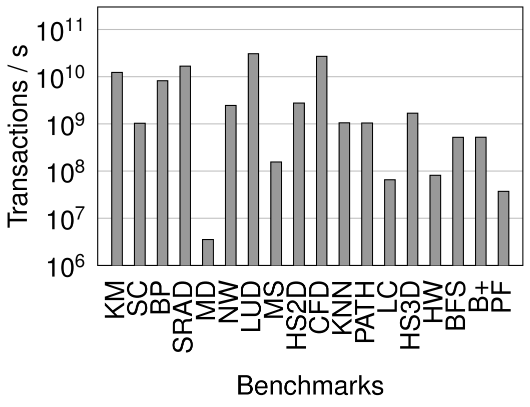
3.3 Results on Knights Landing

This architecture has private L1 and L2 caches. Thus, the cores can allocate data into its private L1 and L2, but on an L2 cache miss, the other L2 caches can be accessed remotely by the coherence protocol. Figures 3.4(a) and 3.4(b) show the L1 and L2 hit

Figure 3.3: Memory operations in the Broadwell architecture. The y-axis is in logarithmic scale in both figures.



(a) DRAM transactions.



(b) Transactions across the QPI.

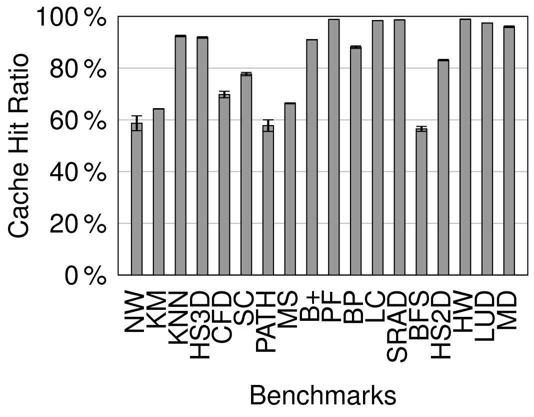
Source: The Author.

ratio of the applications in the Knights Landing architecture. Figure 3.4(c) shows the combined hit ratio. Most applications have L1 hit ratios greater than 80%. However, some applications, including *NW*, *KM*, *PATH*, *MS*, and *BFS*, have L1 hit ratios from 56.6% to 66.4%. These applications performances are affected by the latency of L2 and main memory, which are longer than L1 latency.

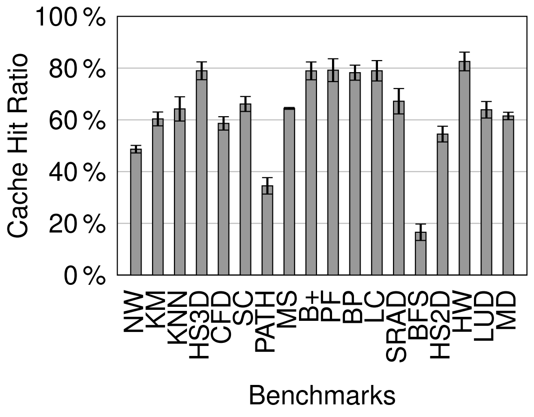
The L2 cache is the last level cache in the Knights Landing architecture and its hit ratio impacts directly in applications' performance. For instance, *HW* has one of the highest performances and also high L2 hit ratio of 82.6%. This application has high hit ratios in L1 and L2, performing better than other applications. The opposite occurs with *NW*, *KM*, and *PATH*, which have low L1 and L2 hit ratios and consequently the worst performances in a Knights Landing. Analyzing the combined hit rate, we can observe that in most cases, applications with a higher combined hit ratio have a better performance. Therefore, the conclusion we arrive is similar to the one in Broadwell. It is more relevant to have a hit on any cache and avoid access to the main memory than to have a high hit rate on any specific level.

The number of memory accesses is also crucial to the performance of the Knights Landing architecture. Figures 3.5(a) and 3.5(b) show results of DRAM and MCDRAM transactions per second, respectively. Applications with low cache hit ratios have a large number of DRAM and MCDRAM transactions, which reduce their performance by accesses to local and remote slow memories. The DRAM transactions per second were in average 0.29×10^9 which is 9.6 times less than the maximum value that is 2.79×10^9 (in *SRAD*). The applications *SRAD*, *NW*, and *LUD*, have the highest values of transactions per second, which limit their performance. MCDRAM transactions also reduce performance,

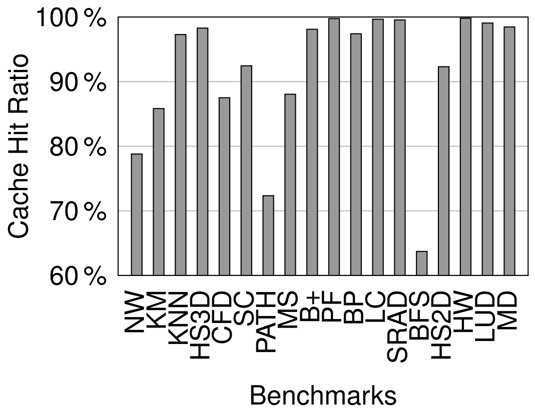
Figure 3.4: Cache hit ratio in the Knights Landing architecture.



(a) Cache L1 hit ratio.



(b) Cache L2 hit ratio.



(c) Combined hit ratio.

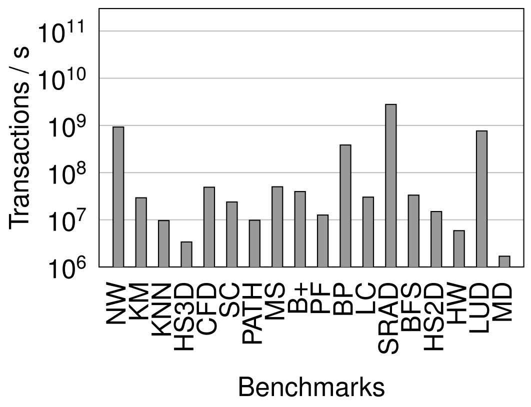
Source: The Author.

but it degrades the performance less than DRAM transactions. Applications with the highest IPC such as *MD*, *LUD*, and *HW* have more MCDRAM transactions that DRAM. The MCDRAM transactions per second were in average 15.25×10^9. The conclusion is that we should reduce the DRAM and MCDRAM accesses, but is better to have access to MCDRAM than DRAM.

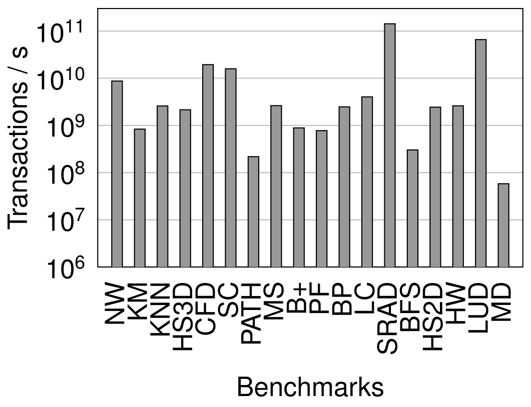
3.4 Conclusions

In this chapter, we performed experiments in two architectures aiming to investigate the performance bottlenecks. We discussed the performance of 18 applications of the Rodinia benchmark. The results show that some metrics help in understand applications performance, but there are cases where a metric alone is not representative. In the next chapter, we use this knowledge to optimize a real-world seismic exploration application.

Figure 3.5: Memory operations the Knights Landing architecture. The y-axis is in logarithmic scale in both figures.



(a) DRAM transactions.



(b) MCDRAM transactions.

Source: The Author.

4 OPTIMIZATION STRATEGIES FOR MULTI-CORE AND MANY-CORE

Chapter 3 investigates multi-core and many-core architectures performance bottlenecks. Based on that, we address some performance optimization strategies aiming to reduce the impact of performance bottlenecks. We first employ the loop interchange technique to improve cache memory usage. Afterward, we put vectorization into account intending to increase the performance of floating-point computations. Later, we apply loop scheduling and collapse to improve load balancing; Finally, we increase the memory hierarchy performance by use thread and data mapping.

Therefore, in order to validate our assumptions, we improve the performance of a real-world seismic exploration application provided by Petrobras. It implements an acoustic wave propagation approximation which 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.

The remainder of this Chapter is organized as follows. First, we introduced the seismic exploration application. Then, we present the optimization techniques and the results for both architectures. Finally, we discuss and summarize the conclusions of this Chapter.

4.1 Seismic Exploration Application

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.

In this dissertation, we focus on improving the performance of a Reverse Time Migration (RTM) program provided by Petrobras, the leading Brazilian oil company.

The program simulates the propagation of a single wavelet over time by solving the isotropic acoustic wave propagation (Equation 4.1), and the isotropic acoustic wave propagation with variable density (Equation 4.2) 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. A 2^{nd} finite differences operator approximates the derivatives.

$$\frac{1}{V^2}.\frac{\partial^2 p}{\partial t^2} = \nabla^2 p \tag{4.1}$$

$$\frac{1}{V^2}.\frac{\partial^2 p}{\partial t^2} = \nabla^2 p - \frac{\nabla\rho}{\rho}.\nabla p \tag{4.2}$$

Next section present the performance evaluation results where we employ the same experimental setup and methodology described in Section 3.1. The difference is the workload that in this chapter is the seismic exploration application. The seismic code was written in standard C and leverage from OpenMP directives for shared-memory parallelism. The stencil used in the experiments was $1024 \times 256 \times 256$ resulting in total memory usage of 2.5 GB.

4.2 Results

The following subsections present the optimizations techniques we used to improve the performance of a real-world application and the experiments performed to validate them. We describe the optimizations and analyze how they address the challenges imposed by multi-core and many-core architectures.

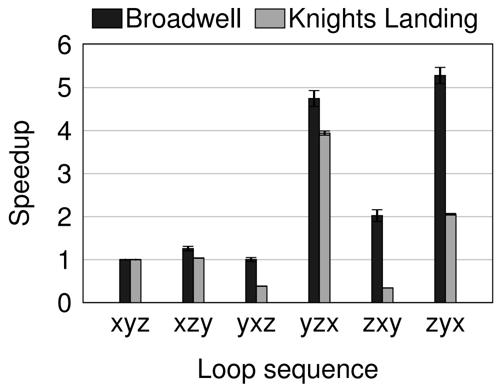
4.2.1 Memory Access Pattern to Improve Data Locality

Current computer architectures provide caches and hardware prefetchers to help programmers manage data implicitly (LEE et al., 2010). 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 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. They 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 4.1, 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 performance and combined cache hit ratio results in Broadwell. The speedup compared with the **xyz** version is 5.3×. 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 4.2(a). The L2 cache hit rate was improved from 14.9% to 77% when the loop sequence was changed to **zyx**. The L3 cache hit was also improved from 76.2% to 92% in Broadwell. However, this was not the case with the L1 cache, as its hit rate decreases from 82.3% to 73%. In Knights Landing, version **yzx** have better performance and combined cache hit ratio results. The speedups were up to 3.9× showing that this optimization impact less on the performance of Knights Landing than Broadwell. The cache hit rates are shown in Figure 4.2(b). The L2 cache hit rate was improved from 9.6% to 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.

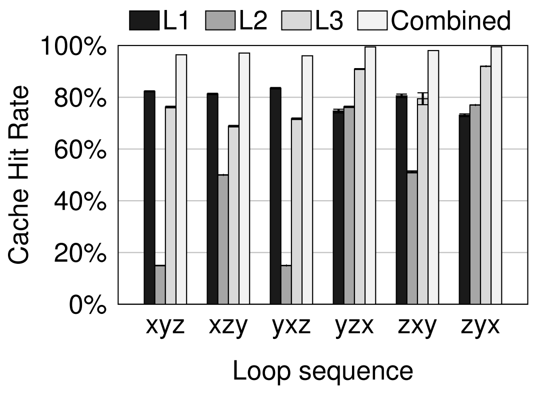
The differences in cache miss happened because the data access stride becomes different when changing the loop sequence, influencing both spatial and temporal localities. Despite the reductions in the L1 hit rate, the increase of the LLC hit rates resulted in the highest performance improvement and is, therefore, the best choice for this application since LLC caches have the highest latencies per access and increase their hit rates consequently reduces latency. The performance improvement in the Knights Landing is lower than in Broadwell because the amount of cache memory available per thread in the Knights Landing is much lower. The best loop sequence for Broadwell is zyx while for Knights Landing is yzx. This difference is due the L2 cache size which is 512 KB per core Knights Landing and 256 KB in Broadwell.

Figure 4.1: Speedup over the xyz sequence.

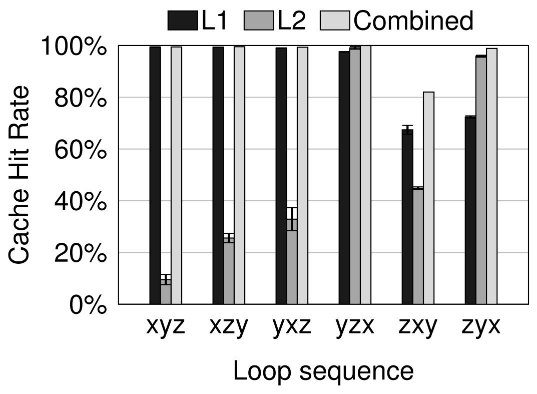


Source: The Author.

Figure 4.2: Cache hit ratio on different architectures.



(a) Broadwell.



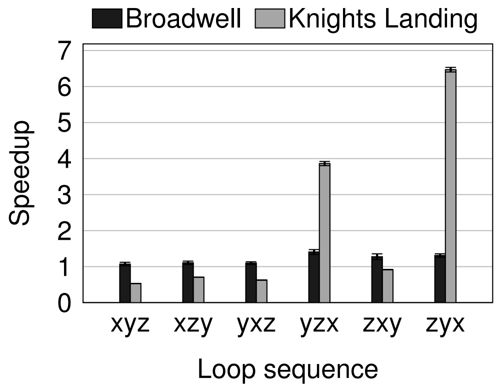
(b) Knights Landing.

Source: The Author.

4.2.2 Exploiting SIMD for Floating-point Computations

Recent hardware approaches increase performance by integrating more cores with wider SIMD (single instruction, multiple data) units (SATISH et al., 2012). This data processing technique, called vectorization, has units that 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. 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 accessed by the same instruction were consecutive along loop iterations.

Figure 4.3: Performance gain using vectorization.



Source: The Author.

We used the Advanced Vector Extensions (AVX) instructions, which is an 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 4.3. 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 Broadwell, the improvement was up to 1.4×. In the Knights Landing architecture, the improvement was up to 6.5×. These differences are due to the size of each architecture's vector unit and the number of cores used.

4.2.3 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 the most time to finish, and thereby by the core with the highest 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 multi-core and many-core systems, load balancing is even more important due to a large number of cores.

The 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 runtime 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 also helps to improve load balancing, because it increases the total number of iterations partitioned across the threads by collapsing two or more loops.

We investigated the impact of different OpenMP scheduling policies with and without loop collapse. Figures 4.4 and 4.4(b) show the speedup of each combination of schedule policy in the Broadwell and Knights Landing architectures. The baseline used to calculate the speedup is the execution time of the default schedule policy, which is static with a chunk size equal to the number of iterations divided by the number of threads.

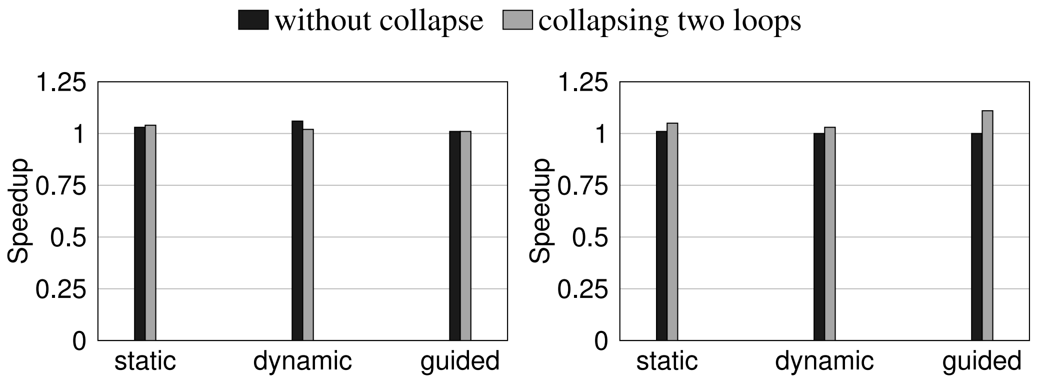
Figures 4.4 and 4.4(b) present the results for Broadwell and Knights Landing. In the experiments without the collapse, we split the outer loop between the threads. The best speedup without collapse in Broadwell was using the guided scheduling policy. It was up to 1.06× faster than the default scheduler. This approach is useful to applications whose workload changes in runtime or have some regions with unbalanced workloads. In Knights Landing, the performance was almost the same as without any scheduler.

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 threads, larger chunks can be used while keeping a good load balance. In order to investigate this, we collapsed the outer loops and evaluated different combinations of the scheduler. Results show that for Broadwell, the best policy is the static scheduler. In Knights Landing, the best policy with collapse is the guided scheduler. The performance in Broadwell was improved by up to 1.04× while in Knights Landing by up to 1.11×.

4.2.4 Optimizing Memory Affinity

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 (CRUZ et al., 2016). Other policies seek a uniform load distribution among the cores and memory controllers.

Figure 4.4: Loop collapse and scheduling performance gain.



(a) Broadwell. (b) Knights Landing.

Source: The Author.

We evaluate different thread and data mapping techniques. We combine thread and data mapping techniques because together they improve the performance even more (DIENER et al., 2016a). The following thread mapping policies were evaluated:

Default (baseline) The default thread mapping of the Linux kernel (WONG et al., 2008), focused on load balancing.

Compact Thread Mapping A compact thread mapping that arranges neighbor threads to closer cores according to the memory hierarchy (EICHENBERGER et al., 2012).

Scatter Thread Mapping A Scatter thread mapping distributes the threads as evenly as possible across the entire system, which is the opposite of compact.

RoundRobin Thread Mapping A RoundRobin thread mapping distributes the threads to the cores in order from 0 to the number of cores minus 1.

The following data mapping policies were evaluated:

Default (baseline) The default data mapping of Linux, the first-touch data mapping policy, where the page is mapped to the NUMA node of the core that accessed the page for the first time.

NUMA Balancing Data Mapping The NUMA Balancing data mapping (CORBET, 2012) migrates pages along the execution to the NUMA node of the latest thread that accessed the page. The NUMA node is detected by introducing artificial page faults along the execution.

Interleave Data Mapping The interleave data mapping arranges consecutive pages to consecutive NUMA nodes. In the Knights Landing architecture, due to having the DRAM and MCDRAM NUMA nodes, we also evaluate the Interleave MCDRAM policies, which distribute pages only to MCDRAM nodes.

The results obtained from different mapping policies in the Broadwell and Knights Landing architectures are shown in Figures 4.5 and 4.6. The speedup was normalized to the baseline mapping with the corresponding loop sequence, such that we can measure the benefits from mapping more precisely. The results of the cache miss, previously shown in Figure 4.2, can help us understand the behavior from different mapping policies. The reason for this result 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.

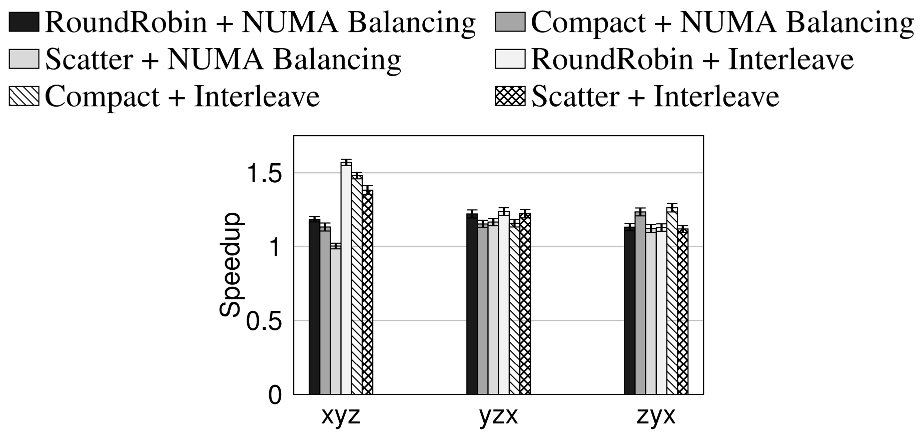
It can be observed that the **xyz** variant, which benefited most from mapping, also had most cache misses. In the other configurations, since the L3 cache hit rate is very high, we have few accesses to the main memory, such that, as explained, the benefit from the mapping is lower. The usage of an interleaved data mapping provided a better distribution of the load between the memory controllers, with the cost of additional inter-chip traffic. Despite the trade-off between the load and inter-chip traffic, the interleaved data mapping provided the best improvements overall.

On the Broadwell architecture, the best speedup was 1.6×, achieved for round-robin with interleave mapping for the xyz version. It combines an even thread distribution with a balanced distribution of the pages. We combine both thread and data mapping because, in most applications, the effectiveness of data mapping depends on thread mapping.

For the experiments in the Knights Landing architecture, we analyze the same techniques as in Broadwell plus interleave using MCDRAM nodes. The best speedup was 4.4× for scatter with interleave MCDRAM for the xyz version. We can observe that in Knights Landing, the best improvements usually happened with policies that focus on load balancing, such as the scatter thread mapping and interleave data mapping.

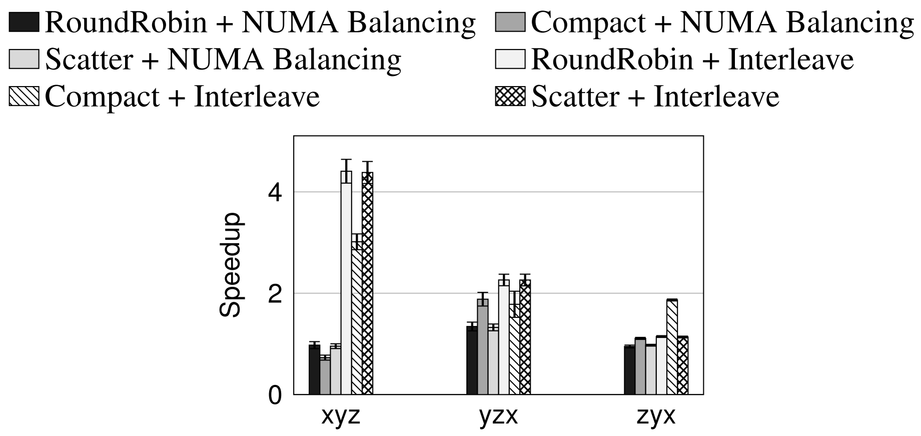
The difference in the memory locality between the cores affects these architectures performance and scalability. In NUMA systems, the time to access the main memory depends on the core that requested the memory access and the NUMA node that contains the destination memory bank. If the core and destination memory bank belong to the same node, we have a local memory access. On the other hand, if the core and the destination memory bank belong to different NUMA nodes, we have a remote memory access. Local memory accesses are faster than remote memory accesses. By mapping the threads and data of the application in such a way that we increase the number of local memory accesses over remote memory accesses, the average latency of the main memory is reduced.

Figure 4.5: Mapping results on Broadwell.



Source: The Author.

Figure 4.6: Mapping results on Knights Landing.



Source: The Author.

Figure 15 shows the number of remote memory accesses that were reduced when we use mapping techniques. For xyz version, we reduced up to 150 GB of remote accesses improving application performance and scalability.

4.3 Conclusions

In this chapter, we applied and analyzed the performance of a set of optimization techniques on multi-core and many-core architectures. We showed that all techniques together could improve the performance of a real-world seismic exploration application by up to 22.7× and 56.7× on Broadwell and Knights Landing, respectively. Regarding the performance, the best was 259.9 GFLOPS on the Broadwell architecture. We emphasize that the optimizations presented in this chapter can also be applied to other applications and architectures.

5 CONCLUSION AND FUTURE WORK

This work performs a detailed analysis of multi-core and many-core architectures and optimizes their performance using different strategies. We evaluated 18 benchmarks and used hardware performance counters to gather accurate measurements of the actual impact of different factors that influence the performance. This evaluation has shown that some metrics help in understand applications performance, but there are cases where a metric alone is not representative.

We use our performance bottlenecks study as a basis for optimizing a real-world geophysics model. We applied the following optimization techniques: (1) loop interchange to improve cache memory usage; (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 better use the memory hierarchy. These optimizations can also be applied to other applications and architectures.

In our performance optimization experiments, we show that loop interchange is a useful technique to improve the performance of different cache memory levels, being able to improve the performance by up to 5.3× and 3.9× on Broadwell and Knights Landing, respectively. These improvements happened because we were able to increase the last level cache hit ratio by up to 95.9%. 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 1.4× and 6.5×. Load Balancing and collapse techniques were also evaluated, but the application balance mitigated their performance improvements. These techniques improve the performance of Knights Landing by up to 1.1×. Thread and data mapping techniques were also evaluated, with a performance improvement of up to 1.6× and 4.4× We also compared the best version of each architecture and showed that we were able to improve the performance of Broadwell by 22.7× and Knights Landing by 56.7× compared with a naive version but at the end, Broadwell was 1.2× faster than Knights Landing.

5.1 Future Work

Future work will focus on proposing an automatic mechanism to optimize the performance of multi-core and many-core architectures. Furthermore, we plan on expanding our evaluation of the performance bottlenecks by using newer architectures and evaluate energy consumption.

5.2 Publications

The following papers were produced during this dissertation. We first list the ones strong related to this work, including those under review:

- **SERPA, M. S.**; CRUZ, E. H. M.; NAVAUX, P. O. A.; PANETTA, J. Otimizando uma Aplicação de Geofísica com Mapeamento de Threads e Dados. In: WSCAD 2018 - 19th Symposium on Computer Systems, 2018, (*Under Review*).
- **SERPA, M. S.**; KRAUSE, A. M.; CRUZ, E. H. M.; PASIN, M.; FELBER, P.; NAVAUX, P. O. A. Optimizing Machine Learning Algorithms on Multi-core and Many-core Architectures using Thread and Data Mapping. In: PDP 2018 - Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2018, **Qualis A2**.
- CRUZ, E. H. M.; DIENER, M.; **SERPA, M. S.**; NAVAUX, P. O. A.; PILLA, L. L.; KOREN, I. Improving Communication and Load Balancing with Thread Mapping in Manycore Systems. In: PDP 2018 - Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2018, **Qualis A2**.
- **SERPA, M. S.**; CRUZ, E. H. M.; DIENER, M.; KRAUSE, A. M.; NAVAUX, P. O. A; PANETTA, J.; FARRÉS, A.; ROSAS, C.; HANZICH, M. Optimization Strategies for Geophysics Models on Many-core Systems. In: International Journal of High Performance Computing Applications, 2018 (*Minor Review*), **Qualis B1**.
- **SERPA, M. S.**; KRAUSE, A. M.; CRUZ, E. H. M.; NAVAUX, P. O. A. Otimizando Algoritmos de Machine Learning com Mapeamento de Threads e Dados. In: ERAD 2018 - XVIII Escola Regional de Alto Desempenho, 2018.
- **SERPA, M. S.**; CRUZ, E. H. M.; DIENER, M.; ROSAS, C.; PANETTA, J.; HANZICH, M.; NAVAUX, P. O. A. Strategies to Improve the Performance of a Geophysics Model for Different Manycore Systems. In: WAMCA 2017 - Workshop on Applications for Multi-Core Architectures, 2017.
- **SERPA, M. S.**; CRUZ, E. H. M.; MOREIRA, F. B.; DIENER, M.; NAVAUX, P. O. A. A Deep Analysis of Memory Performance and Bottlenecks in Multicore and Manycore Architectures. In: International Journal of Parallel Programming, 2017 (*Major Review*), **Qualis B1**.

- **SERPA, M. S.**; CRUZ, E. H. M.; NAVAUX, P. O. A. Impacto de Técnicas de Otimização de Software em Arquiteturas Multicore e Manycore. In: ERAD 2017 - XVII Escola Regional de Alto Desempenho, 2017.
- **SERPA, M. S.**; CRUZ, E. H. M.; MOREIRA, F. B.; DIENER, M.; NAVAUX, P. O. A. Impacto do Subsistema de Memória em Arquiteturas CPU e GPU. In: WSCAD 2016 - 17th Symposium on Computer Systems, 2016.

The following papers were also published during this dissertation:

- PAVAN, P. J.; **SERPA, M. S.**; PADOIN, E.L.; SCHNORR, L.; NAVAUX, P. O. A.; PANETTA, J. Melhorando o Desempenho de Operações de E/S do Algoritmo RTM Aplicado na Prospecção de Petróleo. In: WSCAD 2018 - 19th Symposium on Computer Systems, 2018, (*Under Review*).
- PAVAN, P. J.; **SERPA, M. S.**; CARREÑO, E. D.; ABAUNZA, V. E. M.; PADOIN, E. L.; NAVAUX, P. O. A.; PANETTA, J.; MÉHAUT, J. F. Improving Performance and Energy Efficiency of Stencil Based Applications on GPU Architectures. In: Latin America High Performance Computing Conference, Piedecuesta - Colombia, 2018.
- ABAUNZA, V. E. M.; **SERPA, M. S.**; NAVAUX, P. O. A.; PADOIN, E. L.; PANETTA, J.; MÉHAUT, J. F. Performance Evaluation of Stencil Computations based on Source-to-Source Transformations. In: Latin America High Performance Computing Conference, Piedecuesta - Colombia, 2018.
- PAVAN, P. J.; **SERPA, M. S.**; ABAUNZA, V. E. M.; PADOIN, E. L.; NAVAUX, P. O. A.; PANETTA, J. Strategies to Improve the Performance and Energy Efficiency of Stencil Computations for NVIDIA GPUs. In: WPerformance 2018 - XVII Workshop em Desempenho de Sistemas Computacionais e de Comunicação, 2018.
- ABAUNZA, V. E. M.; **SERPA, M. S.**; NAVAUX, P. O. A.; PADOIN, E. L.; PANETTA, J. Performance Prediction of Geophysics Numerical Kernels on Accelerator Architectures. In: Energy 2018 - The International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies, 2018.
- SCHEPKE, C.; LIMA, J. V. F; **SERPA, M. S.** Challenges on Porting Lattice Boltzmann Method on Accelerators: NVIDIA Graphic Processing Units and Intel Xeon Phi. In: Analysis and Applications of Lattice-Boltzmann Simulations, p. 30-53, 2018.
- KRAUSE, A. M.; **SERPA, M. S.**; NAVAUX, P. O. A. Implementação de uma Apli-

cação de Simulação Geofísica em OpenCL. In: ERAD 2018 - XVIII Escola Regional de Alto Desempenho, 2018.

- CRUZ, E. H. M.; CARREÑO, E. D.; **SERPA, M. S.**; NAVAUX, P. O. A; FREITAS, I. J. F. Intel Modern Code: Programação Paralela e Vetorial AVX para o Processador Intel Xeon Phi Knights Landing. In: WSCAD 2017 - XVIII Simpósio em Sistemas Computacionais de Alto Desempenho, 2017.
- LORENZONI, R. K.; **SERPA, M. S.**; PADOIN, E. L.; PANETTA, J.; NAVAUX, P. O. A.; MEHAUT, J. Otimizando o Uso do Subsistema de Memória de GPUs para Aplicações Baseadas em Estênceis. In: WPerformance 2017 - XVI Workshop em Desempenho de Sistemas Computacionais e de Comunicação, 2017.
- ABAUNZA, V. E. M.; **SERPA, M. S.**; DUPROS, F.; PADOIN, E. L.; NAVAUX, P. O. A. Performance Prediction of Acoustic Wave Numerical Kernel on Intel Xeon Phi Processor. In: CARLA 2017 - Latin America High Performance Computing Conference, 2017.
- LORENZONI, R. K.; **SERPA, M. S.**; PADOIN, E. L.; NAVAUX, P. O. A. Melhorando o Desempenho da Computação de Estênceis em GPUs. In: CNMAC 2017 - Brazilian Society of Computational and Applied Mathematics, 2017.
- LORENZONI, R. K.; **SERPA, M. S.**; PADOIN, E. L.; NAVAUX, P. O. A.; MÉHAUT, J. F. Impacto do Subsistema de Memória da Arquitetura Kepler no Desempenho de uma Aplicação de Propagação de Onda. In: ERAD 2017 - XVII Escola Regional de Alto Desempenho, 2017.
- SILVA, S. A.; **SERPA, M. S.**; SCHEPKE, C. Técnicas de Otimização Computacional em um Algoritmo de Multiplicação de Matrizes. In: ERAD 2017 - XVII Escola Regional de Alto Desempenho, 2017.
- **SERPA, M. S.**; BEZ, J. L.; CRUZ, E. H. M.; DIENER, M.; ALVES, M. A. Z.; NAVAUX, P. O. A. Intel Modern Code: Programação Vetorial e Paralela em Arquiteturas Intel Xeon e Intel Xeon Phi. In: ERAD 2017 - Escola Regional de Alto Desempenho, 2017.
- SILVA, S. A.; **SERPA, M. S.**; SCHEPKE, C. Derrube Todos os Recordes de Ganho de Desempenho Otimizando seu Código. In: ERAD 2017 - Escola Regional de Alto Desempenho, 2017.
- SILVA, S. A.; **SERPA, M. S.**; SCHEPKE, C. Técnicas de Otimização Loop Unrolling e Loop Tiling em Multiplicações de Matrizes Utilizando OpenMP. In: WS-

CAD 2016 - 17th Symposium on Computer Systems, 2016 (*Best Paper Award*).

- KAPELINSKI, K.; SCHEPKE, C.; **SERPA, M. S.** Uma Abordagem Inicial para a Paralelização de uma Aplicação de Simulação de Ablação por Radiofrequência para o Tratamento de Câncer. In: WSCAD 2016 - 17th Symposium on Computer Systems, 2016.
- ROLOFF, E.; CARREÑO, E. D.; VALVERDE-SÁNCHEZ, J. K. M.; DIENER, M.; **SERPA, M. S.**; HOUZEAUX, G.; SCHNORR, L.; MAILLARD, N.; GASPARY, L. P.; NAVAUX, P. O. A. Performance Evaluation of Multiple Cloud Data Centers Allocations for HPC. In: CARLA 2016 - Latin America High Performance Computing Conference, 2016.

REFERENCES

ANDREOLLI, C. et al. Chapter 23 - Characterization and Optimization Methodology Applied to Stencil Computations. In: **High Performance Parallelism Pearls**. [S.l.]: Morgan Kaufmann, 2015.

AUSAVARUNGNIRUN, R. et al. Exploiting inter-warp heterogeneity to improve gpgpu performance. In: IEEE. **2015 International Conference on Parallel Architecture and Compilation (PACT)**. [S.l.], 2015. p. 25–38.

BORKAR, S.; CHIEN, A. A. The future of microprocessors. **Communications of the ACM**, ACM, v. 54, n. 5, p. 67–77, 2011.

CASTRO, M. et al. Seismic wave propagation simulations on low-power and performance-centric manycores. **Parallel Computing**, v. 54, p. 108 – 120, 2016.

CHE, S. et al. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In: **IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.]: Ieee, 2010. ISBN 978-1-4244-9297-8.

COOK, S. **CUDA programming: a developer's guide to parallel computing with GPUs**. [S.l.]: Newnes, 2012.

CORBET, J. **Toward better NUMA scheduling**. 2012. Disponível em: <http://lwn.net/Articles/486858/>.

COTEUS, P. W. et al. Technologies for exascale systems. **IBM Journal of Research and Development**, IBM, v. 55, n. 5, p. 14–1, 2011.

CRUZ, E. H. et al. Lapt: A locality-aware page table for thread and data mapping. **Parallel Computing (PARCO)**, v. 54, p. 59 – 71, 2016. ISSN 0167-8191.

CRUZ, E. H. et al. Lapt: A locality-aware page table for thread and data mapping. **Parallel Computing**, Elsevier, v. 54, p. 59–71, 2016.

CRUZ, E. H. et al. Improving communication and load balancing with thread mapping in manycore systems. In: IEEE. **Parallel, Distributed and Network-based Processing (PDP), 2018 26th Euromicro International Conference on**. [S.l.], 2018. p. 93–100.

CRUZ, E. H. M. et al. Hardware-Assisted Thread and Data Mapping in Hierarchical Multicore Architectures. **ACM Trans. Archit. Code Optim.**, ACM, New York, NY, USA, v. 13, n. 3, set. 2016. ISSN 1544-3566.

DENG, L. et al. Performance Optimization and Comparison of the Alternating Direction Implicit CFD Solver on Multi-core and Many-Core Architectures. **Chinese Journal of Electronics**, IET, v. 27, n. 3, p. 540–548, 2018.

DIENER, M. et al. Affinity-Based Thread and Data Mapping in Shared Memory Systems. **ACM Computing Surveys (CSUR)**, v. 49, n. 4, p. 1–38, 2016. ISSN 15577341.

DIENER, M. et al. Kernel-based thread and data mapping for improved memory affinity. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 27, n. 9, p. 2653–2666, 2016.

EICHENBERGER, A. E. et al. The design of OpenMP thread affinity. **Lecture Notes in Computer Science**, v. 7312 LNCS, p. 15–28, 2012. ISSN 03029743.

FALCH, T. L.; ELSTER, A. C. Register caching for stencil computations on gpus. In: **2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing**. [S.l.]: IEEE, 2014. p. 479–486.

GROPP, W.; SNIR, M. Programming for exascale computers. **Computing in Science Engineering**, v. 15, n. 6, p. 27–35, 2013.

HASIB, A. A. et al. Energy efficiency effects of vectorization in data reuse transformations for many-core processors—a case study. **Journal of Low Power Electronics and Applications**, Multidisciplinary Digital Publishing Institute, v. 7, n. 1, p. 5, 2017.

HE, J.; CHEN, W.; TANG, Z. Nestedmp: Enabling cache-aware thread mapping for nested parallel shared memory applications. **Parallel Computing**, Elsevier, v. 51, p. 56–66, 2016.

HEINECKE, A. et al. **High Order Seismic Simulations on the Intel Xeon Phi Processor (Knights Landing)**. Cham: Springer International Publishing, 2016. 343–362 p.

HSU, J. Three paths to exascale supercomputing. **IEEE Spectrum**, v. 53, n. 1, p. 14–15, 2016.

INTEL. **Intel Performance Counter Monitor - A better way to measure CPU utilization**. 2012. Disponível em: <http://www.intel.com/software/pcm>.

INTEL. **Intel VTune Amplifier XE 2016**. 2016.

JIA, W.; SHAW, K. A.; MARTONOSI, M. Mrpb: Memory request prioritization for massively parallel processors. In: IEEE. **2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.], 2014. p. 272–283.

KIRK, D. B.; WEN-MEI, W. H. **Programming massively parallel processors: a hands-on approach**. [S.l.]: Morgan kaufmann, 2016.

LASTOVETSKY, A.; SZUSTAK, L.; WYRZYKOWSKI, R. Model-based optimization of eulag kernel on intel xeon phi through load imbalancing. **IEEE Transactions on Parallel and Distributed Systems**, v. 28, n. 3, p. 787–797, March 2017. ISSN 1045-9219.

LEE, V. W. et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 38, n. 3, jun. 2010. ISSN 0163-5964.

LIU, G. et al. Optimizing thread-to-core mapping on manycore platforms with distributed tag directories. In: IEEE. **Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific**. [S.l.], 2015. p. 429–434.

MA, L.; AGRAWAL, K.; CHAMBERLAIN, R. D. A memory access model for highly-threaded many-core architectures. **Future Generation Computer Systems**, v. 30, p. 202 – 215, 2014. Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, {ICPADS} 2012 Selected Papers.

MARUYAMA, N.; AOKI, T. Optimizing stencil computations for nvidia kepler gpus. In: **Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna**. [S.l.: s.n.], 2014. p. 89–95.

MEI, X.; CHU, X. Dissecting GPU memory hierarchy through microbenchmarking. **CoRR**, abs/1509.02308, 2015.

MITTAL, S.; VETTER, J. S. A survey of cpu-gpu heterogeneous computing techniques. **ACM Computing Surveys (CSUR)**, ACM, v. 47, n. 4, p. 69:1–69:35, jul. 2015. ISSN 0360-0300.

NALAMALPU, A. et al. Broadwell: A family of ia 14nm processors. In: IEEE. **VLSI Circuits (VLSI Circuits), 2015 Symposium on**. [S.l.], 2015. p. C314–C315.

NASCIUTTI, T. C.; PANETTA, J. Impacto da arquitetura de memória de gpgpus na velocidade da computaç ao de estênceis. In: **XVII Simpósio de Sistemas Computacionais (WSCAD-SSC)**. Aracaju, SE: [s.n.], 2016. p. 1–8. ISSN 2358-6613.

SAO, P. et al. A sparse direct solver for distributed memory xeon phi-accelerated systems. In: **IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2015. p. 71–81.

SATISH, N. et al. Can traditional programming bridge the ninja performance gap for parallel computing applications? In: IEEE. **ACM SIGARCH Computer Architecture News**. [S.l.], 2012. v. 40, n. 3.

SERPA, M. S. et al. Strategies to improve the performance of a geophysics model for different manycore systems. In: IEEE. **2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)**. [S.l.], 2017. p. 49–54.

SERPA, M. S. et al. Optimizing machine learning algorithms on multi-core and many-core architectures using thread and data mapping. In: IEEE. **Parallel, Distributed and Network-based Processing (PDP), 2018 26th Euromicro International Conference on**. [S.l.], 2018. p. 329–333.

SLOTA, G. M.; RAJAMANICKAM, S.; MADDURI, K. High-performance graph analytics on manycore processors. In: **IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2015. p. 17–27. ISSN 1530-2075.

SODANI, A. et al. Knights landing: Second-generation intel xeon phi product. **IEEE Micro**, v. 36, n. 2, 2016.

TOUSIMOJARAD, A.; VANDERBAUWHEDE, W. An efficient thread mapping strategy for multiprogramming on manycore processors. **Parallel Computing: Accelerating Computational Science and Engineering (CSE), Advances in Parallel Computing**, v. 25, p. 63–71, 2014.

WANG, X. et al. A Fast Tridiagonal Solver for Intel MIC Architecture. In: **IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2016. p. 172–181.

WONG, C. S. et al. Towards achieving fairness in the Linux scheduler. **ACM SIGOPS Operating Systems Review**, v. 42, n. 5, p. 34–43, jul 2008.

ZIAKAS, D. et al. Intel QuickPath Interconnect - Architectural Features Supporting Scalable System Architectures. In: **Symposium on High Performance Interconnects (HOTI)**. [S.l.: s.n.], 2010. p. 1–6.

APPENDIX A — RESUMO EM PORTUGUÊS

In this chapter, we present a summary of this master thesis in the portuguese language, as required by the PPGC Graduate Program in Computing.

Neste capítulo, é apresentado um resumo desta dissertação de mestrado na língua portuguesa, como requerido pelo Programa de Pós-Graduação em Computação.

A.1 Introdução

A computação de alto desempenho (CAD) tem sido responsável por uma grande revolução científica. Através dos computadores, problemas que até então não podiam ser resolvidos, ou que demandavam muito tempo para serem solucionados, passaram a estar ao alcance da comunidade científica. A evolução das arquiteturas de computadores acarretou no aumento do poder computacional, ampliando a gama de problemas que poderiam ser tratadas computacionalmente. A introdução de circuitos integrados, *pipelines*, aumento da frequência de operação, execução fora de ordem e previsão de desvios constituem parte importante das tecnologias introduzidas até o final do século XX. Recentemente, tem crescido a preocupação com o gasto energético, com o objetivo de se atingir a computação em nível *exascale* de forma sustentável (HSU, 2016). Entretanto, as tecnologias até então mencionadas não possibilitam atingir tal objetivo, devido ao alto custo energético de se aumentar a frequência e estágios de *pipeline*, assim como a chegada nos limites de exploração do paralelismo em nível de instrução (BORKAR; CHIEN, 2011; COTEUS et al., 2011).

A fim de se solucionar tais problemas, arquiteturas paralelas e heterogêneas foram introduzidas nos últimos anos. A principal característica de arquiteturas paralelas é a presença de vários núcleos de processamento operando concorrentemente, de forma que a aplicação deve ser programada separando-a em diversas tarefas que se comunicam entre si. Em relação às arquiteturas heterogêneas, sua principal característica é a presença de diferentes arquiteturas em um mesmo sistema, cada um com sua própria arquitetura especializada para um tipo de tarefa. A utilização de aceleradores é uma das principais formas adquiridas por arquiteturas heterogêneas, no qual um processador genérico é responsável principalmente pela gerência do sistema, e diversos aceleradores presentes no sistema realizam a computação de determinados tipos de tarefas.

A utilização de arquiteturas paralelas e heterogêneas impõe diversos desafios para se obter um alto desempenho (MITTAL; VETTER, 2015). As aplicações precisam ser codificadas considerando as particularidades e restrições de cada arquitetura, assim como suas características arquiteturais distintas (GROPP; SNIR, 2013). Por exemplo, na hierarquia de memória, a presença de diversos níveis de memória cache, alguns compartilhados e outros privados, bem como se os bancos de memória encontram-se centralizados ou distribuídos, introduz tempos de acesso não uniformes, o que gera um grande impacto no desempenho (CRUZ et al., 2016a). Isso é ainda mais crítico em arquiteturas heterogêneas, visto que cada acelerador pode possuir sua própria, e distinta, hierarquia de memória. Além disso, nas arquiteturas heterogêneas, o número de unidades funcionais pode variar entre os diferentes aceleradores, sendo que o próprio conjunto de instruções pode também não ser o mesmo. Neste contexto, é importante desenvolver técnicas para análise de desempenho e do comportamento de arquiteturas paralelas e heterogêneas, a fim de se propiciar um melhor suporte para otimizar o desempenho de aplicações.

A.1.1 Contribuições

O principal objetivo desta pesquisa é avaliar arquiteturas *multi-core* e *many-core*, e reduzir os gargalos de desempenho através de otimizações para código fonte.

Considerando os objetivos, as principais contribuições deste trabalho são:

- Um conjunto de métricas de desempenho foi analisado sobre aplicações com características distintas de execução paralela, com o objetivo de encontrar uma correlação entre a métrica e o desempenho da aplicação (IPC).
- Um conjunto de estratégias de otimização de desempenho foi aplicado, com o objetivo de aumentar o desempenho de uma aplicação de exploração sísmica no mundo real. As técnicas empregadas foram: *loop interchange* para melhorar o uso da memória cache; vetorização para aumentar o desempenho de cálculos de ponto flutuante; *load balancing* e *collapse* para melhorar o balanceamento de carga; e mapeamento de *threads* e dados para melhor usar a hierarquia de memória.

A.2 Arquiteturas *Multi-core* e *Many-core*

Desde 2003, a indústria vem seguindo duas abordagens para o projeto de microprocessadores (KIRK; WEN-MEI, 2016). A abordagem *multi-core* é orientada à latência, onde instruções são executadas em poucos ciclos de *clock*. Por outro lado, as arquiteturas *many-core* tem uma abordagem focada na vazão, ou seja, um grande número de instruções são executadas por unidade de tempo.

O projeto das arquiteturas *multi-core* e *many-core* é diferente ao ponto que, dependendo da aplicação, o desempenho pode ser muito grande em uma arquitetura e muito pequeno na outra (COOK, 2012). A arquitetura *multi-core* utiliza uma lógica de controle sofisticada para permitir que instruções de uma única *thread* sejam executadas em paralelo. Grandes memórias *cache* são fornecidas para reduzir latências de acesso às instruções e dados de aplicações que tem acesso predominante à memória. Por fim, as operações das Unidades Lógicas e Aritméticas (ULA) também são projetadas visando otimizar a latência.

A arquitetura *many-core* tira proveito de um grande número de *threads* de execução. Pequenas memórias *cache* são fornecidas para evitar que múltiplas *threads*, acessando os mesmos dados, precisem ir até a memória principal. Além disso, a maior parte do *chip* é dedicada a unidades de ponto flutuante. Arquiteturas desse tipo são projetadas como mecanismos de cálculo de ponto flutuante e não para operações convencionais, que são realizadas por arquiteturas *multi-core*. Algumas aplicações poderão utilizar tanto *multi-core* quanto *many-core* em conjunto, sendo cada arquitetura melhor para um tipo de operação.

A.2.1 Trabalhos Relacionados

Com a análise dos trabalhos relacionados, foi possível concluir que o conhecimento profundo do comportamento da aplicação no nível arquitetural permite desenvolver técnicas para obter desempenho. Este trabalho vai além da análise e busca uma maior compreensão do desempenho de diferentes aplicações em sistemas *multi-core* e *many-core*. Arquiteturas *multi-core* e *many-core* porque seu estilo de programação é semelhante, diferente de uma arquitetura de GPU que tem um estilo de programação muito diferente e incomum. Os trabalhos relacionados também mostram que a maior parte do trabalho é focada em otimizações de memória, mas vários trabalhos visam a vetorização, balanceamento de carga e mapeamento. Desta forma, foram realizadas quatro otimizações juntas em uma aplicação do mundo real. Os próximos capítulos descrevem as propostas em detalhes.

A.3 Análise dos Gargalos de Desempenho

Atualmente, existem várias arquiteturas diferentes disponíveis não apenas para a indústria, mas também para consumidores finais. Processadores *multi-core* e *many-core* apresentam características muito diferentes. Essa ampla gama de características representa um desafio para os desenvolvedores de aplicações, porque a mesma aplicação pode ter um bom desempenho quando executado em uma arquitetura, mas mal em outra arquitetura.

Para explicar melhor a motivação, o comportamento de um conjunto de aplicações paralelas nessas arquiteturas é mostrado. A Figura A.1 mostra a métrica IPC (instruções por ciclo), que indica o número médio de instruções executadas por ciclo, para 18 aplicações da suíte Rodinia (CHE et al., 2010). Como esperado, o desempenho de cada aplicação depende da arquitetura. Existem três grupos de aplicações: melhor em Broadwell; melhor em Knights Landing; e quase o mesmo desempenho em ambas as arquiteturas.

Isso motiva o estudo de aplicações e características de arquituras, com o objetivo de entender por que uma aplicação funciona melhor em uma arquitetura e como melhorar seu desempenho. Contadores de desempenho de *hardware* foram utilizados para coletar medidas precisas do impacto real de diferentes fatores que influenciam o desempenho. Ao fazer isso, uma compreensão detalhada de como os diferentes aspectos da arquitetura afetam o desempenho das aplicações foi obtida. Este estudo serviu de base para a próxima seção, onde uma aplicação de exploração sísmica no mundo real foi otimizada.

A.4 Estratégias de Otimização para *Multi-core* e *Many-core*

A Seção A.3 investiga os gargalos de desempenho de arquiteturas *multi-core* e *many-core*. Os resultados mostraram que um dos aspectos mais importantes é o comportamento da memória *cache*, já que a memória *cache* desempenha um papel crucial no desempenho. Da mesma forma, a hierarquia de memória, composta de várias camadas de *cache* e controladores de memória, tem um impacto significativo no desempenho.

Com base nisso, algumas estratégias de otimização de desempenho foram abordadas com o objetivo de reduzir o impacto desses gargalos. Primeiro, a técnica de *loop interchange* foi empregada para melhorar o uso da memória cache. A seguir, a vetorização foi considerada para aumentar o desempenho dos cálculos de ponto flutuante. Após, *load balancing* e *collapse* foram aplicadas para melhorar o balanceamento de carga; Finalmente, o desempenho da hierarquia de memória foi aumentado usando o mapeamento de *threads* e dados.

Portanto, a fim de validar nossas suposições, o desempenho de uma aplicação de exploração sísmica real fornecida pela Petrobras foi melhorado. A aplicação implementa uma aproximação de propagação de onda acústica, que é a referência atual para ferramentas de imagens sísmicas. A aplicação tem sido amplamente aplicada para geração de imagens de reservatórios de petróleo e gás nos últimos cinco anos. Esses mecanismos de propagação acústica devem ser continuamente portados para o mais novo *hardware* disponível para manter a competitividade.

A.4.1 Padrão de Acesso à Memória para Melhorar a Localidade dos Dados

As arquiteturas de computador atuais fornecem *caches* e *prefetchers* de *hardware* para ajudar os programadores a gerenciar dados implicitamente (LEE et al., 2010). A técnica de *loop interchange* pode ser usada para melhorar o desempenho de ambos os elementos, trocando a ordem de dois ou mais laços. Essa técnica também reduz os conflitos do banco de memória, melhora a localidade dos dados e ajuda a reduzir o fluxo de uma computação de matriz. Dessa forma, mais dados que são buscados para as memórias de cache são efetivamente acessados, a reutilização de dados nas *caches* é aumentada, e os *prefetchers* de linha de *cache* são capazes de buscar dados da memória principal com mais precisão. Nessa aplicação, temos três laços que são usados para calcular o estêncil. Eles podem ser executados em qualquer ordem sem alterar os resultados. A sequência de laço padrão é **xyz**.

A sequência do laço foi alterada de **xyz** para todas as permutações possíveis. O laço mais externo é aquele que foi paralelizado usando *threads*. A sequência de laço **zyx** tem um desempenho melhor e resultados de taxa de acertos combinada no Broadwell. O aumento de desempenho comparado com a versão **xyz** é de 5,3 $\times$. Essa sequência é melhor que outras porque os dados são acessados de uma maneira que se beneficia mais das *caches*. A taxa de acertos da *cache* L2 foi melhorada de 14,9% para 77% quando a sequência do laço foi alterada para **zyx**. O impacto do cache L3 também melhorou de 76,2% para 92% no Broadwell. No entanto, esse não foi o caso do cache L1, já que sua taxa de acertos diminui de 82,3% para 73%. Na Knights Landing, a versão **yzx** apresenta melhor desempenho e de taxa de acertos combinada. Os ganhos de desempenho foram de até 3,9$\times$, mostrando que essa otimização impacta menos no desempenho da arquitetura Knights Landing do que da Broadwell. A taxa de acertos na *cache* L2 foi melhorada de 9,6% para 95,9%. Embora a taxa de acertos da *cache* L1 diminua em ambas as arquiteturas, isso mostra que a melhor opção visando o desempenho é aumentar as taxas de acerto da *cache* de último nível, mesmo quando a taxa de acertos do cache de qualquer outro nível diminui.

A.4.2 Explorando SIMD para Computação de Ponto Flutuante

Abordagens recentes de *hardware* aumentam o desempenho integrando mais núcleos com unidades mais amplas de SIMD (única instrução, múltiplos dados) (SATISH et al., 2012). Essa técnica de processamento de dados, denominada vetorização, possui unidades que executam, em uma única instrução, a mesma operação em vários operandos. Para maximizar a eficácia da vetorização, os endereços de memória acessados pela mesma instrução, em iterações de laço consecutivas, também devem ser consecutivos. Dessa forma, o compilador pode carregar e armazenar os operandos de iterações consecutivas usando uma única instrução *load* / *store*, otimizando o uso da memória *cache*, já que os dados já são buscados em blocos da memória principal. Os processadores mais recentes introduzem o suporte para as instruções `gather` e `scatter`, que reduzem a sobrecarga de carregar / armazenar endereços de memória não consecutivos. No entanto, o desempenho ainda é muito maior quando os endereços são consecutivos. Nesse contexto, o código-fonte foi modificado de modo que os endereços de memória acessados pela mesma instrução fossem consecutivos ao longo das iterações do laço.

As instruções AVX (*Advanced Vector Extensions*), que é uma extensão de arquitetura de conjunto de instruções para usar unidades SIMD para aumentar o desempenho dos cálculos de ponto flutuante foi utilizada. Essas instruções usam unidades específicas de ponto flutuante que podem carregar, armazenar ou executar cálculos em vários operandos de uma só vez. Como descrito anteriormente, a eficiência do AVX é melhor quando os elementos são acessados na memória de forma contígua, pois podem ser carregados e armazenados em blocos. O aumento de velocidade mostrado é relativo à sequência do laço sem o AVX. As sequências **yzx** e **zyx** têm melhores resultados porque têm mais elementos sendo acessados de forma contígua. O ganho de desempenho difere de arquitetura para a arquitetura. Na Broadwell, a melhoria foi de até 1,4×. Na arquitetura Knights Landing, a melhoria foi de até 6,5 ×. Essas diferenças se devem ao tamanho da unidade de vetor de cada arquitetura e ao número de núcleos usados.

A.4.3 Melhorando o Balanceamento de Carga

Algumas aplicações têm regiões com diferentes requisitos de carga de computação, por exemplo contornos, potencialmente causando irregularidade no tempo de computação entre as *threads*. O tempo para executar um aplicação paralela é determinada pela tarefa que leva mais tempo para ser concluída e, portanto, pelo núcleo com a maior quantidade de trabalho. Portanto, ao distribuir o trabalho de maneira mais uniforme entre os núcleos, o tempo de execução da aplicação é reduzido. As técnicas de balanceamento de carga reduzem essas disparidades e, portanto, melhoram o uso e o desempenho dos recursos. No contexto de sistemas com *multi-core* e *many-core*, o balanceamento de carga é ainda mais importante devido a um grande número de núcleos.

A especificação OpenMP possui uma diretiva para indicar se o escalonamento é estático, dinâmico ou guiado. O escalonamento estático é o padrão, e atribui iterações as *threads* seguindo um algoritmo round-robin antes do início do cálculo. As abordagens dinâmicas e guiadas distribuem o trabalho durante o tempo de execução de acordo com as solicitações das *threads*, mas, no dinâmico, todos os blocos têm o mesmo tamanho, enquanto no guiado, os primeiros blocos são maiores e seu tamanho diminui ao longo das iterações. Além disso, o *loop collapse* também ajuda a melhorar o balanceamento de carga, pois aumenta o número total de iterações particionadas nas *threads* ao reduzir dois ou mais laços.

o impacto de diferentes políticas de escalonamento do OpenMP com e sem o *loop collapse* foi investigado. Nos experimentos sem *collapse*, o laço externo foi dividido entre as *threads*. A melhor aceleração sem *collapse* no Broadwell estava usando a política guiada. Foi até 1,06$\times$ mais rápido que o padrão. Essa abordagem é útil para aplicações cuja carga de trabalho muda em tempo de execução ou possui algumas regiões com cargas de trabalho desequilibradas. No Knights Landing, o melhor desempenho foi o uso do escalonamento estático com um aumento de velocidade de 1,01$\times$.

Uma maneira de melhorar o desempenho com trechos maiores é com *collapse*. A ideia é que, com mais trabalho a ser dividido entre as *threads*, pedaços maiores podem ser usados, mantendo um bom balanceamento de carga. A fim de investigar isso, os dois laços mais externos foram agregados e diferentes combinações do escalonador foram avaliadas. Os resultados mostram que, para o Broadwell, a melhor política é o escalonador estático. No Knights Landing, a melhor política com *collapse* é a guiada. O desempenho no Broadwell foi melhorado em até 1,04$\times$, enquanto no Knights Landing em até 1,11$\times$.

A.4.4 Otimizando a Afinidade de Memória

O objetivo dos mecanismos de mapeamento é melhorar o uso de recursos, organizando *threads* e dados de acordo com uma política fixa, onde cada abordagem pode direcionar diferentes aspectos para melhorar. Por exemplo, existem técnicas focadas em melhorar localidade, para reduzir os erros de *cache* e os acessos à memória remota, bem como o tráfego nas interconexões inter-chips (CRUZ et al., 2016). Outras políticas buscam uma distribuição de carga uniforme entre os núcleos e controladores de memória.

Na arquitetura de Broadwell, o melhor ganho de desempenho foi de 1,6×, alcançada para técnica *round-robin* com *interleave* para a versão xyz. Ele combina uma distribuição de *threads* uniforme com uma distribuição equilibrada das páginas. O mapeamento de *threads* e de dados foi combinado porque, na maioria das aplicações, a eficácia do mapeamento de dados depende do mapeamento de *threads*.

Na Knights Landing, as técnicas avalidas foram as mesmas que na Broadwell mais *interleave* usando MCDRAM. O melhor ganho foi 4,4× para *scatter* com *interleave* MCDRAM para a versão xyz. Podemos observar que, na Knights Landing, as melhores melhorias geralmente acontecem com políticas que se concentram no balanceamento de carga, como o mapeamento de *threads* de *scatter* e o mapeamento de dados de *interleave*.

A.5 Conclusão e Trabalhos Futuros

Este trabalho realiza uma análise detalhada dos gargalos de desempenho em *multi-core* e *many-core* e otimiza seu desempenho usando diferentes estratégias. Avaliamos 18 aplicações e usamos contadores de desempenho de *hardware* para coletar medições precisas do impacto real de diferentes fatores que influenciam o desempenho. Essa avaliação mostrou que algumas métricas ajudam a entender o desempenho das aplicações, mas há casos em que uma métrica sozinha não é representativa.

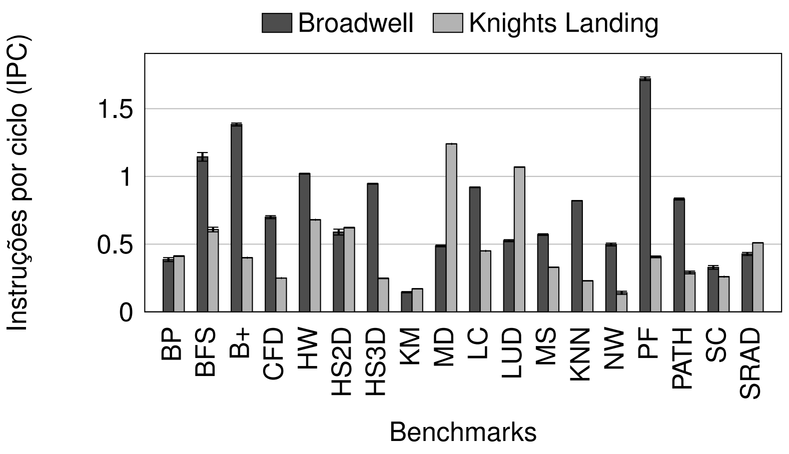
Usamos nosso estudo de gargalos de desempenho como base para otimizar um modelo de geofísica do mundo real. Aplicamos as seguintes técnicas de otimização: (1) *loop interchange* para melhorar o uso da memória *cache*; (2) vetorização para aumentar o desempenho dos cálculos de ponto flutuante; (3) *load balancing* e *collapse* para melhorar o balanceamento de carga; e (4) mapeamento de *threads* e dados para melhor usar a hierarquia de memória. Essas otimizações também podem ser aplicadas a outras aplicações e arquiteturas.

Em nossos experimentos de otimização de desempenho, mostramos que *loop interchange* é uma técnica útil para melhorar o desempenho de diferentes níveis de memória *cache*, sendo capaz de melhorar o desempenho em até 5.3× e 3.9× na Broadwell e Knights Landing, respectivamente. Essas melhorias ocorreram porque a taxa de acertos da cache de último nível foi aumentada em até 95,9%. Além disso, alterando o código de modo que os elementos sejam acessados de forma contígua entre as iterações do laço, o código foi vetorizado, o que melhorou o desempenho em até 1.4× e 6.5×. As técnicas de balanceamento de carga e de *collapse* também foram avaliadas, mas o balanceamento da aplicação atenuou as melhorias de desempenho. Essas técnicas melhoraram o desempenho do Knights Landing em até 1,1×. As técnicas de mapeamento de *threads* e dados também foram avaliadas, com uma melhoria de desempenho de até 1,6× e 4,4×. O ganho de desempenho do Broadwell foi de 22,7× e do Knights Landing de 56,7× em comparação com uma versão sem otimizações, mas, no final, o Broadwell foi 1,2× mais rápido que o Knights Landing.

A.5.1 Trabalhos Futuros

Os trabalhos futuros se concentrarão em propor mecanismos automáticos para otimizar o desempenho de arquiteturas *multi-core* e *many-core*. Além disso, a avaliação dos gargalos de desempenho será expandida usando arquiteturas mais recentes e avaliando o consumo de energia.

Figure A.1: Desempenho de diferentes arquiteturas (maior IPC é melhor).



Fonte: O Autor.