Accelerating Computational Finance Simulations with OpenCL: a case study

Michail Papadimitriou

Accelerating Computational Finance Simulations with OpenCL: a case study

Master's Thesis in Embedded Systems

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Michail Papadimitriou

19th September 2016

Author
Michail Papadimitriou

Title
Accelerating Computational Finance Simulations with OpenCL: a case study

MSc presentation
27th September 2016

Graduation Committee

Prof. Dr. D. H. J. Epema (Chair)	Delft University of Technology
Dr. Ir. Ana Lucia Varbanescu (Supervisor)	University of Amsterdam
Dr. Anna Villanova	Delft University of Technology
Joris Cramwinckel MSc	Ortec-Finance

Abstract

Nowadays, we live in an era where high performance is in particular demand in a very broad variety of fields. For fulfilling these needs there is, a large selection of high performance computing platforms exists. These platforms vary in terms of architecture as well as their flexibility in terms of programming and programming models. Therefore, programmers need to be able qualify their needs and align them to the corresponding platform and tools.

Computational finance is an area where being able to adapt to the constantly increasing amount of data, as well as reducing simulation times is a primary concern. Many computational finance applications, such as option pricing, algorithmic high frequency trading and risk management have their own HPC requirements.

In this thesis, we investigate the computational requirements of a scenario based ALM application, which is part of a commercial product offered by Ortec-Finance. Also, we propose a novel OpenCL implementation, optimized for the Intel Xeon Phi co-processor. Further, we evaluate the performance portability of the proposed solution to other platforms such as a high-end CPU and an NVIDIA GPU.

In general, with deploying simple optimization techniques we manage to achieve a speed-up up to 150x for Intel Xeon Phi, compared with the initial sequential implementation. Also, we prove that for our application, OpenCL yields significantly improved results in comparison with OpenMP on the Phi. In addition, we show that by following basic optimization guidelines, a certain level of performance can be preserved among different platforms (CPU, GPU, and Phi). Overall, our results show that our OpenCL ALM implementation is suitable for multiple accelerators, and it promises to yield significant performance improvements compared with current state-of-the-art implementations.

Acknowledgments

During my studies at TU Delft, I discovered my keen interest in High Performance Computing. Thus, I am pleased that I had the opportunity to undertake my thesis in a relevant topic and strengthen my knowledge. Moreover, I am glad that during my collaboration with Ortec-Finance, I faced problems emerging from the finance industry. Therefore, I am hoping that my work can successfully lead to adoption to new technologies for Ortec-Finance.

I would like to thank everyone who supported me for the past five years during my studies. Also, I would like to acknowledge my friends that got involved with this thesis; special thanks to Achilea Athanasiou for proofreading it. In addition, I would like to express my gratitude to my supervisor dr. ir. Ana Lucia Varbanescu, for her help and guidance through the whole duration of my thesis. Her input proved to be valuable for the progress of this thesis. Last but not least, I would like to thank Ortec-Finance and all people there, for providing me with all the required knowledge and means for completing this thesis. Specially, I would like to thank my supervisor from Ortec-Finance, Joris Cramwinckel, for his daily guidance and influence on the different aspects of this project.

Michail Papadimitriou

Delft, The Netherlands
19th September 2016

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Nowadays, multicore processors[1] can be found in the majority of computing systems around us. This trend exposes users and developers to an extreme amount of computational power. Despite all this computational power broadly available, from smartphones to servers, harvesting its full potential is still a very challenging process. Multicores may seem the next step for increasing the performance of applications with different characteristics, but moving away from the traditional sequential paradigm comes at a cost. This cost is associated with the increasing amount of different parallel programming paradigms, as well as the availability of multicore platforms with diverse architectural features. For example, there are platforms with a few powerful processing units, but also other platforms with thousands of less powerful processing units. Thus matching the right platform and programming model with the given application becomes a very challenging task.

Until recently, a gap existed between multicore platforms and programming models. OpenCL, a fairly new solution, promises to close this gap. OpenCL is a parallel programming model, targeting multicore platforms. It main advantage is functional portability. That means, applications developed and tested on one machine, can be easily ported to another. This portability, as many other technologies under development, comes at a cost: the performance of a portable OpenCL implementation may experience severe fluctuations from platform to platform.

The range of domains which benefit from all these HPC developments is quite extended. Fields of research such as physics, computer science, engineering and finance widely adopt multicores for their day to day processes. This approach allows them to innovate and enhance the quality of applications, products, and services they offer. Therefore, multicores are becoming essential for the fast development of all these fields.

Computational finance is one of the fields where high performance computing is in high demand to enable the use of the increasing amounts of data available.

[1] Across this thesis, we often use "multicore" as a shorter name for "multicore processor".

Also, as a very challenging area of business at the moment, computational finance has a lot of opportunities for cutting edge research. There is a clear need for better models that need to provide more simulations and comply with the continuously changing rules and regulations. In fact, recent statistics hve shown that the 10% of the of the TOP500 supercomputers are being used for computational finance applications[17].

According to the current trends, computational finance covers three main areas of interest: option pricing that targets organizations such as investment banks, high-frequency algorithmic trading for hedge funds, and actuarial science aiming to pension funds and insurance industry.

Scenario-based Asset Liability Management is related with risk and wealth management. Scenario-based ALM is a big part of a successful commercial product offered by Ortec-Finance. Ortec-Finance is the main collaborator of this project and is involved with financial consulting for pension funds. Ortec-Finance identifies an increased need for the ALM model to be able to perform much faster evaluations, along with an increasing number of economy trajectories and fiscal decisions. Therefore, they are extremely interested in using the most modern technologies and methods to improve the performance of their products in general, and of this specific kernel in particular.

Therefore, in this thesis, we investigate the suitability of multicore platforms for financial simulations such as the scenario-based ALM. Our research focuses on extracting a realistic test case, analyzing it, producing a novel OpenCL solution, and eventually increasing the number of simulated scenarios while decreasing simulation time. Also, we explore the performance portability of our proposed solution among different platforms.

1.2 Research Question

This research is set in the context of high performance computational finance. It is a complex investigation combining scenario-based ALM, multiple types of multicore processing units, different programming models, all in the search for the right solution for Ortec's speed requirements. Therefore, our main research question is:

Is OpenCL a suitable programming model for high performance computational finance on modern multicore processors?
We tackle this question by answering the following subquestions:

- What is a good example of a computational finance kernel that requires HPC?
- Can we propose a novel OpenCL implementation of the selected kernel?
- What are the optimization strategies for the proposed OpenCL solution which can improve the performance on the Intel Xeon Phi co-processor?

- How portable is the proposed OpenCL solution among different platforms of different vendors?

1.3 Contributions

The main contributions of this thesis are the following:

- We survey the current trends in terms of modern multicore platforms and programming models.
- We extract a case study model from a commercial product for financial modelling offered by Ortec-Finance.
- We explore the potential of Intel Xeon Phi co-processor for computational finance applications.
- We propose a novel OpenCL implementation of our representative kernel and we analyze its performance in detail.
- We investigate and asses the performance portability of the proposed OpenCL solution on other platforms.

1.4 Thesis Organization

This thesis is organized as follows.

Chapter 1 provided the necessary information regarding the broader context, the main research questions and contributions of this thesis.

Chapter 2 presents the basic background information required to understand our research. We discuss the current trends for multicore systems and their distinct characteristics in terms of programmability and architectural design. Then, the different parallel programming models such as OpenCl, CUDA and OpenMP are presented. Finally, a short introduction to the main field of computational finance and its main applications such as risk management, high frequency algorithmic trading and option pricing are presented.

In Chapter 3 we introduce the related work relevant to this thesis. We present related work on the trends of HPC for computational finance simulations - including options pricing, actuarial science and specific accelerations of ALM simulations. We also discuss work related to the utilization of NVDIA GPUs, as well as work based on accelerating applications with Intel Xeon Phi and OpenCL. Finally, we present work that studies OpenCL's performance portability.

Chapter 4 presents our experimental setup, i.e., platforms, compilers and tools used for this project, as well as all the performance metrics used for evaluating performance.

Chapter 5 contains all the analysis of ALM, the computational finance kernel we selected for acceleration, along with its computational requirements and characteristics. Also, the parallel OpenCL implementation is discussed, including the different optimizations that were used. Finally, all performance results are analyzed.

Chapter 6 analyzes the performance portability of the solution proposed in Chapter 5. Portability is evaluated on three different platforms, and our analysis focuses on the main application and platform characteristics that preserve a certain level of performance portability.

Finally, Chapter 7 concludes this thesis summarizing our main findings and suggestions for future work.

Chapter 2

Background

In this chapter we briefly present the main concepts needed to understand our research.

2.1 Concepts in Parallel Computing

This section presents several basic concepts in parallel computing: Amdahal's law Also: Flynn's taxonomy of parallel machines, levels of parallelism and models of parallel computation.

2.1.1 Amdahal's Law

Parallelization of applications may lead to potential bottlenecks. Thus, in 1967, Gene Amdahl, stated that the gain of sequential application parallelized in multiple processors is bounded by the fraction of code which runs sequentially[3]. This finding is widely known as *Amdahl's Law*.

Amdahl's law is captured by equation 2.1, where **s** is the sequential part of the program and *1-s* the part of the code which can be easily mapped in parallel. Eventually, *S* is the potential spee up in *P* processors.

$$S(P) = \frac{1}{s + \frac{1-s}{P}} \tag{2.1}$$

Today *Amdahl's law* is still one of the most critical means to measure the potential performance gains. Therefore, there are several papers introducing enhancements and revisions on the original law. Such a revision is introduced by J. Gustafon, which added the enhancement of scalable sized model to *Amdhal's law*[21].

2.1.2 Flynn's Taxonomy of Parallel Machines

Based on Flynn's taxonomy introduced in 1966[15], computer architectures can be classified regarding the instruction of data streams available by the processor. This

classification proved for be crucial on the development of multiprocessing CPUs and parallel programming. Themain categories (also see Table 2.1) are:

- **SISD** A standard uniprocessor with no parallelism exploited; a single instruction executes on data stored on a single memory location.
- **SIMD** A multiprocessing architecture for which a single instruction executes over multiple data in the same time.
- **MISD** An architecture in which multiple instructions operate on the same data memory location.
- **MIMD** An architecture in which multiple autonomous processors execute different instructions on different data.

Table 2.1: Flynn's Taxonomy for Parallel Machines

Flynn's Taxonomy		
	Single Instruction	**Multiple Instructions**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

2.1.3 Levels of Parallelism in Software

Three levels of parallelism that can be exploited in applications are presented in this section.

Instruction-level Parallelism

Instruction-level parallelism allows more than one instruction of the same data stream to be executed during a single clock cycle. Usually, ILP is exploited by the hardware or the compiler, but it is often hindered by data dependencies or control dependencies.

Task-level Parallelism

Task-level parallelism can be instructed by the compiler or the end user and is managed through the compiler and the hardware. It allows multiple threads or instructions from the same application to be executed in parallel, but it can be limited by synchronization overhead among the threads[52].

Data-level Parallelism

There are lot of applications where the same operations are applied to different data items. If there are no any dependencies between data elements, these operations can be performed independently among different processing units in parallel. This approach is commonly known as data parallelism.

2.2 High Performance Computing Platforms

Nowadays, there is a very large variety on hardware platforms, targeting applications with diverse characteristics. Choosing the appropriate platform for developing an application involves a combination of different factors such as the application characteristics and the required performance in terms of power consumption or potential speedups. Thus, for fulfilling these requirements, a deep understanding of the key features of each possible platform is needed.

Four platforms of diverse characteristics are presented in this section: general purpose processors (GPPs), co-processors, general purpose graphic processor units (GPGPUs) and field programmable arrays (FPGA). Also, for each category, a cutting edge platform is discussed in-depth along with its unique features in terms of architecture and microarchitecture.

As Intel Xeon Phi consists the main development platform for this project, we take a look in its history, the upcoming new architecture and the main differences with the 1st generation of co-processors that are currently available for commercial use.

2.2.1 General Purpose Processors (GPP)

General purpose processors can be found in many different machines such as servers, laptops and desktops. Usually these processors are designed to serve most workloads with good performance and average power consumptions. Today, the majority of these processors contain several cores and even integrated graphics. Other specific characteristics of GPPs are high clock frequencies, high power consumption, shared memory hierarchy and multi-level caching.

An example of such a processor is the Intel Xeon CPU family which offers up to 18 cores on a single chip, along with clock frequencies up to 4.4GHz[28].

The majority of the semiconductor vendors offering products that fall in this group are: Intel, Apple, AMD, IBM, ARM and Imagination Technologies.

Intel Skylake Microarchitecture

Intel Skylake is the latest generation of GPPs offered by Intel [2][27], its successor is going to be Kaby Lake coming in the end of 2016. Intel produces different products based on the Skylake architecture, targeting applications based on desktop, mobile client systems and server workstation systems.

Skylake based products are built with Intels 14 nm manufacturing process. These processors usually offer from two up to eighteen physical cores for the Xeon family. In addition, some of these cores are enhanced with the Hyper Threading technology (HT), which allows each physical core to be recognized as two logical cores. This enhancement increases significantly the parallel processing power of these products as more elements are able to be processed in parallel on each core. An overview of the Skylake microarchitecute is illustrated in Figure 2.1.

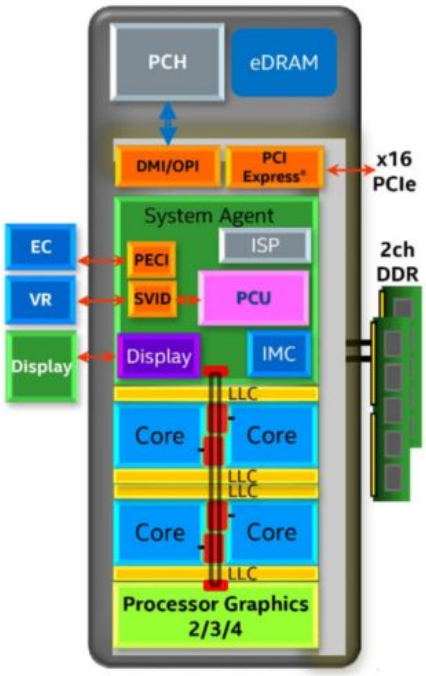

Figure 2.1: Intel Skylake microarchitecute overview [24]

Skylake offers three levels of caching. These levels correspond to an L1 of 64KB per core, L2 of 256 KB per core and 8192 KB of shared L3 memory. Skylake fully supports instruction sets like MMX and FMA3. In addition, it contains all the latest instruction set extensions such as SSE4.2, AVX, AVX-512, VT-x and MPX. Therefore, SIMD operations became more efficient and flexible with Skylake.

Another important aspect of Skylake is that, like its predecessor(Haswell), contains an Integrated Graphics Accelerator unit. These units vary in features depending on the product. For example, Iris Pro Graphics 580 is the high-end model with peak floating point performance of 1152 GFLOPS and 1GHz clock frequency.

A product from the Skylake family relative to this project is the Intel Xeon E3 V5-1230 without integrated graphics available. The processor has a CPU clock frequency of 3.4 GHz, 4 physical cores, Intel Hyper Threading technology available so 8 logical cores, and a peak floating point performance of 130 GFLOPS. Moreover, the chip has a peak bandwidth of 34.1GB/s. Also, based on the specifications provided by the chip manufactures, its power consumption is 80W.

2.2.2 General Purpose Graphics Processing Units (GPGPUs)

The continued demand for rendering high quality graphics made the field of General Purpose Graphics Processing Processing Units (GPUs) evolve at a very fast pace. In addition, the evolution of the multi-billion gaming industry, where real time high quality graphics are need, has lead to a lot of cutting edge research on GPUs. As a result, GPUs are now capable of running thousands of threads over hundreds of physical cores.

The breakthrough in this field was the extension of GPUs to be enabled as fully programmable hardware. The support of floating operations, the ability of handling parallel computational problems and the continuous interest made GPUs HPC target for a large group of applications.

Nowadays, several different vendors offer GPUs as either integrated graphics

solutions or accelerators. Examples are Intels processors with integrated Iris HD graphics and ARM's Mali T60x architecture. The main vendors providing GPUs are NVIDIA, AMD, Intel, ARM, Imagination Technologies and Qualcomm.

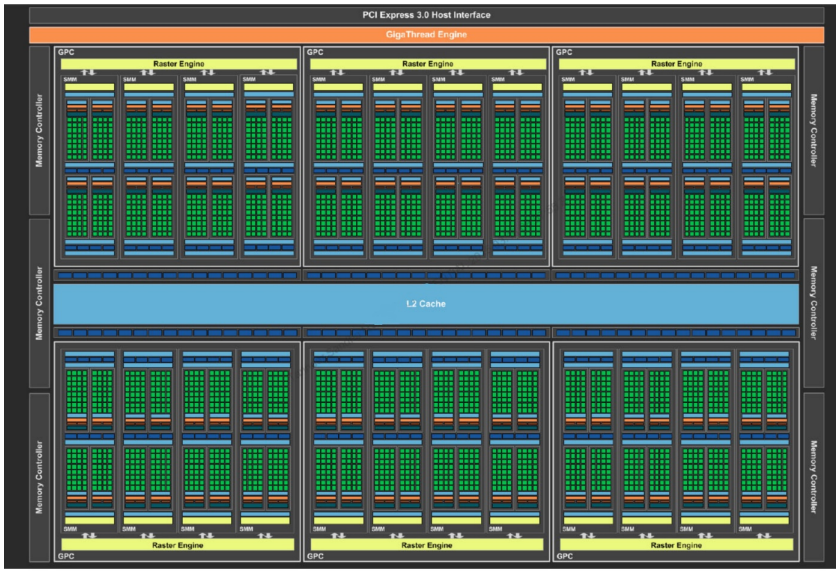

Figure 2.2: The NVIDIA GeForece Titan-X Maxwell architecture [9]

GeForce GTX TITAN X

Titan X uses the 28nm Maxwell architecture on a 551 mm chip[9]. Also, Titan X contains 3072 CUDA cores and 24 streaming multiprocessors, organized in 192 texture units grouped into six Graphics Processing Clusters (GPCs). Its base clock is at 1000MHz and the memory clock is at 3505MHz. In addition, Titan X can offer a theoretical peak of 6.2 TFLOPS with 12 GB of GDDR5 RAM and a bandwidth of 336GB/s. In terms of power consumption, the card can consume 275W through an 8-pin and a 6-pin power connection. Moreover, Titan X has a memory width of 384-bit and an L2 cache of 3MB. Therefore, it keeps a 32K:1 cache:ROP ratio for the Maxwell 2 architecture, which allows the GPU to more cache before try to keep operations of the memory bus. The card can be connected on a PCI Express 3.0 slot.

As it can be seen on Figure 2.2 in terms of architecture, there are six GPCs, which group smaller number of streaming multiprocessors together. Each SMM (Maxwell Streaming Multiprocessor) is a group of 128 CUDA cores, 8 TMUs (Texture Mapping Units), L1 caches, schedulers and dispatchers. Moreover, there are 96 ROPs (Raster Operations Pipeline) enabling the processing of 96 color samples. Finally, a 64-bit wide memory controller for each cluster can be found.

2.2.3 Co-Processors: Intel Xeon Phi

Accelerators are used to accelerate parts of the main application. Their very basic nature is to execute tasks in a way such that the main processor doesn't take part in the computation. Thus, the main outcome is that CPU can save time and power by offloading specialized tasks to specialized processing units. For example, GPUs started as accelerators for rendering complex graphics but, nowadays, they are used as accelerators for many HPC applications[13].

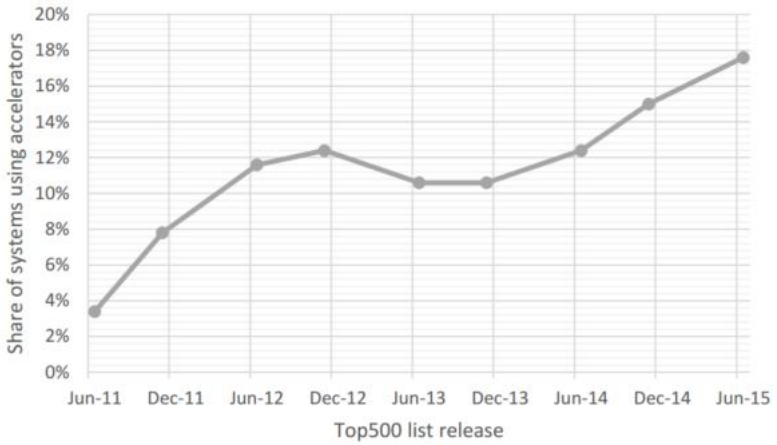

Figure 2.3: Share of TOP500 supercomputers using accelerators/co-processors [13]

As it can be seen on Figure 2.3, the share of systems which use accelerators on the TOP500 list is constantly increasing with an encouraging rate. It can be noted that in a of four years period the use of accelerators increased from 4% to 18%. In addition, it's worth mentioning that in June of 2015 the number one supercomputer (Tianhe-2) on the same list, was equipped with Intel Xeon Phi co-processors.

Table 2.2: Co-processors/Accelerators used by the TOP500[13]

Co-processors/Accelerators	
Vendor	**Model**
Intel	Xeon Phi
NVIDIA	Tesla K20, K40, K80
AMD	FirePro S9150, S9050

Table 2.2 represents the main providers of co-processors and some indications on specific models. Each of these vendors provide their own support in terms of SDKs, compilers and debugging environments. In addition, they have their own programming constructs for their specific hardware. For example, NVIDIA offers CUDA while Intel offers TBB and Cilk Plus. On the other hand, OpenCL provides

a common ground between the different vendors. For the needs of this project, initially we concentrate on Intel Xeon Phi. Therefore, a more in depth description of this specific co-processor follows.

Intel Xeon Phi Co-Processor

In this section a small introduction to Intel Xeon Phi coprocessor is presented, including information regarding the early stages of MIC architecture as well as the current Intel trends is provided.

History

Back in 2006, Intel unveiled the Tera-scale Computing Research which aimed to scale processors from few cores to many. A year later Larrabee was introduced, as an early GPGPU prototype based on x86 cores, only to be discontinued in 2010.

Table 2.3: Intel Milestones on MIC development

Project	Year	Description
TRP	2006	Terra-Scale Research Processor
Larrabee	2007	GPGPU prototype
Knights Ferry	2010	MIC prototype
Knights Corner	2013	1st Generation Xeon Phi
Knights Landing	2015	2nd Generation Xeon Phi
Knights Hill	2017	3rd Generation Xeon Phi

In 2010 Intel introduced the Knights Ferry, which was Intel's first attempt to enter the high performance computing accelerators market. Knights Ferry is considered as a baseline design for the Xeon Phi era and was succeeded by the Knights Corner in 2011. Knights Corner features more cores, more memory and a peak performance that reached one teraflop. Since, 2014 Intel has announced the latest addition to the Xeon Phi family which is named Knights Landing. Commercial distribution started in summer of 2016.

Knights Corner is Intel's 1st generation of Intel Xeon Phi co-processors [25].

With Knights Corner, Intel intended to enter the market with the 1st generation of Xeon Phi co-processors for commercial use.

Intel Xeon Phi [25, 31] contains 57 to 61 cores, depending on the supplied model. As it can be seen in Figure 2.5, these homogeneous cores are connected together through a high speed bidirectional ring. Also, one of the unique characteristics of the MIC design, is that these cores are based on the 1995s Intel P54C Pentium architecture. Although, this traditional design is enhanced and upgraded with 64-bit instructions and 512-bit vector instructions. In terms of caching, Phi has

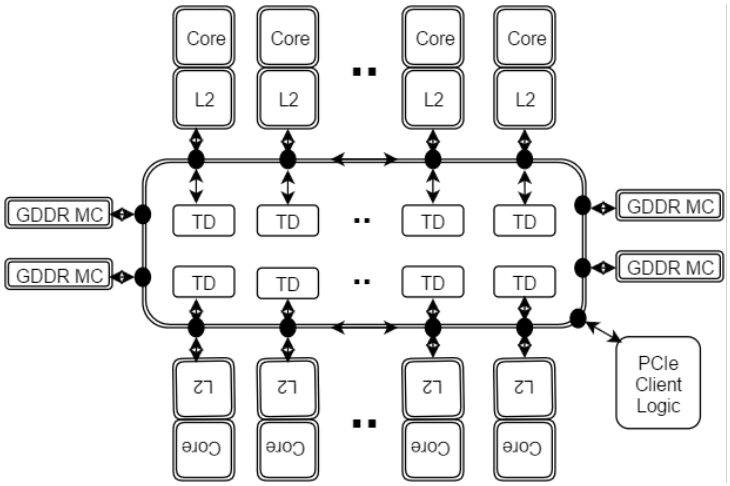

Figure 2.5: Microarchitecture of 1st generation Intel Xeon Phi

two levels of cache hierarchy: L1 cache has 32KB available for data and another 32KB for instructions, along with 512KB L2 cache dedicated for every core[14]. In terms, of floating point performance, the Phi can provide up to 2100GFLOPs and 1100GFLOPs, for single and double precision operations, respectively. The Intel Xeon Phi co-processor runs under a Linux based micro-OS.

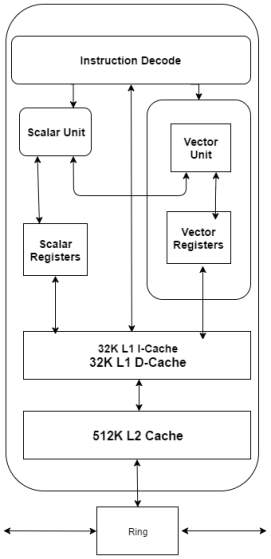

Figure 2.4: Single core configuration for Intel Xeon Phi

It is worth to mention two main components of the Intel Xeon Phi co-processors. Firstly, the fact that single core design is based on the P54C Intel architecture. Thus, the cores cannot be clocked on the current higher frequencies. Therefore, as a result Xeon Phis single core performance is lacking significantly in comparison to to any state of the art modern CPU. On the other hand, the support of up to four threads per core has proved to be helpful for its performance[13]. Although, attention needs to be paid as instructions cannot be issued from the same thread consequently in the same FU and so, always at least two threads per core need to be active at all times.

For flexibility on the development of applications, Intel Xeon Phi supports two execution modes. In native mode, compilation only occurs in the

co-processor and then the execution of application is completely performed on the Phi. On the other hand, applications can run either in native or in offload mode. This, offloading mode is supported by multiple paradigms like OpenMP, MPI and OpenCL, while allows only the computationally intensive parts of an application to be executed on the device rather than the compiler kernel.

Knights Landing is Intel's 2nd generation of Intel Xeon Phi co-processors[1] [5][6]

Knights Landing is the successor of KNC design while uses the latest technologies to increase efficiency and performance. In more detail, KNL is manufactured with the 14*nm* process technology and features up to 72 cores. In addition, for the first time 16*GB* of on-chip Multi-Channel DRAM (MCDRAM) are available which lead up to five times higher bandwidth compared to KNC. Also, with the 72 cores enabled, the processing capabilities can reach 32 FP64 FLOPs/core by using 2 AVX-512 vector units per core. Knights Landing replaced the on-die high speed ring interconnection with mesh. This modification allowed higher bandwidth between cores and memory.

For the KNL, Intel decided to introduce the concept of tiles. Tiles as can be seen in Figure 2.6 correspond to pairs of cores that are sharing 1MB of L1 cache. This alteration on the design can be beneficial as even with minimum data sharing between the cores, it is more likely to have instruction sharing. In addition, the number of elements is reduced (from 72 to 31 pairs) which potentially leads to higher bandwidth and lower latency. A more detailed view of the KNL architecture is represented in Figure 2.6, were all relationships between tiles, DDR4 memory, MCDRAM and PCIe3 are represented.

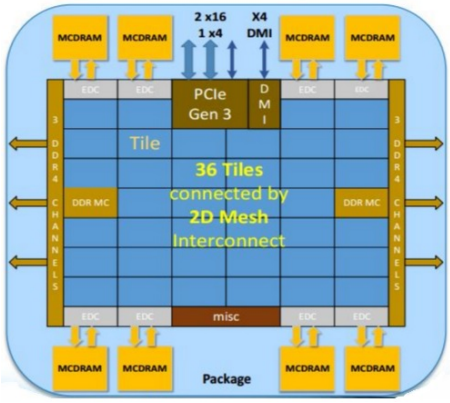

Figure 2.6: Intel Xeon Phi: KNL architectute overview [38]

2.3 Parallel Programming Models

Nowadays, a variety of parallel programming models and tools exist. This group of languages and tools provide enough flexibility for the user to exploit specific hardware characteristics and parallelism present in an application. For this reason, a

[1] Expected to launch in the third quarter of 2016

section containing information regarding the key features of such programming approaches follows. Open Computing Language (OpenCL), Compute Unified Device Architecture (CUDA), Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) are discussed.

2.3.1 OpenCL

OpenCL stands for Open Computing Language, which is a programming model targeting, emerged through the need for portable solutions in the multicore era. OpenCL was launched in 2008, and in 2016 it has reached version 2.2. Currently, OpenCL it's supported on a very broad variety of devices of different vendors such as CPUs, FPGAs, GPUs as well as co-processors like the Intel Xeon Phi and the Cell/B.E[20].

In Figure 2.9, an overview of the OpenCL host-device architecture is illustrated. The host device is responsible for performing the communication and manages the different Compute Devices available. These devices can be a combination of different platforms such as CPUs and GPUs. Thus, OpenCL enables a certain level of heterogeneity for the overall system. Also, these Compute Devices offer a number of Compute Units, which usually are a number of homogeneous cores. Finally, these Compute Units contain Processing Elements, responsible for the execution of OpenCL kernels.

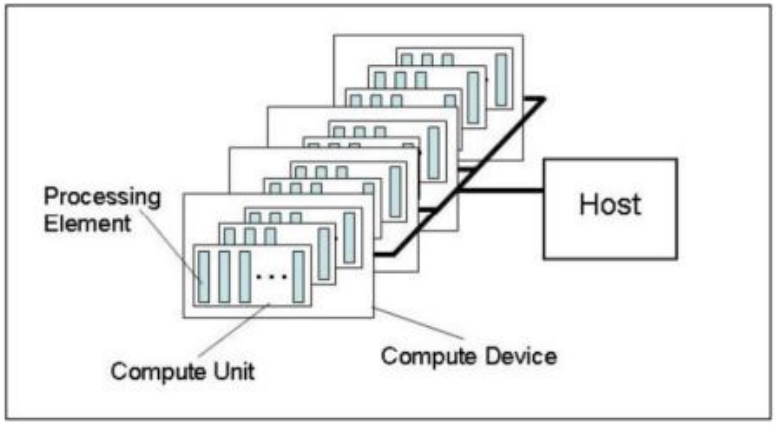

Figure 2.7: OpenCL Host-Device architecture [20]

OpenCL uses its own terminology, similar to the way CUDA does (Section 2.3.2). Thus, in extension of OpenCL functionality, Processing elements are mapped to work-items, while Processing Units are mapped to work-groups. Therefore, an instance of the OpenCL compute kernel is executed by every Processing Element present in a Processing Unit.

Memory in OpenCL is divided into two disjoint regions: the host memory and the device memory. The host memory which the memory available in the main host

and its behavior is out of the scope of the OpenCL and the device memory which is available by the OpenCL compute kernels. The device memory is grouped into four main parts: These parts are the global memory, the constant memory, the local memory and the private memory. The hierarchy of these different memory regions is illustrated in Figure 2.8.

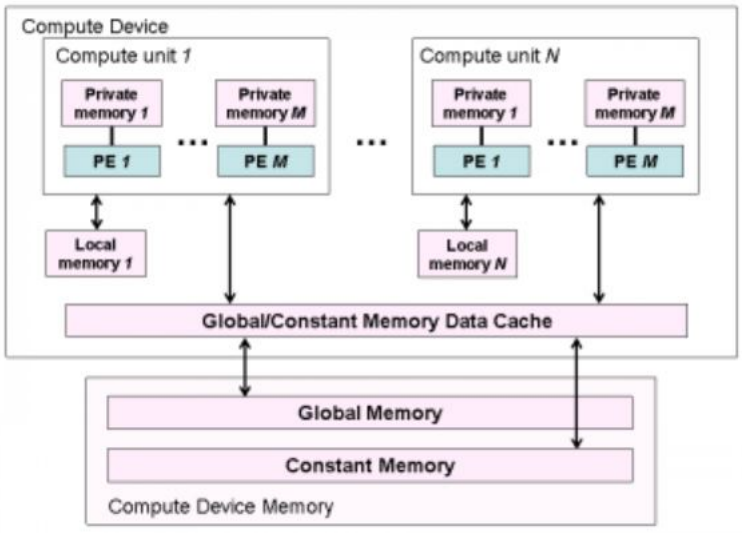

Figure 2.8: OpenCL device memory hierarchy[46]

In an era of heterogeneous code deployment, OpenCL offers the great advantage of functional portability. However, even with application being able to compile and run properly on platforms of different vendors, performance is not far from portable. In order to guarantee performance portability, individual optimizations need be performed for every platform[58].

2.3.2 CUDA

CUDA is a parallel computing platform and application programming interface (API), introduced by NVDIA in order to allow its graphics processor units to perform general purpose computing tasks. The initial release was on the June 23rd, 2017 for version for the compute capabilities, while currently NVIDIA just introduced compute capability 6.0 along with its latest architecture (Pascal).

CUDA works in the same host-device manner like OpenCL, where the CPU is responsible for all the communication and data transfers. On the other hand, on the device side, there can be more than one NVIDIA GPU devices available. Any computationally intensive parts of a program are mapped to the GPU with the form of a kernel. This so called kernel, is executed on the device side, which parallelizes its task among thousands of active threads. CUDA partitions GPU jobs in terms of threads. Threads usually grouped in the following way: threads forming blocks and blocks forming the grid. The grid of threads can have up to three dimensions

(instead of two for previous versions), 1024 maximum block dimensions for x/y and 64 for z.

CUDA compared to other parallel programming paradigms, CUDA incorporates several advantages. Firstly, it allows scatter reads with code, able to be to fetched from different addresses in memory. Also, the latest compute capabilities from 4.0 and 6.0, allow unified virtual memory and unified memory respectively. Finally, CUDA delivers fast access on shared memory among threads, which can allow better caching and allows a certain degree of freedom for the user. On the other hand, CUDA has some serious limitations. For example, CUDA code is only enabled on NVIDIA GPU platforms, unlike its alternative, OpenCL.

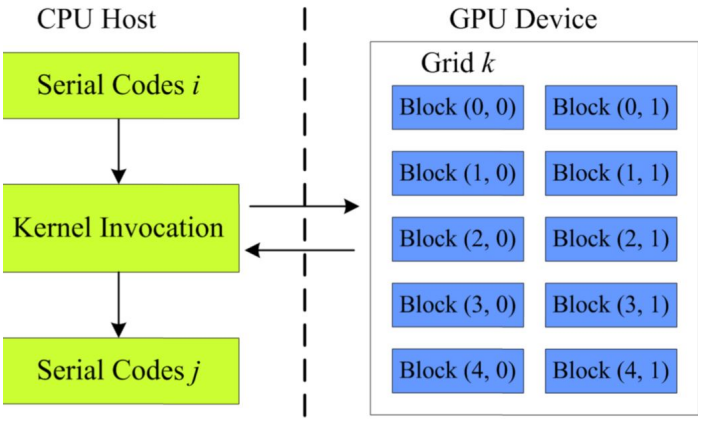

Figure 2.9: OpenCL Host-Device architecture [20]

In conclusion, CUDA constitutes a very accessible solution for writing parallel code for the average software engineer. Although, to be able to optimize and map code to hardware in such way that going to fully exploit its capabilities, needs understanding of the tools available and the specific architectural traits of the platforms is needed. Last but not least, CUDA through its libraries provides an extensive collection of optimized kernels and complementary libraries.

2.3.3 OpenMP

OpenMP which stands for Open Multi-Processing was introduced in 1997 by the Architecture Review Board(ARB). OpenMP is an application programming interface (API)[12]. Which supports different programming languages such as C, C++ and FORTRAN. OpenMP focuses on shared memory parallelism through multithreading. Parallelism is achieved with the use of compiler directives, library routines or environmental variables. Threads are being spawned and run concurrently on different cores.

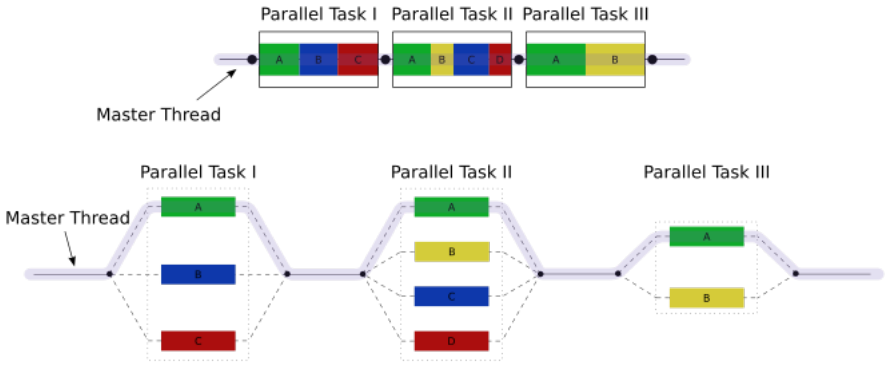

Figure 2.10: OpenMP example representation of multithreading with master thread forks off a number of threads [67]

OpenMP uses a fork-join model, in which a master thread forks. OpenMP is linked to a master thread which forks a number of slave threads. Then, these threads are assigned with specific tasks that are executed concurrently. This behaviour is illustrated in Figure 2.10. Threads are allocated based on usage, load or any other factor specified.

2.3.4 MPI

Message Passing Interface (MPI) is addresses the message-passing parallel programming paradigm, where multiple processes run concurrently and explicitly collaborate and communicate by exchanging messages.[16]. Currently MPI syntax and semantics are available for programming languages such as FORTRAN, C\C++ and Java.

MPI widely used for programming distributed memory computers. MPI is widely used to implement parallel applications for distributed memory machines. Due to its huge popularity, a lot of legacy MPI code exists. With the emergence of many-core systems with distributed (on-chip) memories, using MPI on-chip is an interesting attempt to provide high performance for many-cores like Intel Xeon Phi. Even with MPI standard introduced in 1992, currently the latest release is MPI-3.1 which was released in June of 2015[1].

MPI provides a large selection of collective communication routines. The structure of four of these routines is illustrated in Figure 2.11. These routines are scatter, gather, broadcast and reduction.

In the early stages of development, until the 1990s, MPI targeted distributes memory architectures. Eventually, as architecture trends shifted towards shared memory and distributed systems , MPI adapted in order to embed for functionality for the new designs. These adaptions lead on the current MPI state to be able to handle equally Distributed Memory, Shared Memory and Hybrid systems.

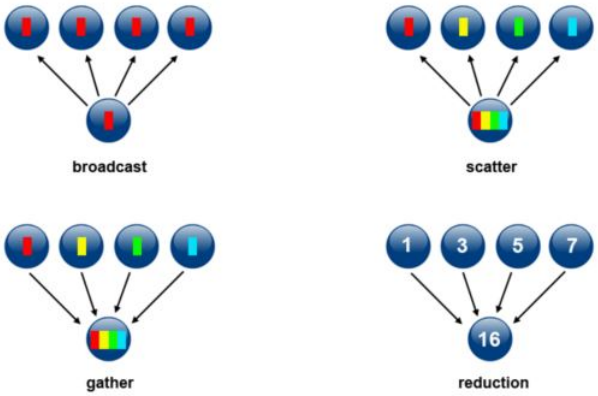

Figure 2.11: Types of collective communication routines [8]

2.4 Computational Finance Simulations

2.4.1 What is Computational Finance?

Computational finance is a domain which brings together computer science, mathematics and finance. In this domain, computationally intensive financial problems attempt to utilize computing systems to improve models and computations. This is achieved by practical numerical methods and techniques for analyzing economical trade-offs. This approach leads to a rather broad spectrum of applications, all requiring HPC capabilities. Also, almost 10% of the TOP500 supercomputers, are dedicated for computational finance purposes[17].

Core research in computational finance covers three main areas: option pricing that targets organizations such as investment banks, high-frequency algorithmic trading for hedge funds and actuarial science aimed towards pension funds, and the insurance industry.

2.4.2 Option Pricing

Options are means which can be used to harvest information emerging from stock movements. Options are formed from the underlying assets. These underlying assets usually take the form of a stock. Thus, the main difference with traditional investment approaches as stocks, is that stocks inherit their values.

These instruments i.e., the options, can be obtained by different means. Usually, these option contracts, provide purchasers or holders with a number of rights. These rights give permission but not the obligation to them for buying stocks with a given value for specific time period. The given value is called strike value while the time period is called expiration date. In the same manner, holders can trade their options in the pre-negotiated value up to a specific date[42].

There is a large selection of methods for option pricing. These methods are roughly 60% based on Montecarlo approaches, 30% based on PDEs and the final

10% is based on other semi analytic methods such as BlackScholes models.

2.4.3 High Frequency Algorithmic Trading

High Frequency Algorithmic trading is a computer based technique for performing buy and sell operations. These operations usually work with financial instruments such as bonds, stocks or options.

Stock market exchanges are highlighted by the tremendous demand for high speed calculations. The introduction of data feeds almost thirty years ago, initiated a new era of high frequency trading. Data feeds are constantly updated, while the stock market is open. These updates are regarding information based on the rates of ordered, canceled and executed stocks. Traders have live access to this information and they are responsible on take action on buy or sell. Also, responses to market events like currency exchange values, buy/sell of big companies or any other significant event, needs to be done in microseconds in order to achieve the best prices. TWith today's explosion of data, sophisticated computing systems are used to perform these actions[50][56].

2.4.4 Risk Management and Asset Lialability Management

Usually, financial decisions are accompanied by a specific strategy. These strategies are optimized with risk-return trade-off in mind. One needs to ensure that implementation, following the strategy and all its processes can be effectively tracked. Thus, as Ortec-Finance stated in[55], *"Risk Management is defined as the process of tracking risks that are related to the above-mentioned process, taking control actions at times when expected or unexpected events appear."* Thus, for individual investors or larger organizations, effective risk management is in particular demand, in order to achieve their goals and minimize the exposure to untolerated risks.

Asset Liability Management (ALM) emerges from a combination of risk management and strategic planning, for the long term scope. ALM policies are highly correlated with being able to draw meaningful insights for optimum asset and liability assessment. While, setting potential goals, these goals may differ up to a certain degree in order to comply with economic anomalies or different fiscal decisions. Usually, there is a very broad range of financial instruments of assets and liabilities. For example, loans, mortgages, bonds, stocks and equities, all consist assets. An example of a Risk Management lifecycle process is illustrated in Figure 2.12. On the other hand, deposits, foreign currency and borrowings are liabilities as they contain a certain amount of risk. Overall, ALM's ultimate goal is to allow for the investor to monitor the potential risk for a number of portfolios. These risks not only need to be handled in an optimum way, but also need to yield a decent amount on returns and earnings.

Insurance companies are main beneficiaries of ALM. Insurance companies, as they are always in need of accurate information regarding the potential risk(s) to

Figure 2.12: Risk Management Process [55]

make valuable decisions. Therefore, Ortec-Finance is a main application provider for such companies with a variety of applications, using sophisticated models. Those applications provide simulation frameworks, able to adapt and predict to rapid financial needs and changes. Thus, these frameworks tend to answer questions such as which strategy is optimal in terms of investment and interest rate policy or what impact should we expect for profitability and capital[47].

Chapter 3

Related Work

This section contains relevant related work for this project. First, we present a review on the high performance computing trends in the finance sector, including work on option pricing, risk management, actuarial science and financial applications using NVIDIA GPUs. Further, we discuss literature related to applications targeting the Intel Xeon Phi co-processor. Finally, we discuss related work in the area of OpenCL performance portability.

3.1 High Performance Computing in Finance

3.1.1 Option Pricing

Option pricing is the dominant area of HPC implementations in computational finance. This area deploys the largest number of computers, research and funds for potential research. One of the many reasons why option pricing is such a popular candidate for accelerations is the nature of the underlying models that are used[11]. For example, typical option pricing applications are Monte-Carlo, making them embarrassingly parallel and therefore suitable for GPU acceleration. Other approaches are PDE based, so GPUs are again a very convenient choice due to the amount of parallelism that can be exploited.

In *Analysis and Optimization of Financial Analytics Benchmark on Modern Multi- and Many-core IA-Based Architectures*[60] , Intel Xeon and Intel Xeon Phi architectures where tested for their suitability on derivative pricing methods. Different pricing methods such as analytical, lattice, Finite-Difference and Monte-Carlo were used. Each of these bases offered a broad variety of pricing such as Black-Scholes, Binomial, Crank-Nicolson and standard MCs in order to test the potential capabilities of both platforms. Their findings show that properly optimized compute bounded kernels on the Intel Xeon Phi can achieve 2.5x speedup, while bandwidth bounded kernels can achieve 2x speedup. Moreover, it was identified that the Ninja performance gap[1] between compiler optimizations and more

[1]Ninja performance gap is the difference in performance between naive and optimized imple-

deep algorithmic alternations, is 1.9x and 4x for Xeon and Xeon Phi architectures respectively.

Most research in HPC for option pricing has focused on using GPUs. The work presented in [41], consists of a solid example in the research carried out in this area. The constant maturity swap (CMS) derivative option algorithm was chosen. Their main focus was on fast accurate double precision normal distribution, parallel grid search algorithm for calculating implied volatility and an optimised data and instruction workflow. Findings showed that when compared to a sequential CPU approach, the run time performance improved to a factor of 10x when using NVIDIA GF100 architecture[41].

3.1.2 Actuarial Science and Risk Management

Actuarial science, risk and return management mostly involve products offered by pension funds, insurance industry and consulting firms. A typical application in this field is asset liability management (ALM),

Barrie Hibbert, which is a branch of Moodys Analytics offers scenario analysis for ALM. Specifically, projections on different portfolios and investment strategies are generated multiple time-steps. A variety of possible risks such as interest rates, inflation and liabilities are captured. For accelerating the calculation of their models, clusters with thousands of cores are deployed[45]. For their simulations, Barrie Hibbert use the Digipede Distributed Computing center which allows the deployment of thousands cores for their computation[51].

Other similar implementations in this field are offered by Towers Watson. Towers Watson introduced the Star models that can be used for generating economic scenarios under either a real-world or risk-neutral measure. Star ESG can be used by clients for setting investment strategy benchmarks, portfolio construction, ALM, pricing, capital setting and the market consistent valuation of options and guarantees.

3.1.3 Accelerating ALM simulations

As the combination of the ALM simulations, combined with scenario based analysis, OpenCL and Intel Xeon Phi, consists a novel approach, we break down the related work. Initially, we investigate other ALM related research on parallel systems. Then the Intel Xeon Phi in-depth and finally OpenCL applications on the Phi and potential optimizations.

As early as 2010, [10] proposed a first ALM approach adapting the Solvency II regulations for a parallel architecture. The authors use Monte-Carlo based trajectories for simulating the different possible projections of the economy for a forward risk neutral evaluation. For each projection of the economy, a certain number of fiscal operations such as applying investment strategies, rebalancing and asset evaluations are performed. In [10], a single real portfolio with horizon of 40 years and

mentations

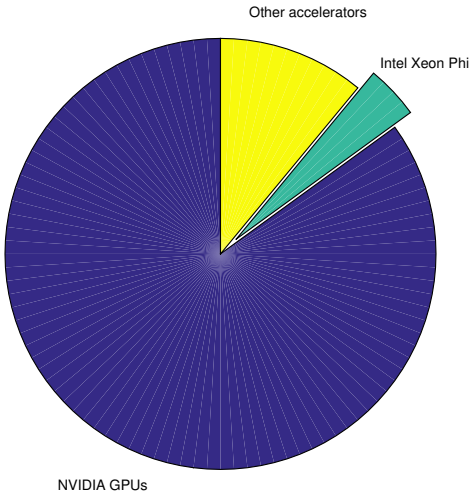

Figure 3.1: Market share in the financial sector for NVIDIA GPUs and other accelerators[43]

two assets (e.g. bonds and stocks) was used. Therefore, they were testing against two different approaches of evaluation. By using a parallel approach on a multi-node CPU cluster and proposing a Fortran 90+MPI extension, they manage to reduce significantly the execution times for the Monte-Carlo evaluations: for 6000 paths the simulation time reduced from 96 minutes to 8 minutes, while for 12000 paths the execution time reduced from 196 to 17 minutes. The data also suggested linear speedup when increasing the number of processors and significant decrease of the standard error.

Another parallel ALM approach was introduced in [19]. The authors provided a survey of the current trends and advances in ALM simulations back in 2011. The Stochastic Programming approach to ALM, along with risk averse modelling and solutions based on Interior Point Methods are discussed in this paper. The Object Oriented Parallel Solver (OOPS) combined with Structure-conveying Parallel Modelling Language (SPML) approach is discussed while aiming for scalability up to 1280 processors. The paper presents an evaluation of 12.8 million scenarios on 1280 processors in 3020 seconds. Furthermore, the included scalability analysis suggests sub-linear speedup for large numbers of processors: for up to 8 processors the resulting speedup was almost a factor of 7x, while for up to 512 processors it could increase to a factor of 27x.

3.1.4 NVIDIA GPUs in Computational Finance

Computational finance tends to use NVIDIA GPUs for accelerating its applications. As Figure 3.1 illustrates, according to NVIDIA, 85% of the clients that use accelerators, prefer NVDIA GPUs. In addition, for the remaining market share, 4% use Intel Xeon Phi co-processors and 11% prefer other methods.[43]. An increas-

ingly large group of companies in the financial industry that chose to use NVIDIA GPUs as accelerating platforms. Examples are JPMorgan[33], nag[40], jedox.[30] and SUNGARD.

JPmorgan moved from the traditional approach of using GPGPUs for their risk management calculation to deploy NVDIA GPUs for that purpose. Risk management computations include equity-derivative calculations. Thus, by moving to Tesla GPUs, they managed to accelerate the performance of their application by a factor of 40x, thus calculating the required risks from hours to just few minutes. This upgrade led to additional power efficiency benefits on their data centers[32].

Another important contribution in that field is from NAG, a supplier for HPC computational software. Usually, their products offer optimizations, parallelization and restriction of numerical applications in order to be able to follow cutting edge trends. Therefore, as they offer products related to algorithmic differentiation for mathematical derivatives aiming to the financial industry, HPC techniques consist their main choice for acceleration. The NAG Numerical Routines for NVDIA GPUs resulted in a 10x improvement for Monte-Carlo based index pricing[39].

3.2 Accelerating applications with Intel Xeon Phi

As far as we know, Intel Xeon Phi has never been used before for ALM simulations. Therefore, this section presents work related to the use of Xeon Phi in applications relevant for computational finance. Monte-Carlo simulations consist the closer match for option pricing. In [49], [22], [57] and [59] Monte-Carlo methods were used on Intel Xeon Phi.

For example, in [59], Intel used as a case study a Monte-Carlo European Option pricing scheme on an Intel Xeon Phi was the main focus of that research. Initially, before they began a stepwise optimization procedure, they introduced a novel sequential implementation for they option pricing scheme. Then, a number of specific optimizations have been performed. Firstly, they switched to an Intel compiler and used the Intel dedicated math library (MKL). Also, vectorization techniques were used before hand, along with OpenMP parallelization. Eventually, native mode, offloading mode and re-compiling on the Intel Xeon Phi, completed their stepwise optimization procedure. Overall, in terms of options per second, they manage to increased their performance results from 32.3 options per second to 496.520 options per second for the fully optimized version running natively on the co-processor.

3.3 OpenCL on Intel Xeon Phi

Most research focusing on Intel Xeon Phi, suggests as a programming paradigm either OpenMP in native or offload mode, or MPI. Thus, as we decided to use OpenCL on this project, research was carried out on a variety of applications using

OpenCL on the Phi. The research presented in [72], [71], [22] and [65] is therefore relevant. All this work demonstrates that OpenCL is indeed functional and usable on the Xeon Phi. Moreover, its functional portability enables easy-to-do comparisons with other platforms

For example, in [71], a bioinformatics case study is discussed. Specifically, a short read alignment algorithm implementation with OpenCL on the Intel Xeon Phi co-processor. An alternative to the short read alignment implementation BarraCUDA [34] is presented in that paper. Initially, the required modifications were made for porting the existing CUDA code in OpenCL. The results in general provide good scalability and linear speedup for increasing problem sizes. In addition, their OpenCL implementation outperforms the existing ones both on the CPU and the GPU. However, authors didn't mention if any specific optimization techniques were used.

3.4 OpenCL Performance Portability

While performance portability of OpenCL has been investigated in the past years, researchers are still far from producing a practical guide of ensuring good performance. Most of the research on performance portability either makes use of very simple cases such as matrix multiplications or FFTs.

For example, in [61], they investigate the potential performance portability between GPUs and CPUs with the use of a portable matrix multiplication kernel. While using a convention from the OpenCL community and by defining CEAN-style transformations, they manage to enable improved SIMD support for the CPUs. Also, they showed that for the proposed CEAN serialization method, their solution achieves performance comparable to native OpenMP solution.

In [53], a practical investigation for performance portability was considered. The novelty of this study was the diversity of platforms under investigation, where a CPU, 2 GPUs from different vendors and a Cell processor were used. This selection, provided enough variety between SIMD, SIMT and VLIW architectures. Then, for a selection of three kernels of different computational loads, the effect of the loop unrolling factor and thread block size was evaluated. Overall, the study showed that different platforms are extremely sensitive to optimization that might benefit other devices. Moreover, the CELL processor seems to be behave similar to CPU and vice-versa for the optimum configuration.

A more comprehensive study regarding performance portability was carried our by the department of microelectronics of TU Wien [54]. A selection of memory bounded linear algebra operations (vector copy, scaled vector addition, the inner product of two vector and matrix vector product) was implemented in OpenCL. Also, GPUs from AMD and NVIDIA, along with HPC-dedicated CPU and co-processor from Intel were tested. For evaluating all potential aspects of performance and its impact on the different platforms, a set of 1900 configurations was used on each device. These configurations affected the local work group size, the

global workgroup size and the data types (e.g. *double2*, *double4* etc). Based on their results, authors provide different tuning approaches for CPUs and GPUs respectively. For CPUs, each work item is advised to operate on consecutive large sets of data in either small or large work groups. On the other hand, for GPUs the work groups should be in sizes of 128 or 256. Furthermore, in some cases, they observed that bandwidth utilization can vary significantly for the same configuration. For instance, for vector copy operation, the best kernel configuration for NVIDIA GeForce GTX 285 (85.3%), yields a utilization of just 0.2% for Intel Xeon Phi.

3.5 Summary

In this section, we survey all the related work that is relevant to this project. We analyze computational finance areas that broadly use HPC techniques. Such a domain is option pricing for which we note that GPUs are the traditional way to tackle this problem. Also, we tracked applications related to risk management that use clusters for accelerating their computations. Furthermore, we discover different domains that use the Intel Xeon Phi as a platform for acceleration, as well as application for which OpenCL consists the main development model. As OpenCLs primary concern is performance portability, we identify research that discusses this issue and we note potential hazards. In conclusion, from the related work, we couldn'tt discover any other project that uses OpenCL and the Intel Xeon Phi for accelerating scenario-based ALM simulations. Therefore, our work in the following chapters proposes a novel approach on accelerating scenario-based ALM simulations with OpenCL.

Chapter 4

Experimental Setup and Performance Metrics

In this chapter we present in detail our experimental setup. Specifically, we discuss the hardware and software details of all the platforms we are using, and introduce our perfomrance metrics.

4.1 Hardware Platforms

In order toanalyze the performance and portability of our parallel application, we had selected three types of hardware platforms.

Initially, we had available two different flavours of GPPs. As seen in Table 4.1, we use an Xeon E5-2630 as a high performance GPP, and an Intel Core i7-5600U as a regular CPU [25]. The Xeon is equipped with 8 cores and 16 threads, while the i7-5600U has just 2 cores and 4 threads. Both processors support Advanced Vector Extensions (AVX) and the i7-5600U supports Streaming SIMD Extensions (SSE).

Table 4.1: Available CPUs specifications

Model/Specification	**Intel Xeon E5-2630-V3**	**Intel Core i7-5600U**
Cores/Threads	8/16	2/4
Processor Frequency (GHz)	2.4/3.2	2.6/3.2
L3 Cache (MB)	20	4
Number of Channels	4	2
Memory Bandwidth (GB/s)	59	25.6
Thermal Design Power (W)	85	15
Instruction Set Extensions	AVX 2.0	SSE4.1/4.2, AVX 2.0

In addition, as Table 4.2 presents, two different accelerators were used. Ori-

ginally, we focused on an Intel Xeon Phi 511P co-processor, with 60 cores and 240 threads. Moreover, the co-processor offers, 8GB of high-speed memory with peak bandwidth of 352GB/s. Phi's theoretical peak single precision performance is 2200GFLOPS[25]. We further added, an NVIDIA GeForce GTX TITAN, a GPU is equipped with 14 cores and 2688 threads. TITAN has a theoretical single precision peak performance of 4500GFLOPS, 6GB of on-hip memory with peak bandwidth of 288GB/s[44].

Table 4.2: Hardware Accelerators Specifications: Co-processor & GPU

Model/Specification	**Intel Xeon E5-2630-V3**	**NVIDIA GTX TITAN**	**Intel Xeon Phi Coprocessor 5110P**
Cores/Threads	8/16	14/2688	60/240
Processor Frequency (GHz)	2.4/3.2	0.837/0.876	1.053
On chip Memory (GB)	-	6	8
Cache Levels	3	2	2
Cache Size (KB)	256 (L2)	512	512
Memory Bandwidth (GB/s)	59	288	352
Thermal Design Power (W)	85	250	245
Throughtput (GFLOPS)	130	4500	2200

Most of the development and testing was performed on DAS-4 and DAS-5[7]. DAS-4 was running CentOS 6.0, equipped with Intel Xeon E5-2620. Also, Intel 13.3 OpenCL driver (OpenCL 1.2) supported the Intel Xeon Phi was available. On the other hands, DAS-5 was running under CentOS 7.02, with Intel Xeon E5-2630-V3 CPUs. In addition, NVIDIA CUDA 7 toolkit with OpenCL 1.2 support was available. Specific hardware/platform specification available in Table 4.3.

Table 4.3: Available Hardware/Software/Driver Configuration

Cluster	DAS-4
	DAS-5
CPU	Intel Xeon CPU E5-2620
	Intel Xeon E5-2630-V3
OS	CentOS 7.2
	CentOS 6.0
Platform	NVIDIA CUDA 7 (OpenCL 1.2 support)
	Intel Runtime 13.3 (OpenCL 1.2 support)
	Intel Runtime 13 (OpenCL 1.2 support)

4.2 Performance Metrics

To quantify and evaluate the performance of our implementation, we select the following metrics:

- **Execution Time (T)** is the main point of observation for understanding the performance of our implementation. The execution time can be obtained from several sources depending on the precision in need and the platform under investigation. We use the following techniques: *wall-time* measurement on the CPU and the *clGetEventProfilingInfo* or vendor specific profilers such as the Intel VTune Amplifier [26].

- **Speedup (S)** is the performance difference between two different implementation with execution times of T_1 and T_2:

$$S = \frac{T_1}{T_2} \tag{4.1}$$

- **Throughput**

  - **Computational Thoughtput** is way to quantify the computational power of a specific system. Thus, it can be calculated as the floating point operations per a given period of time.

$$FLOPS = \frac{FLOPS_{\text{total}}}{T} \tag{4.2}$$

  - **Memory Bandwidth** is the rate at which data is transferred to and/or from the global memory. Thus, the ratio between read and write bytes performed in the execution time in seconds:

$$MB = \frac{bytes_{\text{read}} + bytes_{\text{write}}}{T} \tag{4.3}$$

- **Scenarios per second (SS)** is the number of scenarios [1] that can be fully evaluated by the computation kernel per second:

$$SS = \frac{\#Scenarios}{T_{\text{kernel execution time}}} \tag{4.4}$$

  is a second metric to evaluate hardware utilization. Thus, we calculate the total number of scenarios under consideration divided by the total number of threads present in the HPC platform:

$$ST_{\text{s}} = \frac{\#Scenarios}{\#Threads} \tag{4.5}$$

[1] The meaning and use of scenarios is explained in detail in Chapter 5

Chapter 5

Case study: Scenario-based ALM

Chapter 5 presents our work regarding the chosen case study. In more detail, the exact characteristics of the scenario-based ALM are presented. We further describe the application computational requirements and our parallel algorithm. Moreover, we discuss our OpenCL implementation and the applied optimizations for Xeon Phi. Finally, we present and analyze the performance we have obtained.

5.1 From OPAL to Scenario-based ALM

Asset liability management is a common technique for managing pension assets. ALM aims to cover the need for maximum risk-return trade-off with reduced risk to solvency rations and the risk relative to liabilities more clearly defined.

Our study focuses on scenario-based ALM models. Scenario-based ALM models are preferred because they can asses much more complex interactions and components of an ALM problem. Also, these models allow for a more in-depth examination of the effects of different investment strategies and/or financial decision. Lastly, they can lead to visual analysis of the different scenarios, which allows to more flexibility for the data output.

For the needs of this project, we extracted a Scenario-based ALM model from a commercial application offered by Ortec-Finance, called OPAL. OPAL is a wealth planner that streamlines the entire private wealth advisory process for clients, determining the optimal asset allocation. OPAL selects and monitors investments fully in line with a clients objectives and relevant regulations, and uses ALM methodologies to optimize the client's investment[48].

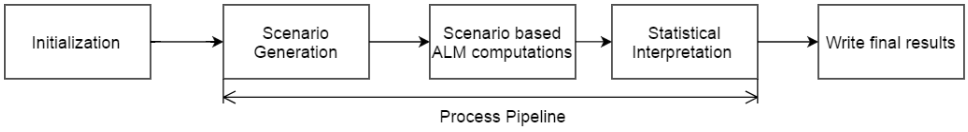

Figure 5.1: OPAL abstract process pipeline representation

As it can be seen in Figure 5.1, there are three main tasks within OPAL: scenario generation, scenario-based ALM computations and eventually a statistical interpretation of the results. Scenario generation produces the input for the compute engine, which corresponds to N different scenarios representing different fiscal conditions. Then, the ALM compute engine is fed with the scenarios and client's portfolio(s) of assets, while applying all investment decisions on each scenario in isolation. Finally, the results are interpreting statistically for calculating an overall potential risk and return for the investment strategies.

The ALM computation part is not only consumes the largest part of the computation time, but also plays a major role in its accuracy. In turn, the accuracy of the simulation is related to the amount of data, which can be effectively handled. For example, larger number of scenarios with more fiscal information regarding the client (e.g. more assets) can produce more precise results and tolerate faulty predictions. Therefore, the ALM computation part is the core focus of this study.

5.1.1 Sceanario based ALM: How it works?

The ALM scenario-based compute engine, receives as an input the pre-generated scenarios and a number of portfolios along with their contents. Each portfolio contains a number of assets summarized in the list below:

- Cash
- Bonds
- Stocks
- European Equities
- Japanese Equities

The selection of these assets allows an arbitrary number of "dummy" investments to take place within the model and therefore, the model's overall complexity can be similar to the actual Ortec-Finance product. Also, in a group of four portfolios, the initial capital is shared in pseudo-randomly among the portfolios. For example, one portfolio might initially have a quarter of the total capital and this amount might be evenly shared among its assets.

Now, for each scenario, a full simulation period corresponds to 64 years and 12 months for each year. Therefore, at each time-step (year), computation is performed to simulate real economical world events:

- Portfolio re-balancing.
- Taxation.
- Cash transfers between portfolios.
- Annual, semi-annual and monthly cash-flows.

In addition, except the above-mentioned financial operations, equations 4.1, 4,2 and 4.3 depict the main computation per iteration. Specifically, equation 4.1 represents how the values for each variable of the different assets is calculated, *meanEquityEU* and *sdEquityEU* are values determined by the scenario, multiplied by the randomized Gaussian values are within zero to one margin. Equation 4.2 shows how the value for each portfolio is determined: by adding all portfolio assets multiplied by a pre-generated factor for the given scenario. Finally, in 4.3 the total values of all portfolios is calculated, in order to perform the remaining operations.

$$ScenarioEquityEu = meanEquityEu + sdEquityEu * RandomizedGaussian \tag{5.1}$$

Therefore, the total value of each portfolio at any given period, corresponds to

$$PortfolioXvalue = \sum_{i=1}^{n} = scenAssetfactor(i) * currentAssetValue(i) \tag{5.2}$$

$$totalValue = \sum_{i=1}^{n} = portfolio(i) \tag{5.3}$$

Algorithm

Algorithm 1 depicts a pseudocode representation of the given scenario-based ALM compute engine. Most of the computation procedures (e.g., compute tax or rebalancing) are make heavy use of branches. For example, compute tax uses up to 6 branches, while rebalancing uses up-to 10. In the worst case scenario, overall, 40 branches need to be evaluated for one iteration of the kernel.More details on the computational structure and complexity of the scenario-based ALM are presented in Section 5.3.

```
Scenario-based ALM case model:
  Input: Scenarios, Years, Months, portA, portB, portC, portD
  Output: totalValue, valueA, valueB, cvalueC, valueD, valueTax
  for 1.....scenarios do
      for 1.....years do
          for 1.....months do
              Sum of Portfolio A assets;
              Sum of Portfolio B assets;
              Sum of Portfolio C assets;
              Sum of Portfolio D assets;
              if month is december then
                  if Total value larger than 1000000 then
                      calculate tax;
                  end
                  else if Total value larger than 100000 then
                      calculate tax;
                  end
                  else
                      calculate tax;
                  end
                  divide tax by portfolios ;
                  subtract tax from portfolios ;
                  rebalance evenly portfolios;
              end
              if semester then
                  withdraw;
              end
              if quarter then
                  transfer amount between portfolios;
              end
              if every other month then
                  other cashflow function ;
              end
              store current value of each portfolio;
              store tax value ;
              store total value of portfolios ;
          end
      end
  end
  return totalValue, valueA, valueB, cvalueC, valueD, valueTax
```

Algorithm 1: Abstract representation of Scenario-based ALM

5.2 Sequential Implementation

The OPAL application is written in JAVA. Thus, the extracted scenario-based ALM compute engine along with the scenario generation need to be rewritten in C. As a result, a sequential C version was implemented. This implementation served the goal of identifying potential performance bottlenecks as well as on defining a parallelization strategy. In addition, it provided enough information in terms of profiling and computational complexity of the model.

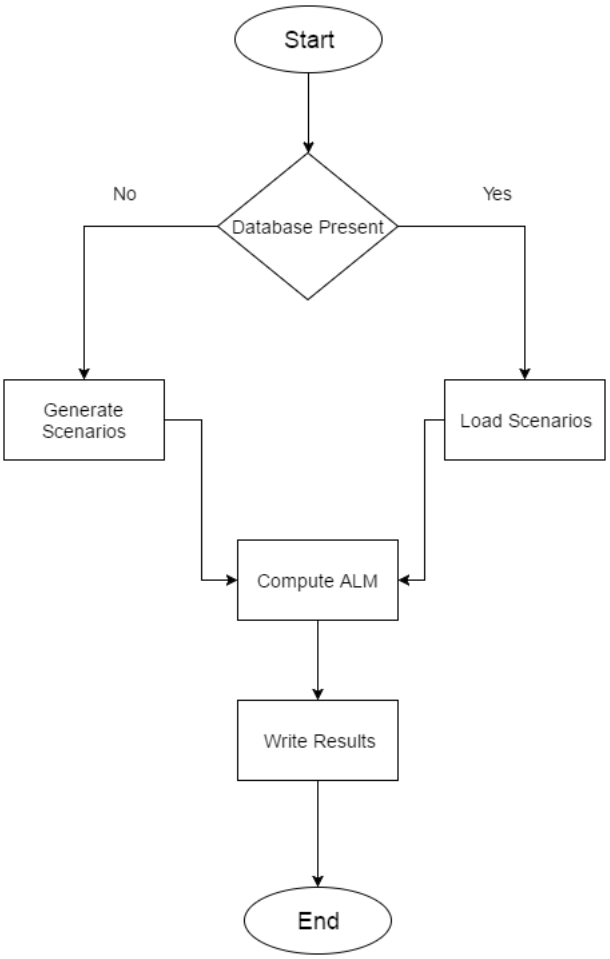

Figure 5.2: Data flow diagram of the extracted C implemetation

Figure 5.2 depicts the main components of the C implementation, extracted from the JAVA application. As it can be seen, the implementation either loads a pre-generated database of scenarios or either generates one itself. For convenience and to allow more degrees of freedom in sense of data types and the length of input (e.g. number of assets, portfolios etc), a choice of generating the scenarios on the fly was made.

5.2.1 Profiling

In order to verify that the compute engine is the most compute intensive part of the application, along with how much of the total execution time is consumed, we run a profiler. For this analysis the GNU *gprof* [18] profilling tool was used. Also, the tests were performed for a single core, single thread execution on an Intel i7-5600U CPU.

Table 5.1, depicts the results obtained from *gprof* for various input sizes. As it can be seen, the compute engine for the ALM calculation consumes from 61% to 66% of the total execution time. In addition, the initialization (e.g. scenario generation) takes up to 24%, while the rest of the tasks 10% to 16% of total execution. Therefore, performanc-wise, it makes sense to accelerate the compute engine.

Table 5.1: Profiling information for single threaded CPU execution

#Scenarios	Function	Time (s)	Percentage
1024	Compute ALM	0.28532	66%
	Initialise	0.10256	24%
	Remaining Tasks	0.0495	10%
10240	Compute ALM	3.1245	61%
	Initialise	1.4254	23%
	Remaining Tasks	0.8125	16%
40960	Compute ALM	9.8645	62%
	Initialise	5.6589	27%
	Remaining Tasks	1.5896	11%

5.2.2 Single vs Double precision requirements

Another aspect of finding potential bottlenecks, was to test the initial sequential implementation for various data types and input sizes. Therefore, the implementation was tested for *single* and *double* precision numbers. Also, different numbers of scenarios were tested. These numbers varied from 1000 to 100000.

Figure 5.3, shows the obtained results for the above-mentioned configurations. This initial testing illustrated the following: 1) for a small size i,e to 8192 scenarios, both single and double precision yield similar execution times 2) for more than 8192 scenarios, the double precision version implementation is slower by 3x/2.5x factor, 3) for inputs larger than 81920 scenarios the double precision implementation can be no longer execute by the GPP configuration present 4) the single precision implementation can handle up to 102400 scenarios, due to the limited memory 5) currently the commercial application can simulate just 500 scenarios in order to comply with specific response time requirements.

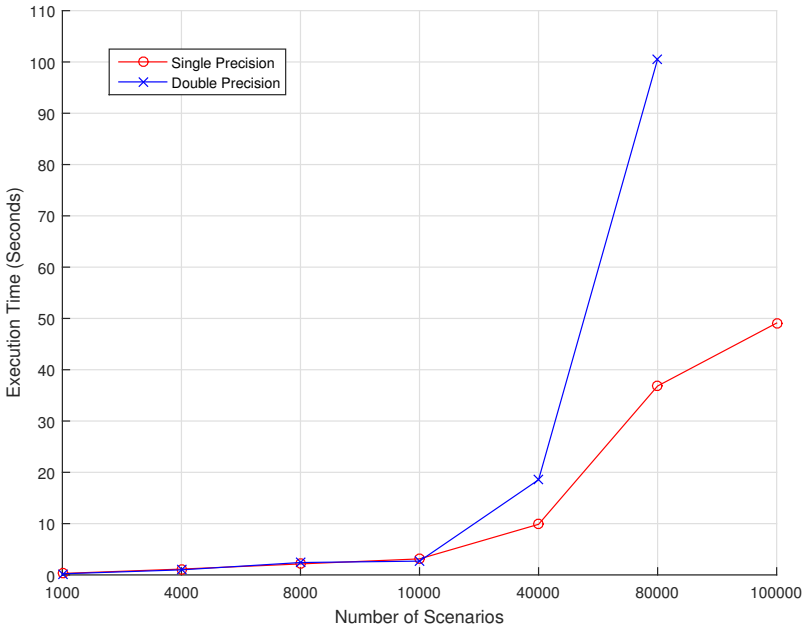

Figure 5.3: Single vs Double precision for single-core/single-threaded execution

5.3 Parallel Implementation

After we extracted all the key information regarding the computational needs and parallelization possibilities, we can we will proceed to create an initial parallel version of our scenario-based ALM.

5.3.1 Operational Intensity Calculation

Before implementing the application we need to quantify its needs in terms of operational intensity. This classification will help to determine the nature of the problem (compute bound vs memory bound), along with expected hardware utilization.

We first approximate the number of operations within the compute kernel. These operations can be floating point additions, subtractions, multiplications, divisions, as well as more complex operations like extracting the square root or computing an exponential value. The total number of these operations is floating point operations (FLOPS).

The total number of FLOPS in our ALM kernel is presented in Equation 5.4. is represented in the following equation:

$$FLOPS = (26 * addf) + (36 * mulf) + (6 * divf) + (20 * subf) \tag{5.4}$$

Now, according to equation 5.4 and the Table 5.2 which represents the cost of its floating point operation for x86 architecture, we can evaluate the total estimated

cost. Therefore, this cost corresponds to 172 FLOPS for our compute kernel.

Table 5.2: Cost in terms of FLOPS for operations[4]

Operation	**Cost (FLOPS)**
addf	1
subf	1
mulf	1
divf	15
sqrtf	15
expf	20

Additionally, before we are able to calculate the operational intensity of the kernel, we need also to find its needs in terms of memory traffic. This memory traffic corresponds to the amount of data that the kernel reads or writes from the memory, in bytes. The kernel memory traffic is estimated in Equation 5.5.

$$memory\_traffic = ((20 * reads) + (5 * writes)) * 4Bytes \tag{5.5}$$

The number of floating point operations and memory traffic, correspond to Work *W* and Memory Traffic *Q* respectively. Then, the Operational Intensity *I* corresponds to the ratio of the Work divided by the Memory traffic. Thus, for our case the operational intensity for the ALM kernel is as follows:

$$I = \frac{W}{Q} = \frac{172}{100} = 1.72FLOPS/byte. \tag{5.6}$$

Roofline Model Analysis

Roofline Model [68], provides the means in order to visualize the potential performance of compute kernel, running on a multi/many-core architecture such as NVDIA GPUs or Intel Xeon Phi.

The model combines information regarding the peak attainable floating point performance, peak bandwidth and the operational intensity of any given platform. Then, from the resulting plot the nature of the application's bottleneck can be defined. The application can either be limited by the memory bandwidth (memory bound problem) or either by the peak computational performance (compute bound problem). The roofline plots contains the attainable floating point performance on the vertical (Y) axis in *FLOPS/s* and the operation intensity in *FLOPS/byte* in the horizontal (X) axis.

Before plotting the Roofline model, some key hardware specifications need to be calculated. These specifications are the peak floating point performance and the peak memory bandwidth. The floating point performance for any given many-core hardware can be determined as follows:

$$Peak = \#chips \times \#cores \times vector\_Width \times FLOPs/cycle \times clock\_frequency \tag{5.7}$$

Now, equation 5.7 and the theoretical peak bandwidth provided with the hardware specifications, help on estimating the attainable GFlops/sec performance, calculated as follows:

$$AttainableGFlops/sec = min \begin{cases} PeakFloatingPointPerformance \\ PeakMemoryBandwidth \times OperationalIntensity \end{cases}$$

Now, having all the necessary information in place, we can proceed to plotting the roofline model for our application. As our main interest on the initial development is the Intel Xeon Phi, the plot uses its performance characteristics: a peak bandwidth of *352 GB/s* and peak floating point performance of *2417GFLOPS*.

Figure 5.4 depicts the ALM roofline model. The figure shows that our compute kernel is memory-bound. This means that our OpenCL implementation should focus on applying performance optimization techniques which improve the memory bandwidth of the application.

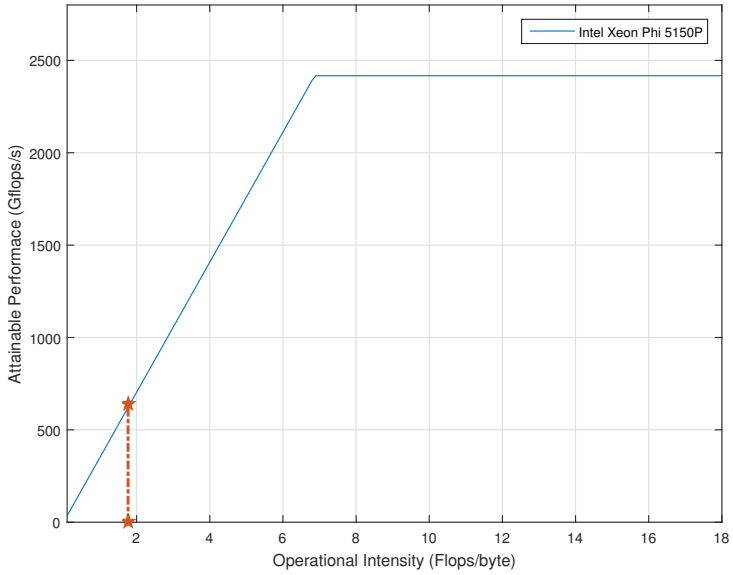

Figure 5.4: Roofline model of the ALM kernel on the Intel Xeon Phi

5.3.2 Initial OpenCL Implementation

The initial approach was to rewrite the sequential C implementation with OpenCL, without taking into account any hardware specific features or dedicated tuning. Because the calculation of each scenario is individual from each other we assign

one *work-item/thread* to be responsible for the evaluation of each scenario. This approach keeps the complexity of the parallelization low. Also, at first only *1D* dimensioning for the work-groups were used, without proper workload balancing. This naive first approach yielded almost 20x performance improvement compared with the original, sequential C-version. This is already a very good result.

5.3.3 OpenCL Optimization and Tuning

As we already discussed the novel OpenCL implementation of the ALM compute kernel, initially targets the Intel Xeon Phi architecture. Therefore, any optimization steps took into account specific architectural features and considerations for the very co-processor. A detailed analysis of the optimization techniques used, along with individual decision and consideration are presented in this section.

Reducing Work-item Branch Divergence

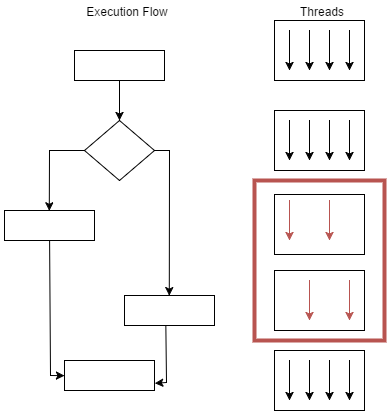

Figure 5.5: Branch divergence effect in GPU threads

It is well-known that branch divergence is one of the main issues affecting the performance of parallel implementations mapped on GPUs and co-processors. There is a lot of relevant research exposing, discussing and proposing solutions on this problem.

We discuss the potential performance loss and effects of branch divergence impact because our naive implementation contains extensive branching. GPUs can experience high performance losses due to branch divergence because of their warp-based execution model. Then, within a warp the potential performance losses are illustrated in Figure 5.5. In this figure we can see that threads within the same warp execute different branches, leading to a potential performance loss of up to 50%.

From the Intel Xeon Phi perspective it can have an advantage for divergent workload. This advantage is that it performs work in much smaller groups. For example, the Phi uses 16 elements per thread, while NVIDIA has 32 threads per warp.

From the above-mentioned information and the nature of our kernel, the following measures were taken. First, we removed branches where possible, we further extracted branches out of loops where it was possible, and finally we included the

minimum number of computations within each branch. Therefore, minimimize the number up branches up-to 10 per iteration for the worst case scenario.

Compiler Optimizaions: OpenCL Flags

One of the greatest advantages accompanied with many-core architecture, is the presence of specialized hardware units, enhanced with the ability of executing mathematically intensive functions. Also, these units are fully supporting floating point operations, which makes them even more valuable feature. Therefore, the computational cost of using such functions can be significantly reduced. For this reason, OpenCL provides certain compiler flags, which for cases where accuracy is not in high demand, it can be sacrificed for faster response.

At first the initial naive implementation didn't make use of any compiler based optimizations. For this reason a research/test process was carried out for finding the most suitable compiler flags for the our kernel. Some math instincts optimization have a significant trade-off between the potential speedup gain and the correctness of the calculations. Therefore, its a major function to consider, while applying different flags.

- *-cl-fast-relaxed-math*, which enables two other optimization flags together. These flags are the *-cl-finite-math-only* and *-cl-unsafe-math-optimizations* . Now, operations that may be against the IEEE 754 and the OpenCL numerical specifications, are allowed to be executed.

- *-cl-single-precision-constant* which allows all the double precision floating point constants to be treated as single precession floating point constants. We verified that this extension does not potentially harm our single and double precision implementations.

- *-cl-denorms-are-zero* is responsible for all single and double precision floating point denormalized numbers. In case of single and double precision denomarlized numbers can be flushed to zero if such an extension is available. This configuration option can be beneficial in the context of performance, as the compiler can choose whether it needs to flush denomarlized numbers.

For ALM, enabling the above-mentioned compiler flags has a negligible impact on performance. This is not surprising, given that there are no special mathematical functions that are needed for the computation, and therefore fast-math is not really any faster than regular math in this context. A more detailed insight on the impact is presented in Section 5.4.1.

Converting Arrays of Structures (AoS) to Structure of Arrays (SoA)

Coalesced memory accesses and caching, play a major role in the performance of many-core architectures. Thus, a very important factor on fully utilize such opportunities, is data layout. Data layout affects can memory access patterns, and,

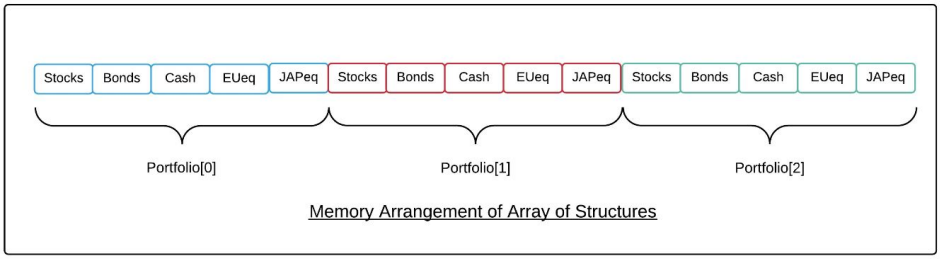

Figure 5.6: Initial data layout in Array of Structures (AoS)

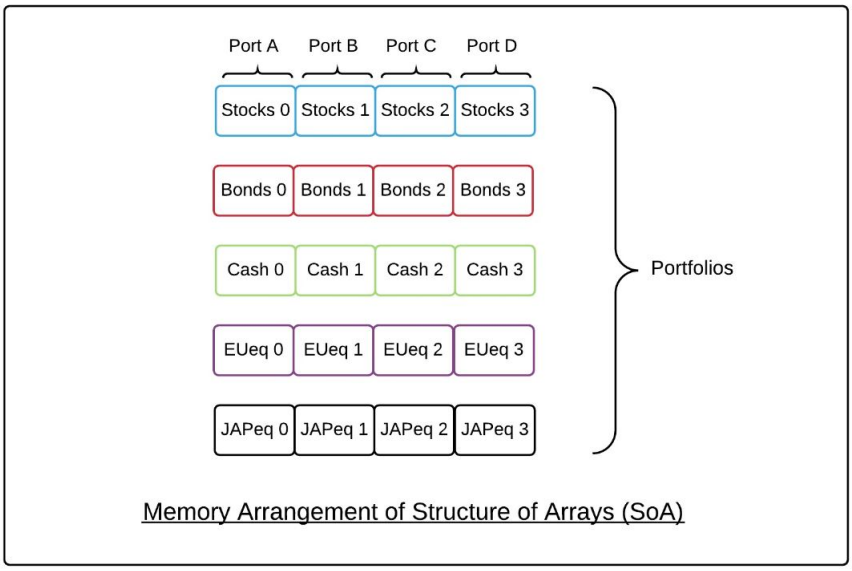

Figure 5.7: Converted data layout into Structure of Arrays (SoA)

consequently the performance of the application as well as the utilization of platform characteristics such as the memory bandwidth. One of the clear guidelines to improve coalescing and caching is to convert arrays of structures (AoS) to structure of arrays (SoA), this conversion yields a more cache and vectorization friendly design [70] [60]. As we are discussing mainly about the effect on the Phi, we need to consider how the vectorization module treats data. Therefore, the advantage of wide SIMD lanes of the Phi, is that the vectorization module transforms scalar operations of the work-items into vector operations. So, the overall number of operations can be reduced significantly based on the available SIMD width.

The original ALM data structures are AoS-based. The array contains a set of different structures of type *portfolio*. Each portfolio structure has inside five *float* variables containing data regarding the asset values. The layout is depicted in Figure 5.6.

On Xeon Phi, using SoA instead of AoS should improve the caching and vec-

torization opportunities. In fact, Intel explicitly suggests AoS on the Phi, only for random sparse accesses[29].

Figure 5.6 illustrates the initial data layout for our compute kernel. AoS was the initial approach as it is a more convenient approach for the developer and allows easier indexing within calculations. On the other hand, Figure 5.7 illustrates the resulted conversion to SoA with one portfolio structure enclosing all the data in need. Now, all aspects are grouped into arrays based on their content and these arrays grouped into a *portfolio* structure. The impact of this conversion is discussed later on.

Work-group Configuration

Common techniques for developing OpenCL kernels for CPU-like architectures similar to the Intel Xeon Phi, suggest local work-group dimensions which map well on the SIMD width. Thus, for the Intel Xeon Phi and single precision data types, the suggested dimensions need to be a multiple of 16. Because of the friendly mapping of threads to compute SIMD units, this approach allows to the OpenCL to fully utilize the auto-vectorization module of the compiler without any manual modifications. In case of manually vectorization of OpenCL kernels, the compiler scalarizes them again before re-applying implicit vectorization[29]. On the other hand, for non-multiplies of 16, the compiler handles the remaining items in a scalar way[63].

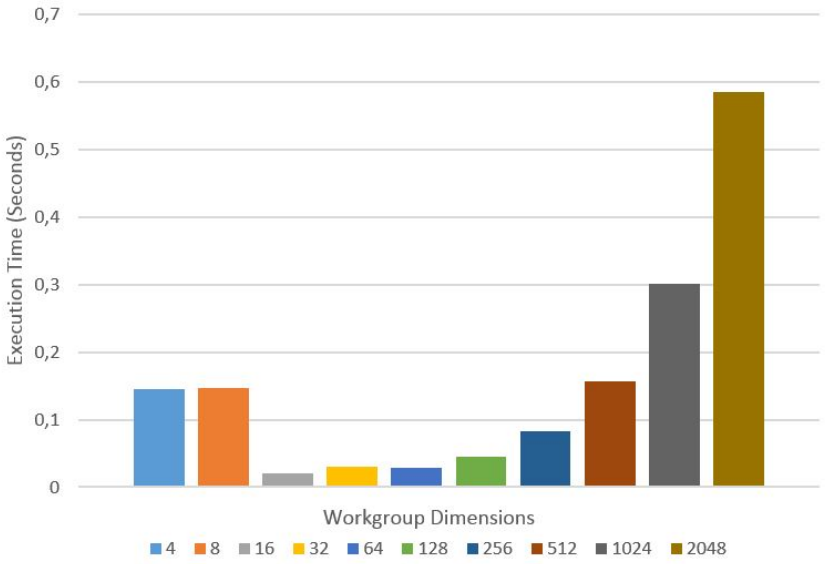

Figure 5.8: Execution times for various work-group configurations

We used a benchmark case for 10240, where 10240 *global* work-items are needed for computation. Then, we tracked the execution times for various work-group di-

mensions. As it can be seen in Figure 5.8, the execution time is the best for 16 work-items making up a work-group. Thus, the optimum configuration for this input size yields, 128 work-groups of 16 work-items each.

Automatic vs Manual Loop Unrolling

Loop unrolling is a traditional technique for improving the performance of parallel loops. With unrolling loops one can minimize the number of executed instructions, minimize the evaluation of each branch and benefit from exploiting more instruction level parallelism where it is applicable.

There are two approaches for loop unrolling: manual and pragma-based. The performance guidelines for programming the Intel Xeon Phi, suggest that manual loop unrolling should be avoided. The reason is that the Intel compiler is able to effectively generate vectorized code when loops are not unrolled. Therefore, alternatives are suggested in order to enhance the vectorization abilities of the processors. For example, loop collapsing and/or vector alignment.

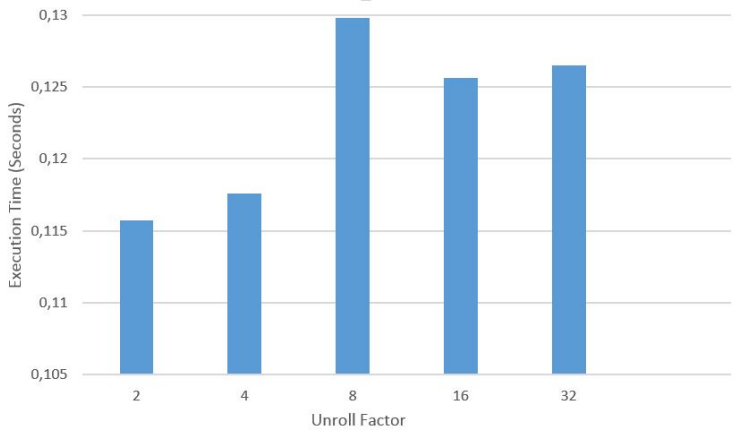

Figure 5.9: Execution time vs loop unroll factor

In Figure 5.9, the execution times for different unroll factors which use OpenCL *#pragmas* are presented. It needs to be mentioned that even with the unroll factor of 2x providing the best improvement in performance, loop unrolling gives slower results than the regular execution. The problem is the corner cases for which dependencies exit. Iteration one and twelve of the inner loop cannot be unrolled effectively. Thus, we tried to use loop peeling on the corner cases (e.g. iteration one and twelve) for which dependencies exit and then automatically unroll shown only negligible improvements.

Loop Collapsing

We further investigated the potential performance gain when loop collapsing is applied. As the whole kernel consists of three nested loops, with the out loop

already mapped to the OpenCL work-item, the inner loop can be optimized further. Therefore, collapsing the two inner loops to one, can give further performance benefits as it can allow the Intel compiler to exploit further its auto-vectorization capabilities if the loops are properly aligned.

```
for 1.....scenarios do
    for 1.....years do
        for 1.....months do
        end
    end
end
```

Algorithm 2: Before loop collapsing

Algorithm 1 and 2, depict the loop structure before and after collapsing. This, re-structuring of the loops proved to be beneficial for the overall performance, as it further reduced the execution time for the kernel. In addition, the intitution was proved that the autovectorization abilities of the compiler have advantageous affect with collapsing loops, in return to a more complex ”indexing”. The results about the benefit in terms of speedup are presented in Section 5.4.

```
for 1.....scenarios do
    for 1.....years * months do
    end
end
```

Algorithm 3: After loop collapsing

Pointer Aliasing

Pointer aliasing can occur when the same memory location can be accessed by two or more different pointers. Thus, the compiler needs to take precautions for eliminating any potential side-effects. For this reason ensuring the compiler that a memory location can be accessed only by a single pointer, allows the compiler to optimize code further. Thus, by using the *restrict* keyword for a given set of pointer in our kernel declarations, spares the compiler from creating unnecessary memory dependencies between non-conflicting load and store operations.

Use of Constant Memory

Constant memory is typically a cached memory region where read-only data can be stored and, due to caching, quickly accessed. Constant memory is typically small in size, but it provides quick access to all compute units of the device. In general, constant memory usage has an impact when a relatively small data structure is used intensively and repeatedly in other computation kernels by all work items. In the case of ALM, constant memory is useful to store a common set of parameters and weights used for asset value computation.

Again, we note that the use of constant memory for scenario-independent weights has a negligible effect: a speedup of only 1.05x. This happens because constant memory is not mapped on any specialized hardware, but simply uses a special region in the global memory. Due to the linear access pattern of the application, the weights stored in global memory are already accessed in a cache-friendly manner, making the use of constant memory ineffective. Furthermore, for larger number of scenarios (more than 10240), data cannot fit in constant memory, so we abandoned this optimization.

Upgrading the OpenCL compiler for Intel Xeon Phi

As we discussed in Chapter 3, all of our testing was carried our on the DAS4 and DAS5 clusters. Multiple OpenCL compilers were therefore available.

For the majority of testing the Intel 3.0 MPSS (Manycore Platform Software Stack) was used as it was recommended when targeting the MIC architecture. Switching to latest available MPSS on the DAS4 (e.g. 3.2.1 at that time), improved the performance by a 16.5x factor. Unfortunately, the latest MPPS, which is 3.7.1 wasn't available at that time in order to evaluate any further performance gains.

5.4 Performance Evaluation and Analysis

This section is dedicated to the performance analysis of our OpenCL ALM kernel. Specifically, for our benchmark case (10240 scenarios), the individual contribution for each optimization is presented. Also, the scalability of our OpenCL kernel for single and double precision is also evaluated. Finally, a comparison to a native OpenMP implementation on the Xeon Phi is also discussed in this section.

5.4.1 Optimizations Efficiency

For each applied optimization, its contribution to the overall speedup as well as the reduction in terms of execution time for compute kernel, captured in Table 5.3.

Table 5.3: Individual effects of each optimization applied to the ALM compute kernel

Version	Time (sec)	Speedup
Sequential (Single Threaded)	3.1245	1x
Initial OpenCL	0.53228	5.8x
Default Optimizations - Enabled	0.15138	20.6x
Optimum Workgroup size	0.03770	82.8x
Compiler Flags	0.03543	88.2x
Converting to SoA	0.03019	104x
Loop Collapsing	0.02889	108.2x
Constant Memory	0.02670	117x
Restrict Pointers	0,02365	132x
Upgrading to the Latest Ocl Compiler	0.02106	148.5x

Figure 5.10 and Table 5.3 capture the effect of each optimization in terms of reducing the execution time and overall speedup, respectively. These results obtained for 10240 scenarios, which is a representative input size. The result show that just porting a nave OpenCL implementation to the Phi, yields an improvement of a factor at least 5.8x. On the other hand, when the basic features of the compiler are exploited, the benefit rises up to 20.6x. But the greatest contribution for speedup, comes from tuning for the optimum local-group dimensions, which provides fine-grained data parallelism and builds-up the speedup to 82.8x. Using specific compiler flags has a negligible effect on performance, as discussed in Section 5.3.3. Similar behavior occurs with the use of constant memory. Other optimizations seem to contribute in a larger grade to overall performance. For example, converting to SoA and use of restrict pointer, contribute by a factor of 16.2x and 15x, respectively. Finally, upgrading to the latest OpenCL compiler available for the Intel Xeon Phi, adds a final 16.5x improvement, reducing the execution time to $0.02106sec$.

5.4.2 OpenCL vs OpenMP on the Intel Xeon Phi

OpenMP is a multi-threading API for exploiting shared memory parallelism. Also, its a more accessible approach on parallelizing and improving efficiency of sequential code. Traditionally, OpenCL is considered a second choice when programming Intel Xeon Phi. Instead, using OpenMP is recommended, as it allegedly outperforms OpenCL on the Phi. Therefore, we need to evaluate our proposed OpenCL solution against an OpenMP version on the Phi.

OpenMP Solution

From our sequential code for our ALM code, an OpenMP version was fairly straight forward to obtain. OpenMP pragmas for ensuring the full utilization of SIMD and

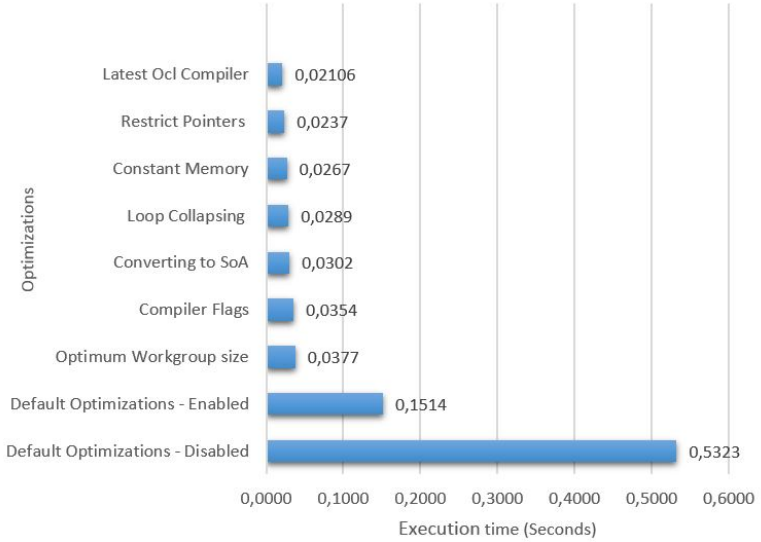

Figure 5.10: Contribution of each optimization on reducing execution times for a benchmark case of 10240 scenarios

vectorization features of the Intel Xeon Phi, were used. In addition, as for the outer loop no dependencies exist, output data partitioning was implemented.

Parallelizing the outer loop of the ALM compute kernel means that each thread of the Intel Xeon Phi can take up to a number of independent scenarios for evaluation. The fact that the scenarios are completely independent of each other, provides the advantage of not using any specific barrier for ensuring communication among threads. This OpenMP version yields the optimum performance. During the implementation procedure other versions consisted of trying to parallelize the inner loop of the ALM compute kernels. Unfortunately, this approach introduced the need of using communication barriers as different threads were trying to write on the same data location. Thus, the results were considerably slower than the outer loop parallelization.

Results & Comparison

Figure 5.11, illustrates a comparison of the results obtained on the Xeon Phi for both OpenCL and OpenMP. OpenMP results show good scalability for problem sizes up to 40000 scenarios. Although, in terms of speedup OpenMP significantly lacks performance compared to our fully optimized solution for the Intel Xeon Phi. For example, for 81,920 scenarios, OpenCL has improved performance by a 7.8x factor, while for smaller input sizes up to 30x improvement compared to our initial scalar code.

These results contradict the current trend in developing application for Intel Xeon Phi, which treats the OpenMP approach as the most feasible solution. Al-

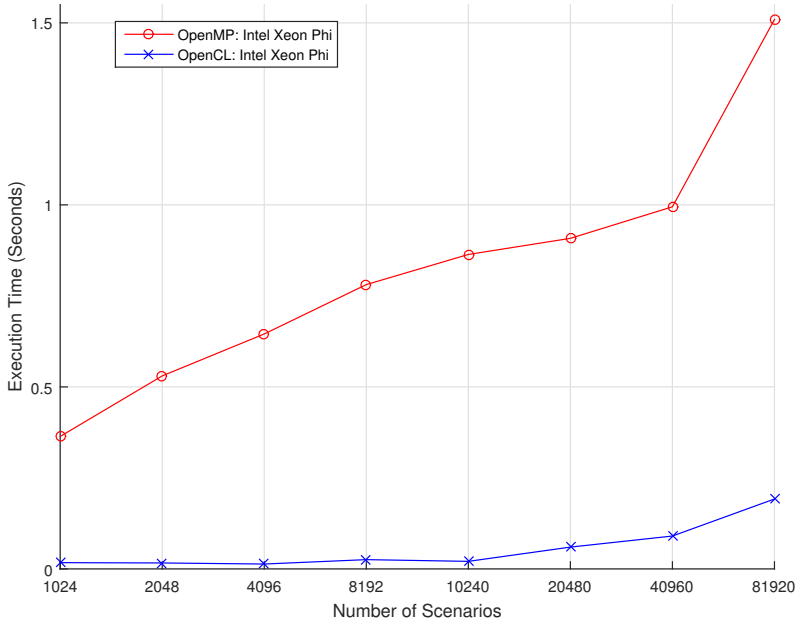

Figure 5.11: OpenMP vs OpenCL execution times on the Intel Xeon Phi

though, there is some related work points that OpenCL in some case can outperform OpenMP. Such research findings are discussed in [62] and [66].

The performance difference most likely emerging by the fact that the OpenCL kernel code is better vectorized by the compiler with optimum use of work-item and ND-range configurations. Thus, OpenCL automatically benefits by the instruction set extensions present on the Intel Xeon Phi, while for OpenMP extra tuning needs to be done for ensuring use of AVX instructions.

Scalability

Another very important aspect of this implementation is the scalability of the produced results, Figure 5.12 captures the scalability for both single and double precision data on the Intel Xeon Phi.

As we can see in Figure 5.12, results for both single and double precision scale in a fairly quadratic manner as it was expected. Also, there is a substantial improvement compared to the sequential results obtained in Section 5.2.2. Finally, we also point out that while for the initial sequential version we were able to simulate up to 80,000 for double and 100,000 scenarios for single precision. On the Xeon Phi we were able to evaluate up to 204,800 for both.

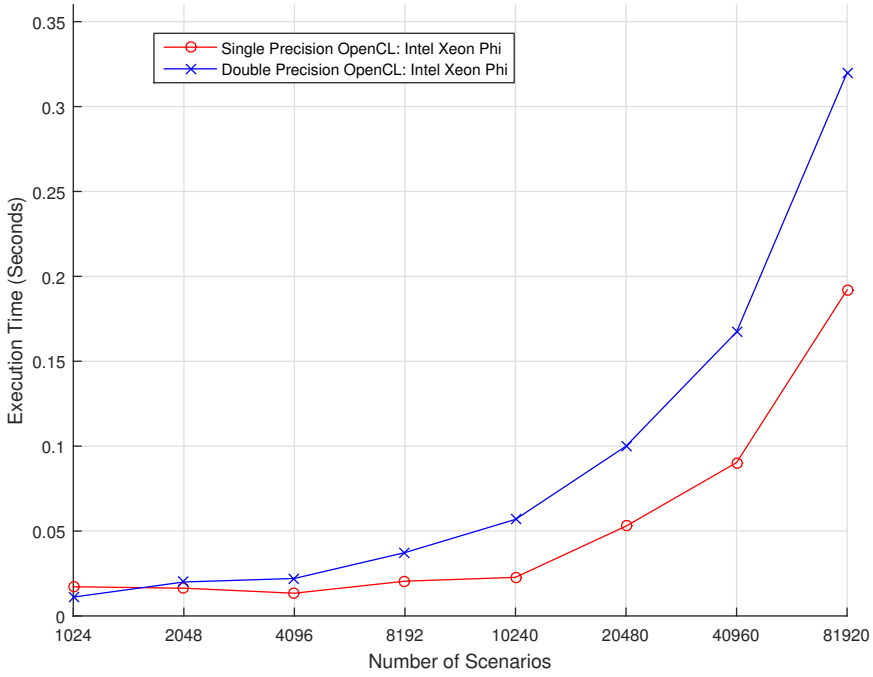

Figure 5.12: Single vs Double precision results for the OpenCL implementation on the Intel Xeon Phi

5.5 Summary

In this Chapter we showed that a representative ALM test case can be extracted in order to produce an OpenCL solution. Also, the use of Intel Xeon Phi along with OpenCL can yield significant improvement in terms of performance when compared against scalar code, provided that a series of optimizations are applied. Furthermore, we prove that our optimized OpenCL solution can also outperform the traditional approach for the Phi, which is OpenMP. Finally, our results also show that our OpenCL implementation is scalable when the number of scenarios increases.

Chapter 6

Performance Portability in OpenCL

OpenCL allows a programmer a certain degree of freedom regarding the potential target platform. However, although functional portability allows the platform of development not to be the platform of deployment, performance portability is a concern. In this chapter we discuss our findings regarding OpenCL's performance portability.

6.1 Motivation

OpenCL offers the great advantage of portable code, but this comes at the price of performance loss. Code with topnotch optimizations, yielding great results in terms of performance and utilization for one platform, may suffer significant changes in performance for another platform. Thus, we might need to start again the optimization process.

In Chapter 5, we presented our case study from a sequential C implementation to an optimized OpenCL version for a given hardware platform - Intel Xeon Phi. We selected Xeon Phi because our kernel uses branching (see Section 5.3.3). It is now time to investigate how well our kernel performs on other platforms.

For benchmarking and analysis, we use the platforms presented in Chapter 4: an HPC CPU for which a lot of comparisons with the Inten Xeon Phi exist [35][36][37] and an NVIDIA GPU.

While implementing and optimizing our novel OpenCL ALM kernel, we tried to use techniques that *should* preserve performance. What we expected/hoped for was to be able to simply run our OpenCL kernel on other platforms and still obtain (similar) performance. Specifically, given our application is memory-bound, we expected *to preserve a certain level of bandwidth utilization*.

For preserving performance, we use the following basic guidelines for developing our kernel:

- We avoid platform specific optimizations. For example, NVIDIA GPUs have texture or shared memory available while other platforms like the Intel Xeon Phi do not have such features. Also, we avoid to depend too much on features such as the constant memory.

- We use a simple 1D work-group organization. While the original application uses a suitable work-group size for the Xeon Phi, this parameter is easy to tune.

- We mostly focused on portable optimizations such as data layout (SoA and AoS), restrict pointers, and compiler optimizations that can have a similar impact on performance for most OpenCL devices.

Whether the resulting OpenCL is, to some degree, performance portable, is the focus of the remainder of this chaper.

6.2 Roofline Analysis for Multiple platforms

Before discussing further the performance on various platforms, we must revisit the roofline model, discussed in Chapter 5. Initially, the roofline analysis was carried out only for the Intel Xeon Phi. Now, we added two other platforms: an Intel Xeon CPU and an NVIDIA GeForce Titan GPU. We use this analysis to see whether the ALM kernel preserves its memory-bound behavior on all three platforms.

As we calculated in Chapter 4, the operational intensity of the ALM kernel corresponds to *1.72FLOPs/byte*. Indeed, as we can see in Figure 6.1, the kernel's performance is memory bound for all these platforms. Thus, as we initially optimized the OpenCL kernel with bandwidth and memory utilization in mind, we expect good performance on all three platforms.

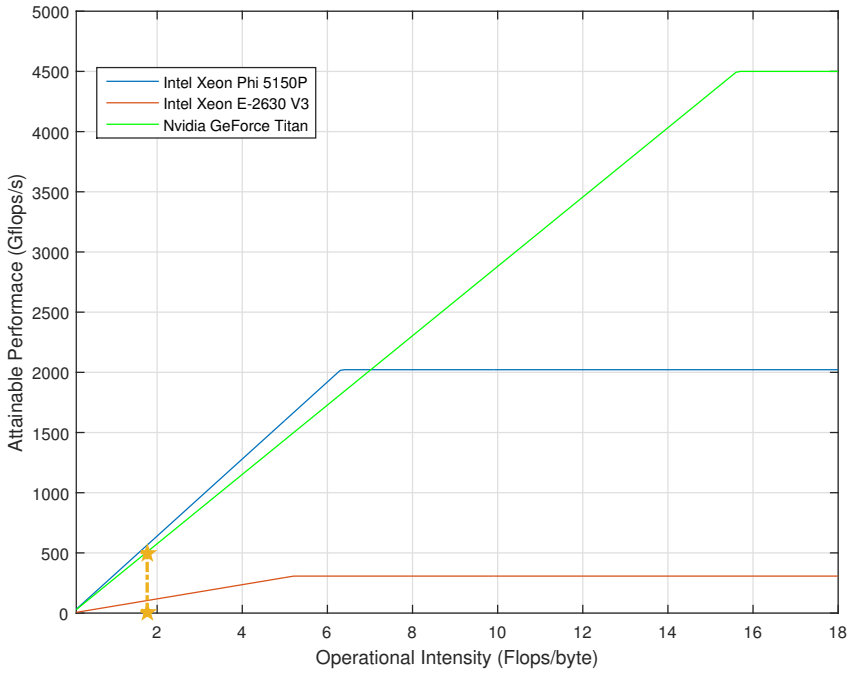

Figure 6.1: Comparison of the Roofline models of the compute kernel

6.3 Optimizations behavior on various platforms

Optimization effect on the NVIDIA GeForce GTX Titan

To gain performance on the Xeon Phi, our kernel uses an SoA data layout and a work-group size of 16. Our goal is to see whether this configuration also performs the best on the GPU. Therefore, we analyze and compare the performance of the original code against different other feasible configurations.

In Figure 6.2, we can see that using the default data layout (SoA) improves performance for any given work-group dimensions. AoS reduces significantly the the performance on the Titan. This result shows that converting to SoA, was the right choice for our kernel.

As Figure 6.3 illustrates, performance can be further improved if except converting to SoA, tune the workgroup size. Furthermore, we can further notice that for a local workgroup configuration of 128 work-items we can achieve the best possible performance.

Overall, the best possible performance was obtained with SoA layout and local workgroup configuration of 128 work-items. Therefore, for performance portability from Phi to GPUs, the data layout can be kept, but the workgrop size must be tuned according to the application and the specific GPU model.

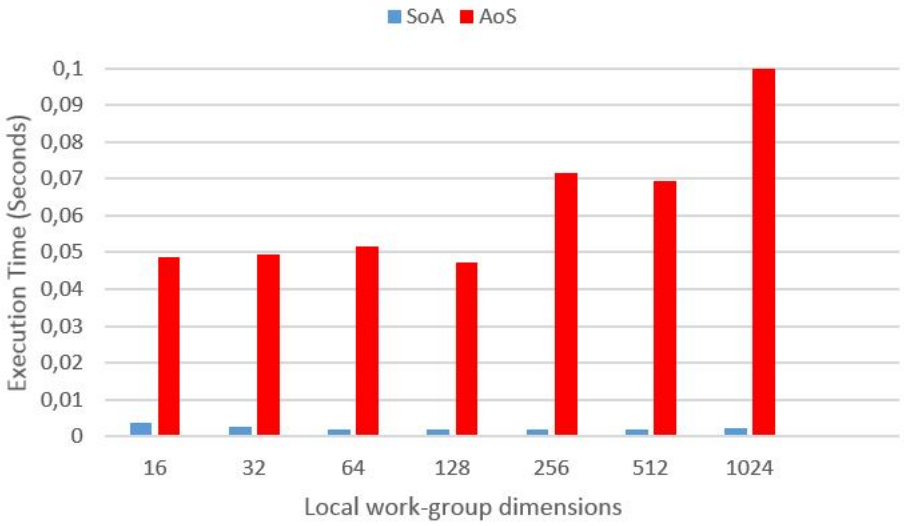

Figure 6.2: SoA vs AoS on the NVIDIA GeForcen GTX Titan

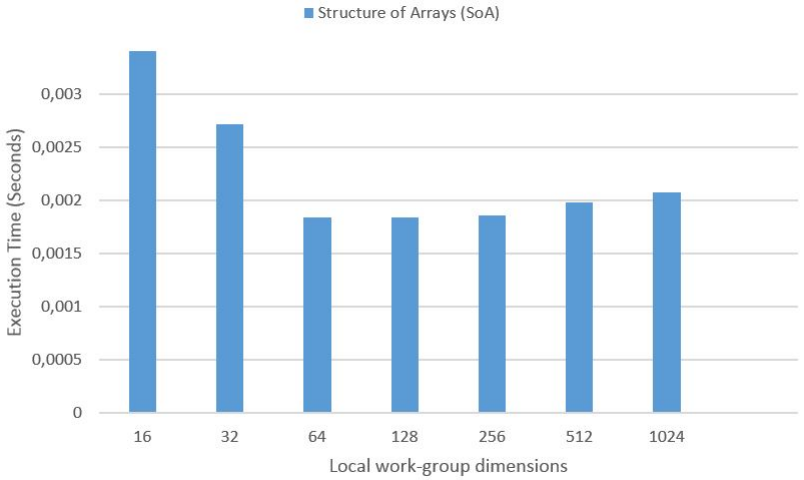

Figure 6.3: SoA: Work-group performance on the NVIDIA GeForce GTX Titan

Optimization effect on the Intel Xeon CPU

Modern high-end CPUs (see Chapter 4) are another platform of interest for computational finance. In addition, the Intel Xeon CPU has similar architectural features with the Intel Xeon Phi. We expect this to further aid the potential results.

As authors at Intel indicated in [23], the Intel Xeon processor also benefits from using SoA, because SoA exploits more vectorization opportunities. Therefore, as we designed our kernel in the same manner for the Intel Xeon Phi, this modification is proven to affect also the performance on the Xeon CPU. Therefore, AoS proved to significantly descrease the performance on the Xeon CPU.

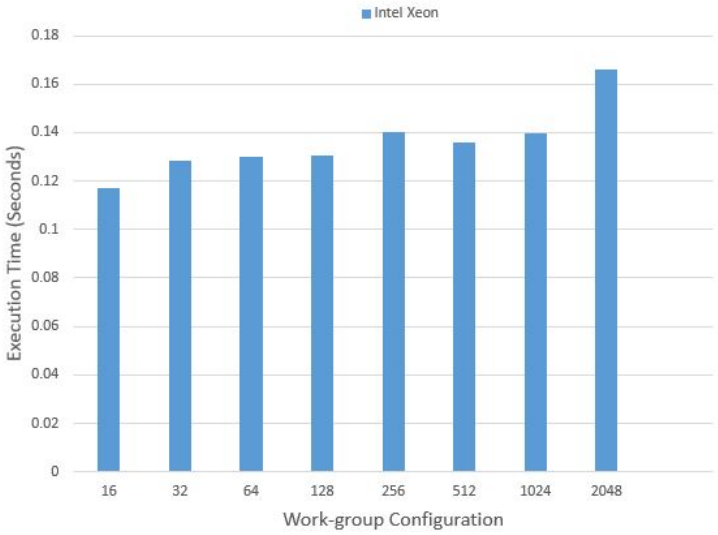

Figure 6.4: Local work-group configuration effect on the Intel Xeon with SoA data layout

With deciding to keep the SoA data layout, only the workgroup configuration was in need for further investigation. As Figure 6.4 depicts, again we tune our kernel for different workgroup configuration. The results showed that the local workgroup size of 16, which the best configuration for the Phi is also the best choice for the Xeon CPU. Therefore, the performance portability for Phi to the Xeon CPU can be achieved without any further tuning.

Performance Evaluation: Speedup

In previous sections we discussed how our kernel behaves on the various platforms. We now compare the actual performance gain. We first investigate execution time as an absolute performance comparison, and we further discuss hardware utilization as a measure of performance portability.

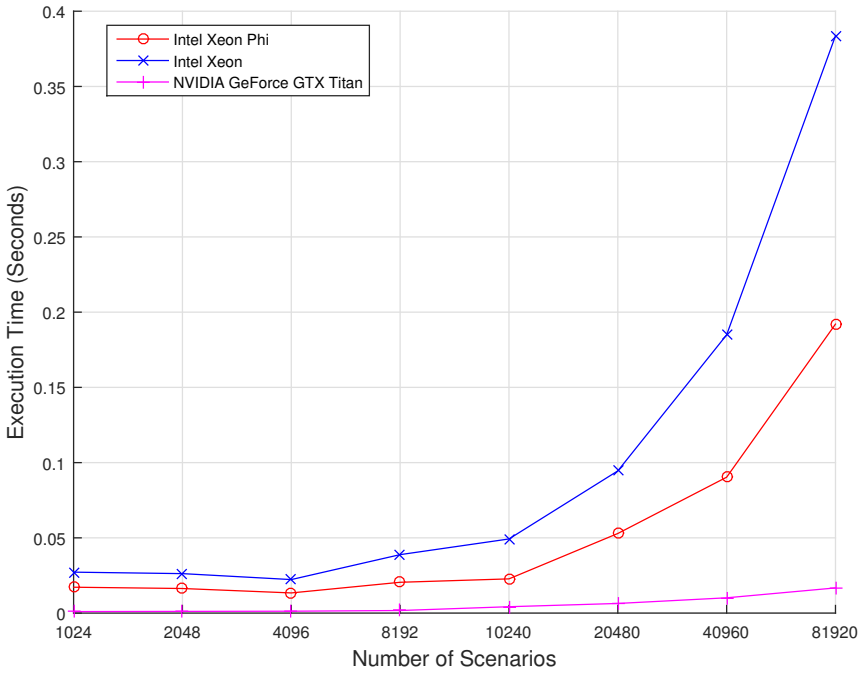

Figure 6.5: Comparison of the execution times for the Intel Xeon Phi, Intel Xeon and the NVIDIA GeForce Titan

Figure 6.5 presents an execution time comparison, while Table 6.1 captures selection of speedups for various input sizes. These two means, provide enough information for a comparison between the different platforms as well as a comparison with our initial scalar code.

In Figure 6.5, a number of interesting insights are illustrated. First we can see that Intel Xeon Phi and Intel Xeon HPC CPU, show similar behaviour. In more detail, both seem to disproportionally increase their execution times for input sizes of more than 40,960 scenarios. On the other hand, the NVIDIA GPU not only produces the fastest execution times, but shows close to linear speedup against the problem sizes.

Table 6.1: A selection of recorded speedups for various problem sizes

Number of Scenarios	Intel Xeon		Intel Xeon Phi		NVIDIA Titan	
	Time (s)	S	Time (s)	S	Time (s)	S
1024	0.0275	x10	0.01719	x17	0.00098	x290
10240	0.04926	x63	0.02106	x148	0.00183	x1707
40960	0.18491	x53	0.08503	x115	0.00862	x1136

Table 6.1 illustrates speedups against the initial scalar code. From these results, we note that the Intel Xeon Phi, in terms of speedup, shows only two times better

performance. This behaviour is unexpected, as the Phi contains thirty times more threads than the HPC-dedicated Xeon CPU.

From the perspective of speedup, NVIDIA GPU yields a 10x improvement compare to the Intel Xeon Phi. Moreover, the NVIDIA GPU provides, for the lowest problem size (e.g 1,240), a 290x speed-up, whithe the Phi and the CPU only reach 17x and 10x, respectively. In addition, all three platforms achieve their best performance can be seen for 10,240 scenarios.

Performance Evaluation: Bandwidth efficiency

As ALM is a memory-intensive application, we originally aimed at a high bandwidth platform. Thus, we selected the Xeon Phi, which features a peak bandwidth of 352GB/s. However, with 59GB/s and 288GB/s peak bandwidth, the Xeon CPU and the GPU are also respectable choices.

In this section, we use bandwidth utilization as a measure of performance portability, aiming to see whether our kernel achieves similar results on all platforms. Our results are presented in Tables 6.2 and 6.3, for 10240 and 81920 scenarios, respectively.

As it can be seen in both Table 6.2 and Table 6.3, even by initially based our development tailored for the Phi, it performs the poorest in term of bandwidth utilization. On the other hand, the NVIDIA GPU has the best performance with 65% and 85% bandwidth utilization, for 10240 and 81920 scenarios respectively.

Table 6.2: Bandwidth Utilization for 10240 scenarios

Platforms	Peak Bandwidth	Bandwidth	Utilization
Intel Xeon	59GB/s	16GB/s	27%
Intel Xeon Phi	352GB/s	29GB/s	8%
Nvidia Titan	288GB/s	186GB/s	65%

Another interesting finding is that both the Xeon and Xeon Phi, behave similarly from the bandwidth perspective. In more detail, while the problem size increases up to a factor of eight, the bandwidth efficiency increases on both by just 2%.For the Intel Xeon Phi, similar results on low bandwidth utilization have been noted before in [64] and[69]. On the other hand, on the same time the efficiency for the NVIDIA GPU increases up to an additional 20%.

Table 6.3: Bandwidth Utilization for 81920 scenarios

Platforms	Peak Bandwidth	Bandwidth	Utilization
Intel Xeon	59GB/s	17GB/s	29%
Intel Xeon Phi	352GB/s	35GB/s	10%
Nvidia Titan	288GB/s	245GB/s	85%

In summary, our original OpenCL kernel written for Xeon Phi achieves better utilization for the CPU and GPU. This result brings evidence that the optimizations we applied are, indeed, not Phi specific and therefore portable, but also demonstrates that, for an architecture like Phi, using the provided bandwidth efficiently does require platform specific optimizations.

6.4 Summary

In this chapter we investigated the performance portability of our OpenCL kernel on the three selected platforms. Initially, we showed that generic optimization techniques are beneficial for all three platforms. For example, using an SoA data layout provides significant impact on all three platforms. The roofline analysis demonstrates that our kernel is memory-bound on all three platforms, which suggests bandwidth utilization as a good portability metric. Our analysis has shown that the Phi has the poorest performance in terms of bandwidth utilization, but the presence of 240 threads compensates this drawback. Thus, our OpenCL ALM kernel optimized on the Phi, performs well in terms of bandwidth on both the NVIDIA GPU and the high-end CPU. We also note that, performance-wise, the GPU is 10x faster than the Xeon Phi, which means the impact of kernel branches is a lot more limited than we originally thought.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

We live in an era where high performance is in high demand in a very broad variety of fields. To match the demand, a large selection of parallel platforms have emerged, using multicore processing units to achieve application acceleration. These platforms vary in terms of both architecture and flexibility in terms of programming. For applications to fully exploit such multicore computing systems, specific techniques and approaches need to be used. Therefore, programmers (1) either need to be able qualify their requirements and align them to the corresponding platform and tools, or (2) use portable programming models and re-use the same implementation on multiple platform. In this work, we focused on the latter scenario.

Computational finance is an area where being able to adapt to the constantly increasing amounts of data, as well as reducing simulation times, are primary concerns. In fact, the core computational finance topics, such as option pricing, algorithmic high frequency trading and risk management define their own trends in HPC platforms and tools.

In this thesis, we focus on a specific computational finance application, and attempt to determine a suitable HPC platform for it. Specifically, we investigate the computational requirements of a scenario-based ALM application, which is part of a commercial product offered by Ortec-Finance. Aiming to use portability to make our search more efficient, we propose a parallel OpenCL algorithm for ALM. Our initial ALM analysis indicated several branches, which could be problematic for GPU processing. Therefore, we selected the Xeon Phi as our main platform candidate. Further, we applied a series of basic optimizations such as data layout conversions, selective use of compiler flags, workgroup configuration and loop collapsing. Moreover, we tried to reduce the branch divergence effect where present. We achieved speedups that varied from 17x for small input sizes to 150x for larger input sizes. We further observed that Xeon Phi can provide good scalability for both single and double precision simulations, compared to a general purpose CPU.

As many Xeon Phi studies suggested OpenMP as the primary programming paradigm for this platform, we developed a fully vectorized OpenMP version of ALM in order to offer a comparative analysis for our OpenCL solution. Our empirical results show that our OpenCL implementation significantly outperforms the OpenMP one.

Furthermore, as OpenCL offers the benefit of functional portability we further investigated the performance portability of our OpenCL implementation. Specifically, we investigated ALM's performance on the most common alternative HPC devices - a HPC CPU and an NVIDIA GPU. The experimental results indicate that our application preserved its good scalability among the platforms. Also, we observed that for both the CPU and the GPU, we manage to achieve good bandwidth utilization (more than 50%) which was one of our issues for the Phi. We also note that as we used very basic optimizations, we needed very little tuning of the kernel for the different platforms. Overall, these results showed that OpenCL can potentially yield portable performance if we optimize for specific metric i.e bandwidth utilization.

Overall, we showed that OpenCL can be used effectively for computational finance applications with characteristics similar to our ALM kernel. Also, we proved that a well optimized OpenCL kernel can outperform OpenMP on native execution on the Intel Xeon Phi. In addition, we illustrated that the use of generic optimizations have not significantly affected the performance levels of OpenCL on other devices, while preserving a decent amount of performance in terms of scalability. Yet performance portability remains a vague term, with no systematic approach for proving/disproving it.

7.2 Future Work

Although we focused our study on a specific test case model, there are many other kernels that need acceleration. Our approach can be extended to other applications than ALM. Furthermore, our portability analysis is rudimentary and based on too few devices, problems that need correcting in the near future.

A first step to extend the findings of this project, is to use OpenCL for other financial applications. As we discussed in Chapter 2 and 3, option pricing applications are dominated by NVIDIA GPUs and CUDA. Thus, there is a very large pool of option pricing established results with CUDA. These results can be used as benchmarks to explore the suitability of OpenCL for option pricing. Potentially, such a design space exploration can be expanded to other generic or custom financial applications (e.g. similar to Ortec-Finance approach).

Further research is also needed to assess the quality of our OpenCL kernel on the latest Phi architecture (i.e. Knights Landing). Unfortunately, at the time we completed this project, KNL wasn't available to the public. KNL promises fundamental architectural upgrades compared to KNC (the device used in this project), and it is the generation that could be used in production in the near future. Such ana-

lysis is therefore necessary. Furthermore, the behavior of optimizations (presented in Chapter 4) on the latest architecture can provide a good estimate of the overall performance gain for the new Phi generation.

Lastly, the performance portability of OpenCL can be further investigated by expanding the results obtained in Chapter 6. In more detail, the novel OpenCL kernel can be ported into other platforms and its performance evaluated; similarly, different sets of optimizations can be applied and their generality can be assessed empirically. Then, by analyzing the results and their patterns, one can establish which hardware specific optimizations can harm performance on other platforms and to what extent.

7.3 Other Contributions

A part of our work (mostly, Chapter 5) was accepted and presented at the *5th International Workshop on Multicore Software Engineering (IWMSE16)*, a workshop associated with the *22nd International European Conference on Parallel and Distributed Computing (Euro-Par 2016)*. The title of the above-mentioned paper is: *Accelerating Computational Finance Simulations with OpenCL.*

Bibliography

[1] *EuroMPI '13: Proceedings of the 20th European MPI Users' Group Meeting*, New York, NY, USA, 2013. ACM.

[2] PC Advisor. Skylake release date and specifications, jul 2016.

[3] Gene Amdhal. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS spring joint computer conference*, 1967.

[4] Folding at Home. Faq: Flops, jun 2016.

[5] Intel Avinash Sodani. Intel xeon phi processor knights landing architectural overview, aug 2016.

[6] Intel Avinash Sodani. Knights landing intel xeon phi cpu: Path to parallelism with general purpose programming, aug 2016.

[7] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, May 2016.

[8] Blaise Barney. Message passing interface (mpi), jul 2016.

[9] Allan 'Zardon' Campbell. Nvidia geforce gtx titan x 12gb, jun 2016.

[10] S. Corsaro, P. L. De Angelis, Z. Marino, F. Perla, and P. Zanetti. On parallel asset-liability management in life insurance: A forward risk-neutral approach. *Parallel Comput.*, 36(7):390–402, July 2010.

[11] Verche Cvetanoska and Toni Stojanovski. Using high performance computing and monte carlo simulation for pricing american options. *CoRR*, abs/1205.0106, 2012.

[12] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[13] Mateusz Iwo Dubaniowski. Exploring performance of xeon phi co-processor. Master's thesis, The University of Edinburgh, aug 2015.

[14] Jianbin Fang, Henk Sips, LiLun Zhang, Chuanfu Xu, Yonggang Che, and Ana Lucia Varbanescu. Test-driving intel xeon phi. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 137–148, New York, NY, USA, 2014. ACM.

[15] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[16] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[17] M. Giles. From cfd to computational finance and back again, nov 2009.

[18] GNU. Gnu gprof: The gnu profiler, jul 2016.

[19] Andreas Grothey. *Massively Parallel Asset and Liability Management*, pages 423–430. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[20] Khronos group. The open standard for parallel programming of heterogeneous systems, jan 2016.

[21] John L. Gustafson. 1the scaled-sized model: A revision of amdahls law.

[22] A. Heinecke. Accelerators in scientific computing is it worth the effort? In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 504–504, July 2013.
[23] Intel. A case study comparing aos (arrays of structures) and soa (structures of arrays) data layouts for a compute-intensive loop run on intel xeon processors and intel xeon phi product family coprocessors, jul 2016.
[24] Intel. Intel skylake idf 2015, jun 2016.
[25] Intel. Intel xeon phi co-processor, apr 2016.
[26] Intel. Intel vtune amplifier 2016, apr 2016.
[27] Intel. Intel xeon processor e3-1535m v5, jul 2016.
[28] Intel. Intel xeon processor e7 family, jul 2016.
[29] Intel. Opencl* design and programming guide for the intel xeon phi coprocessor, aug 2016.
[30] jedox. jedox., jul 2016.
[31] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.
[32] JPmorgan. Using graphic processing units (gpus) in pricing and risk, jul 2016.
[33] JPmorgan. Welcome to j.p. morgan, jul 2016.
[34] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. Improving cuda dna analysis software with genetic programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1063–1070, New York, NY, USA, 2015. ACM.
[35] B. Li, H. C. Chang, S. Song, C. Y. Su, T. Meyer, J. Mooring, and K. W. Cameron. The power-performance tradeoffs of the intel xeon phi on hpc applications. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1448–1456, May 2014.
[36] C. Y. Lin, H. T. Yen, and C. L. Hung. Efficient strategies of compressing three-dimensional sparse arrays based on intel xeon and intel xeon phi environments. In *Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on*, pages 1383–1388, Oct 2015.
[37] G. Misra, N. Kurkure, A. Das, M. Valmiki, S. Das, and A. Gupta. Evaluation of rodinia codes on intel xeon phi. In *2013 4th International Conference on Intelligent Systems, Modelling and Simulation*, pages 415–419, Jan 2013.
[38] Timothy Prickett Morgan. Intel knights landing yields big, aug 2016.
[39] nag. Hpc expertise from nag, jul 2016.
[40] nag. nag, jul 2016.
[41] Q. Nasar-Ullah. Gpu acceleration for the pricing of the cms spread option. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.
[42] NVIDIA. Chapter 45. options pricing on the gpu, jul 2016.
[43] NVIDIA. Introducing nvidia tesla gpus for computational finance, jul 2016.
[44] NVIDIA. Specifications, apr 2016.
[45] American Academy of Actuaries. Actuarial software now. Supplement to Contingencies, 2012.
[46] Khronos Opencl and Aaftab Munshi. The opencl specification version: 1.0 document revision: 48, 2011.
[47] Ortec-FInance. http://www.ortec-finance.com/insurers/asset-liability-management.aspx, aug 2016.

[48] Ortec-Finance. Opal platform goal-based financial planning for private investors, jun 2016.

[49] D. Ozog, A. D. Malony, and A. R. Siegel. A performance analysis of simd algorithms for monte carlo simulations of nuclear reactor cores. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 733–742, May 2015.

[50] R. Pottathuparambil, J. Coyne, J. Allred, W. Lynch, and V. Natoli. Low-latency fpga based financial data feed handler. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 93–96, May 2011.

[51] John Powers. Digipede and barrie & hibbert join forces to grid-enable modeling software, jun 2016.

[52] Thomas Rauber and Gudula Rünger. *Exploiting Multiple Levels of Parallelism in Scientific Computing*, pages 3–19. Springer US, Boston, MA, 2005.

[53] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An experimental study on performance portability of opencl kernels. In *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, 2010.

[54] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Tibor Grasser, and Ansgar Jüngel. Performance portability study of linear algebra kernels in opencl. *CoRR*, abs/1409.0669, 2014.

[55] Elske van de Burg Martijn Vos Sacha van Hoogdalem, Loranne van Lieshout and Ton van Welie. Financial risk management for pension funds. Online, may 2015.

[56] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proc. VLDB Endow.*, 3(1-2):1525–1528, Sept. 2010.

[57] B. Shareef, E. de Doncker, and J. Kapenga. Monte carlo simulations on intel xeon phi: Offload and native mode. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–6, Sept 2015.

[58] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance gaps between openmp and opencl for multi-core cpus. In *Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW 2012)*, September 2012.

[59] Shuo-Li. Case study: Achieving high performance on monte carlo european option using stepwise optimization framework, jul 2015.

[60] Mikhail Smelyanskiy, Jason Sewall, Dhiraj D. Kalamkar, Nadathur Satish, Pradeep Dubey, Nikita Astafiev, Ilya Burylov, Andrey Nikolaev, Sergey Maidanov, Shuo Li, Sunil Kulkarni, Charles H. Finan, and Ekaterina Gonina. Analysis and Optimization of Financial Analytics Benchmark on Modern Multi- and Many-core IA-Based Architectures. *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1154–1162, 2012.

[61] John A. Stratton, Hee seok Kim, Thoman B. Jablin, and Wen mei W. Hwu. Performance portability in accelerated parallel kernels, 2013.

[62] Krishnahari Thouti and S. R. Sathe. Comparison of openmp opencl parallel processing technologies, 2012.

[63] Xinmin Tian, Hideki Saito, Serguei V. Preis, Eric N. Garcia, Sergey S. Kozhukhov, Matt Masten, Aleksei G. Cherkasov, and Nikolay Panchenko. Practical simd vectorization techniques for intel®xeon phi™ coprocessors. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1149–1158, Washington, DC, USA, 2013. IEEE Computer Society.

[64] K. Vaidyanathan, K. Pamnany, D. D. Kalamkar, A. Heinecke, M. Smelyanskiy, J. Park, D. Kim, A. Shet, G, B. Kaul, B. Joo, and P. Dubey. Improving communication performance and scalability of native applications on intel xeon phi coprocessor clusters. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1083–1092, May 2014.

[65] W. Vanderbauwhede and T. Takemi. Twinned buffering: A simple and highly effective scheme for parallelization of successive over-relaxation on gpus and other accelerators. In *High Performance Computing Simulation (HPCS), 2015 International Conference on*, pages 436–443, July 2015.

[66] Sergei Vinogradov, Julia Fedorova, Daniel Curran, Simon McIntosh-Smith, and James Cownie. Openmp 4.0 vs. opencl: Performance comparison. 9 2015.

[67] Wikipedia. Openmp, jul 2016.

[68] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[69] Cheng Zhang, Li Liu, Ruizhe Li, and Guangwen Yang. *Performance Characterization and Optimization for Intel Xeon Phi Coprocessor*, pages 16–33. Springer International Publishing, Cham, 2015.

[70] Yao Zhang, Mark Sinclair, and Andrew A. Chien. *Supercomputing: 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings*, chapter Improving Performance Portability in OpenCL Programs, pages 136–150. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[71] X. Zhao, C. Liu, and G. Tan. Implementation of short read alignment algorithm in opencl on xeon phi coprocessor. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 1633–1636, Aug 2015.

[72] H. Zhu, Q. Zhang, X. Ren, and L. Jiao. Parallel fast global k-means algorithm for synthetic aperture radar image change detection using opencl. In *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pages 322–325, July 2015.