UNIVERSITY OF COPENHAGEN

FACULTY OF SCIENCE

Industrial Ph.D. Thesis

Wojciech Michal Pawlak

wmpk@simcorp.com, wmp@di.ku.dk

Accelerated Financial Algorithms

Derivative Pricing and Risk Management Applications

Department of Computer Science (DIKU): Martin Elsman, Cosmin Eugen Oancea

SimCorp: Allan Peter Engsig-Karup, Carl Balslev Clausen

January 2021

Summary

This thesis concludes a Ph.D. research project in *Accelerated Financial Algorithms*, conducted in collaboration with DIKU and SimCorp. In it, we describe the design and implementation of common compute-intensive financial applications that combine state-of-the-art High-Performance Computing (*HPC*) code optimisation techniques to harness massive parallelism of modern parallel hardware architectures like Graphical Processing Units (*GPU*). We target the acceleration of pricing and risk management of real investment portfolios that consist of complex derivative instruments. We demonstrate our findings through a detailed analysis and practical accelerated implementations of common numerical algorithms. Our research is supplemented with a feasibility study carried out using high-level data-parallel programming languages and frameworks. We propose *Futhark*, which is a purely functional array programming language, as an example of a technology that enables efficient performance, while sustaining modularity, maintainability, and scalability of complex financial algorithms.

We focus on high-level algorithmic specifications and code optimisations that extract enough parallelism from a given algorithm to efficiently map it to high-throughput Graphics Processing Units (*GPU*s). In particular, we use flattening techniques for non-regular nested parallelism; target memory access patterns and size requirements through code optimisation such as data reordering, padding, and access coalescing; introduce inspector-executor approaches to dynamically analyse and adapt to any input dataset; and provide multiple kernel versions that together efficiently cover the entire spectrum of possible datasets.

The *first* contribution addresses an acceleration of a fixed-income derivatives pricing algorithm based on the Hull-White One-Factor Lattice Method (**HW1F**). We introduce a high-level algorithmic specification, which exhibits irregular-nested parallelism, and derive and optimise two hand tuned CUDA implementations: one of which utilises the only outer level of parallelism, while the other utilises both levels of parallelism. The *second* contribution is an accelerated algorithm for equity derivatives pricing that uses a Least Squares *Monte Carlo* Simulation Model (**LSMC**) following a Longstaff-Schwartz model and is implemented in high-level *Futhark* language. We show how an auto-generated implementation beats a manually-optimised one for certain datasets. The *third* contribution is a massive-scale *Monte Carlo* simulation to obtain Value at Risk (**MCVaR**) and other portfolio market risk measures. The simulations comprise multiple nested parallel levels executed on a diverse portfolio of vanilla and exotic derivatives.

Resumé

Denne afhandling afslutter en ph.d. forskningsprojekt i *accelererede finansielle algoritmer*, gennemført i samarbejde med DIKU og SimCorp. I denne beskriver vi udformningen og implementeringen af almindelige beregningsintensive finansielle applikationer der kombinerer avancerede High-Performance Computing (*HPC*) kodeoptimeringsteknikker for at udnytte massiv parallelitet af moderne parallelle hardware-arkitekturer som grafikprocessor enheder (*GPU*). Vi målretter mod acceleration af prisfastsættelse og risikostyring af reelle investeringsporteføljer, der består af komplekse afledte instrumenter. Vi demonstrerer vores resultater gennem detaljeret analyse og praktiske accelererede implementeringer af almindelige numeriske algoritmer. Vi supplerer vores forskning med en teknisk gennemførlighedsundersøgelse udført ved hjælp af højt-niveau dataparallelle programmeringssprog og softwareplatforme. Vi foreslår *Futhark*, som er et rent funktionelt array programmeringssprog, som et eksempel på en teknologi, der muliggør effektiv ydeevne, samtidig opretholde modularitet, vedligeholdelsesevne og skalerbarhed af komplekse økonomiske algoritmer.

I vores forskning fokuserer vi på algoritmiske specifikationer på højt niveau samt kodeoptimeringer, der ekstraherer tilstrækkelig parallelisme fra en given algoritme til effektivt at kortlægge det til grafikprocessor enheder (*GPU*'er) med høj kapacitet. Vi bruger udfladningsteknikker til ikke-regelmæssig indlejret parallelisme; målretter hukommelsesadgangsmønstre og størrelseskrav gennem kodeoptimering ligesom dataregistrering, polstring og adgang til sammenfald introducerer inspektør-eksekutor tilgange til dynamisk analyse og tilpasse sig ethvert inputdatasæt; og leverer flere kerneversioner, der sammen effektivt dækker hele spektret af mulige datasæt.

Det *første* bidrag adresserer en acceleration af fastindkomstderivatprissætning algoritme baseret på Hull-White Enfaktorgittermetode (**HW1F**). Vi introducerer en algoritmisk specifikation på højt niveau, der udviser uregelmæssig indlejret parallelisme, og udlede og optimere to håndindstillede CUDA-implementeringer; hvoraf den ene udnytter det eneste ydre niveau af parallelisme, mens den anden bruger begge niveauer af parallelisme. Det *andet* bidrag er en accelereret algoritme for prisfastsættelse af aktierivater, der bruger en Mindste kvadraters Regression *Monte Carlo* Simulationsmodel (**LSMC**) efter en Longstaff-Schwartz-model og er implementeret på højt-niveau *Futhark* programmering sprog. Vi viser, hvordan en automatisk genereret implementering slår en manuelt optimeret for bestemte datasæt. Det *tredje* bidrag er en massiv *Monte Carlo*-simulering for at opnå Value at Risk (**MCVaR**) og andre porteføljemarkedsrisikomål. Simuleringerne omfatter mange indlejrede parallelle niveauer udført på en forskelligartet portefølje af vanilje og eksotiske derivater.

Preface

This thesis was prepared at the University of Copenhagen in fulfilment of the requirements for acquiring a Ph.D. degree by the Ph.D. candidate Wojciech Michal Pawlak. The project work was carried out in the period between February 2017 and June 2020 at the Programming Languages and Theory of Computing (PLTC) section of the Department of Computer Science (DIKU).

The Ph.D. project was a joint industrial-academic collaboration project between DIKU and SimCorp and was conducted under supervision of researchers and practitioners from both (academic and industrial) environments. DIKU's academic environment was represented by Associate Professor Martin Elsman, who was a Principal Supervisor of the Ph.D. project, and Associate Professor Cosmin Oancea, who was a research co-supervisor. SimCorp's industrial environment was represented by Allan Peter Engsig-Karup Ph.D., who was an Industrial Coordinator of the Ph.D. project, and Carl Balslev Clausen Ph.D., who was a financial business co-supervisor.

SimCorp is one of the largest financial technology (fintech) companies and a leading provider of investment management software solutions for some of the largest financial organisations worldwide. The company is headquartered in Copenhagen, Denmark. During the project, Wojciech was employed as a Senior Research Engineer in SimCorp Technology Labs, a part of SimCorp, which was an internal strategic innovation unit that focused on deriving business value from new technology and enterprise architecture research. The commercial potential of this project was tightly bound and aligned with the in-house system performance strategy in SimCorp. The objective was to produce new state-of-the-art software prototypes, that demonstrate the potential and guide system performance and computational acceleration in SimCorp Dimension, that is the main product offered by SimCorp. The main motivation was tp increase the value of the compute-intensive services that are provided to SimCorp's clients.

The project was supported by a scholarship from Innovation Fund Denmark (Innovationsfonden), which invests in new knowledge and technology, creating growth and employment in Denmark.

Copenhagen, January 2021

Contents

List of Figures

List of Tables

List of Code Listings

List of Abbreviations

ATM	At-the-Money.
AVX2	Advanced Vector Extensions 2 (256-bit).
BSMC	Black-Scholes Monte Carlo simulation.
CDF	Cumulative Density Function.
CES	Component Expected Shortfall.
CUDA	Compute Unified Device Architecture.
CVaR	Component Value at Risk.
EOD	End-of-Day.
EOM	End-of-Month.
ES	Expected Shortfall.
FDM	Finite Difference Method.
GBM	Geometric Brownian Motion stochastic process.
GPGPU	General-Purpose Graphical Processing Unit.
GPU	Graphical Processing Unit.
HPC	High-Performance Computing.
HW1F	Hull-White One-Factor Short Rate Model for Option Pricing.
ITM	In-the-Money.
LSMC	Least Square Monte Carlo for American Option Pricing.
MCVaR	Monte Carlo simulation for Value-at-Risk.

MPI	Message Passing Interface.
MROU	Mean-reverting Ornstein-Uhlenbeck stochastic process.
MS	Market Scenario.
OpenCL	Open Computing Language.
OpenMP	Open Multi-Processing.
OTC	Over-the-Counter.
OTM	Out-of-the-Money.
P/L	Profit and Loss.
PDE	Partial Differential Equation.
PRNG	Pseudo-Random Number Generator.
QRNG	Quasi-Random Number Generator.
RF	Risk Factor.
RN	Random Number.
RNG	Random Number Generator.
SDE	Stochastic Differential Equation.
SIMT	Single instruction, multiple threads.
SM	Streaming Multiprocessor.
SOAC	Second-Order Array Combinators.
SVD	Singular Value Decomposition.
VaR	Value at Risk.

Chapter 1

Introduction

This work deals with the application of *High-Performance Computing* (*HPC*) and high-level programming language techniques executed on massively parallel hardware to accelerate financial algorithms. Although the focus is on applications in finance, the underlying numerical algorithms can be applied to many engineering disciplines such as computational fluid dynamics or climate change prediction. The advanced techniques proposed in this work enable significant efficiency gains through increased computational speed and improved accuracy. These performance benefits may generate significant competitive advantages in all industries that apply such numerical methods.

Financial Perspective

One significant challenge that companies currently face is the need to process massive amounts of data, and do this quickly and efficiently; the financial industry is at the forefront of this challenge. Although the industry has access to a vast amount of data, taking advantage of it requires specialised approaches. In this sense, analysing such data through computational methods has become a key driver of research and innovation in *High-Performance Computational Finance*. Around 10% of all *TOP500* supercomputing power is used for financial computation workloads [TOP20]. A large part of this intensive use of computational power can be attributed to two broad classes of applications: (i) **complex financial instrument pricing**, which represent monetary contracts between entities and are traded in the markets at a large scale, as well as (ii) **financial risk management**, which has moved into the regulatory spotlight since the recent financial crisis of 2007–2008.

The first stems from the fact that complex financial instruments, such as most derivative products, are compute-intensive as the price of most instruments cannot be expressed through closed-form solutions, but instead require numerical approximations. These numerical approximations are based on simulating many variables, often over a long period of time in the future. Simulating one set of variables over one time path can be regarded as one scenario and the more scenarios that are simulated, the more accurate the price. Dealers, who trade such products, rely on instrument prices that are as accurate as possible. In fact, the more accurate the price the dealers have at their disposal, the more precise the profit margin they can calculate in advance, which in turn generates a competitive advantage.

The second is mainly driven by the ambition of financial market regulators to strengthen the resilience and stability of the global financial system. In the early years of financial risk management, the estimation of all risk types was mostly based on relatively simple models, as practitioners focused on the ease of applicability rather than accuracy. Currently the regulators steadily switch from vaguely defined and unrestrictive regulation to direct enforcement of advanced and sophisticated approaches that have well-defined requirements such as Basel III regulatory framework. These advanced methods lead to the introduction of more accurate and thus more complex calculation methods. The aforementioned dealers need to adjust their models to follow these requirements to actively participate in the markets. As a result, they become more confident about the risk that they carry in their portfolios.

Software Engineering Perspective

As a consequence, financial organisations are currently much more inclined to seriously consider the software engineering aspects of the applications that they introduce in their workflows. Ease of code implementation as well as code maintainability, scalability, and reliability, are among the most important factors to be considered in the modern era of many-core computing. The main challenge is the fact that the legacy code bases cannot take advantage of this modern hardware-software paradigm without manual adaptation. Introduction of *HPC* with focus on an efficient use of available hardware resources always involves a significant implementation effort. We claim that the goal of the software engineers should be then to build solutions and tools that make this process as smooth and safe as possible. In attempt to address this challenge in our research work, we explore the current advances in this field from the engineering angle and base our conclusions on a analysis of a series of quantitative experiments.

The *first* thesis of our work is: constructs in high-level programming languages make it possible to express complex code structures that are present in practical algorithms from the financial domain. At first glance, a high degree of performance optimisations is already achievable with the constructs that are available in the established parallel programming frameworks and application programming interfaces (APIs) such as *CUDA* (Compute Unified Device Architecture from *NVIDIA*), *OpenCL* (Open Computing Language), or *OpenMP* (Open Multi-Processing). They are considered to be mature, mainly because they offer a rich tooling and library ecosystem. However, since inception these technologies have always been challenged in widespread and large-scale industry adoption due to their fundamental imperative nature and primary focus on low-level hardware-specific concepts, which leads to a steep learning curve. Effectively, development in the area of parallel programming frameworks is mainly driven by a limited number of specialised engineers that work on directly optimising these technologies for generic use cases, while other companies revert to making calls to their constrained APIs. We recognise that a certain level of manual code optimisation is always going to be required, but claim that, for a vast majority of applications, it is significantly more effective to focus on the algorithmic considerations, while automating the code optimisation process. To support our hypothesis, we decide to shift our focus away from the tools that are currently used in the industry and attempt to show that the situation can be remedied with an alternative programming approach. We choose a high-level hardware-agnostic technique that stems from the latest research in functional programming languages and compiler technology. In our experiments, we observe that the aforementioned high-level constructs can be efficiently mapped to the parallel hardware architecture by an aggressively optimising compiler, effectively hiding complexities of manual parallel code implementation. This feature make the software development process more time efficient and safe. Simultaneously, our results demonstrate that the auto-generated code matches the performance of the manually-optimised.

Business Potential

The industry focuses on meeting their cost-efficiency requirements, which, in particular, are high on the agenda at many financial organisations. We recognise that several approaches exist to meet these objectives, at least when it comes to software applications. Therefore, the *second* thesis of our work is: compute-efficient algorithms, parallel code implementations, and new hardware design that is focused, among others, on energy-efficiency are the most effective and reliable means to make compute-intensive financial computations cost-effective. To illustrate the potential of combining all these concepts, we choose two types of representative applications from the financial domain and adapt them to use these techniques. The applications not only expose a code structure that is typical for financial algorithms, but also lend themselves to aggressive algorithmic and implementation optimisations that in the end show to benefit their performance. Again, the high-level programming approach, which partially automates the optimisation work and allows developers to focus on algorithmic work, is a valuable alternative to a manual implementation process in popular frameworks, which offer a fine-grained control over parallelism.

High performance in computation is the key to success in finance, as portfolio analytics can be perceived as a creative exercise. This ongoing repetitive process, combined with work on the latest available market data, pose a challenging real-time requirement for modern analytical modelling. Analysts initially start with only a rough idea of the current state of the portfolios, most often based on their experience, and filter the available data to eventually reach the conclusions, upon which they can use in their strategic decisions. Therefore, the critical insights, which can change the course of the financial business, depend directly on the techniques and resources that analysts have at their disposal. Analysis results vary greatly, because the systems with long minute-to-hour response times interrupt the analyst's work and reduce the opportunity of creativity, which is essential for analysing and adapting to the constantly changing markets. Rapid execution and frequent alteration for insightful feedback loops is crucial for sufficient exploration of the model space.

At the same time, financial organisations, including investment managers, are notoriously secretive about the models, the algorithms, and the technology that they use. The competition between participants in the financial markets is fierce, so, naturally, they do not share the critical information that can give them a competitive edge. As a result, accelerated computing has to a large extent not been adopted by financial organisations. However, at the same time there are several undisclosed institutions from the financial industry that operate a supercomputer ranked in the aforementioned *TOP500* list year after year [TOP20]. Taking into account the costs of establishing and operating such computational machines, it is evident that there exists a huge demand for such strategic resources.

The increased computational performance, as well as the scalability provided by parallel programming and massively parallel hardware such as Graphical Processing Units (*GPU*s) mean that investment managers are able to increase business at the same time as having tighter control of their risk exposures. With market spreads tightening, it is more difficult to make money and hence it is also more important to be stricter when managing risk to minimise the risk of losses. In addition, *GPU*s, which are designed for high-throughput computing, open new opportunities, for example, by making it tractable to evaluate larger portfolios in shorter periods of time or by enabling analysts to model more complex risk relationships. With an increase in the number of new algorithms being developed, we see an advancement in the mathematical models and trading strategies being deployed on *GPU*s. Other examples of such future applications include intra-day portfolio risk and performance analysis, and use of *Monte Carlo* simulations for a calibration of complex derivative pricing models. These applications were prohibitively expensive 10 years ago, but with modern parallel software and hardware environments, they can be considered mature enough and ready for production.

1.1 Algorithmic and Modelling Scope

This work comprises material that can guide quantitative analysts and developers, who work in finance, into the realm of *HPC* and parallel programming. From a financial perspective, we present implementations of three distinct algorithms that are often used by practitioners, who work daily with large investment portfolios. Two algorithms come from the derivative pricing domain and one from risk management.

In this work, we focus on portfolios of complex exotic derivative instruments. The markets for exotic derivatives are rather illiquid, that is, they are traded infrequently compared to vanilla derivatives such as interest swaps and futures. In other words, a constant price discovery process for exotic instruments does not exist, which stands in contradiction to the liquid markets for vanilla instruments. Furthermore, generally speaking, no analytical solution or closed-form formula exists to compute accurate prices for such exotic instruments. In this case, an accurate price can only be obtained through approximation. Such approximations involve the use of numerical methods such as *Tree Lattices* and *Finite Differences* that construct grids and *Monte Carlo* techniques that run simulations.

The *first* algorithm that we investigate is popular in fixed income and interest rate derivative markets. Here, we want to determine a price of an exotic derivative instrument, a Multi-Callable Floating-Rate Bond, that has a complex cash flow structure and an embedded option that can be exercised at any point in time. In addition, we assume that the derivative instrument is based on a value of a single underlying asset, that is, an interest rate characterised by a yield curve (a term structure of rates). For practical reasons, we need a model that has an ability to calibrate to the current market data. The suitable mathematical model that adheres to the aforementioned constraints is the classical One-Factor Short Interest Rate Model proposed by Hull and White in the series of their articles published in 1990s [HW93; HW94; HW96]. To solve this model, we follow the method proposed in the paper and implement a Trinomial Tree Lattice Model. However, we adjust it further to handle the chosen derivative instrument. We abbreviate and refer further to this algorithm as **HW1F**.

The *second* algorithm is frequently used in the equity derivative markets. In this case, we focus on a valuation methodology for American-style stock options. In contrast to a European stock option, an American option can be exercised at any point in time in the future. Therefore, the valuation problem is in fact a maximisation problem at each point in time in the future with a goal to find the most profitable exercise date. Longstaff and Schwartz, in their, by now classical, article from 2001 [LS01], suggest a numerical solution based on a *Monte Carlo* simulation supplemented with a *Least Squares* regression to approximate the value of such a derivative instrument. We abbreviate and refer further to this algorithm as **LSMC**.

The *third* algorithm comes from the portfolio risk analysis domain. The application is a standard and versatile method to calculate measures that track portfolio value changes and is used to manage investment portfolios of any size and any distribution of assets. The method assumes that future portfolio profits and losses (*P/L*s) are described by a suitable probability distribution that itself is based on prices determined using pricing models. The most frequently used risk measures are *Value at Risk* (*VaR*) and *Expected Shortfall* (*ES*). They can be calculated at the portfolio and instrument levels to determine individual risk contributions. Practitioners use several standard calculation methods to produce these risk measures. These measures vary in (i) how much computation they require and (ii) how well they approximate the *P/L* distribution in the case the portfolio contains more complex derivatives. As we focus on a portfolio of complex derivative instruments that can only be approximated with numerical methods, we choose the most versatile and generic method of a *Monte Carlo* simulation. To evaluate the risk exposure involved in a portfolio, we need to be able to assess the prices of its holdings. Therefore, we use a *Monte Carlo* method for derivative pricing described earlier, and use them as inputs for the consecutive aggregation algorithms. We abbreviate and refer further to this algorithm as **MCVaR**.

1.2 Specific Contributions

The specific contributions of this work relate to how we implement the financial algorithms. Our main goal is to enable the algorithms to be executed efficiently on modern accelerator hardware. We choose a specific platform, *General-Purpose Graphical Processing Units* (*GPGPU*), because the massive parallelism offered by thousands of cores as well as the *Single Instruction Multiple Threads* (*SIMT*) programming model makes it suitable for our investigations. We propose to achieve our goal by using modern software techniques that expose parallelism in such a way that it can be mapped efficiently to parallel hardware.

Hull-White One-Factor Lattice Method (HW1F) for Fixed-Income Derivatives Pricing

We introduce contributions that address the main algorithmic challenge of the examined numerical method, that is the fact that the tree lattices vary highly in size in the realistic portfolio cases. This variability in tree dimensions leads to the issue of workload divergence between threads that handle the derivative pricing. We propose two different implementations in the *CUDA* parallel programming framework that differ in how much parallelism they extract from the algorithm. Based on our experiments, we conclude that providing different versions of the same code (known as *multiversioning*) is the best approach to handle to any portfolio data input. Additionally, we propose and discuss various relevant optimisations and show that, if manually applied with care, the resulting implementation is faster than the code produced with a state-of-the-art aggressively-optimising compiler. A detailed evaluation demonstrates the high impact of the proposed optimisations, as well as the complementary strengths and weaknesses of the two *GPU* code implementations. The two *CUDA* implementations are on average $6.3\times$ faster than a data parallel code produced with a state-of-the-art aggressively-optimising compiler. Moreover, our two *GPU* implementations are on average $2.9\times$ faster than our well-tuned *CPU*-parallel implementation using *OpenMP* multithreading and *AVX2* (Advanced Vector Extensions) vectorisation to take advantage of 104 *CPU* cores. We supplement our results with a detailed validation case that compares our results with a trusted benchmark on a set of different instruments and model parameters. Finally, they are by 3-to-4 orders of magnitude faster than an *OpenMP*-parallel implementation using the popular *QuantLib* library. We emphasise that the numerical method and the optimisations that we apply are transferable to other engineering disciplines. **HW1F** pricing model uses a *bounded* tree, but any Finite-Difference Method that solves a *SDE* or *PDE* (Stochastic or Partial Differential Equation) problem similarly uses a *bounded* grid in both spatial and temporal dimensions. Hence our optimisation techniques are applicable to solving multiple *PDE*s with grids that are variant in time and space.

The *first contribution* is the GPU-OUTER implementation of the algorithm, which optimises only the outer (per instrument) parallelism level of the **HW1F** algorithm and maps individual instruments to separate threads that handle the instruments in isolation. This implementation efficiently *sequentialises* inner parallelism and performs the minimal number of memory accesses. However, it uses only global memory, and does not fully solve the issue of divergence. We show a series of optimisations that address the thread divergence, memory footprint and spatial locality of memory accesses. The optimisations are applied to input data before the computational kernel execution.

The *second contribution* is the GPU-FLAT implementation, which exploits both parallel levels present in the algorithm, that is, not only across the instruments, but also within the tree width associated with the individual instrument. This implementation optimises both levels of divergence and uses predominantly fast shared memory for better temporal locality. However, flattening negatively affects the instructional overhead through indirect accesses and parallel constructs of logarithmic depth (segmented scan). We observe that the spatial dimension of the tree is parallel and map it efficiently to groups of threads that cooperate to handle one or several instruments in parallel. This implementation adheres better to the modern *GPU* architecture with more shared memory available and enables use of this faster memory for thread communication. The trade-off is that we need to perform a dynamic inspector-executor analysis to preprocess input data before we execute the actual computational kernel. The particular flattening step of mapping several valuations to a thread group of a fixed size is highly non-trivial and is a key contribution of this work. Finally, the technique for GPU-FLAT is also beneficial in a distributed-memory (cluster) setting as bin-packing optimises communication and load balancing, while flattening enables intra-node parallelism.

In absolute terms, we can price a portfolio of $100\,000$ exotic fixed-income derivatives, characterised by a random distribution of tree dimensions, in $1.2\,\mathrm{ms}$ on a *NVIDIA Tesla* **V100** PCIe *GPGPU* using GPU-FLAT implementation. GPU-OUTER implementation wins on **GTX 780 Ti** on random and uniform datasets, because (i) the divergence overhead is avoided and (ii) the impact of using shared memory, which is the main GPU-FLAT optimisation, on this *GPU* architecture generation is smaller. In general, the proposed series of optimisations, which address the issues of thread divergence, memory footprint, and spatial locality of memory accesses, need to be currently introduced manually in code, or alternatively be dealt with by an optimising compiler. Our results exhibit the non-trivial areas that can be improved in the current optimising compilers such as inspector-executor methods for reorganising datasets to map them efficiently to parallel architectures. To address this, we demonstrate the necessary steps to generalise our approach and enable our optimisations to become a part of such a compiler architecture.

This work was compiled and submitted in different forms to conferences and journals from a computer science domain, where it underwent a full peer-review process. The experimental research in the current state is under a submission and review process for a domain-specific peer-reviewed conference in supercomputing. In addition, partial results of this work were presented at two poster sessions that were part of Programming and Tuning Massively Parallel Systems (PUMPS) summer school that took place in Barcelona, Spain in

June 2017 as well as at the Ph.D. Forum collocated with 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '18) that took place in Vancouver, Canada in May 2018.

Least Squares Monte Carlo Simulation (LSMC) for American Equity Option Pricing

We propose an approach that addresses the main performance bottleneck of the Least Squares *Monte Carlo* simulation, that is, the regression (optimisation) backward part. This inherently sequential part is used to find the optimal exercise time for American options.

The *first contribution* is a detailed description of the steps that we take to reformulate the mathematical problem to enable partial parallelism in the sequential part. The approach is based on matrix transformations from linear algebra and allows us to isolate the majority of the computation that can be performed in parallel as well as significantly decrease the size of the matrices that are multiplied with each other. The mentioned workload is then precomputed before entering the main sequential loop and saved in global memory for reuse in the loop.

The *second contribution* is the accelerated parallel implementation in *Futhark*, a high-level functional data-parallel language, that targets *GPU*s as the compute platform. We study the feasibility and performance efficiency of expressing a complex financial numerical algorithm with high-level functional parallel constructs. We achieve performance comparable to, and in particular cases up to $2.5\times$ better than, an implementation optimised by *NVIDIA CUDA* engineers. In absolute terms, we can price a vanilla put option, which is a particularly simple instance of an American option, with 1 million simulation paths and 100 time steps in $17\,\mathrm{ms}$ on a *NVIDIA Tesla* **V100** PCIe *GPGPU*. Furthermore, the high-level functional specification is much more accessible to financial-domain experts than the original low-level *CUDA* code, thus promoting code maintainability and facilitating algorithmic changes.

The **LSMC** experimental work was presented at two functional programming conferences. An extended abstract that showed our initial results was presented at ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC '19), which was a part of the ACM SIGPLAN International Conference on Functional Programming (ICFP '19), that took place in Berlin, Germany in August 2019. The full work was presented at International Symposium on Implementation and Application of Functional Languages (IFL '19) that took place in Singapore in September 2019. Consequently, this part of the thesis has been published in the Conference Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL '19).

Monte Carlo Value at Risk Simulations (MCVaR) for Portfolio Market Risk Measurement

The **MCVaR** algorithm is based on the generation of hundreds of thousands of market scenarios that simulate the future portfolio changes. They are used to construct an accurate approximation of the *P/L* distribution for a derivative portfolio. Such distribution enables a calculation of risk measures such as *VaR* or *ES* that are obtained from combinations and aggregations of the portfolio prices in the simulated market scenarios. In our case, *Monte Carlo* simulations are used to both generate the market scenarios and to price the derivative portfolio. In particular, as we deal with a portfolio of complex exotic derivatives, we arrive at a nested simulation problem. When we revaluate a portfolio in each of the future market scenarios, we execute a *Monte Carlo* simulation for each of portfolio holdings. In other words, for each scenario, the portfolio needs to be fully revaluated using simulated input parameters. This nesting structure has a substantial impact on the *GPU* memory that effectively prohibits executing pricing simulations in parallel at the innermost-level, because each of the pricing simulations can alone consume majority of the *GPU* memory. We analyse the risk workload in our implementation by measuring the execution times of *Multicore* and *CUDA* targets compiled from *Futhark* code. We measure execution times of each of its parts and conclude that portfolio pricing part is responsible for majority of the execution time of the complete risk workflow. The portfolio repricing in all market scenarios takes up to $30\times$ on *CPU* and $747\times$ on *GPU* longer than the second most expensive part, that is, market scenario generation. Risk measure calculation is insignificant, even when we calculate the risk measure contributions to portfolio holdings such as *CVaR* and *CES*. Furthermore, we prove our initial hypothesis that we are restricted in our experiments to a sequential execution of portfolio repricing in market scenarios, because **MCVaR** in current implementation consumes prohibitively large amounts of memory in the pricing part. It is especially evident for cases with **LSMC** algorithm used for pricing American Option Portfolio. The already optimised parallel structure of pricing implementations use all the *GPU* compute and memory resources for cases with many sample paths, effectively prohibiting pricing of many portfolio holdings at the same time. However, in our experiments we still observe up to $18.7\times$ speedup of *GPU* execution over *CPU* execution using 32-core on cases, where the inner parallelism over sample paths in the internal pricing simulations is large such as $1\,024\,000$. In particular, we compare execution times between multicore *CPU* and many-core *GPU* platforms. For the largest measured input, the *GPU* version takes $5.9\,\mathrm{s}$ and achieves $3.8\times$ speedup over the highly multithreaded *CPU* version on a portfolio with 10 holdings. The largest single instrument portfolio case that we manage to execute comprises $10\,000$ *MS*s priced using $1\,024\,000$ paths. The *GPU* version takes $41\,\mathrm{s}$ and achieves a speedup of $11.2\times$ over *CPU* version. This proves that *GPU*s are suited for large risk workloads, if we can efficiently map all innermost parallelism (on pricing level) to the core structure of the *GPU* architecture. We conclude that further optimisations that target memory footprint reduction are necessary to extract more performance from the parallel implementation of a complete risk workflow. However, at the current implementation state this needs be achieved through changes in the algorithm. To address these shortcomings, we propose further investigations in use of Quasi-Random Numbers for our implementation. To our knowledge, this is a first large-scale study and application of functional programming to a complete financial risk management workflow.

The *first contribution* is the implementation of several optimisations for minimising the memory footprint, for instance, through reuse of the intermediate sample paths. The code is implemented in the high-level data-parallel *Futhark* language, that can be easily adapted for other portfolio instruments.

The *second contribution* is the approach to deal with a heterogeneous portfolio of different instruments by grouping them dynamically by computational requirements. The computational workload is not equal among instruments, which results in a thread divergence. For example, one can valuate thousands of European options in the same time for one *Monte Carlo* simulation run for an American option.

1.3 General Contribution

From the research perspective, there are two main contributions of this work. The first contribution is an analysis and the practical implementations of selected computational challenges that are present in the field of quantitative finance. The second contribution is a feasibility analysis of using high-level programming languages and frameworks to achieve efficient performance while sustaining modularity, maintainability, and scalability in the implementation of complex financial algorithms. We achieve this through implementing algorithmic prototypes using parallel programming technologies and state-of-the-art code optimisation techniques. We test them on synthetic datasets that simulate different realistic scenarios in financial markets. The discussed results provide a guideline and show how to bridge new technologies to real production systems.

Our work connects advanced results in many disparate research fields. We use standard models from mathematical finance that are well-established in the industry. In the area of computer science, we make extensive use of technologies that are based on the state-of-the-art research in high-level functional programming languages and compiler techniques. In each of the chapters, we provide a focused literature review that attempts to position our specific contributions in the current research spectrum in the specialised areas that we investigate. It is specified in a designated related work section in the end of each experimental chapter, where we relate explicitly to relevant research for each specific contribution from financial and computer science perspectives. Some other contributions are common and can span across chapters as they, for example, relate to specific research in general programming languages and compilation approaches.

1.4 Thesis Structure

The thesis is organised in a following structure.

Chapter 1 sets the scene for the work by providing an overview of the general problem that we deal with in our research, motivates the work presented in the thesis, and gives a summary of the contributions. We focus on how the three research chapters contribute to the overall work thesis.

Chapter 2 introduces some necessary theoretical concepts that are common for all the conducted experiments that are described in the consequent three chapters. Furthermore, we look at considerations and implications of this work from a financial, a computer science, and a business perspective.

Chapter 3 is the first research chapter that presents our work on **HW1F** and discusses our approach to accelerating algorithms from the fixed-income derivative pricing domain.

Chapter 4 is the second research chapter that introduces our work on **LSMC** and discusses our work on acceleration of equity derivative pricing.

Chapter 5 is the third and last research chapter that draws inspiration and builds on some of the work described in the previous two chapters and provides a detailed presentation as well as a discussion of our **MCVaR** experiments and work on accelerated risk analysis.

Chapter 6 concludes the thesis with the main findings of the work and gatherers some promising new directions for future work that we identified during the work on the project.

Chapter 2

Background

The first thesis of our work is: constructs in modern high-level programming language allow us to express complex practical algorithms from the financial domain and have our implementation compiled into an efficient parallel code using an optimising compiler. The second thesis of our work is: compute-efficient algorithms, parallel code implementations, and new hardware design that is primarily focused on energy efficiency, are the most effective and reliable means to make compute-intensive financial computations cost-effective. As a result, this work blends techniques from diverse areas of *computational finance*, *numerical analysis* and *computer science* to demonstrate means to efficiency of financial software implementations.

Computational Finance defines the practical problems in the financial domain using mathematical models that are abstractly expressed using parameterised equations. We choose specific workloads that are recognised as common, and simultaneously are compute-intensive. From a practical point of view, we consider an application workload to be compute-intensive, if (i) it consists of a large number of repetitive mathematical operations applied to large bulks of market data and (ii) it needs to be executed in long-lasting batch jobs rather than on demand, because, for instance, the execution time of its standard sequential code implementation is too long for use during working hours.

Numerical Analysis provides us with the algorithmic tools to discretise and approximate the financial models. In this work, we mostly use classical numerical methods used broadly in many other science and engineering fields. Our main focus is on how to implement these numerical algorithms to build efficient software solutions that can run on modern parallel hardware architectures with the best possible performance. To achieve this, we need to first select the best-suited models from the portfolio of mathematical finance, then apply mathematical transformations where applicable to adapt the code for parallel computation.

Computer Science delivers the tools to implement software that not only makes efficient use of hardware resources and executes quickly, but also enables us to write code that is reliable, scalable, and maintainable. From a software engineering perspective, we aim to express algorithmic complexities and constantly changing logical rules in high level abstractions. This is preferred to delving into the implementation of low-level intrinsic functions, which are used for heterogeneous and parallel hardware architectures. With this view in mind, we look at the modern parallel programming frameworks and high-level functional languages that enable programming parallel hardware. We identify high performance computing techniques and optimisations that benefit the efficiency of financial software.

Theoretical concepts in this chapter are common for all the experiments conducted in this work. In contrast, the following three chapters provide specific and detailed presentation of each experiment. Furthermore, in this chapter we examine considerations and implications of this work from financial and computer science points of view.

2.1 Financial Algorithms

Computational finance uses many quantitative approaches known from other engineering disciplines. For instance, the financial industry bases their business on mathematical modelling to describe phenomena that occur in the financial markets. As markets become fiercely competitive, the need for high-performance computing to simulate and solve these models increases. The situation is further accentuated by the rise of big datasets and real-time data streams that force investment managers to process and analyse data at constantly increasing pace. We examine the field and identify concrete workloads that expose computational challenges. Subsequently, we focus on two specific challenges in our *HPC* research, because they expose algorithmic patterns and scalability issues with regard to a large number of complex mathematical computations as well as large dataset sizes. Exotic derivative pricing as well as portfolio market risk measurement and management are fundamental to the business of all financial organisations today. In this section, we attempt to show why this is a case.

For a general overview of the quantitative finance domain, we refer to Appendix A, where we provide a more detailed classification of its subareas as well as their typical applications. For a more detailed overview of the business setting of this work, we refer to Appendix B, where we specify who the typical users of these applications are, and provide more motivation regarding why it is relevant for them to pursue these techniques.

2.1.1 Computational Challenges in Finance

It is not a coincidence that the numerical algorithms that are most widely used in finance happen also to be the ones that are the most compute-intensive. This aspect leads to a greater-than-ever reliance on high computational power, that an efficient combination of fast numerical algorithms and sophisticated massively parallel hardware can exploit. Computing applications which dedicate most of their execution time to computational demands are considered *compute-intensive*, whereas computing applications which require large volumes of data and dedicate most of their processing time to Input/Output and data manipulation are considered *memory-intensive* or *data-intensive*.

Some of the factors that impact the computation time of a financial application are:

- Complex mathematical models that use multiple correlated stochastic variables to represent changes that drive the financial markets.
- Numerical methods that use simulations, lattices, and grids to discretise and approximate the mathematical models.
- A demand for a high accuracy of market variables as well as resulting prices and risk measures that depend on them.
- Real-time high-frequency streams of market data available online that are fed as inputs to the computations.
- A necessity to perform immediate calculations for diverse input model parameters on demand for intra-day reporting.

In this work, we deal with two specific computational challenges that share all the above factors, that is, exotic derivative pricing and market risk management of large investment derivative portfolio. However, the financial domain is, in fact, rich in compute-intensive applications, which can all be characterised by the aforementioned factors. A list of applications, that is far from being exhaustive, consist of the following examples:

- Parameter calibration of the derivative pricing models based on optimisation algorithms such that the model prices are consistent with market prices [GMS19].
- Stress testing and what-if analysis for portfolio risk management.
- Modelling financial time series and dependencies between them [MFE15].
- Modelling and calculation of key ratios for other types of portfolio risks such as credit, operational, model, etc. [MFE15].
- Modelling extreme tail events in finance [MFE15].
- Computing *xVA* valuation adjustments, such as *CVA*, *DVA*, *FVA*, *KVA*, *MVA*, etc. for hedging against an aggregated counterparty risk of default.
- Investment portfolio optimisation under different market conditions and constraints [CPT18; GMS19].
- Performance measurement on multi-level portfolio investment structures that comprise arbitrarily-nested aggregation levels, such as countries, sectors, currencies, instrument types, etc., characterised by diverse cardinalities.
- Backtesting of trading strategies on historical data.

We mention these applications merely for the sake of completeness and do not discuss them in further detail. With that said, we want to emphasise that the aforementioned challenges are built from the same set of fundamental computational building blocks, so most of the generic implementation techniques covered in this work can be directly applied to efficiently solve them. Therefore, we are especially motivated to focus on the modularity and reusability, if we identify which algorithms are the most fundamental and start by optimising them. A clear example of an interdependence between financial workloads is the link between pricing and calculating measures for risk management. It is impossible to calculate risk measures if we do not know the individual prices of the portfolio instruments. In such a case, if single pricing alone does not exhaust the hardware resources, we can be far more certain to hit the limits when running it multiple times simultaneously to calculate risk measures. In other words, risk management can be perceived as executing pricing at a large scale. As a consequence, when we increase the efficiency of pricing, we immediately impact the overall performance of risk management workloads. Chapter 5 describes such a case, where at the risk level we benefit from the performance optimisations introduced at the pricing level, because we reuse our *Monte Carlo* implementation from Chapter 4.

2.1.2 Derivative Pricing

Despite centuries of academic study, no one can with certainty claim to know what exactly comprise the price of financial assets and what makes it change. It follows from the fact that markets have an infinite number of factors that drive them. Therefore, the best we can do is to prepare a model of how the markets and prices behave. The art of building successful models for derivative pricing is the ability to balance critically determining factors for valuing and hedging a financial instrument while maintaining simplicity in design. The complexity is increased by adding more risk factors that are market variables that change over time, such as an equity price or interest rate. The balance is the key here as the model needs to be tractable and feasible to compute in satisfactory time to be of any use in practice. However, for some cases adding risk factors is inevitable, because there exist derivative types that are constructed deliberately to derive their price directly from many classes of underlying risk factors.

Stochastic Processes

We use stochastic processes to model changes in market behaviour. Different stochastic processes exist: Arithmetic Brownian Motion, Geometric Brownian Motion, Geometric Brownian Bridge, *Ornstein-Uhlenbeck* mean-reverting process, Poisson process, etc. All these stochastic processes can be used to model changes of risk factors that influence derivative prices and portfolio risk. Volatility, which is a measure of the uncertainty of the return realised on an asset, is the main variable driving the value of derivative instruments. In reality, the larger the volatility of an asset underlying an option, the larger the change in the option value.

Early-Exercise and Path-dependent Options

European Call and Put options are simple types of options that are popular in the financial markets, because there exist explicit closed-form formulas that allow us to price such options, for instance, a well-known Black-Scholes model. In practice, if we have a formula, we do not need to use *Monte Carlo* simulations for vanilla options. However, this fact is no longer true for option contracts that have non-trivial cash flows during their lifetime.

Early-exercise options like Bermudan and American options are one example, which we cover in more detail in the next chapters. The underlying asset or assets that the contract relates to can be bought or sold at the agreed strike price prior to the contract maturity, which is the end of the lifetime of the contract. Early-exercise optionality increases the value of any financial instrument, because it gives an option holder a right to quit the financial contract at any time. With some simple exceptions, it is infeasible to model such a multi-faceted optimisation problem with a simple parameterised formula, and thus we need a numerical method to approximate its value. We deal with American and Bermudan options in Chapter 3, where we use a lattice-based numerical method to approximate their value. Furthermore, we price American options in Chapter 4.

Other examples are path-dependent options, for which the payoff, that is, the payment received by an option holder, depends on the history up until the exercise date, which is the date when the contract becomes invalidated. There are many types of option payoffs that are path-dependent to cater for the needs of investors. Several standard examples are:

Basket option An option that depends on the price of multiple underlying assets.

Asian option An option that depends on an average asset price during the lifetime of the option before maturity.

Barrier option An option that depends on the price of the underlying asset breaching a predetermined barrier level.

Cliquet options A series of *ATM* options, where each successive option becomes active when the previous one expires.

All the above options need a numerical approximation like a *Monte Carlo* simulation, where a payoff is evaluated on each time step up until maturity. We show these different examples to illustrate the fact that the complexity of the underlying computations in a generic *Monte Carlo* simulation does not influence the structure of the algorithm, but rather the number of output results that we need to produce to calculate final exotic option payoffs. Naturally, the latter requires more mathematical computations as well as more memory for intermediate and final results. Additionally, these options cover many different underlying asset classes, that is, variables that the price of an option or other derivative depends on. The underlying variables can comprise equities, foreign exchange, commodities, interest rates, inflation, credit, as well as any hybrid combinations thereof. We describe a possible integration of path-dependent options in a portfolio that we consider in Chapter 5. We investigate how they can be priced using a *Monte Carlo* simulation.

Scripting Languages

When we deal with complex exotic derivatives, we fairly quickly conclude that it is difficult to express all the multiple transactions and cash flows over a lifetime of such an instrument. A contract scripting language is a specialised Domain Specific Language (DSL) that is used to separate an instrument definition from a generic pricing algorithm. The language describes contract payoffs in a structured and generic fashion. It can be used to express the aforementioned early-exercise and path-dependent options in a modular way. In fact, it has already been proven that it can be used to express any arbitrary derivative contract. A classical example is LexiFi [PES00] that uses algebraic representation for the contract payoffs. Other work in the area, as presented in [BBE15; And+06; Ege+17; AE18], shows that scripting languages for financial contracts is an active field of research. We do not deal with scripting languages in this work, but we emphasise that high-level abstractions used in functional programming that we apply throughout work can be easily combined with scripting DSLs to build generic pricing engines. Such an approach is one way to increase the functional coverage of our implementations.

2.1.3 Portfolio Market Risk Measurement and Management

This section defines the main financial concepts connected with measuring and managing market risk of investment portfolios.

Risk Management Practice

At present, risk management and analysis is a core part of the day-to-day operations in the financial industry. Responsiveness is the key feature of modern financial risk management applications, because their value is dependent on how fast they can deliver estimates to risk measures like *VaR*, which is the most popular example. They need to be accessible on demand when portfolio managers make key investment decisions, for example, for pre-deal limit checking, that is performed to verify if the risks that the future transactions will introduce in the portfolio do not break regulatory compliance rules or investment capital limits. Furthermore, the applications allows investment managers to meet more widespread systematic stress testing standards that over the recent years have been strictly enforced by regulatory agencies. The *middle office* is the critical department of any investment management (in fact any financial) organisation that deals with market risk management and who hold a critical function. Primarily, they want to understand the market risk profiles of the portfolios that they manage. The ordinary profile includes both short-term *P/L* volatilities and long-term economic risk. Portfolio managers want to quantify how much risk they have amassed in their portfolios and how the total risk compares with their stated risk appetite. Furthermore, they want the group to develop and win regulatory approval of fair treatment of risk-weighted assets, to allow an investment manager to obtain the highest efficiency out of its capital allocation. They use risk measures like *VaR* to model the risk profile. While the requirements for market risk modelling are rather consistent among investment managers, actual practices vary considerably.

Market Risk

Market risk measures the uncertainty of future profits or losses (*P/L*s) that results from changes in market factors (risk factors). Risk factors are prices, interest rates, implied volatilities, FX rates, etc. In other words, they are a complete set of the model parameters that are used in pricing an instrument portfolio. To analyse and report the market risk in the portfolio, we need to apply a risk model. For instance, we combine the relevant instrument pricing and risk factor models that are computed independently from each other, and then apply them to portfolio holdings that are of interest to us.

The goal of market risk measurement is to forecast portfolio return changes based on risk factor changes, their volatility and correlations over time. Risk factors are what makes the portfolio returns change over time. The volatility describes how much the change varies over time. The correlation determines how probable the risk factors are to move together in a given time horizon. As a result, a market risk model evaluates the *P/L* probability distribution of gains and losses (negative gains) to approximate the future value of the analysed portfolio.

Risk Models

Efficient generation of realistic market scenarios, that is, sampling from the joint distribution of risk factors, is one of the most important and challenging problems in quantitative finance.

Historically, sampling from an easy-to-calibrate parametric model of a risk distribution, which is usually represented by a closed-form formula, was a standard solution. An example is a multivariate normal distribution of risk factor logarithmic (non-negative) returns for equity instruments or a Gaussian copula combining a multivariate normal dependence structure with heavy-tailed univariate marginal distributions of individual risk factors that are primarily used for credit instruments. However, although a parametric model is often a poor approximation of reality, mainly its simplicity makes it useful in practice. The goal is to describe the key features of the risk factor distribution with a handful of parameters, achieving the best possible fit to either the empirical distribution derived from the historical data or the prices of traded instruments observed in the market at the time of model calibration. In fact, making the parametric model too complex leads to overfitting and poor generalisation.

A realistic dependence structure is even more difficult to model. A typical parametric approach used in most *Monte Carlo* risk engines starts with modelling the dynamics of relevant risk factors independently, and then imposes a dependence structure by correlating the corresponding stochastic processes. The stochastic processes are, almost invariably, Brownian motions, and the linear correlations between them are enough to construct the joint distribution of risk factors. The key statistics of the return distribution of the risk factors are its first four moments, that is, mainly mean and standard deviation (variance), but also skewness and excess kurtosis, as well as its correlation and tail behaviour. Building a dynamic risk forecast of a global, multi-asset portfolio often requires a choice between a comprehensive fine-grained but possibly complex model and a simple but granular approximated one. There are different types of risk models, among others:

Parametric First-order Delta or Second-order Delta-Gamma approaches are based on Taylor series expansions in the risk factors.

Simulation Historical or *Monte Carlo* approaches are described in more detail in the next paragraph.

In this work, we choose to work with the *simulation* approach, because of its high computational requirements.

Risk Simulation Models

Simulation models do not make any assumptions about distribution of holdings returns, unlike the analytical models. Instead, the distribution of portfolio holding returns is produced by simulating different market scenarios and repricing the holdings with input from these scenarios. Simulation approaches include:

Monte Carlo is a method based on random generation of the market data scenarios. *P/L* distribution is directly dependent on the chosen distribution of the risk factors, for example, normal distribution or Student's t-distribution.

Historical is an intuitive method based on reusing historical data as market data scenarios. History repeats itself, so the distribution of holdings return is probable. The volatility and correlations between risk factors are already embedded in the historical data, so no assumptions about the risk factor or *P/L* distribution.

These two methods differ merely in how we simulate the market scenarios. Once we have the pricing input data, the portfolio valuation and risk measure calculations are identical.

In this work, we take a *Monte Carlo* approach to market scenario simulations. *Monte Carlo* simulations are often compute-, but also memory-intensive, and lead to much longer reaction times compared with the simpler, but less accurate historical simulation. An industrial survey found that average **MCVaR** runtime ranges between 2 and 15 hours [Meh+12].

In stressed environments, where more risk factors are taken into account, these simulations take days on average. As a result, only about 15 percent of investment managers surveyed use *Monte Carlo* simulations as their main approach [Meh+12]. The remaining majority of the organisations do not use them at all or use *Monte Carlo* techniques in limited circumstances, for example, on some complex portfolios or specific risk models.

In reality, the issues of market data quality are often much more important to the investors than the sheer volume of information and the resulting computational challenge. For instance, two main problems with *historical* simulation are selecting the correct historical period to represent expectations about the future, as well as maintaining the data for the selected period. Furthermore, risk models tend to malfunction, when the used data is of low quality, for example data values that exceed their defined limits. Investment managers that use long time series in their historical simulation, for example more than 5 years, often have gaps in data. As a result, they need to use custom algorithms to backfill missing data. On top of that, the burden of computation resulting from enormous volumes is slightly less relevant today. Not only the on-premise IT infrastructure at the financial organisations has already been expanded, but also cloud resource consumption is wide spread. We describe to some degree this important trend of using cloud resources through an infrastructure as a service model in Section 2.3.2.

Market Scenario Evaluation Methods

This section shows that a calculation of risk measures like *VaR* or *ES* involves a combination of a suitable approach to market data scenario generation and the instrument revaluation. In general, to calculate risk measures we first need to establish the *P/L* distribution of the portfolio over the time horizon. The different calculation techniques vary in the computation complexity and their applicability to particular financial instruments. In general, we differentiate two main evaluation methods that translate simulated market data scenarios into the required *P/L* distribution:

Full Revaluation Each risk factor for each holding is reevaluated in each market scenario. This requires an evaluation of a time series for each risk factor over the time horizon.

Partial Revaluation Only the initial scenario is fully valuated. For the remaining scenarios the prices of holdings are approximated with Taylor expansions using *sensitivities* to the relevant risk factors. The *sensitivities*, which are also known as the "Greeks" in derivative pricing, are first-order Delta or second-order Delta-Gamma approximations.

Most investment managers opt to use *partial revaluation* for linear instruments and vanilla derivatives, because in this case both approaches produce identical results. However, *full revaluation* is a more flexible approach that allows a portfolio manager to adjust individual risk factors and correlation assumptions more accurately. For complex non-linear derivatives, *full revaluation* provides a more accurate and realistic calculation of risk that is relevant when the significant risks are present, for instance, in the tails of the *P/L* distribution. In fact, despite the extra cost in time and complexity, the *full revaluation* is mandatory for the non-linear instruments. On top of that, this approach requires a larger number of risk factors than sensitivity simulation. In particular, the *Monte Carlo* approach requires at least 5000–10 000 scenarios to make the model statistically valid in a common case. On the one hand, in *full revaluation*, appropriate use of pricing models is crucial to the accuracy of risk measure estimates, because each portfolio instrument is reevaluated in each scenario. Investment managers that use a *partial revaluation* in this case neglect higher-order risks. On the other hand, in time of high market volatility, executing the *full revaluation* frequently enough turns out to be infeasible for practical purposes. In such circumstances, most investment managers use the approximation approach through *partial revaluation*. In this work, we choose to address a combination that is not only most compute-, but also most memory-intensive: *Monte Carlo* simulations with the *full revaluation* approach. We describe our experiments in more detail in Chapter 5.

2.2 Numerical Methods

2.2.1 Monte Carlo Simulation Method

Monte Carlo is a leading methodology and an essential ingredient in many quantitative investigations. Essentially, a *Monte Carlo* simulation is based on the generation of random objects or processes by means of a computer to model a real-life system such as the evolution of the stock market. There are three typical uses of *Monte Carlo* simulations: sampling, estimation and optimisation. In this work, we are using it to sample from a distribution.

Monte Carlo methods all share the commonality that they rely on random number generation to solve deterministic problems. These methods are especially useful for simulating phenomena with significant uncertainty in inputs and systems with many coupled degrees of freedom. This makes intuitive sense, as the market is difficult to model, has high dimensionality properties, and has infinitely amounts of data to be sampled from. The use of *Monte Carlo* techniques in financial option pricing was popularised in a classic reference [Gla04]. Recently, there have been some significant advances in *Monte Carlo* techniques for stochastic differential equations, which are used to model many financial time series, in particular, [Gil08] and subsequent papers. The *Monte Carlo* method has also proved particularly useful in the analysis of the risk of large portfolios of financial instruments, such as mortgages [MFE15].

Due to tis nature, numerical methods like a *Monte Carlo* simulation can easily benefit from both multithreaded computation on an accelerated compute node and distributed computing at scale. In fact, this approach is a prime example of “embarrassingly parallel” computations. At the same time, a *Monte Carlo* approach to simulations is simple and intuitive enough to apply by practitioners. Most importantly, this simulation approach is the most flexible of all numerical methods, because at its core, it is simply a generic integration scheme. In other words, there are no simulation restrictions to the type of a stochastic process as long as it can be described using an arbitrarily complex *SDE*. Even for models with an analytic solution, *Monte Carlo* simulation is a suitable tool for testing an implementation, for example, to check prices calculated with a complex analytical solution. One of the advantages of *Monte Carlo* method is that its expected error does not depend on the dimensionality of the problem. In fact, for problems in higher dimensions, a *Monte Carlo* simulation is often the only feasible approximation strategy. However, its generic nature comes with a computational price. As a numerical technique, a standard *Monte Carlo* simulation is not sufficiently efficient for practical use, as the sampling error only decreases with the square root of the number of samples, $\sqrt{N}$. In other words, we need to increase the number of simulation paths by 2 orders ($100\times$) if we want to increase the accuracy of the *Monte Carlo* estimation for the expected value by one order ($10\times$). Several techniques exist that improve the convergence rate of a *Monte Carlo* simulation, such as Quasi *Monte Carlo* approach, that we cover in more detail in Section 5.4.2. Other approaches such as antithetic or importance sampling are demonstrated to reduce the computational burden of the *Monte Carlo* approach. However, we do not attempt to introduce them in our experiments at this stage. In this work, we make extensive use of *Monte Carlo* methods in Chapters 4 and 5.

2.3 Accelerated Computation

We stress again that *Computational Finance* is a field where all data is an approximation. This fact makes it different to other engineering disciplines, where we can physically measure quantities to some feasible degree. The feedback loop make it possible for us to test our hypothesis in reality and then adjust our computer simulations. In contrast, the factors that drive financial markets are not directly measurable as they are only statistical and thus abstract notions, which are based on stochastic events. We describe them using *SDE* equations based on stochastic calculus and build algorithms to approximate the solution. The situation is due to some of the mathematical problems in finance are basically too complex to grasp by a simple closed-form formula with several parameters and a finite sequence of elementary operations. The solution to such a complex problem can instead benefit from a fast approximation algorithm. A fast algorithm is one that quickly converges to an accurate solution with an acceptable accuracy to a given significant digit. Modern computer hardware is highly-optimised for such computations based on decades of research in the field [Tre08]. In this section, we focus on the numerical details of the algorithms and the current adoption of massively parallel hardware in the financial sector.

2.3.1 Numerical Accuracy and Floating-Point Precision

In general, numerical algorithms in finance need to be as accurate as possible, but at the same time relatively fast to compute. We start by defining the concepts of numerical accuracy and the floating-point precision as they are fundamental to the computations that we perform in this work.

Accuracy We need to know the number to a specific decimal point, because of its relevance in a given application. Accounting is one financial area, where numbers need to be exact to a certain degree to enable an updated and detailed financial status of the company.

Precision A decimal or real number that can have an infinite number of significant numbers past its decimal point needs to be represented by a limited number of bits representing a floating-point number. Thus, it can be represented only to a certain degree. In practice, this translates to around 11 significant digits for *FP32* and 17 for *FP64*.

One example are computations that involve decimal values that have vastly different exponents need a number representation with larger floating-point precision. Another example from the financial domain are computations that involve large nominals, which represent the number of holdings in an individual portfolio instrument, for instance, number of shares in a given stock equity.

Let us consider an arbitrary portfolio, where we have a large position, that is represented by number of holdings in a particular instrument, and another position that is relatively small in an investment fund that comprises 1000s of different shares. On the one hand, from the computational point of view, the large number of holdings (nominal) in the same instrument, means a risk of running into a floating-point overflow, in a situation when we multiply the instrument price with a large nominal, The *IEEE-754* standard prohibits the representation of such a large number accurately in the hardware memory. On the other hand, simultaneously small values need to be handled accurately too. For instance, a common practice is to buy shares in various investment funds that itself consist of large numbers of shares in various instruments. Effectively, the nominals at the lowest fund constituent level, often represented as fractions per each instrument, can be infinitesimally small. Now let us consider what happens when we want to aggregate all positions in the same instrument across our portfolio. We assume that we not only invest in the large share of the given instrument directly, but also indirectly through the fund. From the computational point of view, it leads to a well known situation in numerical analysis, that addition of a large floating-point number to a relatively small one leads to numerical inconsistencies, if treated without sufficient care.

FP32 vs. *FP64* Precision

From the example in the previous paragraph, we can clearly see that the issue of the sufficient floating-point precision often accompanies the numerical algorithms. Finance is not an exception with its ongoing discussion about how many significant digits are satisfactory to represent the prices of financial instruments. Traditionally, double-precision (64-bit, further abbreviated *FP64*) is the most common choice due to the number of significant digits required for tracking prices. The viability of using less precise single-precision (32-bit, further abbreviated *FP32*) floating-point numbers is a matter of an ongoing debate in the financial sector. On the one hand, many practitioners, especially working in the back office, consider *FP32* as a prohibitively imprecise number representation for their accounting needs. The investment managers are obliged to report calculation numbers in their portfolios precisely due to, for example, regulatory compliance. Moreover, traders, and analysts at the front office need high precision for their mathematical models that grow in complexity. Lowering the precision has a negative impact on the numerical accuracy of an algorithm and diminishes the benefits of the higher approximation accuracy introduced by the models that take more risk factors into account. The impact of truncation errors increases as they involve computations of relatively small changes in these risk factors. Nevertheless, a careful choice and implementation of the numerical algorithms for this purpose is important as well. On the other hand, the computational advantages of using *FP32* can possibly outweigh the need for higher *FP64* precision. Use cases in middle office such as risk management or portfolio performance analysis are based on a large number of statistical samples rather than high precision. In particular risk use cases, less accurate approximations are acceptable or even preferable, so *FP64* is, in fact, not needed at all (see 2.1.3). As a side note, 128-bit or even 256-bit floating-point precision is considered, at least for the purpose of back office accounting. Currently, the largest institutional investment managers that deal with portfolios worth upwards of a trillion dollars run into technical issues, where *FP64* is simply not enough to save the value of the portfolio in a database. Effectively, it is a choice between accuracy and speed, and we always need to accept technical and cost trade-offs.

***FP16* Precision**

The half-precision 16-bit *FP16* floating-point numbers can speed up computation and save up memory space even more than *FP32*, especially on the specialised tensor cores. This precision has been popularised lately due to its use in training of Deep Neural Networks (*DNN*s). These networks require massive amounts of matrix-matrix multiplications, which use so called tensors that are operations on small matrices, that are acceptable to be highly inaccurate (weights of neurons), but need to be fast. As a consequence, the adoption of Data Science and Machine Learning techniques is well under way in finance. For instance, just over the recent 2 years we have observed an active interest in applying Deep Learning techniques to accelerate derivative pricing, which is a workload that involves massive computations on tensors. In concluding Section 6.2 we present a couple of published examples of promising work in a domain of Machine Learning applied to financial algorithm acceleration. In the meantime, we cannot use such low precision in our case, because the insufficient number of accurate significant digits leads to the lack of required numerical accuracy in our simulations. As a result, the algorithms, which we investigate in this work, cannot converge to a result with an acceptable error.

Impact of Parallelism

The differences in the results returned by parallel implementations due to non-deterministic execution order are another technical issue. Operations are not executed in the same sequence, so they can yield different results in different executions, even provided with the same input. In general, different parallel architectures follow the *IEEE-754* standard closely. However, as the discrepancies happen already on the level of the instruction set, their causes are hard to track. In addition, the effect is accentuated further by the fact that accelerator architectures like *GPU*s usually use optimised mathematical instructions at this level. This makes the *GPU* results hard to compare with the *CPU* ones, because the low-level algorithms might differ substantially.

2.3.2 Accelerator Hardware in Computational Finance

As the focus of hardware innovation shifted away from Moore's Law and greater clock speeds, hardware designs moved from single core architectures to multi-core. This fundamentally changed the application execution model, from a single instruction at a time to a set of parallel instructions on many symmetric cores. In fact, future silicon performance wins will come from architectural innovation, rather than from transistor density scaling.

Massively parallel accelerators have been widely adopted for number crunching because of their vast compute capability, highly competitive compute-to-energy ratio, and unprecedented memory bandwidth. For example, *GPU* programming with parallel frameworks like *CUDA* are increasingly viewed as viable technologies for substantial performance improvements of various calculations in financial applications. The highly regular structure of linear algebra primitives frequently used in statistical models can benefit from an enormous reduction of execution time when using *GPU*s. In fact, major banks and investment managers have mostly reported successful deployment of *GPU* solutions to date.

Computational Libraries

Historically, financial software libraries were optimised for single threaded execution on traditional *CPU*s. A prime example is the open-source *QuantLib* library that started as a single threaded library, and only recently started being adapted for thread safety. In the meantime, a mature ecosystem of *GPU*-based generic libraries, both proprietary and open-source, has grown steadily since the launch of *CUDA* in 2007. Among them, we can find full solutions with vendor locked code from specialised financial software companies, who provide custom deployment and maintenance, open source solutions like *QuantLib*, standalone libraries to link from higher level languages such as *Python*, *R* or *MATLAB*, or wrappers around single threaded code.

Low-Level *GPU* programming

Investment managers, that prefer to keep full control of the *GPU* execution of their code, still need to hire quantitative developers with a knowledge of *GPU* programming to implement and maintain their software. It is not straightforward to work with, because single threaded *C* or *C++* programming is still a standard in the financial industry. Learning curve of switching to *CUDA* is steep. In fact, most of the difficulty in using parallel programming frameworks, which is not specific to *CUDA* only, is the lack of experience in thinking using parallel programming concepts. The next challenge is to understand how to work with the thread hierarchy and memory management, as well as how to make use of it to improve performance and scalability of the application. For example, *CPU*s have multiple layers of automatic caching to help mitigate the effect of the memory wall. In contrast, *GPU*s have typically a two-level memory hierarchy with an inner memory, which is shared among a relatively small group of threads, for instance, at most 128 KBs on **V100**. Although *GPU*s have impressively fast memory, the ratio of computation to memory speed is even more uneven than on a *CPU*. As a consequence, accessing memory efficiently is of critical importance to obtain performance improvements.

Cloud computing

Recently large financial corporations have started to migrate their operations to cloud services. Since the cloud providers follow a pay-per-use business model, their customers can use an opportunity to reduce the overall cost of operation (*TCO*). The advantages associated with commodity compute architectures, such as *GPU*s or other accelerators like FPGAs [De 15] and specialised ASICs, have recently granted them a secure place and widespread support in infrastructure services offered by large cloud service providers. This is accompanied by a proliferation of enterprise-ready services based on open-source technologies, that are driven by community-based development. It is becoming more frequent for cloud providers not to limit compute capable hardware solely to large enterprise customers, but rather make them readily available as a service for the general public to use. Operational costs can be tracked and adjusted based on demand using responsive and metered pricing. Moreover, cloud-based compute resources are virtually boundless in scale, which are especially important for distributed computing workloads that execute on clusters of interconnected hardware and use programming paradigms like *MPI*. Finally, it is becoming more common for the latest accelerator devices to be quickly adopted first in the cloud, before other environments like academic clusters catch up. This fact effectively brings down the cost of computational acceleration.

2.4 Parallel Programming Frameworks

Parallel programming denotes programming using multiple hardware cores or processors to gain speed. *Task parallelism* is the simultaneous execution on multiple cores of many different functions across the same or different datasets. In contrast, *data parallelism*, or *SIMD*, is the simultaneous execution on multiple cores of the same function across an array of data elements. In this work, we focus on both many-core parallel programming for *SIMT* or *SIMD* execution model, *GPGPU* programming and, to a lesser degree, multi-core parallel programming for *CPU* programming. We use *CUDA* low-level imperative *GPU* programming style as well as *OpenMP* multithreading and *AVX2* vectorisation in **HW1F** experiments (see Chapter 3). For *CUDA* we provide some essential details in Section 2.4.1. Using *Futhark*, which is described in Sections 2.4.2 and 2.4.3, we can introduce parallel programming in a high-level declarative programming setting and have the compiler perform the automatic dynamic analysis and aggressive code optimisation. We use *Futhark* extensively in our **LSMC** and **MCVaR** experiments (see Chapters 4 and 5).

2.4.1 Imperative Parallel Programming Frameworks

Recent activities of major chip manufacturers, such as *Intel*, *NVIDIA*, *AMD*, and *IBM*, make it evident that future designs of microprocessors and thus large *HPC* systems will be hybrid or heterogeneous in nature. We need specialised languages to program them and enable parallel processing. Currently, if we program in an imperative language like *C* or *C++* and want to have full control over multithreaded execution on a multi-core *CPU*, we most often use a directive-based approach using *OpenMP* or, ever since the *C++*11 standard was introduced, a more fine-grained approach offered by a well-integrated *Standard Threading Library*. These technologies are mature and fully portable across compilers, operating systems and *CPU* hardware. On the contrary, if we want to program a massively parallel *GPU*, we have a choice of a handful of technologies that have matured over the recent 10 years. *NVIDIA* develops the *CUDA* programming toolkit, which is the most popular choice on their *GPU* hardware. The other choices are a portable *OpenCL* API and a directive-based *OpenACC* programming model. Apart from the mentioned technologies, most popular languages today have some kind of an accelerator-enabled backend. For instance, for *GPU* programming *Python* ecosystem offers language wrappers (*PyCuda*), *GPU*-enabled libraries (*CuPy*, *RAPIDS cuDF* and *cuML*, *Dask*) and even *Python-CUDA* compilers (*Numba*). Most of these are contributed to and supported by *NVIDIA* developers. Most popular scientific libraries got their accelerated counterparts that work and scale in the new hybrid and heterogeneous microprocessor configurations. *MAGMA*, *OCCA* and *Kokkos* are just a few examples of *HPC* libraries that manage to bridge the gap between many-core *CPU*s and special purpose hardware and accelerators, especially *GPU*s, that have already outpaced traditionally general-purpose *CPU*s in floating-point performance some time ago.

NVIDIA claims that modern *GPU* generations like Volta and Turing are bandwidth-optimised, general-purpose, parallel processors that implement the *C++* execution model and can run *C++* code. Moreover, the *CUDA* platform that *NVIDIA* has built around its Tesla *GPU*s, with frameworks like *Thrust*, *CUB*, and libraries like *cuBLAS* or *cuRAND*, all of which we use in this project, have made it a lot more straightforward to program

*GPU*s. However, it still involves low-level imperative programming, which is challenging, but rewarding if maximal control of threading is required. To yield good performance, the programmer needs to understand the architectural design principles of the device and have specialised knowledge of imperative code transformations, such as loop interchange, loop distribution, or block and register tiling, for optimising the degree of parallelism and locality of reference. To achieve this, dependence analysis needs to be done manually to reason about the loop-based optimisations, for example, to decide if it is safe to execute a given loop in parallel, or to interchange two loops.

As we use solely *NVIDIA* hardware and software technologies in this work, we adhere to *CUDA*, not *OpenCL* terminology, described below.

Thread in *CUDA* is equivalent to a work-item in *OpenCL*.

Thread block is equivalent to a work-group.

Kernel Grid is equivalent to a NDRange.

Local memory is equivalent to private memory.

Shared memory is equivalent to local memory, typically resident in L1 Cache of size at most 128 KB.

The *CUDA* threads are executed in a lockstep as a **warp**, which is a group of most commonly 32 neighbouring threads. The *GPU* groups threads that execute the same instruction into a warp and then executes them in parallel on *CUDA* cores. Besides warps, the threads are further grouped in thread blocks, which are assigned to a Streaming Multiprocessor (*SM*). Kernel grid has its assigned computer kernel and is mapped to a whole *GPU* device, that comprise several *SM*s. Each *SM* executes instructions in a *SIMT* fashion.

2.4.2 *Futhark* Functional Data-Parallel Programming Language

Distributed and asynchronous processing of interdependent tasks across multiple compute units which can be cores in multi-core *CPU*s, accelerators, or even whole compute nodes, is challenging. It involves management of complex communication among tasks and threads as well as non-trivial synchronisation patterns. The manual design of the aforementioned dependencies is time consuming and error prone. In the ideal case, this low-level layer of complexity is hidden from the developer. We seek a high-level language that allows us to overcome all the aforementioned issues, and instead focus our attention on the algorithmic considerations.

In the last decade, functional programming has gained in popularity due to its declarative characteristics which are effective for parallel computing. Data-parallel programming models seem to be a valid solution to this challenge. They express parallelism declaratively using higher-order language constructs, which build programs from a nested composition of implicitly parallel array operators, which are rooted in the mathematical structure of list homomorphisms. These rich semantics allow high-level reasoning and exploring the large space of strategies for efficiently mapping parallelism in application to the hardware. We reason about the asymptotic (work and depth) properties of our code and consider relevant flattening transformations, which convert (all) arbitrarily-nested parallelism to a more-restricted representation that can be directly mapped to the hardware.

Array-oriented programming languages usually implement data parallelism, because arrays are naturally suited for bulk operations on uniform data. Such languages not only boost the software development productivity, but also provide high-performance parallel execution. As an abstraction, the languages directly mirror high-level mathematical abstractions of linear algebra that are commonly used in financial modelling as well as other engineering sciences. Furthermore, as a language feature, array orientation exposes regular control flow and exhibits structured data dependencies. As a consequence, it makes a programming language more suitable for various types of program analysis, which are focused on code optimisation and correctness. Finally, many modern computer architectures, particularly highly parallel architectures such as *GPU*s and *FPGA*s, are best equipped for an efficient execution of array operations.

Futhark is a statically typed data-parallel functional array language. The language is based on a *ML* or *Haskell* style syntax and is equipped with a number of Second-Order Array Combinators (*SOAC*), such as **`map`**, **`reduce`**, **`scan`**, and **`filter`**. The *Futhark* source language features a higher-order module system [Els+18], polymorphism, and limited support for higher-order functions [HHE18]. Functions cannot be stored in arrays or returned by conditional expressions. Just as higher-order functions and modules are eliminated entirely at compile time, using a no-overhead approach, arrays of records (or tuples) are turned into records of flat arrays. The language also features a uniqueness type system and explicit sequential **`loop`** constructs, which, together with support for array-updates, enable an implementation of imperative-like algorithm in a functional style.

Motivation for Functional Approach

We want to motivate some virtues of functional programming approach that can be particularly valuable to domain experts, that are most often non-technical. Anyone who programs in parallel low-level languages targeting multi-core architectures and does it on a regular basis, observes right away a lack of explicit kernel setup or device memory management in declarative languages. Code remains agnostic to the underlying platform. The aggressive optimisation and parallelisation compiler applies complex code analysis at compilation time to identify suitable performance optimisations. This means that the same *Futhark* code base, which is functional, is also portable across architectures. The same is compiled to a sequential binary running on one *CPU* core as well as to another one running on a massively parallel *GPU* utilising all its parallel cores efficiently. Moreover, when working with *Futhark*, one focuses more on expressing algorithms using and combining the *SOAC*s that mimic the higher-order functions found in conventional functional languages. In *Futhark*, they have sequential semantics, but permit parallel execution, and as such are compiled to parallel code. This means that *Futhark* enables compilation to high-performance parallel code, but still is expressive enough for implementation of non-trivial programs. The purely functional nature of *Futhark* allows the compiler to apply more high-level optimisation, which is its main advantage. In terms of performance portability, *Futhark* supports nested parallelism, so the code can be further autotuned to efficiently support all the available parallelism exposed by the hardware. Otherwise, we face a rather cumbersome and time-consuming task that we need to perform manually.

2.4.3 *Futhark* Compiler

The *Futhark* compiler supports aggressive fusion of parallel constructs [HO13], and specialised code generators for key parallel operators, such as **`scanomap`**, **`redomap`**, and **`segred`** compositions [HLO16; LH17]. Essentially, since all available parallelism is assumed to be explicit in the program, *Futhark* can be seen as a "sequentialising" compiler that attempts to efficiently sequentialise the parallelism in excess of what the hardware can support, thus enabling opportunities for locality of reference optimisations.

In this sense, the compiler supports a code transformation, named *moderate flattening* [Hen+17], that translates arbitrarily-nested, but *regular* parallelism to a flat representation that in a common case can be directly and efficiently mapped to the underlying *GPU* hardware. In general, we deal with "regular" parallelism when the size of the inner-parallel operators is invariant to the outer-parallel nested level. Whereas perfectly-nested parallel constructs can easily be translated to flat parallelism, it is not straightforward to do so with imperfectly-nested constructs.

Furthermore, the *Futhark* compiler implements *incremental flattening* [Hen+19], a notion of generating multiple code versions in situations where the optimal optimisation strategy is sensitive to the input dataset. In this case, the multiple kernel versions are autotuned globally and thresholds are introduced to determine which version is applied. As a result, in most cases, one program offers sub-optimal on all possible inputs. For example, the optimal *GPU* code for dense matrix multiplication depends on the sizes of the matrix dimensions. If enough parallelism is available in the outer two parallel levels, then the dot-product dimension is sequentialised, thus enabling various tiling strategies that optimise temporal locality. Otherwise, the inner dimension is also executed in parallel.

However, given a problem exhibiting irregular parallelism, the *Futhark* programmer is required to implement the flattening strategy manually, which is not necessarily straightforward. The programmer can choose to use a padding approach to flattening and thereby treat all problems to be of the same size. Although this approach can be theoretically work inefficient, in practice it can be the most efficient strategy to use. We note here, that a parallel algorithm is *work efficient* if the work (that is, the number of operations) performed by the algorithm is of the same asymptotic complexity as the work performed by the best known sequential algorithm that solves the same problem. An alternative strategy is to apply a *full-flattening* approach, which leads to a work efficient algorithm, but which is most probably not efficient in practice. Currently, the *Futhark* compiler generates *GPU* code that use *CUDA* or *OpenCL* API as a backend. It can also compile code to run on a multi-core *CPU* through *POSIX Threads* or *OpenCL*. For all the experiments in this work, we use *CUDA*, and in Chapter 4 also to *OpenCL* for comparison of the compilation outcome. In Chapter 5, we additionally use the recently-introduced *Multicore*backend.

Finally, interoperability between parallel languages and mainstream programming environments that are focused on productivity is an issue of practical importance. *Futhark* supports integration with various mainstream programming environments, which can greatly increase its usability in, for instance, larger financial applications. In this sense *Futhark* provides specialised code generators [Hen+16] to languages such as *Python* or *C*, that allow computational kernels that are written in *Futhark* to be easily integrated in larger mainstream projects. An illustrative example is the acceleration of the *BFAST-Monitor* algorithm in *Python* that detects changes in landscape, such as deforestation, by analysing massive amounts of satellite images [Gie+20]. The limitation of *Futhark* is its lack of support for foreign functions to be called from *Futhark* code, albeit certain algorithms, such as radix sorting and matrix multiplication, are more efficiently implemented in specialised *CUDA* libraries such as *cuBLAS* and *CUB*. Other research-oriented parallel languages provide even less integration support. A potential solution could build on the idea of using an interface definition language to specify the interface of type-parameterised software components [OW05b; OW05a; Chi+04], which can be compiled to create standardised wrappers for client or implementation code for each of the supported parallel languages.

2.4.4 Alternative Functional Data-Parallel Languages

Some functional language approaches target a generation of an efficient data-parallel *GPU* code for applications implemented using high-level array language constructs. Such high-level approaches include:

SaC [GS03; GS06; GTS11], a functional array-based language featuring a parallelisable with-loop construct,

Obsidian [SSC11; CSS12; Sve11],

Accelerate [Cha+11], which are both domain specific languages embedded in Haskell.

None of these approaches, however, feature arbitrarily nested parallelism. Approaches that support arbitrarily nested parallelism include the seminal work on flattening of nested parallelism in NESL [Ble96; Ble90], which was extended to operate on a richer set of values in Data-parallel Haskell [Cha+07], and the work on data-only flattening [ZM12]. However, such general compiler-based flattening is challenging to implement efficiently in practice, particularly on *GPU*s [BR12]. Other promising attempts at compiling NESL to *GPU*s include Nessie [RS15], which is still under development, and CuNesl [ZM12], which aims at mapping different levels of nested parallelism to different levels of parallelism on the *GPU*, but lacks critical optimisations such as fusion.

Imperative approaches typically rely on low-level analysis of nested loops using affine indexing and branch conditions for discovering and optimising parallelism, for example the polyhedral model [Pou+11; Bon+08]. Since the affine domain is too restrictive in practice, several techniques use explicit annotations to extend the applicability of polyhedral transformations [RKC16; CSS15]. Other techniques rely on more dynamic analysis coupled with multi-version code generation, for example to identify sufficient conditions under which loops are parallel [MH99; OR11; OR15] or to optimise locality of reference and communication [SHO18; DK99; Rav+14; OM08; Oan+05]. In this sense, our implementation uses multi-version code generation facilities in *Futhark* [Hen+19] to adapt the compilation to the particularities of datasets and hardware.

2.4.5 Data-Parallel Functional Notation

In this section, we present types and semantics of data-parallel operators that are used in this work.

We use **`i32`** and **`real`** to denote the type of a 32-bit integer and *FP32* or *FP64* numbers, respectively, `[n]`α to denote the type of an array with `n` elements of type α, `[`a_1`, ..., `a_n`]` to denote an array literal, and `(a, b)` to denote a tuple (record) value. Function `f` applied to two arguments `a` and `b` is written as `f a b` (without any parenthesis or commas).

The notation supports the usual unary and binary operators as well as (normalised) **`let`** bindings, that is, **`let`** `a = `e_1 **`let`** `b = `e_2 ... **`in`** e_n, and are similar to a block of statements followed by a return denoted by a keyword **`in`**. In-place updates to array elements are permitted and are written as **`let`** `arr[i] = x`. A uniqueness type mechanism supports such updates without any effect on language purity [Hen+17].

Furthermore, the notation supports a conditional **`if`** statement

`if` `c` **`then`** e_1 **`else`** e_2

with semantics similar to the ternary operator `c ? `e_1` : `e_2 from *C* programming language, as well as a sequential **`loop`** expression:

`loop` `(`x^1`, ..., `x^m`) = (`x_0^1`, ..., `x_0^m`)` **`for`** `i < n` **`do`** `e.`

Here, $x^{1\cdots m}$ are **`loop`**-variant variables that are initialised for the first iteration with in-scope variables $x_0^{1\cdots m}$. The **`loop`** executes iterations `i` from `0` to `n-1`, and the result of the **`loop`**-body expression `e` provides the values of $x^{1\cdots m}$ for the next iteration. The initialisation part can be syntactically omitted, that is, **`loop`** `(`x^1`, ..., `x^m`)` **`for`** `i < n` **`do`** `e` is legal. In such cases, the initialisation refers to in-scope variables bearing the same name ($x^{1\cdots m}$).

Above all, the notation supports several key parallel array operators. Their types and semantics are shown in Listing 2.1. We shortly describe each of the operators and supplement them with an example, where relevant. The **`iota`** operator applied to integer `n` creates an array with elements from `0` to `n-1`, that is, an iteration space.

The **`replicate`** operator applied to integer `n` and a value `v` (**`replicate`** `n v`) creates an array of length `n` with all elements having a value `v`.

The **`map`** is a *SOAC* operator that applies its function argument `f` to each element of the input array `as`, resulting in an array of the same length. The function can be declared in the program or can be an anonymous (λ) function. For instance, **`map`** `(λa → a + 1) as` adds 1 to each element of `as`. **`map2`** *SOAC* operator likewise applies its function argument `f` to corresponding elements from its two input arrays `as` and `bs`.

The **`reduce`** *SOAC* operator successively applies a binary-*associative* operator $\odot$ to all elements of its input array `as`, where $e_{\odot}$ denotes the neutral element of $\odot$.

`scan` (also known as parallel-prefix sum) [Ble89] is a *SOAC* operator that is similar to **`reduce`** operator, except that it produces an array of the same length `n` containing all prefix sums of its input array `as`. The *inclusive* **`scan`** (**`scan`**^{inc}) starts with the first element of the array `as`. The *exclusive* **`scan`** (**`scan`**^{exc}) starts with the neutral element $e_{\odot}$.

Segmented **`scan`** *SOAC* operator (**`segscan`**) has semantics of **`scan`** applied to each subarray in an irregular array of subarrays. The latter has a flat representation that consists of (i) a flag array composed of 0s and 1s, where each 1 denotes a start position of a separate subarray, and (ii) a length-matching flat array that contains all elements of all subarrays, ordered accordingly. For example, if `flag = [1,0,1,0,0,0,1]` denotes an array with three rows, having two, four, and one elements, respectively, **`segscan`**^{inc} `(+) 0 flag [1,2,3,4,5,6,7]` results in array `[1,3,3,7,12,18,7]`. In particular, **`segscan`** operator can be implemented in terms of a **`scan`** operator with a modified operator [Ble89], for example, for **`segscan`**^{inc}:

```
λ(f1,v1) (f2,v2) → if f2 != 0 then (f1|f2,v2) else (f1|f2,v1⊙v2)
```

The **`filter`** *SOAC* operator removes all elements in the input array `as` that do not satisfy the predicate `p`. For example, **`filter`** `(<0) [1,-1,2,3,-2] = [-1, -2]`, where `p = (<0)`.

The last *SOAC* operator **`scatter`** updates the array `xs` in-place at indices contained in the index array `is` with the values contained in the array `as`, except that out-of-bounds indices are ignored, that is, not updated. For example, in Listing 2.1, value $\texttt{b}_1$ is not written in the result because its index `-1` is out of bounds.

2.5 Experimental Methodology

We use two *Linux* systems that are described in Table 2.1 across the experiments presented in the Chapters 3, 4, and 5 (Sections 3.6, 4.7, and 5.5).

```
iota: (n: i32) → [n]i32
iota n = [0, ..., n-1]

replicate: (n: i32) → α → [n]α
replicate n v = [v, ..., v]

map: ∀n.(α → γ) → [n]α → [n]γ
map f [a_1, ..., a_n] = [f a_1, ..., f a_n]

map2: ∀n.(α → β → γ) → [n]α → [n]β → [n]γ
map2 g [a_1, ..., a_n] [b_1, ...,b_n] = [g a_1 b_1, ..., g a_n b_n]

reduce: ∀n.(α → α → α) → α → [n]α → α
reduce ⊙ e_⊙ [a_1, ..., a_n] = a_1 ⊙ ... ⊙ a_n

scan: ∀n.(α → α → α) → α → [n]α → [n]α
scan^inc ⊙ e_⊙ [a_1, ..., a_n] = [a_1, a_1 ⊙ a_2, ..., a_1 ⊙ ... ⊙ a_n]
scan^exc ⊙ e_⊙ [a_1, ..., a_n] = [e_⊙, a_1, ..., a_1 ⊙ ... ⊙ a_{n-1}]

sgmscan: ∀n.(α → α → α) → α → [n]i32 → [n]α → [n]α
sgmscan^inc ⊙ e_⊙  [..., 1, 0, ..., 0, 1, ...]
                   [..., a^k_1, a^k_2, ..., a^k_n, a^{k+1}_1, ...] =
                   [..., a^k_1, ..., a^k_1 ⊙ ... ⊙ a^k_n, a^{k+1}_1, ...]

filter: ∀n.(bool → α) → [n]α → [n]α
filter (<0) [1,-1,2,3,-2] = [-1, -2]

scatter: ∀n,m.[n]α → [m]i32 → [m]α → [n]α
scatter [a_0, a_1, a_2, a_3, ..., a_{n-1}]
        [2, -1, 0, 3]
        [b_0, b_1, b_2, b_3] =
        [b_2, a_1, b_0, b_3, ..., a_{n-1}]
```

Listing 2.1: A subset of Data-Parallel Second-Order Array Combinators (*SOAC*), which constititute the operator semantics in *Futhark*. Only operators used in the work are listed.

The compute-oriented *NVIDIA Tesla* **V100** PCIe *GPGPU*, based on the Volta architecture [NVI17; Jia+18], is equipped with 80 Streaming Multiprocessors (32 *FP64* cores per *SM*) that offer a total 7 TFlop/s *FP64* peak performance (14 TFlop/s for *FP32*). The *PCIe 3.0 x16* interconnect bandwidth is 32 GB/s (transfer from/to host main memory). The peak global memory bandwidth (transfer from/to global device memory) is 900 GB/s and peak shared memory bandwidth is 13.8 TB/s. In contrast, for **GTX 780 Ti** the respective bandwidths are 336 GB/s and 3.9 TB/s. In our experiments we observe also that the measured shared memory bandwidth is closer to the peak for **V100** than for **GTX 780 Ti**.

In the **V100** generation the shared memory is merged with L1 Data Cache, so its total capacity per *SM* can be set up to 96 KB. In our experiments, we use a default size of shared memory, that is 48 KB. We do not test the performance on the separate 640 Tensor Cores as we do not use half *FP16* precision for the reasons specified in Section 2.3.1. The impact of using shared memory is higher with newer *GPU*s, because the shared memory size per

D1	**CPU1**	Dual-socket 26-core 2-way HT (104 *CPU* threads) *Intel Xeon* Platinum 8167M *CPU* at 2.00GHz
	RAM	754 GB DDR RAM
	GPU1	*NVIDIA Tesla* **V100** PCIe *GPGPU* Generation: *Volta*, CC 7.0 Compute: 2560 *Volta FP64* cores, 80 *SM*s (32 *FP64* cores per *SM*) Memory: 16 GB HBM2 global, up to 96 kB shared
	CUDA	11.2 and 460.32.03 compute driver
	OS	Ubuntu 20.04
D2	**CPU2**	Dual-socket 8-core 2-way HT (32 *CPU* threads) *Intel Xeon* E5-2650 v2 *CPU* at 2.60GHz
	RAM	128 GB DDR RAM
	GPU2	*NVIDIA GeForce* **GTX 780 Ti** PCIe *GPU* Generation: *Kepler*, CC 3.5 Compute: 2880 *Kepler FP32* Cores, 15 *SM*s (192 *FP32* cores per *SM*) Memory: 4 GB GDDR5 global, 48 kB shared
	CUDA	10.2 and 440.33.01 compute driver
	OS	openSUSE 12.3
D3	**CPU3**	Dual-socket 8-core 2-way HT (32 *CPU* threads) *Intel Xeon* Platinum 8272CL *CPU* at 2.60GHz
	RAM	64 GB DDR RAM
	OS	Ubuntu 20.04

Table 2.1: Device Specification. *CPU*s are dual-socket and have a 2-way Hyper-Threading (HT) enabled, so the effective number of *CPU* hardware threads is $4\times$ the number of *CPU* cores. For instance, for 26-core **CPU1** we have effectively 104 threads to run multithreaded computations in parallel. CC stands for Compute Capability, a unit that *NVIDIA* uses to describe the *GPU* feature set (both hardware and software).

CUDA core and bandwidth seem to grow faster than global memory bandwidth. Table 2.2 shows the improvements of **V100** over **GTX 780 Ti**, cards from two *GPU* generations.

Finally, we want to note that all the experiments are performed on Linux platforms, which in fact has in the recent years become a standard platform for running *HPC* workloads. During the course of the project, we have developed some of the source code on Windows platforms, but we do not perform any benchmark runs on Windows for the experiments that we describe in this work.

	V100	**GTX 780 Ti**	**Per-Core Δ**
#Cores, #SMs, #Cores per SM	5120, 80, 64	288, 15, 192	
Shared memory size per SM	48KB (max: 96KB)	48KB	$3\times$
Peak Shared Memory Bandwidth	13.8	3.9	$2\times$
Peak Global Memory Bandwidth	900	336	$1.5\times$

Table 2.2: Computational and memory resources comparison and evolution demonstrated on the example of two *GPU* generations used in the work, an older **GTX 780 Ti** *GPU* from 2013 optimised for graphics and *FP32* operations and a later **V100** *GPU* from 2018 optimised for compute and *FP64* operations.

Chapter 3

Hull-White One-Factor Lattice Method (HW1F)

Abstract

This chapter reports on the acceleration of a standard, lattice-based numerical algorithm that is widely used in finance for pricing a class of fixed-income vanilla derivatives. We start with a high-level algorithmic specification, exhibiting irregular-nested parallelism, which is challenging to map efficiently to *GPU* hardware. From it we systematically derive and optimise two *CUDA* implementations, which utilise only the outer or all levels of parallelism, respectively. A detailed evaluation demonstrates

1. the high impact of the proposed optimisations,
2. the complementary strength and weaknesses of the two *GPU* versions,
3. that they are on average $6.3\times$ faster than a matching data parallel code auto-generated by a state-of-the-art aggressively-optimising compiler, and that
4. they are on average $2.9\times$ faster than our well-tuned *CPU*-parallel implementation (*OpenMP* multithreading and *AVX2* vectorisation), and by 3-to-4 order of magnitude faster than an *OpenMP*-parallel implementation using the popular *QuantLib* library.

Section 3.1 introduces the specific derivative pricing problem that is present in the fixed-income markets, motivates the importance of accelerated implementation, outlines our approach, and summarises the main contributions.

Section 3.2 presents the relevant financial background necessary to understand the problem and the algorithm used to solve it. Subsection 3.2.1 gives a minimal description of the financial instruments that are priced in this part. Subsection 3.2.2 describes the essential mathematics behind the used financial model. Subsection 3.2.3, then, examines in depth the algorithm that is used to differentiate the model.

Section 3.3 presents the data-parallel structure of the pricing algorithm with emphasis on how the algorithm can be parallelised.

Section 3.4 shows how on-level GPU-OUTER implementation uses only the outer level of parallelism, while optimising spatial locality, memory footprint and (partially) thread divergence by data reordering.

Section 3.5 derives and discusses the two-level GPU-FLAT implementation which flattens the nested parallelism at block level, thus optimising both levels of divergence and which executes mostly in the fst shared memory.

Section 3.6 presents the methodology, the experiments and their results as well as the detailed evaluation to demonstrate the impact of proposed optimisations, and the overall performance of the *GPU* code versions and discusses the results.

Section 3.7 relates the work to other projects on acceleration for similar pricing problems and compares the used optimisation techniques with the current research.

Section 3.8 concludes on the main outcomes.

3.1 Introduction

Pricing is a fundamental computational component for the *risk management* of any investment portfolio. It applies mathematical modelling and compute-intensive algorithms to accurately approximate the value of any complex financial derivative instrument. A derivative is a contract that derives its value from other, more basic instruments like fixed-income bonds. A classic example is an Over-the-Counter (*OTC*) instrument baring a custom cash flow, which is not liquid (not traded frequently enough) and thus its price cannot be observed at the exchange (or *marked to market*). In this case, the price needs to be *approximated* by a mathematical model (or *marked to model*) that parametrises certain features that have an impact on the behaviour. The model needs first to have its parameters *calibrated* with the current market situation to yield a meaningful result to practitioners. The calibration is usually based on the simpler more liquid exchange-traded instruments, for which prices are quoted in the market.

Modern derivative portfolios, managed by large institutional investors, consist usually of a large number of such contracts, which differ significantly in their characteristics and cash flows. This diversity has a severe impact on the required computational effort. In addition, the combined size of the whole interest rate derivative market alone is immense with the notional number currently outstanding in the order of hundreds of trillions of American dollars [Int].

This chapter reports on the *GPU* (and *CPU*) parallelisation of an algorithm that solves a standard pricing model, which is commonly used in practice. Hull-White One-Factor Short-Rate (**HW1F**) model [HW94] defines the instrument value by a stochastic differential equation, that represents the random changes in the interest rate over time. The algorithm used to discretise and solve them is a *Trinomial Tree*, a lattice-based numerical method proposed and described in detail in [HW94; HW96]. In this method, pricing a single derivative instrument comprises two main stages. The forward stage builds a tree of bounded width, representing the propagation of the interest rate until the maturity of the underlying bond is reached. The tree has bounded width, because the interest rate tends to revert to the mean value over time. The backward stage then performs the instrument valuation from maturity back to the current time. The computational structure thus consists of two sequential time-series **`loop`**s, with number of iterations equal to the height of the tree, in which each iteration performs several semantically-parallel operations, which have the same length as the bounded width of the tree. For example, the forward step computes each node at the current level from 3 nodes at the previous level in the tree.

Practical use cases require pricing a (large) portfolio of instruments, which gives raise to an outer level of parallelism, as instruments can be priced independent of each other. Practical use cases require pricing a (large) portfolio of instruments, which, in principle, can be parallelised in a straightforward manner by mapping individual instruments to different threads. All current approaches related to the acceleration of this pricing method:

- have considered the simple case in which all trees that have the same width and height, and
- have utilised only the outer parallelism level of the instruments in the batch, while sequentialising the inner parallelism available in the valuation of one instrument.

Section 3.7 comprise a survey and a comparison with the current research.

The main challenge is related to the fact that in realistic scenarios *the width and height of the trees is highly variant* across the portfolio instruments. This gives raise to a case of irregular-nested parallelism that is difficult to map efficiently to *GPU* hardware. In particular, related approaches, surveyed in detail in Section 3.7, either assume the homogeneous case [HT05; Ger04] in which all trees have the same height and width, or acknowledge the problem, but do not offer a solution [Gra+13]. To our knowledge, this work is the first to address the *GPU*-acceleration of the **HW1F** model solved by the bounded trinomial-tree numerical method. Related work considers other pricing models or numerical methods used to solve them.

This chapter studies this challenging case of a heterogeneous portfolio, and reports on two parallelisation strategies and their subsequent optimisations.

The first strategy follows the common wisdom that the outer levels of parallelism are more profitable to exploit than the inner ones. As such, given a big enough portfolio, it sequentialises the inner parallelism and performs one instrument valuation per thread. Section 3.4 presents our implementation of this strategy, named GPU-OUTER, together with a set of optimisations aimed at improving memory footprint and spatial locality (that is, coalesced accesses to global memory).

In particular, the high variance of widths and heights across a portfolio introduces two levels of thread divergence on *GPU*. These levels correspond (i) to the sequential time **`loop`** of variant height and (ii) to the inner width-parallel operations, which are sequentialised. GPU-OUTER allows to optimise *one level of divergence, but not both*; for example, by precomputing the widths (or heights) of the instrument' trees, and by sorting the portfolio in decreasing order of their widths (or heights).

The second strategy is to optimise (i) the height level of divergence by sorting as before, and (ii) the width level by efficiently exploiting the (irregular) inner level of parallelism at thread block level, which also allows to maintain most of the data in the fast (shared + register) memory.

The idea is (i) to *pack instruments into bins*, such that their summed widths fit the size of the thread block (bin), and then (ii) to "*flatten*" (merge) the available two-level parallelism. The first level corresponds to the instruments in a bin, and the second level corresponds to the width-length parallel operations that appear in the implementation of each instrument.

The flattening step is highly non-trivial and is a key contribution of this work. In Section 3.3, we document it by presenting an initial simplified nested-parallel specification that uses operators such as **`map`** and **`reduce`**. In Section 3.5, we then demonstrate how flattening is performed for the specific application, that is a subject of this chapter, resulting in a program named GPU-FLAT.

A detailed experimental evaluation on two *NVIDIA GPU*s shows that:

- The proposed optimisations (data reordering, padding, coalescing global memory accesses) have high impact: for GPU-OUTER as high as $6.0\times$ and on average $4.1\times$ and for GPU-FLAT as high as $5.8\times$ and on average $2.7\times$.

- *GPU* performance is sensitive to both dataset and hardware characteristics. On older *GPU* hardware, GPU-OUTER is slightly faster than GPU-FLAT by a factor as high as $2.3\times$ and on average $1.8\times$ on large portfolios with constant or randomly-distributed heights and widths. GPU-FLAT is faster in all the other cases, that is, on small portfolios ($3.4\times$ on average), or when the distribution of dimensions (widths or heights) is skewed (as high as $15.3\times$ and on average $5.9\times$) or even random on newer *GPU* hardware ($1.8\times$).

- To demonstrate that the current compiler technology does not currently support the proposed optimisations, we compare GPU-OUTER and GPU-FLAT with "matching" implementations written in the data-parallel language *Futhark* [Hen+17]. Our best *CUDA* version is faster than *Futhark* on all datasets by a factor as high as $29.8\times$ and on average $6.3\times$.

- Finally, the best *GPU* version is faster by a factor as high as $6.7\times$ and on average $2.9\times$ than our tuned multithreaded and vectorised *CPU* implementation (*OpenMP* + *AVX2*), named CPU-MT+VECT. In addition, our *CPU* implementation is $3 - 4$ orders of magnitude faster than an *OpenMP*-parallel implementation built on the popular *QuantLib* library [Bal21; Bal14], named QUANTLIB-PAR.

3.2 Financial Background and Algorithm

3.2.1 Option as a Derivative and Bond as an Underlying Asset

Fixed-income Instruments like Bonds

In this work, we deal with a specific class of bond instruments, that are heavily traded in fixed income markets by the largest financial institutions. The debt market, where bonds, among others, are traded, is by far the largest and most liquid of all the financial markets. Bonds are characterised by a cash flow during their lifetime. We consider a bond paying a coupon rate on specified dates up until bond maturity. The value of coupons is most often fixed for fixed-rate bonds, but varies for floating rate notes (FRNs) depending on some agreed reference rate. The payment dates are usually periodical, but we assume they can occur on any future date. The bonds having a cash flow are traded more frequently as they offer larger premiums in contrast to zero-coupon bonds. However, due to the fixed coupon, the market value of a fixed-rate bond is susceptible to fluctuations in interest rates, and therefore has a significant amount of interest rate risk.

Derivative Instruments like Options

A derivative (a contract) derives its own value from the value of the underlying asset. A *call* or *put* bond option is a derivative contract that gives an investor the right, but not the obligation, to, respectively, buy or sell the underlying bond for a predetermined strike price at a future exercise date before the bond maturity. In general, options as financial instruments are used for hedging of portfolio risks or market speculation.

Embedded (Callable, Puttable) Options

In the domain of fixed income instruments, a *call* is used to benefit from a decline in interest rates and a respective increase in bond prices, while a *put* from the inverse, an increase in interest rates and a respective decrease in bond prices. These options are considered to be embedded in the bonds and valued together accordingly. In a practical setting, a call option on a *callable or redeemable bond* gives the bond issuer the right to redeem the bond. The bond holder has sold a call option to the issuer. The issuer is protected and receives an opportunity to refinance its debt at a more attractive rate. On the other hand, a put option on a *puttable or retractable bond* gives the holder the right to demand early redemption. The holder has purchased a put option and receives a protection against an unattractive rate.

Option Types and Styles

We differentiate traditional *vanilla* options and more complex and custom *exotic* options. A call or a put are typical examples of vanilla options. Furthermore, we differentiate a few styles of a vanilla options. Exercise time of the options determines its style. *European* can only be exercised on one particular date, while *Bermudan*, on many specific, usually periodical, dates, and *American*, on any date up until the last agreed exercise date. Analytical formulas exist for an immediate and exact valuation of European options, but the value of the other two can only be approximated by numerical methods. Exotic options differ from vanilla options in their payment structures, expiration dates, and payoff functions. They are based on the assumption that the underlying asset varies over time, that leads to more hybrid investment alternatives. In fact, exotic options are often packaged as financial products and customised to fit the needs of the investor.

Instrument Assumptions

In this work, we deal with a *multi-callable or puttable bond with a fixed or floating coupon*. We focus on Bermudan or American optionality, but capture any bond cash flow. Moreover, we assume that bonds have only one underlying factor, an interest rate of the currency, they are traded in. For a more detailed descriptions of the interest rate derivatives, we suggest the standard literature on derivative pricing [Hul17] as well as material focused on interest rate instruments [BM06; AP10].

3.2.2 Hull-White One-Factor Short Rate Model for Option Pricing

One of the most popular short-rate models for interest rate derivative pricing is *Hull and White* (1990, 1994a) [HW90; HW94]. We consider a one-factor model, which has a single source of risk or uncertainty, a *short rate*, that is an interest rate applicable at instantaneous or very short periods of time. The model is based on a diffusion stochastic process, which describes a probabilistic evolution of a random short rate over future time. The process assumes the future interest rates are a function of the initial ones with their movement reverting to the mean over time.

In this work, we consider a simplified version of the *Hull-White* extension of the *Vasicek* model, where volatility σ and mean reversion rate a parameters are constant. In strict mathematical terms, the dynamics of **HW1F** are expressed by a *Stochastic Differential Equation* (*SDE*):

$$dr = [\theta(t) - ar]dt + \sigma dW \tag{3.1}$$

The equation defines a continuous process, which can be decomposed into a drift term and volatility (Brownian motion) term. The drift coefficient of (first) dt term in the Equation 3.1 drives the evolution of the process. The term includes the constant short rate r, function $\theta(t)$, mean-reversion rate a and an infinitely small increment of time dt. r follows a mean-reverting *Ornstein-Uhlenbeck* (*MROU*) stochastic process, that is pulled back toward a central value with a rate a. $\theta(t)$ is a deterministic function of time chosen to fit the theoretical bond prices to the market yield curve and defines the average direction that r moves with rate a at time t. The volatility coefficient of (second) dW term in the Equation 3.1 determines the amount of "noise" or variability of a process, and as such models risk. The term comprises of volatility σ and the stochastic variable dW, which follows the random normal distribution with mean 0 and standard deviation 1. In practice, σ models the absolute level of short rate volatility, while a determines the relative volatilities of long and short rate. A high value of a causes short-term rate movements to damp out fast, and thus reduce the long-term volatility. **HW1F** is an arbitrage-free model and, as such, is appreciated by the practitioners for its ability to reproduce an arbitrary market yield curve and enable calibration of model parameters to the observed market prices.

3.2.3 Hull-White Trinomial Tree as a Numerical Method

Hull and White (1994a, 1996) [HW94; HW96] outline a trinomial tree construction procedure for solving **HW1F** model. We present an algorithmic procedure and refer to the original research for a more rigorous mathematical description of the method.

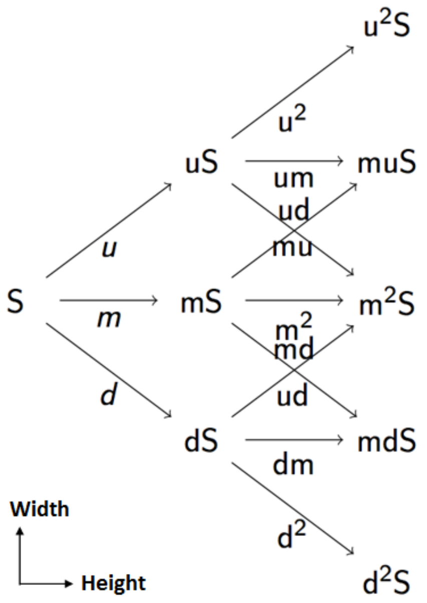

Figure 3.1: A trinomial tree as in Boyle (1986) [Boy86] that shows a tree construction procedure for a stochastic variable S.

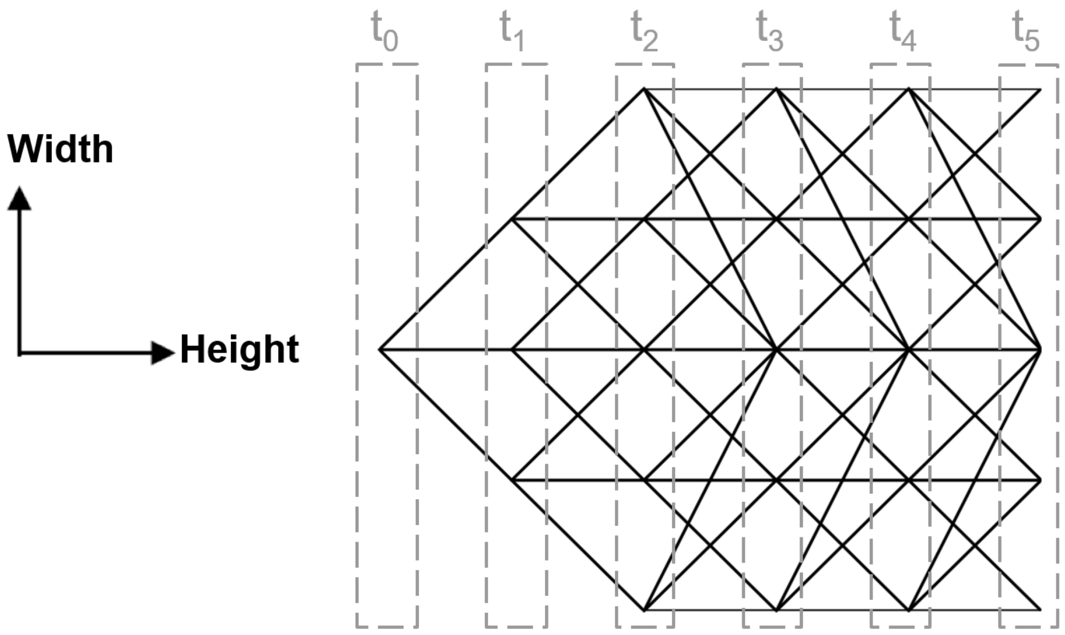

Figure 3.2: A Hull-White Tree [HW94], which is a trinomial tree with a bounded width that incorporates the mean reversion phenomenon.

Forward Propagation: Tree Construction

The tree is constructed in a breadth-first (in this work, width-first) manner. It starts from the root, which corresponds to the current time, and ends at the leaves, which correspond to the bond maturity time. The nodes at a certain height level denote possible values for the short rate at that time step. The tree *height* is defined by the remaining time to the maturity T of the bond, specified as a year fraction. The tree *width* is determined by the mean reversion rate a (the lower the rate the wider the tree), specified as a basis point. The *height* varies across bonds, while the *width* is determined by a calibration of a to the current market data to estimate the volatility of the underlying interest rate. Moreover, both dimensions depend on the frequency of time steps, that is, how often we monitor changes in the interest rate. Every bond depends on a *term structure* (or a *yield curve*), a relationship between interest rates and maturity terms. It represents a market expectation of interest rate evolution over future time, and in practice is constantly updated based on the market situation. The original chapter [HW94] introduces two stages in the *forward propagation*, *tree construction* and *tree displacement*, which implies that the whole tree is traversed twice.

The first stage constructs a preliminary tree, where $\theta(t) = 0$ and initial value $r = 0$. We define r as a continuously compounded Δt-period rate. We denote the expected value across a Δt-period $r(t + \Delta t) - r(t)$ as $-ar(t)\Delta t$ and the variance of $r(t + \Delta t) - r(t)$ as $\sigma^2\Delta t$. The time step size is $\Delta t = T/n$, where T is the bond maturity and n is the number of desired time steps. The interest rate step size is then $\Delta r = \sigma\sqrt{3\Delta t}$, which is a theoretical choice. Δt progresses along *height*, while Δr are distributed along the *width*. We denote the tree node by (i, j) for which $t = i\Delta t$ and $r = j\Delta r$, where i denotes the time step along the *height*, and j the rate step along the *width*. By convention, the node in the middle of a tree level has index $j = 0$.

A *trinomial tree* represents a random propagation of the interest rate in time. In our case, the value of node (i, j) is the Arrow-Debreu price $Q_{i,j}$, which corresponds to the value of a security that pays 1 if node (i, j) is reached and 0 otherwise. In Figure 3.1, the weights on the edges represent the probabilities of transitions from one value to the other. The tree is considered "trinomial", because the computation of each node value at a current time step i depends on three node values from the previous time step $i - 1$. In particular, at each node there is a choice u, m and d to branch upward, horizontally, and downwards, with probabilities p_u, p_m, and p_d, respectively, where $p_m + p_u + p_d = 1$, as in Figure 3.1. The tree probabilities are dependent on a chosen n and a and are chosen to match the expected change and variance of Δr over the next interval Δt, and are calculated from equations for each branching.

In practice, the total sum of branching probabilities becomes negative or greater than 1 for large negative or positive values of j. To mitigate this problem, Hull and White propose to subdivide the original short-rate trinomial tree into two time regions along the *height* dimension to obtain a *Hull-White tree*. They match parameters of their model with empirical observations and determine that the region changes from one to the other, when the step index reaches $j_{max} = -0.184/M$, where $M = e^{(-a\Delta t)} - 1$. In particular, the *normal* region propagates the probabilities as in an ordinary trinomial tree from time 0 to some particular height i, at which width of the tree reaches $2j_{max} + 1$. The *bounded* region (from time $i + 1$ onwards) changes what happens with the probabilities at the boundary widths j_{max} and $-j_{max}$.

In more detail, the current node (i, j) contributes to the three nodes at the next time step $(i + 1)$, as follows:

- if $j = -j_{max}$ then horizontally to node j, and upwards to nodes $j + 1$, and $j + 2$;
- if $j = j_{max}$ then horizontally to node j, downwards to nodes $j - 1$, and $j - 2$;
- otherwise horizontally, upwards, and downwards to nodes $j, j + 1$ and $j - 1$, respectively.

Figure 3.2 shows an example, where $j_{max} = 2$, width is 5, height is 6, $i \in [0, 5]$ and $j \in [-2, 2]$.

The change in the tree geometry not only allows for bounding a width dimension, which makes the algorithm more tractable from a computational perspective, but also leads to faster tree construction (less nodes to traverse) and more accurate pricing (the algorithm converges faster). We use this fact in Section 3.5 to extract parallelism from the tree algorithm and efficiently map the implementation to the *GPU* architecture, where amount of parallelism we can exploit is limited by hardware. However, the most important motivation from the modelling perspective is the fact that this tree geometry intuitively fits the empirical observation that the interest rate reverts to the mean over time. In other words, particular high and low interest rates values are considered unlikely to happen. The current market situation in 2020s questions this assumption and forces us to reevaluate our models. The Hull-White model treats interest rates as normally distributed. This leads to scenarios in which interest rates are negative, though there is a low probability of this occurring as a model output.

The second stage adjusts the constructed tree from the first stage to match the initial term structure observed in the market. We displace the nodes (that is, the Arrow-Debreu prices Q) at time $i\Delta t$ by an amount α_i, that is, the new value of $Q_{i,j}$ in the displaced tree is then equal to $Q_{i,j} + \alpha_i$. The value of α_i is determined from a sum of Arrow-Debreu prices Q across all nodes at the previous time step $(i - 1)$ and the bond price $P(0, i)$ with maturity in the current time step i. In particular, $\alpha_0, \alpha_1, \alpha_2, \ldots$ for time steps $0, 1, 2, \ldots$ are determined successively to ensure that the prices of zero-coupon bonds with maturities $\Delta t, 2\Delta t, 3\Delta t, \ldots$ are matched, respectively.

Our implementation combines the *tree construction* and *displacement* stages, by performing them one after another at each time step to avoid traversing the tree twice.

Backward Propagation: Option Pricing

Once the complete term structure has been calculated at each node, the tree can be used to value a wide range of derivatives. We use a *backward propagation* to value a bond with embedded options from the maturity back to the current time at the root. At each step we adjust for the cash flows, accrued interest or the eventual option exercise and discount the bond price. We end up with the estimated bond price as of the calculation day at the root of the tree. This computation reuse α values computed during *forward propagation*.

In the following sections, `Qs` denotes the Arrow-Debreu prices, which are computed in the *forward propagation* stage and `Cs` denotes the bond prices with an embedded optionality computed in the *backward propagation* stage. Finally, we note that both forward and backward propagations need to be implemented as sequential **`loop`**s, because the values of the current time step depend on those computed in the previous time step.

3.2.4 Motivation behind the Model and Algorithm Choice

In this section, we present a number of arguments that motivate our model and algorithmic choice.

Versatile Pricing Model

First of all, we emphasise that although in this work we use the Hull-White model to find the price of a callable bonds with a Bermudan or American optionality, in practice the model is commonly used to price many other exotic interest rate derivatives like multi-legged interest rate swaps, bermudan swaptions, or spread options. Our proposed implementation optimisations are also applicable to these other types of derivatives.

Parallelisation Enables Simultaneous Valuations

By parallelising **HW1F** Trinomial Tree algorithm, we enable a valuation of many instruments at the same time, which is highly beneficial for risk workloads like **MCVaR** that valuate thousands of instruments against millions of scenarios. Therefore, fast valuation is a building block for larger computational workloads, where performance is expected to scale with the size of the input. In fact, we need a large number of trees available in big computation cases to reach an optimal performance on a massively parallel architecture like a *GPU*. An alternative method that can be used in a **HW1F** case are *Monte Carlo* simulations. This approach is popular on *GPU*s, because of an embarrassingly-parallel nature of the algorithm. Sets of paths are independent of each other map and can thus be directly to a large number of threads available on a *GPU*.

Real-time Computation Enables New Use Cases

Let us define real-time performance execution as one that executes in a matter of a single second. One use case that is a realistic candidate for a real-time computation is a portfolio risk analysis. It requires ad-hoc simulations, where a change in one position in portfolio requires full portfolio repricing. In an ordinary case with typical sequential implementation, an analysis on a relatively small portfolios is the only feasible option. However, if pricing one tree is sub-millisecond than large portfolios with thousands of positions can be repriced on demand in real-time, Such a case is an improvement for portfolio decision-making.

Trade-offs between Execution Time and Accuracy

Let us now focus on the trade-off between the execution time and accuracy. Each **HW1F** tree itself can comprise of arbitrarily large number of nodes that are determined by its height and width dimensions. In this work, we constrain us to relatively small trees dimensions and still manage to get significant execution time speedups, because we are able to efficiently group and then map many smaller trees to all threads available to handle them in parallel and maximise the *GPU* occupancy. However, the same is true, if we increase the dimensions of the trees for these instruments, for which it is necessary. The way how we map tree nodes to threads does is indifferent to the size of the tree as long as it is bounded and smaller than the maximal number of threads that can be executed at the same time on a*SM*. The better the approximation the more accurate is the price. We achieve that using a more fine-grained lattice. The consecutive changes (differences) between tree nodes in rate and time dimensions are decreased and the size of the tree grows as a result. If we can handle many nodes in parallel, we can handle better approximations faster. As a result, some trees, that are infeasible to run under practical time constraints, can now be processed. We present an example, where we show an advantage of having high accuracy. In our experiments, tree height is at most equal to 1200, which allows for high accuracy in representing any bond cash flow during its lifetime, for instance, on a day-by-day basis. Let us now assume an extreme case where we have one time step for each business day in the year, 252 of them a year. 1200 time steps means that we can represent about 5 years until bond maturity, which is a realistic case. Let us further assume a case where there are semi-annual coupons on a fixed-rate bond during the lifetime of the bond with a early-exercise opportunity on each day (an American option). Due to daily treatment of the time steps, we can map a coupon to an exact day. Moreover, width has a limit of 1024, which enables a fine-grained treatment of short-rate probability value. This number of nodes across width allows us to handle the changes in the underlying stochastic process with a sufficient accuracy for industrial needs. As a result of such a setup, we can simulate changes in interest rate on a day-by-day basis with high accuracy.

Tractability

Our choice of a model that can be represented by a bounded trinomial tree is also motivated by the tractability of this numerical method, which is a valuable feature for practitioners. The trinomial tree, as well as the binomial tree, gives us tractability, because we have the possibility to test for early-exercise at each time step in the tree, for example, each day, until the maturity of an option. With this approach, we do not need to reduce the number of monitoring dates. One more model feature follows from the fact that a tree can be represented as αs vector. Thus, a portfolio manager can save the tree in this form and retrieve it later for offline inspection.

Comparison with Other Models

Lattice models are more computationally stable and accurate in low dimensions (one-factor, one stochastic process) than other numerical methods. The first alternative numerical methods that can be used for solution of **HW1F** model is a Finite-Difference Method, for example through a standard *Crank-Nicolson* discretisation scheme. The problem is formulated as a *PDE* and then discretised as a grid of finite differences to approximate bond and option prices. In fact, binomial and trinomial trees are perceived as easy to use alternatives to *FDM* for implementing one-factor models of the short rate. *FDM* are similar to lattice methods, because a trinomial tree can be seen as a particular implementation of an explicit *FDM*. *FDM* suffer from their own complications like (i) finding accurate assumptions to the boundary conditions and (ii) adapting the procedure to match the initial term structure. The second alternative is to use a *Monte Carlo* simulation in forward step to generate random paths for changes in bond prices. This method requires *SDE* formulation of the model.

Model Extensions

As shown by Hull and White (2001) the tree building procedure can be extended to accommodate non-equal time steps and values for a and σ that are functions of time. Hull and White (2016) show that their method can be expanded to three dimensions, where two curves are used to price exotic options. Hull and White (2017) generalises the idea of tree construction to accommodate in a realistic way the existence of or a possible future occurrence of negative interest rates.

3.3 Simplified Nested Data-Parallel Specification

This section uses functional notation that was introduced in Section 2.4.5 to present the nested-parallel structure of the pricing algorithm defined in Section 3.2.3.

In comparison to a lower-level **`loop`**-based notation, such as *CUDA*, the functional notation has the advantage that it (i) enables a concise specification of all available (nested) parallelism in terms of well-known data-parallel operations such as **`map`**, **`reduce`**, and **`scan`**, and (ii) allows to demonstrate at a high level the rewrite rules, that are applied to derive GPU-FLAT. However, we omit the description of the rewrite rules in this work.

Listing 3.1 sketches the (simplified) implementation of the pricing algorithm, which nevertheless accurately captures the nested-parallel structure. The **`main`** function (at the bottom of the Listing 3.1) receives a portfolio of instruments and performs a valuation of each by an outer embarrassingly-parallel **`map`** operation, that can be easily distributed across

```
 1 let valuate (ins : Instrument) : real =
 2   let (w,h) = f1(ins)
 3   let Qs = replicate w 0.0
 4   let Qs[w/2+1] = f2(ins)
 5   let αs = replicate h 0.0
 6   let α_i = f3(ins)
 7   let αs[0] = α_i
 8   let (_,αs) =
 9     loop (Qs, α_i, αs) for i < h-1 do
10       let Qs' = map (λj →
11         let q0 = Qs[j]
12         let q1 = if j > 0   then Qs[j-1] else 1.0
13         let q2 = if j < w-1 then Qs[j+1] else 1.0
14         in  g1(i, j, α_i, q0, q1, q2)
15       ) (iota w)
16       let α_v = reduce (+) 0.0 Qs'
17       let α_i'= g2(α_v, ins)
18       let αs[i+1] = α_i'
19       in  (Qs', α_i', αs)
20   let Cs = replicate w 100.0
21   let Cs =
22     loop (Cs) for ii < h-1 do
23       let i = h - 2 - ii
24       let α_i = αs[i]
25       in map (λj →
26         let c0 = Cs[j]
27         let c1 = if j > 0   then Cs[j-1] else 1.
28         let c2 = if j < w-1 then Cs[j+1] else 1.
29         in  g3(i, j, α_i, c0, c1, c2)
30       ) (iota w)
31   in Cs[w/2+1]
32
33 let main (portfolio : []Instrument) =
34   map valuate portfolio
```

Listing 3.1: Nested-Parallel Implementation of **HW1F**.

threads, *GPU*s, or nodes. We can use such an approach, because, according to our portfolio assumptions, each instrument depends only on a single unique underlying asset.

Function `valuate` receives an instrument data as an argument and computes its price approximation. Computation starts by determining the width `w` and height `h` of the trinomial tree (at line 2), and by initialising arrays `Qs` of size `w` (lines 3-4) and α`s` of size `h` (lines 5-7). The two sequential **`loop`**s of indices `i` and `ii` implement the forward and backward tree propagation.

The first **`loop`** (lines 9-19) fills in the values of an array α`s`. First, the **`map`** operation of length `w` (at lines 10-15) computes each element across width at the current time step (height) level in the tree, that is, `Qs' [j]`, by aggregating the three different values belonging to the previous time step level, that is, `Qs[j-1]`, `Qs[j]`, `Qs[j+1]`. Note that only the current and previous time step elements across width levels, rather than the entire tree, are manifested in memory by arrays `Qs'` and `Qs`.) The newly created array `Qs'` is then summed up, through the **`reduce`** operation at line 16. and provides the value of α`s[i+1]`. Note that both parallel operations occur inside the outer **`map`** operation that is applied to the whole portfolio, and thus giving raise to nested parallelism. Finally, the resulted values of `Qs'`, α`_i'` and α`s` are bound to the **`loop`**-variant variables `Qs`, α`_i` and α`s` for the next iteration.

The second **`loop`** (lines 22-30) traverses the tree backwards, from the maturity to the present date. At each step, the iteration computes the prices associated to the current width level by a similar **`map`** operation. The price of the instrument today is at the root of the tree, corresponding to `Cs[w/2+1]` after the **`loop`**.

3.4 GPU-OUTER: Outer-Parallel Version and Optimisations

GPU-OUTER is derived by mapping each instrument to one thread, thus sequentialising the inner parallelism available in the `valuate` function. While most of the low-level (*CUDA*) implementation is straightforward, one non-trivial issue refers to the fact that arrays such as `Qs` and α`s` need to be *expanded* across all valuations in the portfolio and to be stored in global memory. There exists no suitable statically-known upper bound for their length, and as such they cannot be stored in *CUDA* shared memory. This section discusses two performance critical optimisations. The first finds a good layout for the expanded arrays, $\mathtt{Qs}^{exp}$ and $\alpha\mathtt{s}^{exp}$. The optimisation enables coalesced access to global memory, while minimising the memory pressure. The second diminishes the overhead of one of the two levels of thread divergence, that arises from the per-thread computation across both the width `w` and the height `h` of the tree. The optimisation addresses the cases, where both `w` and `h` vary significantly across threads (instruments).

3.4.1 Naive Expanded-Array Layout

We assume that the portfolio consists of `n` instruments. Subsequently, we determine a naive layout by using a **`map`** to pre-compute the width and height of each of the `n` trees into two arrays `ws` and `hs`. We sum up these two arrays to produce the total length of $\mathtt{Qs}^{exp}$ and $\alpha\mathtt{s}^{exp}$. Next, we compute the starting offsets into $\mathtt{Qs}^{exp}$ of the logically-local arrays `Qs`, one for each iteration of the outer **`map`**, by applying a $\mathbf{scan}^{exc}$ operation on `ws`. The operation results in an array named `Qs_offs`. For example, the logical arrays `Qs` corresponding to iteration (thread) `i` of the outer **`map`** is represented by the slice $\mathtt{Qs}^{exp}$`[Qs_offs[i]:Qs_offs[i+1]]` of length `ws[i]`. Similar considerations apply to $\alpha\mathtt{s}^{exp}$. The inspector code is presented below, where **`unzip`** transforms an array-of-tuples to a tuple-of-arrays, that is,

$$\mathbf{unzip}\ [(a_1,b_1),\ldots,(a_n,b_n)] = ([a_1,\ldots,a_n],\ [b_1,\ldots,b_n]).$$

```
let (ws, hs) = unzip (map f1 portfolio)
let Qs_offs  = scan^exc (+) 0 ws
let αs_offs   = scan^exc (+) 0 hs
let len_Qs^exp = Qs_offs[n-1] + ws[n-1]
let len_αs^exp = αs_offs[n-1] + hs[n-1]
```

In practice, the implementation of the parallel inspector above fuses the **map** and the two **scan** operations, leading to an efficient code, especially if the operation is based on a single-pass **scan** implementation [MG16]. However, this layout is problematic, because it leads to poor spatial locality. Consecutive threads access global memory with a large stride, that is equal to the values of `w` (or `h`). This leads to accessing global memory in an uncoalesced fashion, which can be prohibitively expensive on *GPU*s.

3.4.2 Global Padding Enables Coalesced Access

Matters can be improved by computing the maximal width $\texttt{w}^{m}$ and height $\texttt{h}^{m}$ across all `n` trees and padding each *local* array to this size, that is, $\texttt{Qs}^{exp}$ and $\alpha\texttt{s}^{exp}$ are now two-dimensional arrays of sizes $\texttt{n}\times\texttt{w}^{m}$ and $\texttt{n}\times\texttt{h}^{m}$, respectively. Modulo thread divergence (imbalanced) issues, fully coalesced access to global memory can be achieved by working with the *transposed* versions of $\texttt{Qs}^{exp}$ and $\alpha\texttt{s}^{exp}$. The inner array dimension of size `n` is indexed by the thread (instrument) number, hence consecutive threads access consecutive memory locations, thus achieving coalesced access to global memory. The main downside is a potential explosion in the memory footprint, for instance, under skewed distribution of width or height values.

3.4.3 Block or Warp-Level Padding: Coalesced Access at a Small Memory Overhead

The memory explosion can be remedied by padding at finer granularity, for example, at thread block or warp level. Let us denote the block (or warp) size by `B`. Padding is then accomplished by (i) finding the maximal width (and height) for each group of `B` instrument trees, (ii) followed by padding to the maximal size within the group, and (iii) by working with the transposed version of the arrays, as before. In the following we assume for simplicity that `n` is a multiple of `B`.

```
let wbs' = reshape (n/B, B) ws
let wb_max = map (reduce (+) 0) wbs'
let pad_lens = map (λw → w*B) wb_max
let blk_offs = scan^exc (+) 0 pad_lens
```

The implementation of the parallel inspector is shown above: (i) the precomputed array of widths (heights) is **reshape**d as a $\frac{n}{B}\times B$ two-dimensional array, then (ii) a **segred** operation finds the maximal width of each group, (iii) the padded lengths of each group is obtained by multiplying the maximal widths by `B`, and (iv) the start offsets into the expanded array for each block (warp) are computed by a $\texttt{scan}^{exc}$ operation.

For example, the expanded array for block `b` is the slice

$$\texttt{Qs}_b^{exp} = \texttt{Qs}^{exp}\texttt{[blk\_offs[b]:blk\_offs[b+1]]},$$

which is seen as a two-dimensional array of shape $\texttt{B} \times \texttt{wb}_{max}\texttt{[b]}$, This means, that the start-index of logically-local array `Qs` corresponding to local thread `i` is

$$\texttt{blk\_offs[b] + i}\star\texttt{wb}_{max}\texttt{[b]}.$$

To obtain coalesced access, the implementation manifests the **`transpose`** of $\texttt{Qs}_b^{exp}$ of shape $\texttt{wb}_{max}\texttt{[b]} \times \texttt{B}$. The operation requires that we find the maximal width and height for group at the level, and then pad and transpose at that level, which can be achieved using $\texttt{scan}^{exc}$.

3.4.4 Data Reordering Optimises One Level of Thread Divergence

The code in Listing 3.1 exhibits two levels of divergence, because the body of the `valuate` function is executed (sequentially) by each thread, which corresponds to an instrument in the **`map`** over the entire portfolio. The recurrences appearing inside `valuate` are (i) the two forward- and backward-traversal **`loop`**s of count `h`, and (ii) the enclosed (inner) **`map-reduce`** computations of length `w`. Since both the height `h` and width `w` of the tree varies significantly across instruments, it follows that both parameters are sources of a thread divergence and their combination further exacerbates it. For example, if two threads executing in lockstep have $(\texttt{w}_1,\texttt{h}_1)=(1,\texttt{m})$ and $(\texttt{w}_2,\texttt{h}_2)=(\texttt{m},1)$, then their execution takes $O(\texttt{m}^2)$ time, rather than the expected $O(\texttt{m})$ time. With the GPU-OUTER implementation, one of the levels of divergence (but not both) can be optimised by a pre-processing step that sorts the portfolio of instruments after the heights or widths of their corresponding trees. In practice, sorting in the decreasing order of the *widths* (rather than heights) is more beneficial because it improves the degree of coalesced access to frequently-accessed array such as `Qs` and `Cs`, especially for the version performing padding at block or warp level. To conclude, we report that all pre-processing (inspector) overheads are small, because they sum up to under 2% of the total execution time.

3.5 GPU-FLAT: Flattening Two-Level Parallelism

This section demonstrates how the GPU-FLAT implementation, which utilises both levels of parallelism, is derived from the nested-parallel code of Listing 3.1, in a way that simultaneously optimises (i) both levels of divergence and (ii) temporal locality. We keep the discussions here intuitive and specific to the trinomial tree algorithm. However, the transformation can be formalised by a set of inference (rewrite) rules that can be integrated in the repertoire of a compiler. We omit presentation and discussion of the rewrite rules, because they are not considered a contribution of this work.

Essentially, GPU-FLAT uses both levels of parallelism, which allows to simultaneously optimise (i) the temporal locality and (ii) both levels of divergence. The idea is to first sort the instruments in decreasing order of their heights, thus optimising the divergence of the forward and backward traversal **`loop`**s. Then we pack the input (instruments) into bins, such that the summed widths of their trees do not exceed the thread block size, which determines the capacity of the bin. We choose the maximal size 1024 that is the upper bound on the number of threads in a thread block. The two parallel levels, one of the instruments in a bin and the other of the inner parallelism inside `valuate` function, are then combined (flattened) and mapped at the thread block level, while the parallelism across bins is mapped on the *CUDA* grid.

On the one hand, this implicitly optimises the width-level of thread divergence, because the flattened parallelism has roughly the size of the thread block. On the other hand, temporal locality is also optimised because the data created by inner-parallel operations (inside `valuate`), such as the arrays `Qs` and `Cs`, is maintained in fast (scratchpad or shared) memory. The array α`s` is maintained in global memory, in the same manner as in GPU-OUTER, because it is not guaranteed to fit in the shared memory. Moreover, it is again padded and transposed at block level to optimise coalescing and memory footprint. Unfortunately, the flattening transformation introduces two issues: instructional overhead and shared memory or register pressure.

3.5.1 Flat-Parallel Version in Fast Shared Memory

Listing 3.2 shows the code that is obtained from applying the flattening transformation. In this section, we make an extensive use of the functional notation described in Section 2.4.5, because we need the parallel-basic constructs (**`reduce`**, **`segscan`**), or otherwise it is infeasible to demonstrate the transformation producing GPU-FLAT.

The `bin` array corresponds to a batch of `q` instruments, for which summed widths are less than the thread block size. The result is an array of length `q` of **`real`** numbers denoting the prices of these instruments at the current time. The flat code is obtained by distributing the (outer) **`map`** operation, that is applied over the `q` instruments of the bin, around each **`let`** statement of the original `valuate` function shown in Listing 3.1. In essence, distributing the **`map`** (i) across a scalar statement results in a **`map`** of size `q`, and (ii) across a parallel operation of size `width` results in a parallel operation of size $\Sigma_{k=0}^{q-1}\texttt{width}_k$, that is padded to thread block size.

For brevity and clarity of exposition, our discussion in Listing 3.2 (and the inference rules) ignores the complications related to (i) padding parallel arrays to thread block size and (ii) using the non-trivial offsets in the arrays of all instruments and of α`s` corresponding to the sub-arrays of the current block. Despite being tedious to derive, they are straightforward to add.

```
let valuate^bin (q: i32) (bin: [q]Instrument) : [q]real =
  let (ws,hs) = map f1 bin
  let B_w = scan^exc (+) 0 ws
  let len_flat = B_w[q-1] + ws[q-1]
  let tmp = map2 (λs b → if s == 0 then -1 else b) ws B_w
  let flag = scatter (replicate len_flat 0) tmp (replicate q 1)
  let tmp = scan^inc (+) 0 flag
  let out_inds = map (λx → x-1) tmp
  -- map (λw → iota w) ws
  let tmp = map (λf → 1-f) flag
  let inn_inds = sgmscan^inc (+) 0 flag tmp
  -- map (λw → replicate w 0) ws
  let Qss = replicate len_flat 0.0
  -- map2 (λQs w → Qs[w/2+1] = f2(ins)) Qss ws
  let tmp_i = map2 (λb w → b + w/2 + 1) B_w ws
  let tmp_v = map f2 bin
  let Qss = scatter Qss tmp_i tmp_v
  -- init regular (transposed) h_max×q matrix αss^T
  let h_max = reduce max 0 hs
  let α_is = map f3 bin
  let αss^T = scatter (replicate (h_max*q) 0.0) (iota q) α_is

  -- map-loop interchange; loop count padded to h_max-1
  let (_,_,αss^T) =
    loop(Qss,α_is,αss^T) for i < h_max-1 do
      -- map2 (λis α_i→map (...) is) inn_inds α_is
      let Qss' = map2 (λj oi →
        let (b,h) = (B_w[oi], hs[oi])
        in if i ≥ h-1 then Qss[b+j]
        else
          let q0 = Qss[b+j]
          let q1 = if j > 0   then Qss[b+j-1] else 1.0
          let q2 = if j < w-1 then Qss[b+j+1] else 1.0
          in g1(i, j, α_is[oi], q0, q1, q2)
      ) inn_inds out_inds
      -- map (reduce (+) 0) Qss'
      let scQs = sgmscan^inc (+) 0.0 flag Qss'
      let α_vs = map2 (λb w → scQs[b+w-1]) B_w ws
      -- map(λα → α[i+1] = g2(...)) αss
      let tmp_i = map2 (λh k →
        if i ≥ h-1 then -1 else (i+1)*q + k) hs (iota q)
      let α_is' = map2 g2 α_vs bin
      let αss^T = scatter αss^T tmp_i α_is'
      in (Qss', α_is', αss^T)
  let Css = replicate len_flat 100.0
  ...
```

Listing 3.2: Flat-Parallel Implementation of **HW1F**. We omit the second *backward propagation* `loop` as it has a similar code structure.

Listing 3.2 starts by computing the widths and heights of the trees for all the `q` instruments (line 2). In practice, the widths and heights are precomputed by an inspector, which is separated from the original code. We run in before the main kernel in the preliminary steps, where the instruments are first sorted after their heights to optimise the divergence of the sequential **`loop`**s, and then their widths. We need to use this approach to pack the instruments into bins.

For demonstration, we assume that `q=2`, and the widths and heights are `ws=[2,4]` and `hs=[4,3]`. Lines 3-11 compute three helper arrays (`flag`, $\texttt{out}_{inds}$ and $\texttt{inn}_{inds}$) that are used in the code transformation. The first array `flag` is the flag component in the flat-representation of an irregular array of shape `ws`, such as `Qss`. We recall that the flag arrays are required by **`segscan`** operations. An irregular array of shape `[2,4]` has two rows of lengths `2` and `4`, respectively, and its flag array marks with `1` the start of each subarray and has `0` elements, otherwise. It follows that we expect `flag` to be equal to `[1,0,1,0,0,0]`. We compute `flag` by applying a $\textbf{scan}^{exc}$ on `ws`, resulting in $\texttt{B}_w$`=[0,2]`, then by computing the total number of elements $\texttt{len}_{flat}$ `= 2 + 4 = 6`, and finally, by the **`scatter`** operation at line 6, that writes `1`s at the indices in $\texttt{B}_w$`=[0,2]` into an array of `6` (equal to $\texttt{len}_{flat}$) `0`s, hence `flag=[1,0,1,0,0,0]`, as expected.

The second array $\texttt{out}_{inds}$ records, for each of the width entries associated with an instrument, the index of that instrument in the current bin. Thus, we expect $\texttt{out}_{inds}$ to be equal to `[0,0,1,1,1,1]`. We compute it by applying $\textbf{scan}^{inc}$ to the `flag` array, which results in `[1,1,2,2,2,2]`, and by subtracting `1` from each obtained element (lines 7-8).

The final array $\texttt{inn}_{inds}$ is the expansion of **`iota`** `w` across the `q` widths, hence we expect $\texttt{inn}_{inds}$ to be equal to `[0,1,0,1,2,3]`. We compute $\texttt{inn}_{inds}$ at lines 10-11 by negating the flag array, resulting in `[0,1,0,1,1,1]`, and applying $\textbf{segscan}^{inc}$ on the result under the flag array `flag`, that is, independent **`scan`** on the two logical rows of two and four elements, respectively.

Lines 12-17 in Listing 3.2 are the flattening across `q` instruments of the lines 3-4 in Listing 3.1, which initialises `Qs` elements to zeroes and sets index `w/2+1` to value `f2(opt)`. The zero-initialisation of `Qs` is translated to **`replicate`** of length $\texttt{len}_{flat}$, that is, the summed widths of the `q` instruments, The update at index `w/2+1` is translated to a **`scatter`** on the expanded `Qss`, in which

- the updated indices are computed at line 15 by summing the offset in `Qss` of each instrument, denoted by $\texttt{b}\in\texttt{B}_w$, with `w/2+1`, where $\texttt{w}\in\texttt{ws}$, and
- the updated values are the result of mapping `f2` on the `q` instruments at line 16.

The initialisation of α`ss`, the expansion of α`s`, is obtained by padding each row to the maximal height $\texttt{h}_{max}$ of the `q` instruments, hence total length is $\texttt{q}\times\texttt{h}_{max}$, and by using a **`scatter`** to overwrite the first entry in every row with the result of calling `f3`. This procedure is implemented in lines 19-21, except that αss^T, **`transpose`** of αss, is used to achieve coalesced access to global memory. Next, the forward **`loop`** is padded to count

$\mathtt{h}_{max}$, the outer **map** of length q is interchanged inside the **loop**, and the outer-**map** distribution continues on the **loop**-body statements.

Lines 27-35 correspond to flattening the **map**, that is applied to **iota** w to compute array Qs' in Listing 3.1, lines 10-15. Since the flattened code corresponds to applying the original **map** simultaneously to all entries of all q instruments, it is translated to **map2** over $\mathtt{inn}_{inds}$ and $\mathtt{out}_{inds}$ in such a manner that:

- $\mathtt{inn}_{inds}$ is precisely the expansion of **iota** w across the q instruments, hence j takes the same values as in Listing 3.1.
- $\mathtt{out}_{inds}$ is used to access values that are the same across the width threads assigned to process the current instrument, but are needed by each thread. We recall that the $\mathtt{out}_{inds}$ values record the index of each instrument in each of the width entries associated with it. For example, $\mathtt{out}_{inds}$ is used to (indirectly) index into length-q arrays $\mathtt{B}_w$, hs and $\alpha\lambda$_is to compute the start offset b into array Qss, the height h and the α value corresponding to the current instrument, respectively.
- The body of the mapped function is protected by **if** i≥h-1 condition, that checks that the tree traversal has not already terminated for the current instrument, because the **loop** count is padded to maximal value $\mathtt{h}_{max}$. If so, then the input value of Qss is directly returned.

The code between lines 37-38 is the flattening across all q instruments of the (original) **reduce** at line 16 in Listing 3.1, which sums up the values of array Qs'. This procedure is implemented by first performing $\mathbf{segscan}^{inc}$ on the expanded array Qss', which by definition, computes the q corresponding sums in the last entry of each logical subarray of the result scQs. Then these last entries are extracted by **map** operation of length q. The index of the last entry of the $\mathtt{i}^{th}$ subarray is $\mathtt{B}_w$[i]+ws[i]-1, because $\mathtt{B}_w$ and ws record the offset and the size of each subarray, respectively.

Finally, lines 40-43 implement the expansion of the update to αs[i+1] at line 18 in Listing 3.1. This procedure is translated to a **scatter** that updates αss^T at the q flat-indices belonging to row i+1 (stored in tmp_i) with the values tmp_v obtained by applying g2 to all α_vs and instruments in the batch. Note that if the **loop** index i is greater or equal than the logical **loop** count h-1 then the return index is -1, hence the update is ignored. Similar ideas apply for the translation of the backward **loop**, which is not shown. Our *CUDA* implementation of GPU-FLAT fuses aggressively the inner-parallel operations and reuses shared memory buffers whenever possible: for example, Qss, Qss', Css, Css' use the same buffer. Arrays of size q are typically stored in the shared memory (since they save register space), and arrays $\mathtt{out}_{inds}$ and $\mathtt{inn}_{inds}$ are held in registers. The shortcomings of GPU-FLAT are following: (i) it introduces instructional overhead, that is, the code is more complex than the nested-parallel one, (ii) it introduces significant register pressure and that (iii) the parallel operations of size q underutilise the block-level parallelism, which is typically much larger than $\mathtt{q}$. In particular, *NVIDIA CUDA* compiler *nvcc* reports that $74 - 76$ registers per thread are used by default and a speedup of up to $1.66\times$ is achieved by limiting the number of registers to 32.

3.6 Experimental Evaluation

We run the experiments and compare the performance on two systems, **D1** and **D2**, described in Section 2.5.

We measure total application execution time, excluding host-device memory transfers, but including kernel execution as well as all preprocessing steps, such as data reordering and bin-packing, that involve extra computation. It is our deliberate choice not to overlap computation and memory transfers, to obtain a fair comparison among different runs and technologies. However, for the tested datasets the time involved in memory transfers has an insignificant impact on total execution time (less than 1%). Execution times are averaged across 10 runs (standard deviation $< 1.5\%$) and are reported in $\text{GFlop}^{\text{SPEC}}/\text{s}$, which counts the number of floating-point operations as they appear in the high level specification (GPU-OUTER). We argue that $\text{GFlop}^{\text{SPEC}}/\text{s}$ is better suited for comparing across different implementations of the same algorithm, because it represents normalised runtime. In contrast, we risk that GFlop/s reports the slower version as having better performance.

3.6.1 Datasets

The evaluation uses 7 synthetic datasets that not only model the instrument distributions in real portfolios used in practice, but also, more importantly, clearly demonstrate different workload divergence properties and our performance improvements. All datasets consist of 100 000 valuations, except for **U1**, which uses 3000 and is intended to measure the impact of under-utilising hardware parallelism by GPU-OUTER. Portfolios have diverse sizes in practice, so **U1** is also realistic.

U1 and **U2** use constant width 259 and height 606 across all trees (hence no divergence). These datasets represent a case, where we price a single instrument against many different market scenarios, where one class of the risk factor parameters varies, for example yield curves. In such a case, although the tree dimensions do not change, the values at the tree nodes do.

R1, **R2**, and **R3** (**R***) use random distributions of widths and heights in intervals $[7, 511]$ and $[13, 1200]$, respectively. **R1** uses uniform distribution for both, **R2** uses uniform and standard-normal distributions for widths and heights, respectively, while **R3** does the opposite. These datasets model an instrument distribution typical for an interest-rate derivative portfolio. The results indicate that such datasets exhibit similar performance.

Finally, **S1** and **S2** (**S***) present skewed distributions. In **S1** 1% of the dataset consists of widths and heights in the interval $[461 - 511]$ and $[1082 - 1200]$, respectively, while the rest has widths and heights uniformly distributed in $[7-57]$ and $[12-131]$. **S2** presents the same figures, but separates skewness over dimensions: 1% combines skewed widths with uniform heights and another 1% the reverse. These distributions imitate a portfolio case often met in practice, where a small set of bonds have significantly different characteristics, for instance, much longer maturities or much smaller volatilities than the remaining majority of bonds. Our optimisation techniques address such an example of an extreme workload divergence, that inflict huge performance penalties *GPU* architectures.

We emphasise that it is realistic to consider portfolio composed of instruments with diverse heights and widths of their trees. For instance, the tree height depends among others on the underlying bond maturity, which can be arbitrarily large. The tree width depends, among others, on the interest rate volatility, which is usually obtained through calibration of the liquid vanilla interest rate derivatives traded in the markets. We can expect a significant variation here as well, since there exist many different types of interest rates and vanilla instruments, and on that they are connected with some particular currency that the rate is denominated in.

3.6.2 Result Validation

To validate our **HW1F** results and be confident that they are correct and sufficiently accurate, we compare them with the output from a Hull-White model implementation in *C++* that is a part of an internal *SimCorp* pricing library. This library has been developed and used in production environments by *SimCorp* clients for decades, so we can trust that the modelling approach is accurate for our purposes and the prices that it produces are correct.

We choose a fixed-rate bond (see Section 3.2.1) case with different embedded optionalities, as described in Section 3.2.1. Different input parameters and instrument characteristics are presented in Table 3.1. We choose deliberately to exemplify the impact of various relevant model parameters on the final instrument price. We include both call and put options, as described in Section 3.2.1, as well as all three optionality styles, European, Bermudan, and American, as described in 3.2.1. We want to demonstrate that the option prices are sensitive to changes in model parameters and are different for different types and complexities of instruments.

We present the validation results in Table 3.2. We want to stress that the inconsistencies in price values do not step from parallel execution or a iterative numerical method artefacts. They follow from different implementation of the model, where the internal library has a more sophisticated implementation of calendar dates as well as it supports variable time step, while our implementation keeps the time differences between steps fixed. This feature allows the model to map the time steps to actual exercise date exactly and incorporates an impact of time differences between days in the model (weekends, holidays, etc.). This leads to slight changes in the probability values in the nodes and αs, which aggregate to a different resultant price.

Instrument	9-year fixed rate bond
Model Parameters	
Coupon	Annual 10.0 (9×)
Repayment	100.0 at maturity (1×)
Yield Curve (YC) type	Zero curve based on zero-coupon bonds
YC Rates Discounting method	Continuous compounding
YC Interpolation method	Linear
Calendar Convention	Actual/365
Strike price (K)	63.0
Mean Reversion Rate a	10%
Volatility (σ)	1%
Option-Adjusted Spread (OAS)	3 cases: 0.0, 1.0, 10.0
Optionality	**Exercise dates**
European Option	At the end of the 3^{rd} year
Bermudan Option	For 8 years every 6^{th} month
American Option	At the end of each of 8 years once a month (every date)
Time steps/Early Exercise dates	1 per month

Table 3.1: Set of model and trinomial tree method parameters for the **HW1F** validation. The case comprises various fixed-rate bonds that differ in their embedded optionalities.

	w/o OAS spread		w/ 1.0 OAS spread		w/ 10.0 OAS spread	
Optionality	**Benchmark**	**HW1F**	**Benchmark**	**HW1F**	**Benchmark**	**HW1F**
European Call	78,826609	78,8**70576**	76,770095	76,**812798**	60,562330	60,5**97479**
European Put	116,031936	116,**126855**	108,791987	108,**880131**	63,570053	63,**614187**
Bermudan Call	69,374872	69,37**5598**	68,684580	68,68**5299**	56,257581	56,2**87480**
Bermudan Put	116,031936	116,**126855**	108,791987	108,**880131**	64,150250	64,1**33718**
American Call	69,374872	69,37**5598**	68,684580	68,68**5299**	56,257581	56,2**86167**
American Put	116,031936	116,**126855**	108,791987	108,**880131**	64,150250	64,1**78329**

Table 3.2: **HW1F** pricing accuracy is compared with a production-ready benchmark for different types of options and model parameters. The differences in significant digits is marked in red.

3.6.3 Performance Results

We start a discussion of our performance results by mentioning that we compare performance of our implementations to the external *QuantLib* library [Bal21; Bal14]. We adapt its relevant pricing modules to be thread-safe and implement an *OpenMP* multithreaded version, QUANTLIB-PAR, that makes each library call run in parallel on the outer valuation level. However, despite this obvious optimisation, we still obtain 3-to-4 order of magnitude speedups with both our multi-core *CPU* implementation CPU-MT+VECT as well as both GPU-OUTER and GPU-FLAT kernels, when compared to the QUANTLIB-PAR. Consequently, we do not proceed to compare and discuss these experimental results any further.

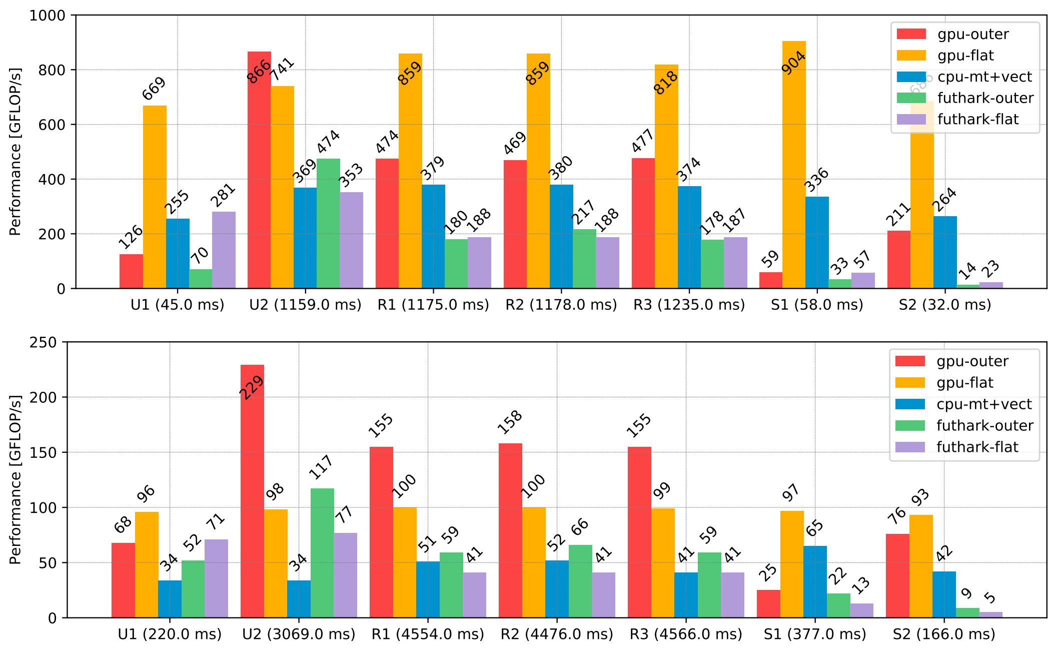

Figure 3.3: Best performance in GFlop$^{\text{SPEC}}$/s for *FP64* on **D1** (**CPU1** and **GPU1**, upper chart) and *FP32* on **D2** (**CPU2** and **GPU2**, lower chart) across all datasets. Devices are described in Section 2.1). Datasets **U1**, **U2**, **R1**, **R2**, **R3**, **S1**, **S2** are described in Section 3.6.1. Best runtime for each dataset is specified in ms in parentheses next to dataset abbreviation.

	R1			**R2**			**R3**			**S1**			**S2**		
GPU-OUTER	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}
V^{ns}_{o1}	103	29	3.42	84	27	3.42	85	26	3.42	56	13	0.52	92	32	0.52
V^{ws}_{o1}	69	24	3.42	61	23	3.42	60	21	3.42	54	12	0.52	77	24	0.52
V^{ns}_{o2}	97	33	6.55	103	35	6.55	114	44	6.55	48	21	6.40	101	41	6.40
V^{ws}_{o2}	423	114	6.55	427	120	6.55	427	136	6.55	57	23	6.40	189	52	6.40
V^{ns}_{o3}	103	45	6.47	109	49	6.28	120	53	5.98	49	23	0.82	111	52	0.82
V^{ws}_{o3}	461	154	3.62	454	157	3.58	461	154	3.60	58	24	0.56	210	75	0.55
V^{ns}_{o4}	108	45	6.38	114	49	6.14	126	53	5.77	49	23	0.82	115	53	0.82
V^{ws}_{o4}	**474**	**155**	3.53	**469**	**158**	3.51	**477**	**155**	3.52	**59**	**25**	0.54	**211**	**76**	0.53
Speedup/Mem.Δ $\times$	4.60	5.34	1.03	5.58	5.85	1.03	5.61	5.96	1.03	1.05	1.92	1.04	2.29	2.38	1.02
GPU-FLAT	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}
V^{ns}_{f1}	521	69	1.98	529	70	1.98	496	66	1.98	266	37	1.83	118	17	1.83
V^{ws}_{f1}	553	73	1.98	554	73	1.98	525	72	1.98	268	33	1.83	119	15	1.83
V^{ns}_{f2}	517	69	1.98	526	70	1.98	492	66	1.98	266	36	1.83	118	16	1.83
V^{ws}_{f2}	550	73	1.98	552	73	1.98	524	71	1.98	268	33	1.83	119	15	1.83
V^{ns}_{f3}	740	87	1.30	779	91	1.21	710	84	1.26	783	97	0.24	577	76	0.24
V^{ws}_{f3}	**859**	**100**	1.09	**859**	**100**	1.10	**818**	**99**	1.09	**904**	**97**	0.17	**686**	**93**	0.17
Speedup/Mem.Δ $\times$	1.65	1.45	0.55	1.62	1.43	0.56	1.65	1.50	0.55	3.40	2.62	0.09	5.81	5.47	0.09

Table 3.3: Summary of GPU-OUTER and GPU-FLAT performance P in GFlop$^{\text{SPEC}}$/s and required global device memory M^{64} for *FP64* in GB. Speedup and memory difference are specified as a ratio between the unoptimised (V^{ns}_{o1}/V^{ns}_{f1}) and the most optimised versions (V^{ws}_{o4}/V^{ws}_{f3}).

***GPU* optimisations**

Performance measures in GFlop$^{\text{SPEC}}$/s are reported in Table 3.3 on machines **D1** and **D2** across 5 datasets, (**R***) and (**S***). GPU-OUTER ($_o$) and GPU-FLAT ($_f$) use *FP64* (^{64}) on **D1** and *FP32* (^{32}) on **D2**. Version V_1 corresponds to a version that does *not* optimise for coalesced global memory accesses, while the remaining versions achieve coalescing by padding and transposing at global (V_2), block (V_3) or warp level (V_4), respectively. Superscripts ^{ns} and ^{ws} correspond respectively to versions without and with data reordering through sorting (in descending order). For GPU-OUTER the trees are sorted by width first, while for GPU-FLAT by height first. Column M^{64} reports the memory footprint in *GB* for *FP64* versions. *FP32* versions use half of the M^{64}. The uniform datasets **U*** are not reported in Table 3.3, because, due to constant tree dimensions, sorting, and block or warp level padding optimisations do not yield a significant effect on the performance.

Main observations for GPU-OUTER are:

- The impact of memory coalescing optimisation varies across **U*** and devices. On **D1** it results in a minor $1.1\times$ speedup on **U1**, but a large $20\times$ speedup on **U2**. On **D2** the optimisations results in $2.7\times$ for **U1** and $14.1\times$ for **U2**. The other two optimisations have negligible impact.

- Our optimisations, data reordering and coalescing by padding, have small or even *negative* impact when applied in isolation. For example, on **D1** and **R1** , the unoptimised version V_{o1}^{ns} is in fact $1.5\times$ faster than V_{o1}^{ws}. However, when combined they have significant impact: between $4.6 - 5.96\times$ on **R*** and between $1.05 - 2.38\times$ on **S*** datasets.

- Warp-level padding V_{o4}^{ws} achieves coalescing at the cost of a modest 3% increase in global memory footprint (M^{64}) w.r.t. V_{o1}^{ns}. Moreover, it is the fastest version on all datasets and on both **D1** and **D2**. In comparison, V_{o2}^{ws} increases M^{64} by $1.9\times$ and $12.3\times$ on **R*** and **S*** datasets.

Important observations for GPU-FLAT are:

- The impact of optimisations on **R*** is smaller than with GPU-OUTER: V_{f3}^{ws} is about $1.6\times$ faster and uses 55% of the memory of the unoptimised version (V_{f1}^{ns}).

- The impact of reordering is positive in most cases, but is relatively small, for example sorting alone produces a speedup as high as $1.2\times$.

- The impact of optimisations is much higher on the skewed datasets **S***, for example, on **S2**, reordering and coalescing at block level (V_{f3}^{ws}) account for speedup factors $5.8\times$ and $5.5\times$ on **D1** and **D2**.

- V_{f3}^{ws} also reduces the memory footprint by $11\times$ in comparison to V_{f1}^{ns} on **S***.

GPU-FLAT VS. GPU-OUTER

Figure 3.3 compares the performance of the best versions of GPU-FLAT (V_{f3}^{ws}) and GPU-OUTER (V_{o4}^{ws}).

- The performance of GPU-FLAT is stable across datasets, between $669 - 904$ on **D1** and $93 - 100$ GFlop$^{\text{SPEC}}$/s on **D2**, while the performance of GPU-OUTER is highly variant.
- On the **R*** datasets, GPU-FLAT is about $1.8\times$ faster than GPU-OUTER on the newer hardware **D1**, but about $1.55\times$ slower on **D2**.
- GPU-FLAT is significantly faster than GPU-OUTER on (i) the skewed datasets **S***, because it Optimises better the divergence and locality of reference by the use of shared memory, and (ii) the small dataset **U1**, because the outer parallelism given by 3000 valuations is insufficient to utilise the hardware well.
- Finally, latest hardware seems to benefit GPU-FLAT, which performs better than GPU-OUTER on **D1** across all datasets, except for **U2**, where it is only $1.17\times$ slower. For instance, on **S1**, GPU-FLAT is $15.3\times$ faster than GPU-OUTER on **D1**, but only $3.9\times$ faster on **D2**.

GPU* vs. *CPU

Figure 3.3 compares the best GPU-OUTER and GPU-FLAT configurations with our multi-core implementation using *OpenMP* multithreading and *AVX2* vectorisation, named CPU-MT+VECT. Even though we use powerful *CPU*s with 104 (**D1**) and 32 (**D2**) hardware threads, the *GPU* versions are faster than the CPU-MT+VECT, with speedups as high as $6.7\times$ on **U2** and $3.8\times$ on **R3**, and on average $2.4\times$ on **D1** and $3.3\times$ on **D2**. On the one hand, we do not observe any case, where CPU-MT+VECT is faster than GPU-FLAT. On the other hand, for particular combinations of datasets and devices CPU-MT+VECT is faster than GPU-OUTER, for example, on **D1** CPU-MT+VECT is $2.0\times$ faster on small **U1** due to insufficient *GPU* parallelism and $5.7\times$ on skewed **S1**, while being only 1.3 faster on **S2** due to thread divergence. These results are a reason why GPU-FLAT is a preferred version on the aforementioned datasets. On machine **D2**, which has a smaller number of cores, the performance gains of CPU-MT+VECT diminish. The only faster case is $2.6\times$ on **S1**. Nevertheless, GPU-FLAT is always faster than CPU-MT+VECT.

As presented in Table 2.1, we note that **CPU1** in **D1** machine is an expensive and powerful system with 104 hardware threads, whereas **CPU2** in **D2** has 32. In comparison to this powerful *CPU*, cost efficiency favours *GPU* hardware.

CUDA* vs. *Futhark

Furthermore, to demonstrate that current compiler technology does not effectively support the code transformations presented in this work, we compare the performance of GPU-

OUTER and GPU-FLAT to the auto-generated code implemented in the data-parallel language *Futhark* [Hen+17; Hen+19]. Figure 3.3 presents the performance results for these two matching implementations executed on **D1** and **D2**. FUTHARK-OUTER uses only the outer level of parallelism and matches the GPU-OUTER code structure from Listing 3.1. FUTHARK-FLAT utilises both levels of parallelism, and is intended to match the GPU-FLAT version, except that, due to compiler limitations, flattening is performed globally across all valuations (rather than at block level), and the intermediate results are recorded in global (rather than shared) memory. The best *CUDA* version is faster than *Futhark* by a factor as high as $29.8\times$ and on average $6.3\times$ on all datasets.

3.7 Related Work

3.7.1 Accelerated Implementations of Lattice Methods

In practice, the trinomial tree numerical method is the standard choice for solving the **HW1F** model. It is especially suited for pricing low-dimensional bond instruments, that we focus on in this work, which depend on 1 or 2 underlying factors. Its clear advantage lays in its simple deterministic execution path, that enables valuation tractability. In comparison, methods based on *Monte Carlo* simulations are more general, but they are also more compute-intensive, and introduce the randomness element, which distorts the understanding of the pricing process. We are unable to find work that applies *GPU* acceleration to the exactly same problem. Thus, we relate our work to the current research in the field that addresses comparable problems, and focus on the main performance inhibitor, namely the divergence introduced by the trees having different dimensions.

Grauer-Gray et al. (2013) [Gra+13] adapt *QuantLib* implementation of Bond and Repo pricing through iterative bootstrapping and adapt it to work with *GPU*s. They implement and compare different *GPU* programming frameworks like HMPP and OpenACC apart from standard *CUDA* and *OpenCL*. Although none of the experiments use a trinomial tree, the authors report an experiment on a diversified bond portfolio, where they parallelise the computation on the outer level, across all the instruments, in the manner that we do in GPU-OUTER. They achieve a speedup of up to $40 - 80\times$ compared to original sequential code. Just like us, they identify the issue of thread divergence, as well as numerous global memory accesses, but they do not propose any solution to address it.

Schabauer et al. (2008) [SHP08] also present an outer-level parallelisation scheme for pricing path-dependent interest rate products on bounded trinomial lattices that resembles GPU-OUTER. They use *Fortran* and *MPI* for distributed computing across as many as 16 nodes with single-core *CPU*s and report speedups of up to $13\times$ depending on the tree height and number of compute nodes. However, their evaluation uses a homogenous set of trees, which does not exhibit workload divergence across priced instruments. Like us, they identify a large number of intermediate communications between lattice nodes on the innermost level (across tree width) to be a performance issue, but abandon parallelisation on this level in their implementation. We remark that our technique for GPU-FLAT addresses this issue and improves the performance. Valuations can be bin-packed across nodes, and inner parallelism can be exploited in shared memory at node level.

Gerbessiotis (2003) [Ger03] describes a distributed implementation for a trinomial tree method for pricing vanilla equity options. They use *MPI* and achieve up to about $16\times$ speedup on 16 dual-core *CPU*s. They address a problem of a *single* trinomial tree computation, that has a large number (32768 or 65536) of time steps. Therefore, the parallelism on the outer level does not exist. As such, they apply a technique that divides the computation of the tree into blocks, which are then computed in parallel on different nodes. The blocks are constructed in such a way to minimise the communication between compute nodes. This approach addresses issues that occur when using an unbounded tree and thus cannot be directly compared to GPU-FLAT.

GPU parallelisation of a simpler binomial lattice method is better represented in the literature.

Gui et al. (2013) [Gui+13] parallelise the binomial tree model using *CUDA* to price a standard vanilla equity option. The problem is restricted to the *specialised* case when the tree dimensions are *the same* across options, thus no divergence occurs. The approach is to exploit inner parallelism by pricing each option on one thread block, which is much less challenging than the procedure for flattening irregular parallelism used for GPU-FLAT.

Suo et al. (2015) [Suo+15] use *GPU*s, through *CUDA* and *OpenCL*, to implement binomial tree method and compare it with a *Monte Carlo* simulation to price *a single* vanilla equity option. They introduce two tree parallelisation schemes. The first uses many thread blocks to price one tree, where each of the thread blocks handles many tree nodes. The second addresses the synchronisation issues of the first approach and minimises communication between thread blocks by taking advantage of a triangular shape of the binomial tree. Such an approach is mainly applicable to unbounded trees, and cannot be easily extended to price in parallel a portfolio, where trees have different dimensions.

Gerbessiotis (2004) [Ger04] is the first to consider acceleration of a binomial method for pricing an option using a distributed setup, but they price many identical instruments with trees that have the same dimensions. The approach resembles GPU-OUTER, but does not consider divergence issues.

Zhang et al. (2012) [ZLM12] present a hybrid implementation that constructs and traverses a binomial tree on *CPU* and *GPU* simultaneously to price a *single* American equity option. Their technique however does not naturally extend to a portfolio of options having irregular tree dimensions. It partitions a binomial tree into blocks of multiple levels of nodes, and assigns each such block to multiple processors. Each block, processed in a sequential order backwards from the leaf nodes, is divided into sub-blocks of equal size (except the last one), which are processed in parallel by distinct processors. In their experiments however, they price a single instrument and only vary the number of time steps in the tree. In addition, they identify the benefit of using of on-chip shared memory to reduce the number of accesses to the off-chip device memory.

Zubair and Mukkamala (2008) [ZM08] show a binomial tree implementation on a single- and multi-core processor for pricing a single vanilla equity option. They propose two algorithms that take advantage of hierarchical model of memory to maximise locality of data access. They use cache and register blocking techniques for improving the performance. These approaches can be mapped to *GPU*s.

Huang and Thulasiram (2005) [HT05] develop parallel algorithms for pricing path-dependent American exotic options with up to 10 underlying assets using binomial tree method. They use *MPI* model to program multi-core *CPU*s and consider the performance changes due to different number of time steps for a variable number of assets. However, the options they consider have trees of the same dimensions, and thus they do not address the problem of workload divergence.

Other work studies different numerical methods to solve the *Hull-White* pricing model.

Theiakos et al. (2015) [The+15] target *GPU*s, but price a mortgage contract with one underlying by using a *Finite Difference Method (*FDM*)* to solve the problem. Dang et al. (2012, 2013, 2014) [DCJ12; DCJ13; DCJ14] use *GPU*s to price cross-currency interest rate derivatives having multiple risk factors and path-dependent features. They also use FDM to solve the high-dimensional system of partial differential equations. Both approaches consider that the all instruments share the same grid dimensions (no divergence).

Bernemann et al. (2010) [BSS10] use *GPU*s, but use *Monte Carlo* simulations and more sophisticated *Heston Hull-White* model with local volatility to price structured equity instruments. We mention that this extended version of the model can only be solved using a *Monte Carlo* simulation. Finally, a large body of work is dedicated to *GPU* acceleration of *Monte Carlo* simulations used for *pricing derivatives* [Alb07; Lee+10; Oan+12; NL11], model *calibration* [And+16] or *risk management* [DCK09].

Finally, Albenese (2007) [Alb07] price a number of different callable instruments by means of a large lattice implemented on a *GPU*. In fact, they argue that short rate models and are better suited for *GPU* architectures then market models, that are considered to be more advanced and precise. However, the latter can only be solved using expensive *Monte Carlo* simulations. They consider portfolios of consecutively more sophisticated instruments, but the complex mathematical approach that they use to formulate the pricing problem using semi-analytical formulas allows them to reduce it to a large matrix-matrix multiplications, which are implemented efficiently using standard *BLAS-3* function calls. They also assume that all the required discounted transition probability kernels and data structures that describe cash-flows fit the available device memory. They organise the pricing functions for all the instruments in the portfolio in a large matrix, where one column vector is attributed to one instrument, and perform the backward induction collectively on the entire portfolio. This work is an example of using formulas available for some short-rate models and redefining a problem algorithmically so it suits massively parallel hardware, and such cannot be directly compared to our approach, where parallelism is extracted from a classical approach to parallelism.

In conclusion, we are unaware of any work aimed at accelerating on *GPU*s the **HW1F** model solved by bounded trinomial tree for pricing a portfolio of callable bonds. However, the techniques applied to generate GPU-FLAT and to optimise GPU-OUTER seem feasible to be applied to other compute-intensive pricing methods.

3.7.2 Compiler Techniques

Our implementation draws inspiration from a number of compiler techniques. The GPU-FLAT version builds on the flattening transformation [Ble+94; BR12], which maps irregular nested parallelism into a sequence of flat-parallel ones. There are two key differences here. The first one is that flattening pushes all sequential recurrences outside the parallel code, and it introduces many prefix-sum operations that are executed in global memory and thus limit performance gains. Instead, we bin-pack inner parallelism at block level so that it can be efficiently executed in shared memory. The second difference is more subtle and refers to the fact that the traditional flattening transformation replicates variables that are unbounded in the inner parallel constructs. This procedure leads to a possible memory explosion, which prevents the use of shared memory. In comparison, our procedure does not expand such variables, but instead indirectly accesses them by means of auxiliary arrays, such as $\texttt{out}_{inds}$, $\texttt{inn}_{inds}$, $\texttt{B}_w$ in Listing 3.2. The latter can be seen as part of the shape representation of an irregular array of arrays, reused between similarly-shaped arrays, such as `Qss` and `Css`.

Finally, our techniques for optimising the two-level divergence by sorting after the tree dimensions are inspired by data reordering transformations aimed at improving locality and communication patterns [SCF03], and by inspector-executor restructuring transformations [SHO18; RAP95; OR11]. However, we are not aware of any compiler framework that is able to generate the GPU-FLAT code version from the nested-parallel specification of Figure 3.1. The same is true for the GPU-OUTER optimisations. The work presented in this chapter provides useful insights into how to integrate such techniques into the repertoire of a compiler.

3.8 Conclusion

This chapter investigates the problem of providing an efficient *GPU* acceleration to a model, that is commonly used in practice, which uses a bounded trinomial-tree numerical method to price a class of derivative instruments based on one underlying interest rate. The instruments are characterised by non-trivial cash flows and Bermudan or American optionality.

The main challenge resides in the fact that, in practice, the portfolio instruments are characterised by trees of very different dimensions. This leads to high performance penalties due to a thread divergence overhead and suboptimal memory access patterns.

In this regard, we identify several relevant optimisation techniques, but we omit a demonstration the rewrite rules that are necessary for compiler integration. We start from a high-level specification that makes explicit all the available (nested) parallelism in terms of data-parallel basic constructs (**`map`** and **`reduce`**), and show how to systematically derive and optimise two different low-level *CUDA* implementations. GPU-OUTER prices each instrument on a different thread, while GPU-FLAT exploits the parallelism available inside the valuation of each instrument by bin-packing instruments to block level, and by efficiently flattening this parallelism in the shared memory.

In addition, we present several transformations aimed (i) at improving spatial locality and memory footprint, by padding intermediate arrays at block or warp levels and by operating on a transposed-array layout, and (ii) at optimising the thread divergence, by reordering the instruments by their widths and/or heights.

Finally, we present a detailed experimental evaluation that demonstrates that (i) the proposed optimisations have high impact when they are applied together, and much-reduced impact in isolation, (ii) GPU-FLAT is more efficient than GPU-OUTER in most cases, especially on recent *GPU* hardware, (iii) current compiler infrastructure does not support the proposed code transformations, for example, our *GPU* implementations outperform the *Futhark*-generated code, and (iv) our *GPU* implementation is faster than our parallel (*OpenMP* multithreading + *AVX2* vectorisation) *CPU* implementation, and it outperforms by $3-4$ orders of magnitude the multi-core implementation using *QuantLib* library, which is commonly used in practice.

In terms of point (i), we stress that thread divergence, is caused by lock-step execution and fixed by data reordering in our implementation. The uncoalesced accesses are fixed by a combination of padding snd transposition. These two are independent sources of overhead, which “hide” each other. In other words, threads, that are waiting due to divergence, result in fewer in-flight memory transactions, whose latency is better hidden by multithreading. The observed and total overhead is closer to the maximal of the two overheads, rather than the sum. Thus, both overheads need to be optimised before observing significant performance gains. The same is true for optimisations such as constant folding, copy/constant propagation, and common subexpression elimination, which are only effective when applied together, and not in isolation.

Chapter 4

Least Squares Monte Carlo Simulation (LSMC)

Abstract

We study the feasibility and performance efficiency of expressing a complex financial numerical algorithm with high-level functional parallel constructs. The algorithm that we investigate is a Least Squares regression-based *Monte Carlo* simulation for pricing American options.

We propose an accelerated parallel implementation in *Futhark*, a high-level functional data-parallel language, that targets *GPU*s as a computational platform. We achieve a performance comparable to, and in particular cases up to $2.5\times$ faster than, an implementation optimised by *NVIDIA CUDA* engineers.

In absolute terms, we can price a put option with 1 million simulation paths and 100 time steps in 17 ms on a *NVIDIA* Tesla V100 *GPU*. Furthermore, the high-level functional specification is much more accessible to the financial-domain experts than the low-level *CUDA* code, and thus promotes code maintainability and facilitating algorithmic changes.

Section 4.1 introduces the specific derivative pricing problem that is present in the equity markets, motivates the importance of accelerated implementation, outlines our approach, and summarises the main contributions.

Section 4.2 provides rudimentary background to American option pricing.

Section 4.3 examines the problem solved by our application and describes the high-level algorithm for American option pricing.

Section 4.4 shows how American option pricing algorithm can be naively implemented in *Futhark* language.

Section 4.5 provides a detailed description of the linear algebra transformations required to extract partial parallelism from the traditionally sequential optimisation part and significantly reduce the sizes of matrices and computational burden.

Section 4.6 discusses the three parts of the implementation. It demonstrates how we can turn the inefficient version of the algorithm into a more efficient version targeting *GPU*s. We also describe the design and implementation choices that were necessary for obtaining an efficient version of the algorithm.

Section 4.7 presents the methodology, the experimental test cases and discusses the results.

Section 4.8 relates the work to other projects on acceleration for similar pricing problems and compares the used optimisation techniques with the current research.

Section 4.9 summarises the main findings of our work.

4.1 Introduction

Pricing American options is a fundamental business case in the financial services sector, because such financial instruments are widely traded in the derivative markets. American options can be exercised at any time between the present date and the time to maturity. This aspect puts them in contrast to European options, that can only be exercised at their maturity. In the usual case, the option holder is expected to exercise the option as soon as it is more profitable to do so rather than wait until its expiration. Effectively, the value of an American option is the value achieved by exercising it at the optimal time. This embedded optimisation (optimal stopping) problem is the main challenge. Assuming no general closed-form formula solutions [Hau07] exist, we attempt to approximate the option value accurately with a numerical simulation. which is a memory-exhaustive and time-consuming computational effort. It has to be significantly reduced to become acceptable for time-critical applications in financial practice. Therefore it is a compelling case for accelerating the computation on highly parallel hardware, such as *GPU*s.

Currently, the most efficient accelerated simulations are implemented in dedicated languages and frameworks. Most examples are implemented in *CUDA* [Abb+14; FP13; PW12], but there are also implementations in other technologies like *OpenCL* [Var+15b; Bru+15], *MPI* [Cho+08], *OpenMP* [Zha+17], etc. [CHL15]. The challenge with these implementations is the poor expressibility, which makes them inaccessible to domain experts. A specialist developer has to be appointed to implement and maintain such kind of code. An interaction between domain experts and developers is needed every time a change in the algorithm is required. It also results in code that is difficult to maintain. On top of that, the developer needs to be aware of the low-level properties of the underlying hardware architecture, and have good knowledge of code transformation aimed at optimising potentially-conflicting factors such as locality of reference and degree of parallelism.

We propose a functional approach to the implementation of an accelerated option pricing model. The use of high-level parallel constructs lets us express the algorithm in an intuitive manner, without the implementation concerns of mapping the code to the architecture. The *Futhark* language and the optimising compiler behind this language are used to simplify the implementation process [Hen+17].

The main contributions of this work are the following:

1. We present a high-level data-parallel implementation of the Longstaff-Schwartz algorithm for pricing American options using a *Monte Carlo* Simulation with Least Squares Regression (abbreviated **LSMC**) [LS01]. The implementation serves as a reference implementation and can easily be ported to other functional languages. Moreover, the algorithm makes explicit the available parallelism by using high-level data-parallel constructs.

2. We give a detailed description of the algorithmic changes required to achieve an efficient parallel implementation of the **LSMC** algorithm.

3. We present an optimised efficient version of the algorithm and describe how the original algorithm is reimplemented in *Futhark* to achieve performance results that in most cases matches a *CUDA* version, which is a hand-tuned implementation of the algorithm, implemented in a dedicated low-level programming model by *CUDA* engineers. In addition, we present specific cases, in which *Futhark* version achieves up to $2.5\times$ speedup over the *CUDA* version.

4.2 Monte Carlo Simulation and American Option Pricing

An option contract is defined by its payoff function. For the vanilla-type options like *calls* and *puts*, it compares the strike price K and the current asset spot price S and thereby determines the cash flow of an option. The strike price K is an agreed fixed price, at which the option holder can buy (in case of a call) or sell (in case of a put) the underlying asset. The spot price S is a price of the underlying asset like stock or commodity, that the option derives its value from. S varies over the time. This progression in time is random and thus can be described using a stochastic process. Such a process is defined by a *SDE*, which cannot be solved directly using a closed-form formula. Instead, we use a numerical simulation method.

In practice, the *Monte Carlo* simulation is the most widely used and robust method for solving general *SDE* problems, and option pricing problems in particular. It is popular in the quantitative finance community, because (1) it is relatively simple to parallelise and (2) it allows for complex instrument pricing that depend on many underlying assets. Such instruments cannot be priced with deterministic methods such as *FDM* or lattice models suitable for low-dimensional instruments. In fact, a *Monte Carlo* simulation is the only numerical method that can be used for multi-factor pricing in dimensions greater than four, because pricing many underlying assets lead to a system of equations that is too complex to express with a regular grid, which is a discretisation of *PDE*s in many dimensions [Gla04].

The *Monte Carlo* Simulation method is based on two theorems of probability theory. The first one is the *Strong Law of Large Numbers*, which guarantees the convergence of a certain series of independent random numbers having the same distribution to a value of an integral. The second is the *Central Limit Theorem*, which determines the convergence rate of the first law [Bil12; Shi16].

In a standard *Monte Carlo* simulation, the paths of the state variables are simulated forward in time, which is, in particular, the case for pricing European options. Given an option payoff, a forward (future) price is determined for each path at the maturity date. To estimate the present price of the instrument, the future cash flows, that is, prices in different points in future time, have to be discounted to the present time using some established discount (interest) rate. We perform it to adjust the price with a time value of money that states that the money sum available now is worth more than the identical sum in the future. Finally, an unbiased estimate of the current option price is a mean average of these prices.

In contrast, the American option pricing progresses backward in time. First, the optimal exercise price is determined at the maturity and later is recursively propagated and discounted backward in time using dynamic programming until the current time. The current price is estimated this way. Although an American option can be exercised at any time, the exercise times are discretised and restricted to a fixed set of times, for example, daily or monthly. The overall goal here is to provide an approximation of the optimal stopping rule that maximises the value of the American option.

At the maturity $t = T$, the option holder exercises the in-the-money (*ITM*) option, that is, if the value of the exercised option generates a positive cash flow. For instance, in case of a call option spot price higher than the strike price is preferable, while for put option the inverse is favourable. For any other time t_i, the holder needs to choose whether to exercise the option or continue holding it. The option value is at the maximum if the exercise happens as soon as the immediate exercise cash flow is greater than or equal to the continuation value, that is, the discounted expected option value at the next instance in time. However, this continuation value at any given time t_i is not known, so it needs to be estimated.

In practice, we want to price large portfolios of options through simulation within seconds for real-time decision making. We assume that *GPU*s, which allow for a high degree of parallelism due to its massive number of cores, are a good fit for such large computational workloads.

4.3 The Longstaff-Schwartz Algorithm

Several authors have proposed the use of regression to estimate continuation values from simulated paths and thereby enable American option pricing through simulation. It is, especially, the initial studies of [Car96], the most renown [LS01] as well as [TV01], who performed similar studies at the same time. There are two reason that motivate the choice of Longstaff-Schwartz approach [LS01] for our implementation. It is the algorithm with the most widespread adoption in the financial industry and it is based on a *Monte Carlo* simulation that is easy to parallelise. In addition, we mention that the authors of [CLP02] prove the convergence of the Longstaff-Schwartz algorithm and analyse the dependence of its convergence rate on the number of simulated paths.

4.3.1 Least Squares Regression

To start with, we turn our focus to the core challenge of the algorithm, that is the estimation of the continuation values C_i at each time step i. Let us assume that S_{ij} is an asset spot price at time step i on a path j. Each continuation value $C_i(S_{ij})$ is the regression of the option value in the next time step on the value of S_{ij} in the current time step and path. The procedure is to approximate C_i by a linear combination of basis functions of the current state and use regression to estimate the best coefficients for this approximation. The approximation accuracy depends on the choice of functions used in the regression.

Following the Longstaff-Schwartz approach, we apply an ordinary Least Squares regression across the simulated paths at any given time t_i to estimate the continuation value C_i. The Least Squares regression is a method for finding approximate solutions to overdetermined systems of linear equations, that is, there are more equations to solve than variables to choose. The method minimises the sum of the squares of the errors in the equations [BV18]. However, since the decision to exercise the option is relevant only when the option is in-the-money, we regress only paths that are in-the-money. This choice results in an improved algorithmic efficiency without negative impact on accuracy. Both the convergence of the algorithm and how the algorithm converges with the number of simulated paths were analysed in [CLP02].

To begin with, the *ITM* paths are parameterised using a quadratic polynomial $\beta_0 + \beta_1 S_{ij} + \beta_2 S_{ij}^2$, where S_{ij} is a variable, an asset spot price at a time step i, here for some *ITM* path j. In particular, each term of a polynomial, a monomial with basis $1, x, x^2, \ldots$, is a basis function. We chose to use up to second order polynomials in the basis. Having said that, we mention that this choice is usually left as a parameter to the algorithm. In contrast, we deliberately settle on a particular type of basis function. This makes it possible to implement specialised versions of matrix transformations, and as a result enable performance optimisations. The number of used polynomials matches the number of different time steps i, because one regression is performed for each step. A point cloud of asset spot prices $S_i j$, distributed for each path across the time steps, is obtained from a simulation. The regression problem is then to find, separately for each of these time steps, the best fit in terms of β coefficients and basis functions for a quadratic polynomial. We discuss this procedure further in Section 4.6.

Essentially, we deal with an overdetermined system of equations, because the number of equations is larger than the number of unknown coefficients. In our experiments we use 3β coefficients. For the sake of presentation, let us assume we solve a system of equations $Ax = b$. We minimise the objective function $\| Ax - b \|^2$ by finding a variable vector $\hat{x}$ from all possible choices of x. In other words, it is a least squares approximate solution to $\| A\hat{x} - b \|^2 \leqslant \| Ax - b \|^2$. In our case, we assume that the number of *ITM* paths itm is significantly larger than the number of the basis functions (3). It follows that A is a tall data matrix of size $itm \times 3$ and b is a data column vector of size itm. The variable of this system of equations is the column vector x of size 3. In particular, A is a matrix of powers of asset spot prices S_i built from the chosen polynomial. b is a vector of cash flows $\hat{V}_i$ dependent on the payoff function $p(S_i)$, specific for an option that is priced. Finally, x is a vector of polynomial coefficients β_k, that we want to fit with the least squares method. We arrive at the following system of equations:

$$\begin{bmatrix} 1 & S_0 & S_0^2 \\ & \dots & \\ 1 & S_j & S_j^2 \\ & \dots & \\ 1 & S_{itm-1} & S_{itm-1}^2 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} p(S_0) \\ \dots \\ p(S_j) \\ \dots \\ p(S_{itm-1}) \end{bmatrix} \tag{4.1}$$

The textbook solution for solving this Least Squares problem is to multiply both sides by A^T, resulting in $A^T A\hat{x} = A^T b$. Since $A^T A$ is a square matrix, we can now multiply both sides with its inverse what leads us to the unique solution: $\hat{x} = (A^T A)^{-1} A^T b$. This follows from a fundamental assumption of the least squares method that the columns of matrix A are linearly independent, therefore $A^T A$ is always invertible. The matrix $(A^T A)^{-1} A^T$ is the pseudo-inverse of the matrix A, denoted $A^\dagger$. The approach to constructing $A^\dagger$ is the main algorithmic challenge and the source for optimisations. There exist different methods to build $A^\dagger$. For the first naive implementation, we use the formula directly, applying matrix multiplication, transpositions, and inversion on A. In Section 4.5.1, we present an efficient algorithm for a pseudo-inverse construction, adapted for massive parallelism offered by *GPU*s.

4.3.2 Detailed Algorithmic Structure

The generic structure of a simulation algorithm that uses a linear least squares regression according to Longstaff-Schwartz algorithm [Gla04] can be summarised as follows.

We assume as input we are given the number of time steps m, the number of paths n, and the option-specific payoff function $p_i(S_{ij})$, where S_{ij} is an asset spot price at time step i and path j. Payoff p_i is discounted back from the maturity T (time step $m-1$) to current time (time step i).

A generic linear combination of basis functions is denoted by a polynomial function $\psi_i : \mathbb{R}^r \to \mathbb{R}$ and constant coefficients β_{ik}, where r is the highest degree of the chosen polynomial with each term being a basis function $k = 0, \dots, r-1$. Moreover, $\beta_i = [\beta_{i0}, \dots, \beta_{im-1}]$ and $\psi(S_i)^T = [\psi_0(S_i), \dots, \psi_{m-1}(S_i)]^T$.

1. Generate a matrix $W(n, m)$ of random numbers drawn from a standard normal distribution.

2. Using W, simulate (by *forward induction*) n independent paths $S_{0j}, \dots, S_{m-1j}$, $j = 0, \dots, n-1$ of Geometric Brownian Motion stochastic processes for the underlying asset prices.

3. At the last step $m-1$ (at maturity T), compute the option value $\hat{V}_{mj} = p_m(S_{mj})$, $j = 0, \dots, n-1$ applying the payoff function p at the last step $m-1$.

4. Apply *backward induction* for each step $i = m-2, \dots, 1$ to compute cash flows:

   a) Select the *ITM* paths.

b) Build the matrix ψ_i from asset prices S_i and the right hand side cash flows vector $\hat{V}_{i+1}$ only for the *ITM* paths for the Least Squares linear equation $\psi_i(S_i)\beta_i = \hat{V}_{i+1}(S_{i+1})$.

c) Use regression to calculate $\hat{\beta}_i$ by solving a pseudo-inverse

$$\psi(S_i)^\dagger = (\psi(S_i)^T\psi(S_i))^{-1}\psi(S_i)^T$$

in

$$\hat{\beta}_i = \psi(S_i)^\dagger\hat{V}_{i+1}(S_{i+1}).$$

d) Approximate the continuation function $\hat{C}_i(S_i) = \hat{\beta}_i\psi(S_i)^T$.

e) Decide to early-exercise based on the value of the continuation function $\hat{C}_i$ for each *ITM* path j:

$$\hat{V}_{ij} = \begin{cases} p_i(S_{ij}), & p_i(S_{ij}) \geqslant \hat{C}_i(S_{ij}); \\ \hat{V}_{i+1,j}, & p_i(S_{ij}) < \hat{C}_i(S_{ij}). \end{cases} \tag{4.2}$$

5. Return $\hat{V}_0 = (\hat{V}_{10} + \cdots + \hat{V}_{1n-1})/n$ discounted to time step $i = 0$.

4.4 Naive Implementation

We first present a *Futhark* implementation of the naive **LSMC** algorithm, as specified in the original paper. The *Futhark* function `lsmc_naive`, which implements the main part of the algorithm is listed in Figure 4.1. The function takes as arguments (1) a two-dimensional array containing generated paths, (2) the maturity time (`T`) as a year fraction, (3) a risk-free interest rate `r` used for discounting, and (4) a payoff function `pFun`.

We want to emphasise that, in our *Futhark* implementation of **LSMC**, we follow the same algorithmic choices as taken by *NVIDIA* in their *CUDA* implementation [Dem14]. However, we can not find any published material on the exact linear algebra transformations that are carried out. decide to specify the process in detail in the following sections. Therefore, we present the algorithmic consideration and an efficient approach to an implementation of a financial algorithm for a widespread case of a *Monte Carlo* simulation for American Option Pricing, which is frequently reimplemented across financial institutions. Our work is, to our knowledge, the first state-of-the-art high-level implementation approach to this particular problem available to the public.

4.5 Mathematical Considerations

The main inefficiency of the naive (straightforward) implementation of the **LSMC** algorithm is the necessary computation of a full $A^\dagger = (A^TA)^{-1}A^T$ matrix at each time step. We optimise this by means of a linear algebra reformulation and an algorithmic refinement, which aims to separate the part of the computation of $A^\dagger$, that is intrinsically sequential and

```
let lsmc_naive [paths] [steps] (Ss: [paths][steps]real)
    (T: real) (r: real) (pFun: real → real)
  : real =
  let Sst = transpose Ss
  let dt = T / (real (steps-1))
  -- compute discount factors
  let disc =
    map (λi → exp(r*real(-(i+1))*dt)) (iota (steps-1))
  -- prepare initial payoffs
  let Ps = map (λji →
    let j = ji / steps
    let i = ji % steps
    in if i < steps-1 then zero else pFun(Ss[j,i])
  ) (iota (paths*steps))
  -- iteratively update the payoffs going backwards
  let (Ps, _) = loop (Ps, h) = (Ps, steps - 1) while h >= 1 do
    -- compute paths that are in-the-money
    let selectedpaths =
      filter (λj → pFun(Sst[h,j]) > zero) (iota paths)
    -- prepare for and perform regression
    let Y = map (λj →
      map (λi → disc[i]*Ps[j*steps + h+i]) (iota (steps-h))
      |> reduce (+) zero) selectedpaths
    let Xt = map (λi →
      map (λj → Sst[h,j] ** (real i)) selectedpaths) (iota 3)
    let X = transpose Xt
    let β = Mat.matvecmul_row
      (Mat.matmul Xt X |> Mat.inv) (Mat.matvecmul_row Xt Y)
    let exVals = map (λj → pFun(Sst[h,j])) selectedpaths
    let contVals = map (λj →
      let sst = Sst[h,j]
      in loop (racc,sacc) = (0.0,1.0) for k < 3 do
          (racc + sacc * regY[k], sacc * sst)
      ) selectedpaths |> (.1)
    let (updInds, updVals) = map (λji →
        let j = ji / steps
        let i = ji % steps
        let j' = selectedpaths[j]
        in if (contVals[j] < exVals[j])
        then if (i != h)
          then (j' * steps + i, zero)
          else (j' * steps + i, exVals[j])
        else (-1, zero)
      ) (iota ((length selectedpaths)*steps)) |> unzip
    let Ps = scatter Ps updInds updVals
    in (Ps, h - 1)
  -- compute the discounted mean
  let prices = map (λj →
    map (λi → disc[i]* Ps[j*steps + i+1]) (iota (steps-1))
    |> reduce (+) zero
  ) (iota paths)
  in Stats.mean prices
```

Listing 4.1: *Futhark* code for the naive **LSMC** algorithm.

has dependencies within the time-step loop from the one that does not have these issues. The latter, "independent" part can thus be precomputed in parallel before the time-step loop is entered. The algorithmic change consists, at a very high level, of working with a QR decomposition of $A = QR$, where the R matrices have small dimensionality (3×3 in our case) and can be efficiently precomputed in parallel for all time steps. With this, the computation inside the time-step loop is reduced to $A^{\dagger} = R^{-1}Q^{T}$.

Furthermore, the Q^T matrix is not manifested in memory at any moment, but rather computed on the fly from the sample matrix and fused in the multiplication with R^{-1}. This requires some redundant computation, but in a general case it significantly decreases the number of accesses to global memory, which are orders of magnitude slower than scalar arithmetic. Finally, the sample matrix is computed in transposed layout to optimise spatial locality, that is, coalesced accesses to global memory on *GPU*.

4.5.1 Building a Pseudo-Inverse Efficiently

We take our design goals into consideration and change the algorithm to adhere to parallel computation on *GPU*s. This approach was first adapted in the original *CUDA* implementation by *NVIDIA* that we are trying to match in performance [Dem14; NVI].

For the sake of brevity, we assume that we work with one system of equations $Ax = b$, although one per each time step i needs to be solved. We follow the standard practice by applying Singular Value Decomposition (*SVD*) $A = U\Sigma V^T$ to reduce the dimensions of A and build the Moore-Penrose pseudo-inverse of a tall matrix $A^{\dagger} = V\Sigma^{-1}U^T$. Next, $A^{\dagger}$ is used to compute the solution $\hat{x} = V\Sigma^{-1}U^T b$.

Thus, our goal is to build $A^{\dagger}$ efficiently, because we need to construct one matrix A for each time step. Naive *SVD* computation of A requires extensive execution time and memory. To remedy this problem, we start with a QR decomposition of $A = QR$, and use the fact that R is much smaller than A. We specialise the algorithm to work with a 3-degree parametrisation using a quadratic polynomial. As a result, in our case R is of size 3×3. We compute *SVD* of $R = U_R\Sigma_R V_R^T$ to build the *SVD* of $A = QU_R\Sigma_R V_R^T$.

The QR factorisation is typically performed using *Householder transformation*. Its naive application involves $3 \times n \times n$ memory access, which is the number of matrix elements that need to be updated. In contrast, the efficient solution comes from the observation that in our case matrix R can be constructed using only 8 scalars:

$$S_0, S_1, S_2, \sum_{i=0}^{itm-1} S_i^0, \sum_{i=0}^{itm-1} S_i^1, \sum_{i=0}^{itm-1} S_i^2, \sum_{i=0}^{itm-1} S_i^3, \sum_{i=0}^{itm-1} S_i^4,$$

where S_i is the asset spot price on the *ITM* path i. Three first scalars are asset spot prices for the first *ITM* paths found when traversing the paths from path 0. The first sum can be translated to a total number of *ITM* paths found. The remaining four sums are consecutive powers $(1, \ldots, 4)$ of spot prices associated with each found *ITM* path. These scalars are prepared as part of *SVD* preparation. We implement a custom function that uses these scalars to build the matrix R.

Furthermore, QR factorisation provides us with a simple formula for the pseudo-inverse. We use the fact that A is left-invertible, so its columns are linearly independent. We have

$$A^T A = (QR)^T (QR) = R^T Q^T Q R = R^T R,$$

so

$$A^\dagger = (A^T A)^{-1} A^T = (R^T R)^{-1} (QR)^T = R^{-1} R^{-T} R^T Q^T = R^{-1} Q^T.$$

The final equation that we solve in the main loop for each time step is then as follows: $\hat{x} = R^{-1} Q^T b$. The R^{-1} is precomputed for each time step before the loop using *SVD* like this: $R^{-1} = V_R \Sigma_R^{-1} U_R^T$. The orthogonal matrix $Q^T = R^{-T} A^T$ does not need to be stored for each time step and can instead be computed on-the-fly. We again use the fact that (1) R^{-1} is by now already precomputed using *SVD* and (2) matrix A can itself be computed on-the-fly, where, for our case, each row comprises 3 elements: $1, S_{ij}, S_{ij}^2$, that is, it can be computed from vector S_i that consists of asset spot prices for *ITM* paths for a given time step i. Naturally, they need to be processed in transformed form.

4.6 Optimised Algorithm and Implementation

We do not claim any contributions to this algorithm and instead closely follow the implementation proposed by *NVIDIA* [Dem14; NVI]. We focus on the goal to match the performance of this public benchmark implementation. The obtained algorithm outlined in Figure 4.2 is implemented using a nested composition of sequential **loop**s and parallel **map**, **reduce**, and **scan** constructs.

```
-- Path Generation
map(n)
  loop(m)
transpose
-- SVD Preparation
map(m)
  loop(n)
    scan(chunk)
  map(n) |> reduce(n)
  map(n)
map(m)
-- Main Regression Loop
loop(m)
  map(n)
  map(n)
reduce(n)
```

Listing 4.2: The high-level view of the implemented optimised algorithm structure presented as a combination of parallel constructs. It consists of 3 parts with n denoting the number of paths and m denoting the number of time steps. **transpose** performs matrix transposition.

The code in Figure 4.3 demonstrates the main `lsmc_opt` function of the optimised algorithm. The function takes the following arguments: (1) number of time steps `m`, (2) number of paths `n`, (3) a function to verify if the option is *ITM* `is_itm`, (4) a payoff function `payoff`, (5) the time step size `dt` as a fraction of year, (6) initial asset spot price at the current day `S0`, (7) a risk-free interest rate `r` used for discounting, (8) volatility σ, (9) `seed` for Random Number Generator and 2 helper parameters that determine the amount of computation that is performed sequentially. The function returns the calculated option price.

In the next sections, we give a detailed description of the implementation and optimisation involved in the algorithm. The three main parts of the algorithm are: path generation in Section 4.6.1, *SVD* preparation in Section 4.6.2, and main regression loop in Section 4.6.3. Each section is accompanied with a code listing presenting a *Futhark* implementation of a function that implements the given part.

```
let lsmc_opt (m: i32) (n: i32)
    (is_itm: real → i32)
    (payoff: real → real) (dt: real)
    (S0: real) (r: real) (σ: real) (seed: i32)
    (min_itm: i32) (CHUNK: i32)
  : real =
  -- Path Generation
  let paths = generate_samples_and_paths
      prng_seed m n S0 dt r σ payoff
  -- SVD Preparation
  let Sst = transpose paths
  let (svds, all_otms) = prepare_svds Sst is_itm
      min_itm CHUNK
  -- Main Regression Loop
  let expmrdt = exp(-r*dt)
  let (cashflows, _) =
    loop (cashflows, i) = (Sst[m - 1], m-2)
    while i >= 0 do
      let βs = compute_βs is_itm svds[i] Sst[i]
          cashflows all_otms[i]
      let new_cashflows = update_cashflows payoff
          expmrdt βs Sst[i] all_otms[i] cashflows
    in (new_cashflows, i - 1)
  in expmrdt * (reduce (+) zero cashflows) / n
```

Listing 4.3: Main function of the optimised *Futhark* implementation of the **LSMC**.

4.6.1 Path Generation

The computational effort of a *Monte Carlo* simulation is determined by the number of paths and time steps. A large number of paths n, that is usually 100 000 to 1 000 000, needs to be generated to obtain an accurate value approximation [Gla04]. In American option pricing case, the number of time steps m is bound to the number of early-exercise opportunities and is usually much smaller than n. Path generation part consists of two sub-parts: random number generation and path generation.

For the random number generation purposes in the first part, we use Newer "Minimum standard" *PRNG* in a parallel skip-ahead manner. The *Futhark* code is ported from *C* implementation of `minstd_rand`. After seeding the *RNG*, we draw $m \times n$ random samples by splitting the *RNG* into n sub-*RNG*s for each path and than sequentially drawing a sample for each time step of the given path. The samples are independent from each other. For the process, that we want to simulate, we need samples drawn from Gaussian (standard normal) probability distribution. We achieve it in two steps. First, we draw samples from uniform probability distribution using the *RNG*. Afterwards, we use the *Inverse Normal Cumulative Density Function* (*CDF*) to produce normally-distributed samples out of the generated uniforms. As there exists no exact formula for Inverse Normal *CDF*, we need to use an approximation algorithm. We implement the *Beasley-Springer-Moro* algorithm known for its speed and accuracy following the procedure described in [Gla04].

For the simulation purposes in the second part, every sample needs to be turned into an asset spot price instance at every simulation time step or, alternatively, an early-exercise opportunity. We chose the standard *Geometric Brownian Motion* $GBM(r, \sigma^2)$ with a mean (drift) equal to the risk-free interest rate r and variance (diffusion) equal to a square of volatility σ^2. We use the generated normally-distributed samples to simulate a stochastic process. In practice, as the process is a Markov chain, we know that the current step is independent of the past realisations of the process.

In our case, we deal with one stochastic factor, an underlying asset spot price, as the priced American option is only dependent on one underlying variable. This means the paths are independent from each other. This allows us to parallelise the generation efficiently across the paths by having one thread generate one whole path. Nevertheless, we want to emphasise that our assumption of one underlying is not a limitation to parallelism in the implementation. Many stochastic processes can be generated in parallel as long as we adjust the simulations with the correlations between the stochastic variables, a necessary step in practice.

The code in Figure 4.4 presents a compact *Futhark* implementation of path generation. For each path and step, `UnifRealDist.rand` function draws first a single random number from a uniform distribution. Then function `compute_gbm_normal_step` transforms it to a standard normal distribution and computes a current *GBM* step. It uses function `NormRealDist.invNormalCdf` to approximate the Inverse Normal *CDF* of a uniform sample. At the last time step the cash flow is always known, because the option is invalidated and we cannot exercise it any more after that time step. Therefore, the value is set to the payoff value.

Performance Enabler We gain most performance here by fusing the random sample generation and path generation together into one step like in lines 21–29 in the code in Figure 4.4. We perform these actions for each step based on the observation that we work on the same array for both actions as well as each sample is independent from the all other ones. This way we read from and write to the device global memory only once and thereby save the redundant intermediate memory accesses, that are costly to execute compared to compute instructions.

```
let compute_gbm_normal_step
    (drift: real) (vol: real) (x: real)
    : real =
  drift * exp(vol * (NormRealDist.invNormalCdf x))

let generate_samples_and_paths
    (seed: i32) (m: i32) (n: i32)
    (S0: real) (dt: real) (r: real) (σ: real)
    (payoff: real → real)
    : [n][m]real =
  let rng = minstd_rand.rng_from_seed [seed]
  let rn_range = (0.0, 1.0)
  let rngs = minstd_rand.split_rng n rng
  let drift = exp((r-0.5*σ*σ)*dt)
  let dtSigma = σ * r_sqrt(dt)
  in map (λr →
    let path = replicate m 0.0
    let (path', _, _) =
      loop (path, rng, acc) = (path, r, 1.0)
      for i < m do
        let (rng, num) = UnifRealDist.rand rn_range rng
        let W = compute_gbm_normal_step drift dtSigma num
        let acc' = acc * W
        let v = acc' * S0
        let v' =
          if i < m - 1
          then v
          else payoff v
        let path[i] = v'
        in (path, rng, acc')
    in path'
  ) rngs
```

Listing 4.4: *Futhark* code for the Path Generation part.

4.6.2 Preparation of Singular Value Decomposition

This part is run before entering the main regression loop and covers the main algorithmic optimisation. The main advantage of this approach is that *SVD* for R in each time step can be processed in parallel. As R is small, the intermediate variables easily fit into registers of one streaming multiprocessor (SM). The number of *SVD* matrices to prepare depends on the number of time steps m. This part is compute-intensive, but at the same time the parallelism is limited by the fact that m is usually much smaller than n. A sequential loop is needed to find the first three *ITM* paths to get the asset spot prices to build R matrix. In the body of `prepare_svds` function, the eight required scalars are gathered and computed. They are subsequently passed to `svd_3x3` function that performs the QR decomposition and *SVD* decomposition for R and R^{-1}. It start with assembling the R matrix from the 8 scalars and afterwards uses the iterative Jacobi method to determine the inverse matrix R^{-1}. The function returns 6 upper elements of matrix R and 6 upper elements of the inverse matrix R^{-1}, as the matrices are orthogonal.

Performance Enabler The key to performance here is the fact that we not only perform computation in parallel on the outer level across all time steps m, but we also enable inner parallelism in computation for each of the time steps itself.

First of all, *SVD* preparation benefits significantly from the intra-group parallelism in the first map (lines 5–31 in the code in Figure 4.5), that finds the first three *ITM* paths. The spot prices on these paths are needed for construction of matrix R. It works by taking the `CHUNK` paths at one time and working on them in parallel. It is achieved by a combination of parallel constructs like **`map`**, **`reduce`**, **`scan`**^{exc} and **`scatter`**. `CHUNK` parameter depends the level of intra parallelism here. The smaller its value, the faster this part performs. However, the value cannot be smaller than a number of *ITM* paths, that are needed for constructing R *SVD* matrices. In our implementation, it is determined by `min_itm` parameter and fixed to 4, one more than dimension size of R. The next optimisation that benefits the overall performance is the application of segmented reduction in the lines 33–48 that enables parallelism in gathering the remaining 5 scalars. Afterwards, a map in the line 49 works in parallel across time steps, but internally, for each time step, it uses the matching scalars to compute sequentially the partial *SVD* in `svd_3x3` call.

4.6.3 Main Regression Loop

This part is where the least squares system of equations is regressed, the continuation value is computed and the cash flow per each time step is updated. It can be seen in the code in Figure 4.3 in lines 16–23. This loop has $m - 1$ iterations. The computation in the loop is greatly simplified in the *SVD* preparation step, which is performed before entering the loop, because most of the sequential computation is performed there. The code in Figure 4.6 shows the implementation. Function `compute_`β`s` computes β coefficients required for regression through a multiplication of pseudo-inverse $A^\dagger$ and cash flow vector. Afterwards, the `update_cashflows` estimates a payoff based on βs for each path and determines continuation value for each of them, comparing the estimated payoff with a current payoff of the option.

Performance Enabler Due to the *SVD* preparation before start of the main regression loop, the computational work in each loop iteration is significantly reduced. The other reason is the reduced size of the matrices that are being processed. All the remaining computa-

```
let prepare_svds [m] [n] (Sst: [m][n]real)
    (is_itm: real → i32)
    (min_itm: i32) (CHUNK: i32)
  : ([m][12]real, [m]i32) =
  let svds = map (λSs →
    let svds = replicate 12 zero
    -- Loop to find 3 first ITM paths
    let (svds, _, _, _) = unsafe
      loop (svds, found_paths, path_offs, exit)
          = (svds, 0, 0, false)
      while (!exit && found_paths < 3
          && path_offs < n) do
        let Ss_chunked = map (λi →
            if i + path_offs < n
            then Ss[i+path_offs]
            else zero) (iota CHUNK)
        let itms = map is_itm Ss_chunked
        let exit = reduce (&&) true <| map (==0i32)
          itms
       let scn_ms = scanExc (+) 0 itms
        let tot_sum = scn_ms[CHUNK-1] + itms[CHUNK-1]
        let inds = map2 (λin_m sm →
            if in_m == 1i32 && found_paths+sm < 3
            then found_paths+sm
            else -1)
          itms scn_ms
        let svds = scatter svds inds Ss_chunked
        let found_paths = found_paths + tot_sum
        in (svds, found_paths, path_offs+CHUNK, exit)
    in svds
    ) Sst
  let (ms, sums, all_otms) = unzip3 <|
    map (λSs →
      let itms = map is_itm Ss
      let ms = reduce_comm (+) 0i32 itms
      let sums = map2 (λin_m S →
        if in_m == 1i32
        then (S, S*S, S*S*S, S*S*S*S)
        else (zero, zero, zero, zero)
      ) itms Ss
      |> reduce_comm tuple4_sum_op
          (zero, zero, zero, zero)
      let all_otm =
        if (ms < min_itm)
        then 1i32
        else 0i32
      in (ms, sums, all_otm)
    ) Sst
  let svds = map3 svd_3x3 ms sums svds
  in (svds, all_otms)
```

Listing 4.5: *Futhark* code for the *SVD* preparation part.

tion is performed in parallel across n paths. In Figure 4.6, `compute_βs` use a **`map`** in line 9 followed by a **`reduce`** line 17. The `update_cashflows` follows with one more **`map`** in line 24. The performance of the loop is highly dependent on the number of time steps, as they need to be processed sequentially, because of data dependency between consecutive time steps.

```
 1  let compute_βs [n] (is_itm: real → i32)
 2      (svds: [SLOTS]real) (Ss: [n]real)
 3      (cashflows: [n]real)
 4    : [](real, real, real) =
 5    let R00 = svds[0]
 6    -- Initialise R and W matrices from svds
 7    -- and compute inverse of R and W
 8    λdots
 9    in map2 (λS i →
10      -- Compute Qis. The elements of the Q matrix
11      -- in the QR decomposition.
12      λdots
13      let cashflow = if (is_itm S) == 1i32
14        then cashflows[i] else zero
15      in (WI0*cashflow, WI1*cashflow, WI2*cashflow)
16    ) Ss (iota n)
17      |> reduce_comm tuple3_sum_op (zero, zero, zero)
18
19  let update_cashflows [n]
20      (payoff: real → real) (expmrdt: real)
21      (β: [](real,real,real))
22      (Ss: [n]real) (cashflows: [n]real)
23    : [n]real =
24    map2 (λS path →
25      let old_cashflow = expmrdt * cashflows[path]
26      let cur_payoff = payoff S
27      let (β0, β1, β2) = β
28      let estimated_payoff =
29        (β0 + β1 * S + β2 * S * S) * expmrdt
30      in
31        if cur_payoff <= estimated_payoff
32        then old_cashflow
33        else cur_payoff
34    ) Ss (iota n)
```

Listing 4.6: *Futhark* code for the main regression loop. It consists of a computation of βs for assessing the continuation value with a subsequent update of the cash flows.

4.7 Experimental Results

In this section, we present different experimental tests and discuss their results. We validate the accuracy of the simulations and measure the performance of the implementation by comparing it against other established benchmarks. We run the experiments on **D1** system described in Section 2.5.

4.7.1 Accuracy

To start with, we compare the pricing results with an established benchmark to validate correctness of our implementation. Table 4.1 presents a comparison of different implementations of American Option Pricing. We are pricing an American put option with a fixed strike and constant risk-free rate. The remaining parameters vary as specified in the original paper by Longstaff-Schwartz [LS01]. Columns *FDM*^{orig} and **LSMC**^{orig} denote the results from the original paper. The two remaining columns comprise **LSMC** results for *CUDA* implementation, that we used as a benchmark algorithm, and our *Futhark* implementation. In addition, we mention that the **LSMC** simulation from the original paper uses antithetic sampling, which cuts the number of random samples in half. This algorithmic optimisation leads to a reduced variance of the sample paths as well as an improvement in the overall accuracy of the simulation.

The simulation results compared to *FDM* method have a low error. The difference between simulation results varies slightly for different sets of parameters, but, in general, they are insignificant and are the outcome of using different *RNG*s. The same is valid for both the original *CUDA* and *Futhark* implementations.

S_0	σ	T	***FDM***^{orig}	**LSMC**^{orig}	**LSMC**$^{\textit{\textbf{CUDA}}}$	**LSMC**$^{\textit{\textbf{Futhark}}}$
36	0.20	1	4.478	4.472	4.460	4.465
36	0.20	2	4.840	4.821	4.821	4.826
36	0.40	1	7.101	7.091	7.077	7.092
36	0.40	2	8.508	8.488	8.514	8.518
38	0.20	1	3.250	3.244	3.232	3.239
38	0.20	2	3.745	3.735	3.736	3.739
38	0.40	1	6.148	6.139	6.131	6.147
38	0.40	2	7.670	7.669	7.670	7.661
40	0.20	1	2.314	2.313	2.307	2.313
40	0.20	2	2.885	2.879	2.873	2.878
40	0.40	1	5.312	5.308	5.290	5.319
40	0.40	2	6.920	6.921	6.914	6.909
42	0.20	1	1.617	1.617	1.613	1.612
42	0.20	2	2.212	2.206	2.205	2.205
42	0.40	1	4.582	4.588	4.578	4.590
42	0.40	2	6.248	6.243	6.231	6.234
44	0.20	1	1.110	1.118	1.104	1.104
44	0.20	2	1.690	1.675	1.682	1.680
44	0.40	1	3.948	3.957	3.945	3.952
44	0.40	2	5.647	5.622	5.628	5.637

Table 4.1: Comparison between results from the original paper (*FDM*^{orig} and **LSMC**^{orig}) and **LSMC** implementations in *CUDA* and *Futhark*, **LSMC**^{CUDA} and **LSMC**^{Futhark}, respectively. The strike price of the put option is 40 and the risk-free rate is 0.06. The remaining parameters are as indicated. All **LSMC** simulations are done with $100\,000$ paths and 50 time steps per year.

4.7.2 Performance Validation Case

Model Parameters	**Value**
Option Type, Payoff	Put, $\max(K - S)$
Initial Spot price (S_0)	80.0
Strike price (K)	90.0
Time to maturity (T)	1 year
Risk free rate (r)	5%
Volatility (σ)	30%
Simulation Parameters	
Time steps/Early Exercise dates	100
Paths	1 024 000
Ref. value (Binomial Tree)	13.804

Table 4.2: Set of model and simulation parameters for the American option pricing. We provide an option price obtained from a different numerical method (binomial tree) for reference.

The pricing test case is presented in Table 4.2. It is an example of a typical put option that is *ITM* on the calculation day. The performance results are presented in Table 4.3. The correctness is validated against the benchmark binomial tree numerical method for the same problem. We want to obtain a value that is as close as possible to this benchmark. *Futhark* compiler is able to translate high-level functional language to different backend parallel languages. We test *CUDA* (**V1**) and *OpenCL* (**V2**) backends, and observe identical execution times for both versions. In terms of the speedups, **Ref** is only $1.11\times$ faster than **V2**, what can be considered insignificant. Furthermore, we do not observe large discrepancies in terms of partial execution times among the three main parts of the algorithm. This fact demonstrates that *Futhark* auto-generated low-level code is similar in complexity and on par in performance with one that is hand-tuned.

Indeed, some parts of the algorithm run faster, while other can benefit from further optimisation, for example changing the algorithm. For instance, The (17%) overhead in **Path** part is due to choice and an internal implementation of the *RNG*. We have used a different *RNG* than **Ref**, which uses `CURAND_RNG_PSEUDO_MRG32K3`, a member of the Combined Multiple Recursive family of pseudo-random number generators. On the other hand, the 13% speedup in **Main** stems from a more optimal handling of memory copies in comparison to **Ref**. Furthermore, the effect is amplified, because the loop has 99 iterations.

4.7.3 Performance Scalability Tests

In this section, we present the performance scalability results of our experiments.

	Path	***SVD***	**Main**	**Total**	**Δ**	**Val**
Ref	4.7 (30%)	1.8 (12%)	8.9 (58%)	**15.4**	$1.11\times$	13.778
V1	8.0 (47%)	1.4 (8%)	7.7 (45%)	**17.1**	$1.00\times$	13.789
V2	8.0 (47%)	1.4 (8%)	7.7 (45%)	**17.1**	$1.00\times$	13.789

Table 4.3: Execution times for the verification case. **Ref** is the original *CUDA* benchmark, while **V1** is *Futhark* compiled to *OpenCL* and **V2** is *Futhark* compiled to *CUDA*. Both total and partial execution times for each part of the algorithm are shown. The execution times are given in *ms* and averaged based on 250 runs. **Path** denotes the Path Generation part, **SVD** the *SVD* Preparation, and **Main** the Main Regression Loop. In Δ column, we compare the speedups against the slowest execution time. The obtained values are presented in **Val** column.

Fixed Number of Paths, Various Number of Time Steps

As the next step we test the scalability behaviour of *Futhark* implementation **V2** by gradually changing the number of time steps or paths. These two dimensions are the main parameters that determine the size of the computation involved in a *Monte Carlo* simulation. Consequently, we reuse the test case from Table 4.2 and compare against the benchmark **Ref** for different combinations of these two simulation parameters.

First of all, we fix the number of paths to a relatively high number $1\,024\,000$ and test against 5 different numbers of time steps. High number of paths allows for massive parallelism across the path dimension and thus full utilisation of the *GPU* hardware. Figure 4.1 shows the results of this experiment. Contributions of three algorithmic parts are distinguished and sum up to a total execution time for each tested case.

The main observation is that for the low number of time steps *Futhark* **V2** is faster than benchmark *CUDA* **Ref**. In particular, for very few time steps like 10, it is $\sim 2.5\times$ faster. The difference diminishes with increasing number of time steps to match at ~ 100 time steps. For more time steps, **Ref** is slightly faster than **V2**, but the ratio is maintained with increasing time steps. For instance, it is $1.25\times$ faster for 250 time steps.

The next observation is that for the large number of time steps the **Path** part grows much faster and becomes the main computational bottleneck. It takes 40% for 10 time steps, but more than 52% for 250. This slower execution time is caused by necessary transposition of the layout, which the sampled paths matrix is organised in, from one ordered by the paths to the one ordered by time steps. As a result, we enable coalesced accesses to global memory on *GPU* across time steps that benefits the performance of **SVD** part. We also experience that the contribution of **SVD** part decreases with more time steps. The **V2** implementation is more efficient, because it always executes faster than one in **Ref**. Finally, we achieve better performance on the sequential part **Main** by ensuring that we do not launch any device-to-host memory transfers. On *GPU* architectures such transfers introduce significant execution delays. Hence, the solution is to keep all intermediate data on the device if possible. As a side note, `compute_`β`s` and `update_cashflows` functions in 4.6 exchange an array of βs that is computed sequentially, not in parallel, as it only comprises of 3 elements.

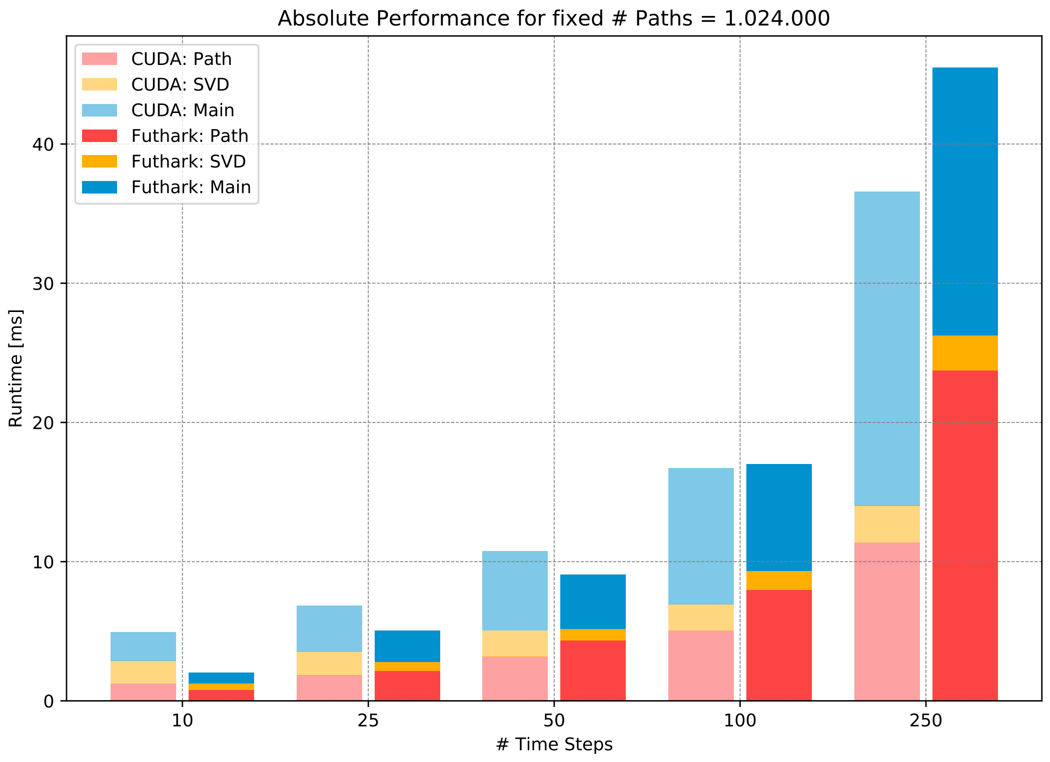

Figure 4.1: Execution time comparison of *CUDA* (**Ref**) and *Futhark* (**V2**). Absolute performance is presented for a fixed number of $1\,024\,000$ paths. Execution time is given in ms.

Fixed Number of Steps, Various Number of Paths

We observe analogous scalability, when the number of paths is increased. *Futhark* **V2** is faster than benchmark *CUDA* **Ref** on low number of paths. In particular, for $10\,240$ paths, it is $\sim 1.25\times$ faster. The difference diminishes with increasing number of time steps to match at $\sim 1\,024\,000$ paths.

Various Number of Time Steps and Paths

Figure 4.3 presents the results for different combinations of time steps and paths. The ratio between them is kept so, that the required work as well as memory requirements are constant. These cases saturate the memory available on **V100**. We can see that the impact of the time steps on the overall execution time is slightly higher than the number of paths. For **V2** it goes from 39 ms to 45 ms. The execution time changes, because the computations in **Main** loop that need to be run one step at a time. In general, the performance of both **Ref** and **V2** is stable across different configurations, which shows that *Futhark* matches the *CUDA* performance. We can observe that *Futhark* **V2** is slightly (1 ms to 13 ms) slower than **Ref** on all cases.

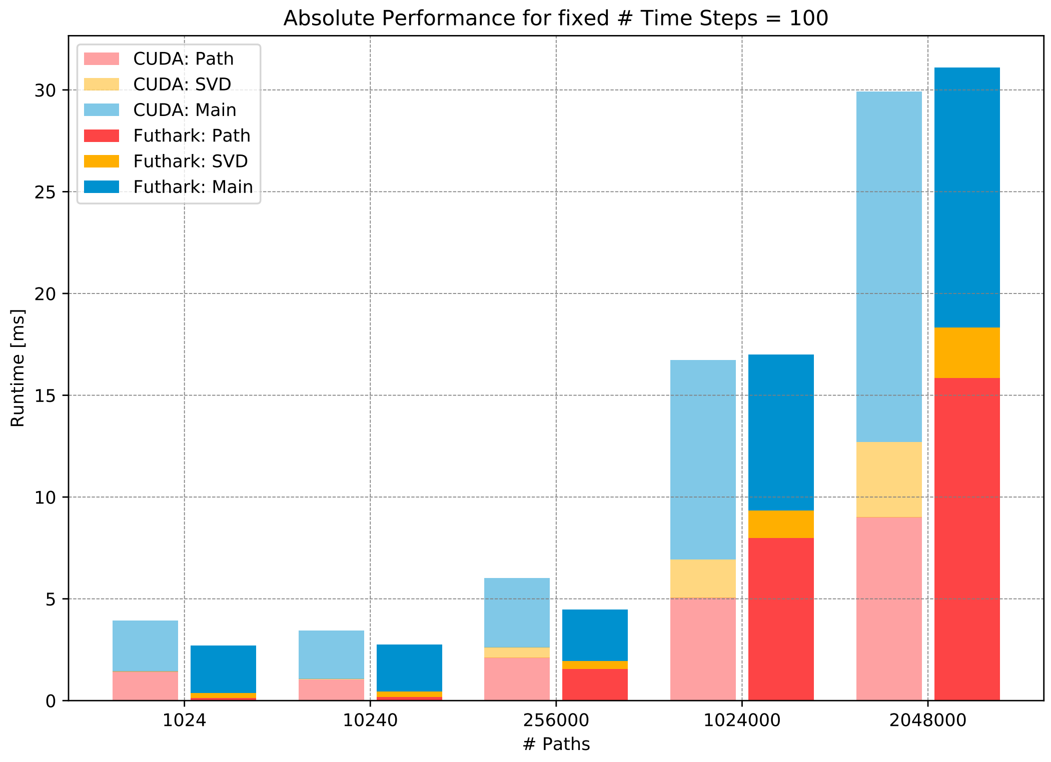

Figure 4.2: Execution time comparison of *CUDA* (**Ref**) and *Futhark* (**V2**). Absolute performance is presented for a fixed number of 100 steps. Execution time is given in ms.

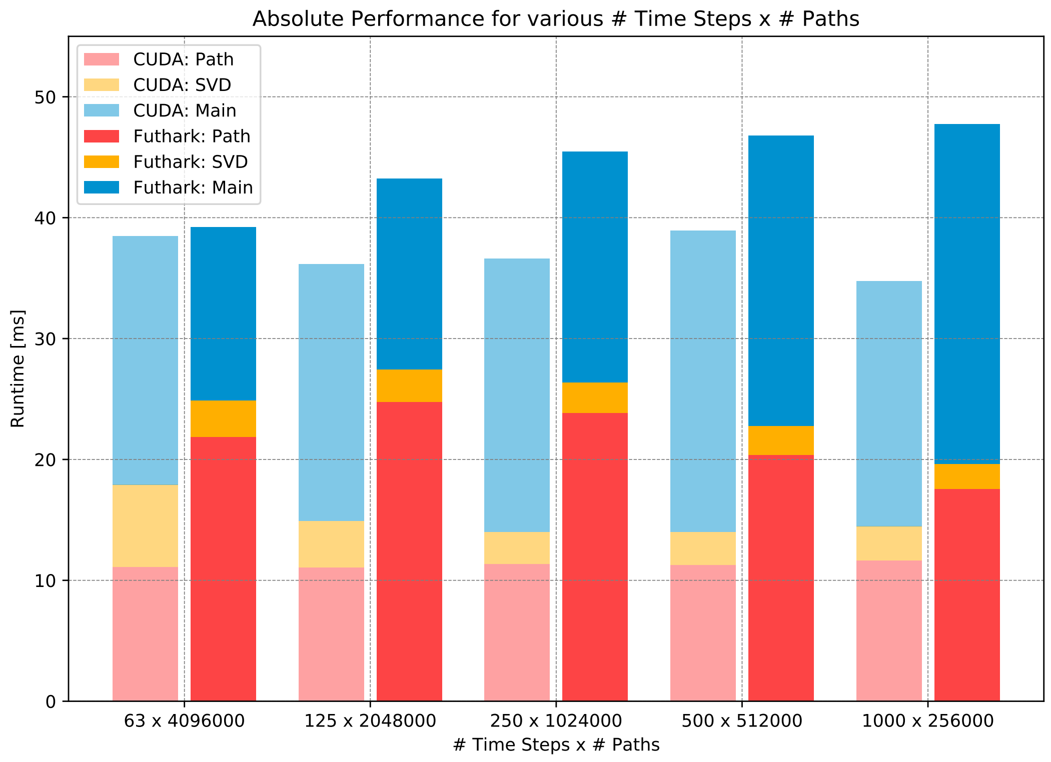

Figure 4.3: *CUDA* (**Ref**) and *Futhark* (**V2**) are compared in terms of execution times. Absolute performance is presented for different combinations of number of time steps and paths. Execution time is given in ms.

4.8 Related Work

We present the related work that is specific for acceleration of **LSMC** problem.

4.8.1 Accelerated Implementations of Monte Carlo Simulations

The most efficient implementations of *Monte Carlo* simulations for American option pricing are implemented in low-level dedicated data-parallel languages and frameworks, For instance, there are many examples using *CUDA* [Abb+14; FP13; PW12; Zha+17]. Other efficient parallel implementations are based on task-parallel approaches [Cho+08; CHL15], which are suitable for multi-core architectures. We are not aware of any accelerated implementations of American option pricing using functional languages.

Previous work has investigated the use of *Futhark* for implementing *Monte Carlo* simulations for European option pricing [And+16; Oan+12]. Even though European option pricing is simpler than, and in fact it is a special case of, American option pricing, the previous work covered a number of advanced features that the present work does not consider. In particular, the previous work on European option pricing considered European options with multiple underlying assets, Sobol sequence generation [HEO18], and options that are *path dependent*, which are options with a price that not only depends on the value of the underlying instrument at maturity, but also on its intermediate values. Using the module language features of *Futhark*, it is possible to parametrise the implementation in such a way that multiple underlying assets and path dependence are supported by the implemented American pricing engine. We consider such advanced features future work. Due to code modularity, it is also straightforward to replace a pseudo-*RN* generation (*PRNG*) with a quasi-*RN* one that uses Sobol sequences (*QRNG*).

4.9 Conclusion

In this work, we present the results of the accelerated implementation of a well-known **LSMC** algorithm for a common financial use case of American Option Pricing. We choose to use a high-level functional approach to the implementation, express the algorithm using succinct parallel constructs and let the optimising compiler auto-generate an efficient parallel code that targets massively parallel hardware. For this purpose, we use the *Futhark* language and address *GPU*s as a suitable compute platform. We demonstrate that with this approach it is possible to achieve the execution times that are, in general, at the same level as the hand-tuned implementations in dedicated languages like *CUDA*, but there exist particular smaller cases, where the implementation beats the benchmark by up to $2.5\times$. This promising finding motivates further work on the optimisation compiler and the algorithm.

We consider the high-level functional specification as being much more suitable and accessible for the financial-domain experts than the low-level dedicated code, that is usually implemented by expert software developers. Its expressibility and modularity enables code maintainability, hiding the implementation details targeting particular parallel architecture, and instead turning focus to algorithmic and domain-specific consideration. It also facilitates algorithmic changes, so prevalent in the financial industry, for example due to a multitude of financial instruments traded in the global markets.

This work has a potential to be integrated as a module in a larger risk management system aimed at large investment portfolios. Such modular and fast pricing capabilities enriched with, for example, sensitivity calculations, can be combined into standard Value at Risk (*VaR*) or more involved Value Attribution (xVA) risk portfolio analytics [Hul18].

Chapter 5

Monte Carlo Value at Risk Simulations (MCVaR)

Abstract

We accelerate *Monte Carlo* simulations for estimating the key risk measures like Value at Risk (*VaR*) and Expected Shortfall (*ES*) using high-level data parallel language to target highly parallel architectures like *GPU*s. The nested *Monte Carlo* simulations that involve compute-intensive simulations for both *MS* generation and revaluation of the portfolio instruments are the most flexible approach to the computation of risk measures. We analyse the risk workload in our implementation by measuring the execution times of *Multicore* and *CUDA* targets compiled from *Futhark* code. We measure execution times of each of its parts and conclude that portfolio pricing part is responsible for majority of the execution time of the complete risk workflow. The portfolio repricing in all *MS*s takes up to $30\times$ on *CPU* and $747\times$ on *GPU* longer than the second most expensive part, that is, *MS* generation. Risk measure calculation is insignificant, even when we calculate the risk measure contributions to portfolio holdings such as *CVaR* and *CES*. Furthermore, we prove our initial hypothesis that we are restricted in our experiments to a sequential execution of portfolio repricing in *MS*s, because **MCVaR** in current implementation consumes prohibitively large amounts of memory in the pricing part. It is especially evident for cases with **LSMC** algorithm used for pricing American Option Portfolio. The already optimised parallel structure of pricing implementations use all the *GPU* compute and memory resources for cases with many sample paths, effectively prohibiting pricing of many portfolio holdings at the same time. However, in our experiments we still observe up to $18.7\times$ speedup of *GPU* execution over *CPU* execution using 32-core on cases, where the inner parallelism over sample paths in the internal pricing simulations is large such as $1\,024\,000$. For the largest measured input, the *GPU* version takes $5.9\,\mathrm{s}$ and achieves $3.8\times$ speedup over the highly multithreaded *CPU* version on a portfolio with 10 holdings. The largest single instrument portfolio case that we manage to execute comprises $10\,000$ market scenarios priced using $1\,024\,000$ *Monte Carlo* paths. The *GPU* version takes $41\,\mathrm{s}$ and achieves a speedup of $11.2\times$ over *CPU* version. This proves that *GPU*s are suited for large risk workloads, if we can efficiently map all innermost parallelism (on pricing

level) to the core structure of the *GPU* architecture. We conclude that further optimisations that target memory footprint reduction are necessary to extract more performance from the parallel implementation of a complete risk workflow. However, at the current implementation state this needs be achieved through changes in the algorithm. To address these shortcomings, we propose further investigations in more efficient use of quasi-random numbers in our implementation.

Section 5.1 outlines the context of the research, explains the identified gap in the problem, presents the main original ideas in the solution, and summarises contributions.

Section 5.2 establishes the necessary financial background for software implementations addressing the problems in risk management and analysis.

Section 5.3 identifies and explains the nested parallel structure of the problem.

Section 5.4 shows how we derive a *Futhark* implementation of the nested simulation and describes the proposed optimisations.

Section 5.5 describes the experimental approach and various test cases as well as discusses the performance impact of each implementation and optimisation.

Section 5.6 compares our work with previous research in the area.

Section 5.7 concludes the chapter and discusses the main outcomes.

5.1 Introduction

Risk management of large investment portfolios involves large-scale calculations on an daily basis. Portfolios across investment managers vary in size and instrument composition. They usually include diverse set of illiquid derivative instruments that do not have a price quoted in the markets. For such exotic derivatives, *Monte Carlo* simulations [1] are often the only viable approach to get an accurate price approximation from a suitable mathematical model. Moreover, to assess the risk in portfolio with a certain confidence we approximate the distribution of relevant risk factors (*RF*s) [2] that have an impact on the portfolio. We choose an analysis date in the future, for which we want to asses the risks of our investments. The next step is to estimate the evolution of each *RF* from calculation date (usually today) over a given future time horizon. The goal is to estimate values of the *RF*s on the analysis date and then use them as input to price each portfolio holding. For *RF* estimation, *Monte Carlo* simulation is, again, a most versatile choice, as it can assume any *RF* distribution. A desired distribution is obtained through a generation of synthetic market scenarios (*MS*s) [3] that specify *RF* values, in a stochastic way using random numbers (*RN*s) [4] , for each date over time horizon. As a result, *MS*s, when combined together, cover the complete space of *RF* values for the risk distribution of the portfolio.

Although *Monte Carlo* simulations are a robust approach, they need a significant number of samples to be accurate. In effect, *Monte Carlo* simulations are traditionally considered to suffer from high computational complexity. We aggravate the complexity further, because we want to not only generate *MS*s through simulation, but also use a *Monte Carlo* simulation to revaluate each derivative in the portfolio in each of these scenarios. Certainly, this flexible approach leads to a larger set of nested simulations that is a problem traditionally considered as computationally infeasible. At present, investment managers run such workloads as *EOD* batch jobs that run over night to prepare the results for the next day, because most often they cannot meet the compute power requirements. Even worse, they need to repeat this costly and time-consuming process from scratch on every business day as they cannot keep their analytics up-to-date on a running basis. When we work with risk analysis, we forecast future values instead of working with historical data. However, the today data that can be considered historical is fed as input to the simulation. Therefore, the results today are not going to be of any value tomorrow, and also cannot be reused in the calculation on the next day. As a result, we need to perform these simulations from scratch and this is a process most often not fast enough to be executed many times during the business hours. This prohibits application of such simulations in intra-day on-demand usage of portfolio rebalancing or what-if simulations. Moreover, lack of intra-day recalcu-

[1]For clarity and not to repeat ourselves, wherever it is applicable we refer to a *Monte Carlo* simulation as just a simulation further in this chapter, because this is the only method (or algorithm) to perform a stochastic simulation that we consider and use in this chapter.

[2]We abbreviate a risk factor to *RF* further in this chapter, because we use this concept frequently.

[3]We abbreviate a market scenario to *MS* further in this chapter, because we use this concept frequently.

[4]We abbreviate a random number to *RN* further in this chapter, because we use this concept frequently.

lations leads to the portfolio composition remaining constant throughout the day. This is an unrealistic assumption and huge restriction in the time of highly-volatile markets. This is also means that investment managers do not have the high-level up-to-date picture of the portfolio risk situation at any given moment. This severely impacts their flexibility and confidence in taking investment decision. To alleviate the described situation, the goal of our work is to enable nested *Monte Carlo* simulations for the on-demand, real-time and continuous risk measurement of complex portfolios through use of efficient algorithms and accelerated software implementations.

The nested simulations result in a *P/L* vector that approximates the risk distribution of a portfolio. Based on these vectors, we calculate the key ratios (measures), that are a set of descriptive indicators that the regulation authorities standardise and market participants coordinate among each other to track risk in their portfolios. In our work, we choose to calculate not only Value at Risk (*VaR*) and Expected Shortfall (*ES*) for the whole portfolio, but also for each of its components (abbreviated *CVaR* and *CES*, respectively) to determine how each instrument and a corresponding *RF*s contribute to a complete portfolio risk. We follow the industry practice and use **MCVaR** abbreviation to identify the whole process of nested simulations for the calculation of key ratios like *VaR*.

We attempt to verify if *GPU* acceleration of **MCVaR** enables *VaR* estimation of large portfolios with thousands of *RF*s and hundreds of thousands of future *MS*s within seconds. Despite the compute power of the modern *GPU*s, **MCVaR** is still a large computational problem that requires extensive memory resources to track the intermediate computation results. In consequence, our implementation needs to take into account the extensive, but still finite compute capabilities and limited memory resources of a modern *GPU*. For instance, 16 GB of device memory on a **V100**-generation architecture puts a hard limit on the number of simulations we can perform at any time. Furthermore, for practical reasons we need to assume any distribution of instruments in the portfolio. Even though we use simulations for each instrument, the required workload size differs across pricing models for these instruments. This fact leads to divergence between individual computations. To address this issue, we need to group computations in a way that maps to a *GPU* most efficiently and execute them in a sequence that minimises the thread divergence. Lastly, we need to consider the number of pricing computations that we can run in parallel as this risk problem has three potential levels of parallelism as specified below.

1. Outer-level parallelism across *MS*s.
2. Inner-level parallelism across portfolio holdings.
3. Innermost-level parallelism across the sample paths of the simulation for derivative pricing.

The input parameters determine the size of each level of parallelism.

1. The common number of *MS*s determines the outer level.

2. The number of portfolio holdings determines the inner level.

3. The common number of sample paths for each simulation pricing determines innermost level.

By common, we refer to the fact that all *MS* and pricing simulations use the same number of sample paths. Our approach is to parallelise or sequentialise each level depending on the input parameters to identify the most efficient configuration for each input case. For instance, we can parallelise the outer level and sequentialise the remaining nested ones.

To meet the requirements of real-time **MCVaR** computations, we combine the development productivity of the *Futhark* programming language, algorithmic optimisations, and the high-performance capabilities of the highly-parallel code that is compiled for execution on *GPU*. High-level functional orientation of *Futhark* allows us to prototype fast as well as modularise the code to enable reuse. The described risk problem is itself scalable in nature, because revaluations across market data scenarios are independent from each other. We harness this fact to map the problem to the *GPU* architecture. Moreover, we reuse and build on top of the derivative pricing implementations from Chapter 4. In this regard, we combine the pricing workloads for individual portfolio holdings to build a larger portfolio pricing workload. As a result, large number of simulations provide us with sufficiently large floating-point computations to fully utilise the massive parallelism of a *GPU*. For this purpose, we exploit the inbuilt *moderate parallelism* functionality of the *Futhark* compiler, by manually tweaking the code to guide the compiler.

To map the risk problem to a *GPU*, we need to optimise for memory footprint and reuse the intermediate results. We address a problem of divergence between different types of pricing algorithms by grouping the calculations together by their computational complexity. The other algorithmic optimisation that we introduce is to use statistical properties and domain knowledge to adapt the algorithm to generate only a single set of *RN*s for each *MS* and reuse them across internal (pricing) simulations. Our functional-oriented codebase is high-level, platform agnostic, maintainable, and expandable. These characteristics are crucial for risk management software, because these highly-heterogeneous applications evolve considerably over their deployment time to adapt to ever-changing financial markets. At the same time, these applications need to scale effortlessly to meet changing functional and non-functional requirements.

We evolve the application from a naive implementation by applying algorithmic optimisations to reduce the memory footprint and reach a performance speedup on the parallel hardware platforms, to enable on-demand and real-time execution, rather than *EOD* and overnight. We asses the performance of these simulations on a synthetic mixed portfolio of European and American equity options. We describe the code implementations as well as discuss the experimental results of our **MCVaR** acceleration work. We discuss the performance impact of different parallelisation techniques and optimisations that we implement. In particular, we compare execution times between multicore *CPU* and parallel *GPU* platforms. For the largest measured input , the *GPU* version takes 5.9 s and achieves 3.8× speedup over the highly multithreaded *CPU* version on a portfolio with 10 holdings. The largest single instrument portfolio case that we manage to execute comprises 10 000 *MS*s priced using 1 024 000 paths. The *GPU* version takes 41 s and achieves a speedup of 11.2× over *CPU* version.

5.2 Risk Measures

In this section, we describe the mathematics behind the commonly used portfolio risk measures (or key ratios) that we deal with in this chapter. We calculate a total *VaR* and *ES* risk measures for the whole derivative portfolio. In addition, we calculate their component equivalents per portfolio holding, *CVaR* and *CES*, to asses contribution of each *RF* to the whole portfolio risk. All mentioned risk measures use the sorted *P/L* vector, which is calculated across *MS*s. We calculate the component ratios through aggregation of component prices across *MS*s. When we calculate risk measures, we need to typically decide on the confidence level, the time horizon, and the statistical distribution for the individual *RF*s. The last component can only be accurately modelled using simulations. The choice of distribution determines how realistic the *RF* model is. Moreover, particular distributions are better in describing the fat tails that represent improbable extreme market events. We describe the computational complexity of calculating these measures by using a simulation in Section 5.3. Here, we only provide minimal description to be able to match the linear algebra transformations and formulas to the implementation code. For more detailed and generic description of the mathematics behind these measures, we refer to standard reference in risk management [Hul18; MFE15].

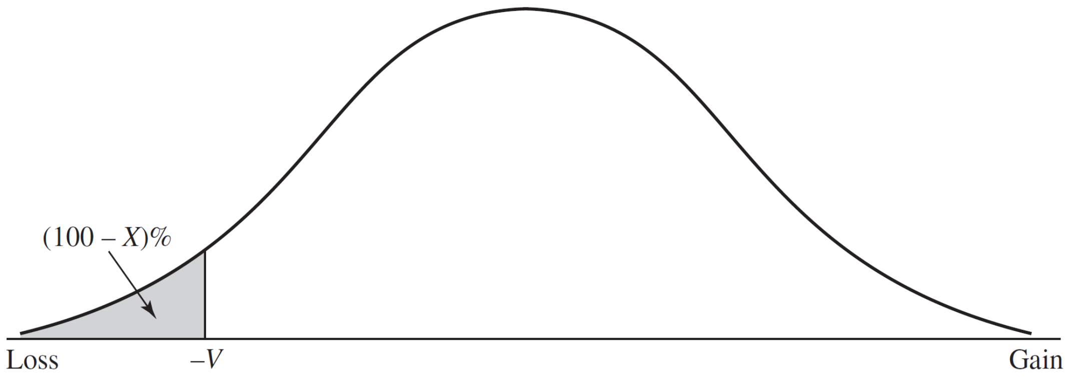

Figure 5.1: Calculation of *VaR* and *ES* from the *P/L* probability distribution of the gain in the portfolio value as presented in [Hul18]. Losses are negative gains, confidence level is X%, *VaR* level is equal to V, and *ES* is the greyed-out space under the curve (integral) between $-\infty$ and $-V$.

5.2.1 Value at Risk (VaR)

Value at Risk (*VaR*) is a standard measure used by portfolio managers to asses the amount of money that can be lost or gained over a specific time horizon given some confidence level. *VaR* measures the risk at a particular confidence level, points to a critical scenario, and indicates the estimated losses for that scenario, that is, it only describes what happens at but does not describe what happens beyond in the tail to the left of the *VaR* level. Therefore, *VaR* is a single *P/L* value at a particular index in the *P/L* vector. For example, *VaR* at 99.0 percent captures 1-day-in-100 events. We make an assumption that holdings remain constant over the time horizon.

$$\mathit{VaR}_\alpha(L_{t+\tau}) = \inf_{l\in\mathbb{R}}\{F_{t+\tau}(l) \geqslant \alpha\} = \sum_{i=1}^{N} h_i \frac{\partial \mathit{VaR}_\alpha}{\partial h_i} \tag{5.1}$$

VaR is a not a subadditive measure, which means that the risk measure for two portfolios after they have been merged should be no greater than the sum of their risk measures before they were merged.

5.2.2 Expected Shortfall (ES)

Expected Shortfall (which is abbreviated *ES*, and is also called Conditional *VaR*) is closely related to *VaR* and its confidence level. In contrast to *VaR*, *ES* is the average of the estimated *P/L* in the tail to the left of *VaR*. In other words, *ES* is a mean of an aggregation of a subset of *P/L*s in this vector. This measure indicates how large the loss is on average for the scenarios not covered by the confidence level.

$$\mathit{ES}_\alpha(L_{t+\tau}) = \frac{1}{1-\alpha}\int_\alpha^1 \mathit{VaR}_\gamma(L_{t+\tau})\mathrm{d}\gamma = \sum_{i=1}^{N} h_i \frac{\partial \mathit{ES}_\alpha}{\partial h_i} \tag{5.2}$$

Furthermore, in comparison with *VaR*, *ES* is subadditive.

5.2.3 Component Risk

We calculate the marginal contributions of individual holdings to the risk measure of the whole portfolio. These measures give insights into which holding in reality increases the portfolio risk and how large is its impact.

$$\mathit{compVaR}^i_\alpha = h_i \frac{\partial \mathit{VaR}_\alpha}{\partial h_i} \approx -k h_i S_i \tag{5.3}$$

$$\mathit{compES}^i_\alpha = h_i \frac{\partial \mathit{ES}_\alpha}{\partial h_i} \approx -q h_i S_i \tag{5.4}$$

5.3 Nested Simulations in Monte Carlo Repricing Approach

This section describes the computational structure of the nested simulation problem.

5.3.1 Problem Assumptions

Practical risk modelling is a complex application that depends on many parameters and consists of many intermediate steps. Our goal is to build the core computation engine for a risk modelling application. Therefore, to allow us to focus on its main computational challenges in our experiments, we need to make some assumptions and put constraints on the risk modelling problem. As a result, the resulting model is less practical and realistic, but sustains the structure of the complete workflow that allows us to draw conclusions about its overall performance. In fact, once we have the core implemented, the application can be then made more practical by using its modularity and gradually extending it to support more functionality.

We make the following assumption in our experiments.

1. We look at a problem of assessing financial risk of an investment portfolio that consists of a varying number of European and American equity options.
2. We use nested simulations and a full revaluation approach, so we not only use simulations to generate *MS*s, but also use them to price the portfolio instruments.
3. We choose short time horizons of 1 day or 10 days, but longer horizons are also met in practice.
4. We set the confidence level to 99% in our experiments, but 95% and 99.9% are also commonly used in practice.
5. Each individual instrument depends on many different *RF*s. However, we only simulate a single characteristic *RF* using a standard stochastic process, while we keep the remaining constant.
6. Equity derivatives are dependent on a spot price modelled with a log-normal *GBM* stochastic process. The volatility and risk-free rate are constant over time horizon.
7. *RF*s are not correlated with each other, that is, their individual correlations across time steps are equal to 0. As a result, we assume a covariance matrix of *RF*s to be an identity matrix.
8. We do not change the instrument composition of the portfolio over the time horizon, that is, there are no transactions during the simulated time horizon.
    a) We assume no cash flows on the portfolio instruments, so its modelled price determines the value of each instrument on each given date.
    b) There are no reinvestments in the original instruments, so the nominals do not change.
    c) The set of *RF*s driving the portfolio risk is fixed over the time horizon.

9. Time steps in simulations are always equal to one day.

10. We use fixed ranges to initialise the values of the *RF*s to be able to recreate our experiments.

11. We use a fixed seed for *RNG*s to be able to recreate our experiments.

In a realistic setup, we compute correlations between individual *RF* returns along their evolution in time and build a covariance matrix for each time step. This way we adhere to an intuitive and observable market phenomenon that the changes in any given *RF* influence the remaining to variable degree.

In contrast to single-underlying options, from a computational point of view, a simulation of the derivatives that derive their value from the behaviour of multiple *RF*s grows linearly, proportional to the number of these factors. In other words, for each underlying asset or *RF* we use a separate sample path. In fact, for practical purposes, adding more *RF*s and underlyings assets is always accompanied with an extra computational cost on pricing level, because we need to calculate correlations between the processes for these underlying assets.

5.3.2 External Simulation: Market Scenario Generation

We start with simulating a *RF* distribution. At this stage, we need to generate *MS*s to simulate how the portfolio reacts to different changes in *RF*s. *RF*s are a proxy to translate future conditions in the financial market. Each *MS* in a simulation is defined by an individual sample path that propagates one time step at a time to simulate "what can happen the next day". In our experiments, we deal with daily time series. We generate up to 100 000 *MS*s for each of the *RF*s in the portfolio. We choose to use a simulation to generate different *MS*s as it allows us to generate any probability distribution for the relevant *RF*s. The probability distribution that we choose is the normal distribution, because the stochastic process that we use in simulation is based on this distribution. For the *RF* driving equity derivatives, we use *GBM* to simulate spot prices of underlying stocks over the specified time horizon.

5.3.3 Internal Simulation: Derivative Pricing

To asses a value of a portfolio at the time horizon at each *MS*, we need to find a price of each instrument on that date. The classical numerical approaches to derivative pricing involve simulations, apart from lattice models and finite-difference methods, to solve underlying stochastic and partial differential equations. In finance, a more generic *Monte Carlo* simulation estimates the value of a financial instrument in contexts where we cannot use faster more specialised methods like analytic, numerical integration, or *FDM*. This means that we can apply the derivative pricing models that we describe in Chapter 4.As we notice there, the pricing itself requires extra computational power as well as high *FP64* precision to yield accurate approximations to the future prices of derivative instruments.

Market Scenario Generation provides us with an input to pricing. We take the simulated *RF*, the spot price `S0`, on the last day of the simulated time horizon and set it as an initial spot price in our derivative pricing model. We then simulate the process until the maturity of the given derivative. In this project the priced portfolio consists of different instruments that require different models. For European Equity Options, we use a pricing algorithm that is also based on a *Monte Carlo* simulation for a standard Black-Scholes pricing model (*BSMC*). We achieve a better approximation accuracy at the cost of the computational complexity by increasing the number of paths used for pricing one instrument. On top of that, to investigate the required computational effort, we investigate European path-dependent option pricing, such as barrier options with the knock-out feature, the Asian options with the averaging feature and the lookback options with a payoff dependent on the realised extremum value of the underlying asset price. If necessary, we can easily extend the latter model with cash flows varying over the lifetime of a derivative, for example, for dividends paid out on a regular basis. For each simulated path, we can control the number of steps, to match the cash flow dates. Finally, for American Equity Options, we use the Least Squares *Monte Carlo* simulation, described in Chapter 4.

5.4 Accelerated Implementation

In this section, we provide a detailed description of our implementation of the risk measurement system in *Futhark* that incorporates all functionality that we describe in the earlier theoretical sections.

5.4.1 Technical Challenge

Performance-oriented implementations that target highly parallel computing platforms like *GPU*s are highly sensitive to implementation factors such as data layout or placement and thread synchronisation. For highly-numerical applications like risk analysis, these factors lead to code that is infeasible to read and maintain. In standard parallel programming frameworks, complex algorithmic logic blends with verbose low-level optimisations that address thread and memory management. We tackle this problem and motivate use of high-level languages for such use cases to hide the aforementioned low-level implementation details. The actual end goal is to allow the financial domain experts, who typically are not trained to deal with the more sophisticated technical challenges, to write high-performance code on modern architectures.

At a first sight, a simulation procedure for **MCVaR** calculation is well-suited for a *GPU*, because the problem involves a large amount of computation, has multiple levels of parallelism, and is embarrassingly parallel on each level. These characteristics are reflected in code structure as a simulation handles each sample path, while pricing each instrument in each *MS* in isolation, independently from all the others. In fact, the bulk of computation originates from our choice of a repricing evaluation method for the risk problem. We choose to revaluate each instrument in a portfolio, which can consist of hundreds of thousands of different instruments. At the same time, the risk flow assumes a small set of input parameters, and outputs a compact set of risk measures as a result. Favourably, these features lead to a small number of host-device memory operations between simulation steps, as the intermediate results are saved and reused locally in the device global memory. On the contrary, extensive use of device memory comes at a cost and leads to an increased memory footprint of the computation at any given time, which, in fact, effectively limits how many *MS*s and instruments can be handled in parallel. Therefore, to address and minimise these memory problems, we propose a set of algorithmic optimisations and make an effort to reuse data where applicable.

As mentioned before, the algorithmic structure of **MCVaR** involves nested simulations. On the outer risk level we generate *MS*s that use a simulation. *MS*s are shared among instruments and in general can be generated independently from pricing. On the contrary, the *MS* step has to complete before we move to the instrument pricing, because *MS*s are its input. Afterwards, on the inner pricing level each instrument in the portfolio is again priced using a simulation in each scenario. In more detail, the steps in the naive sequential implementation of **MCVaR** algorithm are as follows:

1. Value the portfolio today using the current values of *RF*s.
2. Sample once from the multivariate normal probability distribution of the Δx_i.
3. Use the sampled values of the Δx_i to determine the value of each *RF* at the end of time horizon.
4. Revalue the portfolio at the end of time horizon using *RF* values from previous step.
5. Subtract the value calculated in step 1 from the value in step 4 to determine a sample ΔP.
6. Repeat steps 2 to 5 many times to build up a probability distribution for ΔP.
7. Use probability distribution to calculate risk measures like *VaR* or *ES*.

5.4.2 Random Number Generation

To perform any *Monte Carlo* simulation, we need ot generate *RN*s. We use two types of Random Number Generators (*RNG*) to produce *RN*s for simulations.

PRNG Pseudo-random numbers that use Minimal Standard `minstd_rand` generator.

QRNG Quasi-random numbers that use Sobol sequences.

Selection of *RNG* does not only influence the rate of convergence of a simulation, but also, more importantly, has a significant impact on the performance of the whole simulation provided that the *RN* sequences are generated in parallel. We use *RN*s in each level of simulations that all comprise **MCVaR**. For each of the levels, we generate a separate matrix of *RN*s, because each of the simulations can have different dimensions.

PRNG using `minstd_rand`

We use a fixed seed for the *PRNG* to be able to recreate our experiments.

Listing 5.2 show how we use *PRNG* in our implementation.

```
module UnifRealDist =
  uniform_real_distribution f64 minstd_rand

let getPRNGs
  (seed: i32)
  (rngCount: i64)
=
let rn_range = (zero, one)
let rng = UnifRealDist.engine.rng_from_seed [seed]
let (rng, _) = UnifRealDist.rand rn_range rng
let rngs = UnifRealDist.engine.split_rng rngCount rng
in (rn_range, rngs)

let getMinStdRandPRNs
  (seed: i32)
  (pathCount: i64)
  (stepCount: i64)
  (convert: real → real)
: [pathCount][stepCount]real
=
let (rn_range, rngs) = getPRNGs seed pathCount
in map (λrng_init →
    let path = replicate stepCount zero
    let (path', _) =
      loop (path, rng) = (path, rng_init) for i < stepCount do
        let (rng, num) = UnifRealDist.rand rn_range rng
        let path[i] = convert num
        in (path, rng)
    in path'
  ) rngs
```

Listing 5.1: *Futhark* function for the generation of the pseudo-*RN*s.

QRNG using Sobol sequences

Sobol sequences are quasi-random low-discrepancy number sequences often used in *Monte Carlo* simulations. They are superior to traditional *PRNG*s for solving numeric integration problems, where a *Monte Carlo* simulation is most often used. The goal for numerical integration is to fill the many-dimensional hypercube as evenly as possible. Because of its low discrepancy, Sobol sequences span the integration space better than *PRNG*s avoiding the holes and clusters that are inevitable with random sequences of pseudo-*RN*s that follow a less structured pattern. This feature also means that we need less numbers to converge to the satisfactory solution.

Sobol sequences may be multi-dimensional and a key property of using Sobol sequences is that we can freely choose the number of points that should span the multi-dimensional space. In contrast, if we set out to use a simpler uniform sampling technique for spanning two dimensions, we can only span the space properly if we choose the number of points to be in the form x^2, for some natural number x. This spanning problem becomes worse for higher dimensions[EHO18]. The choice of direction numbers determines how many independent dimensions of Sobol sequences we can generate. The most widely used direction number sets are the ones prepared by Joe and Kuo [JK03; JK08] with dimension up to 21 201 as well as by Jäckel [Jäc02] with dimension up to 8 129 334. Convergence rate for *QRNG* with Sobol sequences is $1/n$, while for *QRNG* the rate is only $1/sqrt(n)$. Sobol sequences are problematic, because they need to be generated all at once. They are usually stored in a fixed size array, so the number of dimensions needs to be known at the compile time. This leads to a large memory footprint.

Listing 5.2 shows how we use *QRNG* in our implementation.

Normallly-Distributed Random Numbers

We want to sustain some realism in our distribution. Thus, the next step is to convert the generated uniformly-distributed *RN*s to normal distribution. This is typically achieved through computation of the inverse of Cumulative Density Function (*CDF*) of a Normal Distribution. We need to approximate the function, because no specific closed-form formula for normal distribution exists. There exist different algorithms that vary in computational complexity. We choose the Beasley-Springer-Moro approximation algorithm following [Gla04, p. 68, fig. 2.12 & 2.13] and implement it as a *Futhark* `invNormalCdf` function. We gather the implementations of this and other probability conversion algorithms in a `norm_real_dist` custom *Futhark* module for convenience and reusability. In general, it is beneficial to assemble such a library of reusable statistical algorithms, because function like the inverse *CDF* are a commonly used in different financial applications. Furthermore, we implement the `getNormValRange` function to ensure that the values that we draw are in a specific interval.

5.4.3 Equity Option Portfolio Generation

In a real production setup, the current state of the portfolio is retrieved from the database and the updated market data, for instance, instrument prices or volatilities for calibration, are streamed to the system in real-time from a market data provider. At the current setup, we assume, for the sake of brevity, that we are not provided with a realistic portfolio and do not have access to the realistic market data. Instead, we choose to generate input test data on-the-fly. We create a portfolio of a size specified at the input and populate it with different derivative instruments: in-the-money (*ITM*), at-the-money (*ATM*) and out-of-the-

```
open import "lib/github.com/diku-dk/sobol/sobol-dir-1000"
open import "lib/github.com/diku-dk/sobol/sobol"

let maxSmallQRNGDim = 10i64
module SmallSobolQRNG = Sobol sobol_dir {
  let D = maxSmallQRNGDim }
let maxMediumQRNGDim = 100i64
module MediumSobolQRNG = Sobol sobol_dir {
  let D = maxMediumQRNGDim }
let maxLargeQRNGDim = 1000i64
module LargeSobolQRNG = Sobol sobol_dir {
  let D = maxLargeQRNGDim }

let getSobolQRNs (sobolRnT: rngType) (chunkIndex: i32)
  (pathCount: i64) (stepCount: i64)
  : [pathCount][stepCount]real
  =
  let QRNs = match sobolRnT
    case #QRNGSMALL → SmallSobolQRNG.chunk chunkIndex pathCount
    case #QRNGMED → MediumSobolQRNG.chunk chunkIndex pathCount
    case #QRNGLARGE → LargeSobolQRNG.chunk chunkIndex pathCount
    case _ → MediumSobolQRNG.chunk chunkIndex pathCount


  in map (λi → map (λj → QRNs[i, j] )
    (iota stepCount)) (iota pathCount)

let getRNs
    (rnT: rngType)
    (seed: i32)
    (pathCount: i64)
    (stepCount: i64)
  : [pathCount][stepCount]real
  =
  match rnT
    case #PRNG →
      getMinStdRandUnifDistPRNs seed pathCount stepCount
    case #QRNG →
      getSobolQRNs #QRNGMED seed pathCount stepCount
    case #QRNGSMALL →
      getSobolQRNs #QRNGSMALL seed pathCount stepCount
    case #QRNGMED →
      getSobolQRNs #QRNGMED seed pathCount stepCount
    case #QRNGLARGE →
      getSobolQRNs #QRNGLARGE seed pathCount stepCount
    case _ →
      getMinStdRandUnifDistPRNs seed pathCount stepCount
```

Listing 5.2: Call to *Futhark* Sobol library for the generation of the quasi-*RN*s from Sobol sequence.

money (*OTM*) European and American options. Listing 5.3 presents the code for this part of **MCVaR** calculation.

Four of them describe the instrument type and cash flow In our case, seven different variables characterise each derivative. and are fixed at the instrument creation, when two counterparties make an agreement on the contract between each other. The remaining three are *RF*s are external to the contract, are subject to change over time, and impact the value of the derivative during its lifetime. Based on choice of the pricing model, these *RF*s are used as input parameters to the model.

Option Style The choice is between *European* or *American* options. In our implementation, the style determines the choice of pricing algorithm. *European* options are priced with *BSMC*, whereas *American* options with **LSMC** algorithm.

Option Type A vanilla options that we support in our implementation are a *Call* or a *Put*. This parameter determines the option payoff depending on the current market situation.

Maturity/Step Count Maturity `T` of the option determines when the last date that the option can be exercised on. We use a year as the maturity time unit, so the values can be fractional. Using model parameters expressed as annual values is a standard practice in derivative pricing. Step Count is derived from the maturity and is expressed in equally sized time steps, that is, available business days during the year. We need a day value representation to be able to deduct it from time horizon, that is in practice much shorter than the opinion maturities and expressed in days rather than in years. In addition, step count is necessary to determine the number of sequential **`loop`** iteration in the approximation algorithms.

Strike Price The exercise price `K` of the option.

Spot Price The spot price `S0` of the equity underlying the option.

Risk-Free Rate The interest rate `r` used for discounting of the option.

Volatility The annual volatility σ of the option used for the derivative pricing. For risk measurement purposes, the annual volatility is scaled to a daily form by dividing the annual value by $\sqrt{250}$, where 250 is the assumed average number of business days in the year.

The above-mentioned values are randomly drawn from a standard normal distribution and assigned to each portfolio instruments. To achieve that, we need to generate *RN* sequences for each of the variables. The process is simple, because the fixed number of dimensions is known at the compile time. To start with, we use *PRNG* or *QRNG* to generate 7 independent sequences of random values in a uniform distribution. In our case, we fix the intervals at the compile time to allow us to recreate the experiments. Year values for maturity as well as percent values of interest rate and volatility need to be scaled so they match the right unit. In our experiments we apply both daily and annual variables, so we need to pay special attention to which one we use, and we adapt our code to use annual or daily volatility across risk and pricing parts, respectively.

```
type TEquityOptData = {
  OptionStyle    : i8    -- option style: 0: EU, 1: US
  , OptionType   : i8    -- option type, 0: Call, 1: Put
  , StepCount    : i32
  , T            : real -- maturity, [years]
  , K            : real -- option strike price
  , S0           : real -- spot price at initial time 0
  , r            : real -- risk-free rate
  , σ            : real -- volatility
}

let generateEquityOpt
    (optDimRNs: [optDimCount]real)
    (pfDist: EPfDist)
    (termStepCount: i32)
  : TEquityOptData
  =
  -- Risk Factor Value Intervals
  let (minOptStyle, maxOptStyle, minOptType, maxOptType,
    minT, maxT, minK, maxK, ...) : (real, ...) = ...

  let T =
    ((getNormValRange optDimRNs[2]
      (minT*busDinY) (maxT*busDinY) 1000) / busDinY)
  in
  {
    ...
    StepCount = r2i (T * i2r (termStepCount)),
    T         = T,
    K         = (getNormValRange optDimRNs[3] minK maxK 100),
    ...
  }

let generateEquityOptPf [pfHCount]
    (RNs: [pfHCount][optDimCount]real)
    (pfDist: EPfDist)
    (termStepCount: i32)
  : [pfHCount]TEquityOptData
  =
  map (λnums →
    generateEquityOpt nums pfDist termStepCount
  ) RNs

-- Generate American/European Equity Option portfolio
let pfHs : [pfHCount]TEquityOptData =
  generateEquityOptPf pfGenRNs pfDist termStepCount
```

Listing 5.3: *Futhark* code for the generation of the portfolio of European equity options.

5.4.4 Outer Parallelism Level: Market Scenario Generation

This part describes a simulation implementation that allows us to simulate changes in *RF*s over the specified time horizon to use them to asses the *P/L* distribution, which is then in the final step used to compute the risk measures. In our implementation, we represent a *MS* as a set of *RF* values on a analysis date. We are usually interested in the *MS* on the day at the end of time horizon. The time horizon defines the number of business days into the future that we want to calculate the risk measures for the portfolio. The typical choice used in practice is 1 day or 10 days (2 business weeks), but it is not unusual to see longer time horizon like 1 month or 1 year.

In a realistic setup all *RF* values change over time, but in our case we simulate changes in only a single *RF*, that is, spot price `S0`. We keep values of the other *RF*s constant over the time horizon. We make this assumption to simplify the problem for the sake of presentation. Moreover, we assume that each option in the portfolio has a different underlying equity that use a different spot price `S0` at the initial step for each of them. At the same time, we assume no correlation between any two equities.

The simulation uses *GBM* process to generate different stock path evolutions for each portfolio instrument. Again, in a realistic setup, a separate and suitable stochastic process is used for to simulate changes in each of the *RF*s. For interest rate, it can be a *MROU* stochastic process described in Chapter 3. To simulate changes in volatility, a popular choice is *GARCH* time series model. It is also beneficial to take into account the correlations between all these stochastic processes. Simulating each of these processes comes with their own computational cost.

Listing 5.4 presents the code for this part of **MCVaR** calculation. Market Scenario Generation is again based on *RN*s that come from *PRNG* or *QRNG*. We generate a separate *RN* set for this purpose. We draw them upfront from uniform distribution and pass them to the generation function, which translates them to normal distribution using inverse *CDF*. This time we fuse it with the computation of a *GBM* step in a separate `compute_gbm_normal_step` function.

We need to simulate spot price values on each of the days up until time horizon, for each of the *MS*s. To achieve this, we need the other *RF*s, that is, the initial risk-free interest rate and volatility for each portfolio instrument. We get them from the portfolio generation step. Time horizon determines the number of steps in the external **MCVaR** simulation, that is, `extStepCount`. This dimension of the simulation is sequential in its nature and cannot be parallelised, because there are data dependencies between the consecutive steps. The number of *MS* is the input to **MCVaR** algorithm. Number of *MS*s determines the number of sample paths in the external simulation, that is, `extPathCount`. This dimension of the simulation can be parallelised, because each sample path can be generated independently. We end up with a computational problem of producing an array of size `extPathCount` $\times$ `extStepCount` of real numbers. However, following our problem assumptions, we can simplify the computation and output and array of size `extPathCount`. We are able to do it, because we simulate one *RF* and only need the intermediate spot values inside the generation function. More importantly, we are only interested in values at the time horizon for each of the *MS*s, which can be found in the last step of the simulations. We pass the result values as model parameters to derivative pricing, which is the next step. This implementation involves a fast **`map-reduce`** in line 22. We generate two separate *MS* sets for European and American options. This structure stems from the fact that different option types are driven by different *RF*s. Such approach gives us flexibility to adjust the *RF*s for each them in isolation.

```
 1 let generateEquityMSsForPfH
 2   (extPathCount: i64)
 3   (extStepCount: i64) -- time horizon
 4   (rnT: rngType)
 5   (S0: real)
 6   (r: real)
 7   (σ: real)
 8 : [extPathCount][]real
 9 =
10 let seed = getUniqueSeedExt S0 r σ
11 let extRNs : [extPathCount][extStepCount]real =
12   getRNs rnT seed extPathCount extStepCount
13 let extStepCountp1 = extStepCount + 1
14 let dt    = one / businessDaysInYear
15 let drift = r_exp((r-half*σ*σ)*dt)
16 let dtVol = σ * r_sqrt(dt)
17 let Ws    = map (computeGbmNormalStep drift dtVol) (flatten extRNs)
18 let paths = map (λpath →
19     map (λday → if (day == 0) then S0 else Ws[path*extStepCount + day-1])
20       (iota extStepCountp1)
21   ) (iota extPathCount)
22 in map (reduce (*) one) paths
23
24 ...
25 let euOptPfMSs : [][extPathCount]real =
26    map (λpos →
27    generateEquityMSsForPfH extRNs pos.S0 pos.r pos.σ)
28   euOptPfHs
29 let usOptPfMSs : [][extPathCount]real =
30   map (λpos →
31   generateEquityMSsForPfH extRNs pos.S0 pos.r pos.σ)
32   usOptPfHs
33 ...
```

Listing 5.4: *Futhark* code for the generation of *MS*s through simulation of different underlying equity paths for each of the derivative positions in the portfolio. It starts with generating *RN*s reused across external simulations using the *RNG* specified on the input. Spot `S0` is the only simulated *RF*. The paths are generated using *GBM*.

Performance Enabler

1. Both *PRNG* or *QRNG* can be used to obtain *RN*s for *MS* generation that uses outer simulations, but *QRNG* fills the sample space more uniformly and thus approximates more accurately the distribution of relevant *RF*s. However, if number of scenarios is small, the calculated risk measure values vary considerably, because simulated values across *MS*s differ from each other. In a practical case the number of *MS*s needs to be large, at least 10 000, to account for enough variability in market data.

2. The simulated *RN*s cannot be reused across portfolio holdings (`pfHs`). We cannot reuse the same set of scenarios, that is, sample paths of the outer simulation, across all portfolio derivative holdings, because each holding is described by a unique set of model parameters. We use these model parameters to generate in isolation a set of *MS* that only describes the evolution of that particular holding. In other words, we make an assumption that these *MS*s are uncorrelated with *MS*s generated for the other holdings.

3. In our case, external simulations comprise sample paths of spot prices for equities (stocks) that underlie each derivative in the portfolio. We have a one-to-one mapping, because each derivative depends only on a single underlying. This assumption means that we simulate the same kind of *RF* for different types of holdings in the same time horizon. Thus, we have no thread divergence in this part of the algorithm.

5.4.5 Inner (Nested) Parallelism Level: Derivative Pricing

In this section, we describe our implementation of the internal simulation that is used to price each portfolio holding in each *MS* simulated in the external simulation. As described in Section 5.4.3, portfolios in our case consist of both European and American options. We price the former with *BSMC* algorithm and the latter with **LSMC** algorithm.

As described in Section 2.1.2, a simulation can price a derivative with any optionality and cash flow structure. For example, it can price a path-dependent option, which payoff depends on how the underlying asset performs during the whole lifetime of the option, Alternatively, it can price an option that depends on many underlying assets. Consequently, with a simulation the choice of possible options to price is only limited by the existence of a stochastic processes and the capability of the pricing model to grasp its variability. The general structure of the algorithm remains the same and do not depend on the pricing model we decide to choose.

Memory requirements across simulations vary highly due to differences in option characteristics and input parameters used in pricing model. On the one hand, for European options we only need to store an array of size equal to the number of paths used for each *RF* that we simulate, because we only need values at the maturity of the option. On the other hand, for American options, which is depended on the whole time series data, we need to store *RF* information about each time step on each sample path. This means we need to store a two-dimensional array of size equal to a number of paths $\times$ a number of time steps.

This procedure is repeated for each *RF* that we simulate. When we price path-dependent options, we need to accumulate even more information when we propagate through path over time. For example, for a barrier option we track the maximum and minimum value of the *RF* per path, whereas for the Asian option we track an average value per path. Obviously, keeping track of this additional information introduces a computational complexity and extra memory footprint for each value on each path. The extra cost is proportional to the number of time steps and paths. In contrast, when we add a new underlying asset that an option depends on, the computational complexity grows linearly by a number of assets, because each of them is described by its own set of sample paths with a number of time steps. On top of that, we also want to incorporate the correlations between different simulated *RF*s. To achieve that, we build an additional square covariance matrix of the size equal to the number of *RF*s used in the derivative pricing. Nevertheless, options with many underlyings or multiple *RF*s, which drive their value, as well as treating correlations between them are beyond the scope of our experiments in this chapter.

The *BSMC* can only be used to price European options with a single exercise date at the option maturity. Some path-dependent options can also be priced with this model as long as we track the relevant data to compute a payoff functions of such an option.

The value of any derivative is dependent on its future value. In other words, we can only derive a value of a derivative at a certain time if we know if is going to be exercised subsequently or not, as well as if it occurs before or at its maturity. As a results, we need to know the evolution of the option over its whole lifetime up until its maturity. Option maturities are often significantly longer than the risk time horizon used in **MCVaR**. While a typical time horizon is 1 day or 10 days into the future, the option maturities are set months into the future. To compute the value at the end of time horizon, we move the initial point for derivative pricing to the time horizon by deducting the time horizon business days from the overall maturity of the option.

In conclusion, we approximate the distribution of the stock price outcomes over the remaining lifetime of the option to calculate the expected value. We need *RN*s to execute a simulation. Each time step along the sample path represents one dimension, for example, modelling a 1-year option with 250 time steps means that we have 250 dimensions. This in turn means that we need 250 independent *RN* sequences, each of length equal to the number of sample paths. Again, we use *PRNG* or *QRNG* to generate *RN*s. We then use the *RN*s and *GBM* process, which underlies the Black-Scholes model, to generate stock price paths for simulation. At this point, we emphasise that this model can be replaced, if necessary, with more sophisticated stochastic processes like, for example, Heston model with a stochastic volatility that changes randomly over time. As a result, We relax the assumptions about constant volatility and incorporate more realistic features in our modelling.

In the simulation, we use an iterative method to discretise the *GBM*[] process over time steps of the sample path. The most basic, but at the same time most popular, iterative method is the Euler-Maruyama scheme. We use the scheme to simulate each sample path.

In addition, we need to be careful about how we treat the process increments when using this scheme. Consequently, we apply logarithmic scaling to ensure that the spot prices do not end up being negative.

In the Euler-Maruyama scheme, parallelism exists in both spatial and temporal dimensions. We are thus able to flatten the array that contains stochastic variables and compute each step independently. To compute increments of the process, we use **`scan`** operation. The result of this operation allow us to track the evolution of the option along future time to check for early-exercise or path-dependent features. Finally, to compute the final expected value of the option, we **`reduce`** paths with a sum operator and 0 neutral element.

European Path-Dependent Option Pricing with *BSMC*

In this section, we describe the implementation of *BSMC* that is used to price a European option. In Listing 5.6 we show *Futhark* code that prepares input for pricing European equity option portfolio with the `bsmc` parallel implementation.

We start with pricing each holding in the portfolio at the start of the time horizon to then aggregate the prices and get the base portfolio price on the calculation day. From a computational perspective, the code comprise nested parallel maps first over portfolio holdings, then over *MS*s and finally over sample paths used for option pricing.

In `bsmc_genPaths` function that we call for each portfolio holding we generate a set of sample paths using *PRNG*. We apply an optimisation that reduces the memory requirements of the simulation. We normalise the values in the paths in the internal simulation by setting the initial spot price `S0` to 1.0 before passing it to `bsmc_genPaths` function in lines 24–26. The normalisation enables a path reuse across *MS*s for each portfolio holding. In other words, we make an assumption that *RN*s that are used for each *MS* are the same. We treat the *RN* vector as an input to the *Monte Carlo* simulation and expect the same output. To illustrate this situation better, let us compare a stochastic *Monte Carlo* simulation with a deterministic numerical method like a lattice method from Chapter 3. For the same input parameters, it always returns the same values. This is valid, because we are pricing different holdings in the same *MS*s. Market scenario describes a specific market For each *MS*, we need to initialise *RNG* with a different seed. We simply shift all the values on each path by the spot price taken from the given *MS*.

The result is a vector of spot prices of size of `intPathCount`, because we only need to know the spot price at the maturity. Once we have spot prices at the maturity for a normalised case, we multiply the normalised spot price by the spot price from *MS*, apply the option payoff function to each path using **`map`**, **`reduce`** all the paths with sum operator, apply the discounting and divide it by the number of paths (`intPathCount`). `bsmc_pricePaths` function in lines 19–32 executes the above procedure and returns a price of a European option.

```
let bsmc_genPathsPathDep
  (seed: i32) (pathCount: i32) (stepCount: i32)
  (T: real) (S0: real) (b: real) (σ: real)
: [pathCount][stepCount]real =
...
in map (λrn →
    let path = replicate stepCount zero
    let (path', _, _) =
      loop (path, rng, acc) = (path, rn, S0)
      for i < stepCount do
        let (rng, num) = UnifRealDist.rand rn_range rng
        let W = computeGbmNormalStep drift dtSigma num
        let acc' = acc * W
        let path[i] = acc'
        in (path, rng, acc')
    in path'
  ) rngs

let bsmc_pricePathDep_Denorm [pathCount]
  (normSs: [pathCount][stepCount]real) (T: real) -- years
  (S0: real) (r: real) (payoffFun: real → real)
: real =
let normSs_PathDep : [pathCount](real, real, real, real) =
map (λS_path →
  map (λS_step → (S_step, S_step, S_step, S_step)) S_path
  |> reduce_comm tuple4_diff_op (one, zero, zero, zero)
) normSs
let payoffs = map (λ(S, avgS, minS, maxS) →
  payoffFun (S0 * S, S0 * avgS, S0 * minS, S0 * maxS))
normSs_PathDep
let sum = reduce (+) zero payoffs
in ((r_exp (-r*T)) * sum) / (i2r pathCount)

let bsmc_priceFullPathDep
  (seed: i32) (pathCount: i32) (stepCount: i32)
  (opT: t_optT) (T: real) (K: real) (H: real) (k: real)
  (S0: real) (r: real) (b: real) (σ: real)
: real =
let payoffFun
    ((S, avgS, minS, maxS) : (real, real, real, real))
  : real = ...
let Ss : [pathCount][stepCount]real =
  bsmc_genPathsPathDep seed pathCount stepCount T S0 b σ
in bsmc_pricePathDep Ss T r payoffFun
```

Listing 5.5: Simplified *Futhark* code for *BSMC* pricing for a path-dependent option. The . . . symbol denotes code or function arguments omitted for brevity.

```
let priceEuPathDepOptPfHs [pfHCount] [extPathCount]
    [intPathCount] [maxIntStepCount]
  (extStepCount : i32)
  (intRNs      : [intPathCount][maxIntStepCount]real)
  (pfMSs       : [pfHCount][extPathCount]real)
  (pfHs : []TEquityOptData)
: (real, []real, [extPathCount]real, [][extPathCount]real)
=
-- Calculate price of the portfolio on the calculation day
let basePfHPrices : []real =
  map (λpos → bsmc_priceFullPathDep ... ) pfHs
let basePfPrice : real = reduce (+) zero basePfHPrices

-- Calculate price for each scenario and each holding
-- at VaR time horizon
let msPfHPrices : [pfHCount][extPathCount]real =
  map (λpos →
    ...
    let intStepCount =
        pfHs[pos].StepCount - extStepCount
    let T =
        (pfHs[pos].T * busDinY - (i2r extStepCount))/busDinY
    let normSTs =
      bsmc_genPathsPathDep_Norm ... intPathCount intStepCount
        T one -- normalise by setting S0 = 1.0
        pfHs[pos].r pfHs[pos].σ
    in
      map (λmsS0 →
        bsmc_pricePaths_Denorm normSTs T msS0 pfHs[pos].r ...
      ) pfMSs[pos]
  ) (iota pfHCount)

-- Sum holding prices for each λMS{}
-- to get λMS{} portfolio prices
let msPfPrices =
  map (reduce (+) zero) (transpose msPfHPrices)
in (basePfPrice, basePfHPrices, msPfPrices, msPfHPrices)
```

Listing 5.6: Simplified *Futhark* code that executs European option pricing by passing the relevant parameters to *BSMC* parallel implementation. `busDinY` is the number of business days in the year by default set to 250. The . . . symbol denotes code or function arguments omitted for brevity.

American Option Pricing with LSMC

In this section, we describe our approach to pricing a portfolio of American options for the aim of successive calculation of its risk measures. A simulation for pricing American options is on average at least two orders of magnitude more compute- and memory-intensive than a corresponding simulation for European options. The American pricing algorithm not only saves all steps across all simulation sample paths, but also performs a backward regression part that traverses all these paths from maturity to the calculation day. The first constraint leads to a memory allocation for a two-dimensional array of size `intPathCount` $\times$ `intStepCount`, which severely restricts the number of parallel pricing simulations, especially when the `intPathCount` and `intStepCount` are large. In our experiments we assume that in the worst case we want to use up to $1\,000\,000$ sample paths and up to 3 years of weekly time steps. in **LSMC** pricing to get an accurate approximation to American option price. As a result, the memory requirements of saving in worst case ($1\,000\,000 \times 150 \times 8$ B $= 1.12$ GB) make it infeasible to price several *MS*s at the same time on modern *GPU*s (Table 2.1), not to mention parallel processing of several portfolio holdings. The second constraint is caused by data dependency between time step results, thereby involving a sequential processing of price values across sample paths one time step at a time, which is described in Section 4.3.1. Consequently, we need to adapt the computational structure of the code for sequential execution on the aforementioned outer simulation levels of portfolio holdings and *MS*s. In particular, we begin with an outermost sequential loop over portfolio holdings in lines 17–42 and inside of it nest a second sequential loop over *MS*s in lines 31–40. On the innermost level for each *MS* we execute a parallel implementation of American option pricing using a **LSMC** algorithm. We describe the corresponding pricing algorithm in more detail in the context of a single instrument in Section 4.6. In particular, Listing 4.3 presents the code implementation.

Listing 5.7 demonstrates *Futhark* code that prepares input parameters for pricing American equity option portfolio with the parallel **LSMC** implementation in function `lsmc_opt`. In `priceUsOptPfHs` function, we start by calculating the base price of the portfolio on the calculation day, that is, in case of **MCVaR** at the start of time horizon. Portfolio pricing in involves pricing each of portfolio holdings in isolation followed by summarising the results. The next step involves pricing each portfolio holding in each *MS*. Here, we follow the analogous procedure to the one described for *BSMC* in Section 5.4.5. We reuse the sample paths across the simulated *MS*s, but generate a separate set of *RN*s for each portfolio holding. Function `lsmc_generateSst` is responsible for generation of *RN*s that are reused for all pricing calls for given *MS* by normalising the spot by setting `S0 = 1.0`. In fact, `lsmc_generateSst` is the same as `generate_samples_and_paths` in Listing 4.4, except it does not compute payoff on the last step and the resulting paths are returned in transposed form. Afterwards, we price each portfolio holding using `lsmc_pricePaths_par`. We adjust, that is, denormalise, the initial normalised spot price for each *MS* by multiplying the spot values on each path and each step by the original initial spot price. We retrieve the necessary initial value from *MS* structure `pfMSs[pos, ms]` at the *VaR* time horizon. This time we price the portfolio holdings at the end of *VaR* time horizon in future, so this horizon expressed in days needs to be subtracted from each options maturity. Finally, we sum holding prices for each *MS* to get *MS* portfolio prices in the same way as for base portfolio price.

As a result, the parallel structure of *Futhark* adheres to the memory requirements constraints. While the portfolio pricing in base scenario is performed in parallel in line 10, the portfolio pricing in different simulated *MS*s in lines 17–42 is done in a sequential **`loop`**. Each portfolio holding is priced one after another in each *MS* in a sequential manner. However, inside the **`loop`** iteration for *MS*, we have an innermost parallel level, because we derivative pricing itself uses a parallel implementation of the **LSMC** pricing.

Grouping Options for Pricing at Portfolio Level

This section moves on to describe in greater the approach we take to pricing a portfolio that comprise several different instrument types. We repeat that to find a value of an investment portfolio that comprise many instruments, we need to first know the value of each instrument, which comes from pricing it in isolation. Each instrument is priced with a different model and often a different numerical method is used to solve it. In our case, we deal with a portfolio of various European, path-dependent, and American options. A naive approach involves pricing such instruments together in parallel, that leads to a divergence in a computational workload among threads. To remedy this problem, an intuitive approach is to differentiate instruments based on their type and handle their pricing by grouping them together in batches.

We implement the grouping in the code excerpt shown in the Listing 5.8. This part of the main risk function groups instruments in batches based on their instrument type. `PfH` or `pfH` abbreviations in the variable names in all Listings in this chapter stand for portfolio holding. `EuOpt` prefix in the variable names denotes a European option, while `UsOpt` is an American Option. `ext` prefix indicates the external simulation, while `int` is the internal simulation. Finally, `base` prefix denotes a value of a portfolio variable at the calculation day, while `ms` indicates the portfolio variables in *MS*s. We launch the inner simulations to price European and path-dependent options using `priceEuPathDepOptPfHs` function in Listing 5.6 from Section 5.4.5 and American options using `priceUsOptPfHs` function in Listing 5.7 from Section 5.4.5.

Performance Enabler

1. We reuse sample paths across *MS*s for each portfolio holding. During each internal simulation for derivative pricing, all parameters besides spot price are kept constant across *MS*s. That means we assume that the only *RF* that changes in each *MS* is the one that we simulated in the external simulation. We obtain the spot price at the end of time horizon from each of market data scenarios that are simulated during

```
let priceUsOptPfHs [pfHCount] [extPathCount]
    (extStepCount : i32)
    (intPathCount : i32)
    (pfMSs        : [pfHCount][extPathCount]real)
    (pfHs         : []TEquityOptData)
  : (real, []real, [extPathCount]real, [][extPathCount]real)
  =
  -- Calculate price of the portfolio on the calculation day
  let basePfHPrices : []real =
    map (λpos → ... in lsmc_opt ... ) pfHs
  let basePfPrice : real = reduce (+) zero basePfHPrices

  -- Calculate price for each scenario and each holding
  -- at VaR time horizon
  let msPfHPrices_init =
    map (λ_ → replicate extPathCount zero) (iota pfHCount)
  let msPfHPrices : *[pfHCount][extPathCount]real =
    loop msPfHPrices = msPfHPrices_init
    for pos < pfHCount do
      ...
      let intStepCount =
        pfHs[pos].StepCount - extStepCount

      let T =
        (pfHs[pos].T * busDinY - (i2r extStepCount))/busDinY
      let normSst : [intStepCount][intPathCount]real =
        lsmc_generateSst ... intPathCount intStepCount
          T one -- normalise by setting S0 = 1.0
          pfHs[pos].r pfHs[pos].σ

      let msPfHPrices_pos_init = replicate extPathCount zero
      let msPfHPrices_pos : *[extPathCount]real =
        loop msPfHPrices_pos : *[extPathCount]real =
          msPfHPrices_pos_init
        for ms < extPathCount do
          let price =
            lsmc_pricePaths_par normSst ...
              T pfMSs[pos, ms] pfHs[pos].r ...
          let msPfHPrices_pos[ms] = price
        in msPfHPrices_pos
      let msPfHPrices[pos] = msPfHPrices_pos
    in msPfHPrices

  -- Sum holding prices for each λMS{}
  -- to get λMS{} portfolio prices
  let msPfPrices =
    map (reduce (+) zero) (transpose msPfHPrices)
  in (basePfPrice, basePfHPrices, msPfPrices, msPfHPrices)
```

Listing 5.7: Simplified *Futhark* code that executs American option pricing by passing the relevant parameters to **LSMC** parallel implementation shown in Listing 4.3 and described in Section 4.6. `busDinY` is the number of business days in the year by default set to 250 The . . . symbol denotes code or function arguments omitted for brevity.

```
...
let (baseEuOptPfPrice, baseEuOptPfHPrices,
  msEuOptPfPrices, msEuOptPfHPrices) : (real, []real,
    [extPathCount]real, [][extPathCount]real) =
  priceEuPathDepOptPfHs
    extStepCount intRNs euOptPfMSs euOptPfHs
let (baseUsOptPfPrice, baseUsOptPfHPrices,
  msUsOptPfPrices, msUsOptPfHPrices) : (real, []real,
    [extPathCount]real, [][extPathCount]real) =
  priceUsOptPfHs
    extStepCount intPathCount usOptPfMSs usOptPfHs

let basePfPrice = baseEuOptPfPrice + baseUsOptPfPrice
let basePfHPrices : [pfHCount]real =
  concat_to pfHCount baseEuOptPfHPrices baseUsOptPfHPrices
let msPfPrices : [extPathCount]real =
  map2 (+) msEuOptPfPrices msUsOptPfPrices
let msPfHPrices : [pfHCount][extPathCount]real =
  concat_to pfHCount msEuOptPfHPrices msUsOptPfHPrices
...
```

Listing 5.8: Code excerpt for Inner Nested Simulations: **MCVaR** with **LSMC**. Calculate the longest maturity in the portfolio. Generate *RN*s reused across internal simulations. Calculate prices for the European and American options in the portfolio in two separate functions. The sizes of `baseEuOptPfHPrices` and `baseUsOptPfHPrices` arrays are unknown at the compile time as the composition of the portfolio is dynamically specified at runtime. This fact limits the compiler optimization possibilities.

external simulation. We then observe that two independent external simulations use two independent *RN* sets. Therefore, spot prices in two separate market data scenarios cannot be the same.

2. Because the same *RN*s are reused and all other pricing parameters are constant, the internal simulation for each *MS* generates the same paths. We can obtain exactly same set of paths for each internal simulation if we set the spot price to 1 and compute a normalised version only once, and then shift the all paths by multiplying every value by a spot price that comes from external simulation for each *MS*. The shift means that we reuse the normalised paths by a simple multiplication of values on each path and each step by the spot price from the given external simulation. This saves the memory, because we load only a single one-dimensional array (for European options or two-dimensional for American options) per *MS* in memory at any time for a further reuse. We save also computational time, because we do not need to generate paths every time.

3. For European option, we only need to save the spot values at the option maturity, because the pricing requires only the values at the last time step. For European path-dependent options, we need additional values for each path at the maturity step, that

is, we need an average spot price `avgS` over the whole path for an Asian option and a minimum `minS` as well as a maximum spot `maxS` over the whole path for a barrier option. In contrast, for American option we need to save all steps for all paths, which means that we need to save a full two-dimensional array.

5.4.6 Risk Measure Calculation

The last essential step is to calculate the risk measures for the portfolio at the end of time horizon, that is, at time τ in future. As mentioned, this calculation is based on the future *P/L* distribution of the portfolio. To asses the *P/L* distribution of the portfolio, we need to:

1. calculate the base portfolio value at the start of time horizon t_0,
2. calculate the value of each of the portfolio instruments in each simulated *MS* at the end of time horizon τ,
3. sum instrument values to get the portfolio value for each scenario at τ,
4. subtract base portfolio value at t_0 (step 1) from portfolio values at τ for each *MS* (step 3),
5. sort the portfolio changes (step 4) in the ascending order from the largest loss to the largest gain.

We notice that in our case the value of the spot price `S0` is the value of the initial spot price at t_0 (the initial time step 0). It is different for each of the portfolio instruments. We calculate risk measures based on the value of spot at τ (the time step `timeHorizonDayCount - 1`). The portfolio *P/L* changes over the time horizon τ from the above procedure are represented as a *P/L* vector the describes the simulated *P/L* distribution over the specified time horizon.. The largest portfolio loss is indicated by the *MS* with the largest negative portfolio value. Analogously, the largest portfolio gain is indicated by the *MS* with the largest positive portfolio value. Now we have all the components to calculate any risk measures for the portfolio at the end of time horizon τ.

We are mostly interested in the left tail of the distribution, where we observe the largest losses. Risk measures like *VaR* or *ES* that treat the largest losses consider only this part of the distribution. Each risk measure is accompanied by its probabilistic confidence level α that corresponds to the percentile of the distribution. The most popular choice for the confidence level are 95, 99 and 99.9%. Because our *P/L* vector is sorted, we can directly index the critical scenario that lays on the confidence level. *VaR* is the (negative) value of the *P/L* vector element at the index associated with the confidence level, whereas *ES* is the mean of the values up to that index. For example, for 100 *MS*s sorted in the ascending order the 99^{th} percentile of the *P/L* distribution is located at the index 1 of the sorted *P/L* vector.

Listing 5.9 presents the described procedure for risk measure calculation implemented in *Futhark*.

```
let calculateRiskMeasures [extPathCount] [pfHCount]
    (basePfPrice: real)
    (basePfHPrices: [pfHCount]real)
    (msPfPrices: [extPathCount]real)
    (msPfHPrices: [pfHCount][extPathCount]real)
    (α: real)
  : ([extPathCount]real, real, real, ...,
    [pfHCount]real, [pfHCount]real, ...)
  =
  let pfPricePnLs =
    map (λprice → price - basePfPrice) msPfPrices
  let (sortPfPricePnLs, sortIndices) =
    zip pfPricePnLs (iota extPathCount)
    |> radix_sort_float_by_key (.0) r_num_bits r_get_bit
    |> unzip
  let percIndex = extPathCount - r2i (α * i2r extPathCount)
  let VaR = sortPfPricePnLs[percIndex]
  let ES = Stats.mean sortPfPricePnLs[0 : percIndex + 1]
  ...
  -- component VaR for each holding
  let varIndex = sortIndices[percIndex]
  let (compVaRs, compDeltaVaRs)
      : ([pfHCount]real, [pfHCount]real) =
    map2 (λbasePfHPrice msPfHPrices →
      let comp = msPfHPrices[varIndex] - basePfHPrice
      in (comp, comp / VaR * hundred)
    ) basePfHPrices msPfHPrices |> unzip2
  -- let compDeltaVaRs = map (λcomp → ) compVaRs
  -- component ES for each holding
  let esIndices = sortIndices[0 : percIndex + 1]
  let (compESs, compDeltaESs)
      : ([pfHCount]real, [pfHCount]real) =
    map2 (λbasePfHPrice msPfHPrices →
      let comp = Stats.mean (
        map (λmsPfHPrice → msPfHPrice - basePfHPrice)
          (map (λpos → msPfHPrices[pos]) esIndices)
      )
      in (comp, comp / ES * hundred)
    ) basePfHPrices msPfHPrices |> unzip2
  -- let = map (λcomp → comp / ES * hundred) compESs
  in (sortPfPricePnLs, VaR, ES, deltaVaR, deltaES,
    compVaRs, compDeltaVaRs, compESs, compDeltaESs)

...
let (sortPfPricePnLs, VaR, ES, ...
    compVaRs, compDeltaVaRs, compESs, compDeltaESs) =
  calculateRiskMeasures
    basePfPrice basePfHPrices msPfPrices msPfHPrices α
...
```

Listing 5.9: Code excerpt for Inner Nested Simulations: **MCVaR** with **LSMC**. Calculate *VaR* based on portfolio gains/losses in different simulated *MS*s.

5.5 Experimental Results

We run the experiments on **D1** system described in Section 2.5. All code is implemented in *Futhark*. For each of our experiments, we build two versions of executables: *Multicore* compiled with *Futhark* `multicore` backend targeting multicore *CPU*s and *CUDA* compiled with *Futhark* `cuda` backend targeting *GPU*s. *Multicore* uses *POSIX Threads*, also known as `pthreads`, libraries in Linux to spawn many concurrent threads for execution on multicore *CPU*s. We have also experimented with *OpenCL* backend on **CPU3** and **GPU1**. At the time of experiments, we are not able to successfully start a benchmark run on **CPU3** platform that was using the latest *OpenCL* driver from *Intel*. On **GPU1**, the execution time results are consistently slower than matching ones for the *CUDA* backend. Therefore, we decide to abandon it entirely in our **MCVaR** experiments. That is why. we revert to using *Multicore* backend We execute *Multicore* versions on **CPU3** (32 threads) on and *CUDA* version on **GPU1** (**V100**) (refer to Section 2.5). We note here that we select a *CPU* platform with a 16 hyper-threaded cores that is classified as a *HPC*-grade server system, which is optimised for compute-intensive workloads. Alternatively, it can easily be considered to be a many-core platform. We claim that it is more fair to compare a **V100** system against a multi-threaded execution on a *CPU* platform that has access to a substantial number of threads than measure speedups against sequential runs. This is even more evident, when we consider workloads for risk analysis, which are normally executed on clusters of distributed computation nodes. We consistently execute each benchmark run 5 times to reduce the deviations in the measurements. Using more runs makes prohibitively long for feasible measurements on larger workloads, which in any case are not as sensitive to deviations caused by system operations, etc., as short micro- and millisecond runs. In our long-running **MCVaR** experiments on the chosen platforms, we observe that the measurements become stable with Relative Standard Deviation > 0.01, if they take more than $10\,\mathrm{ms}$ on **GPU1** and $100\,\mathrm{ms}$ on **CPU3**. The experiment case sizes are mainly motivated by (i) domain knowledge based on the client cases observed in the industry (that is, the number of *MS*s and the number of portfolio holdings), but restricted by (ii) memory available to the **GPU1** device. We deduce the latter based on our experiments, because we are able to run all the *Multicore* cases on **CPU3**, but on larger cases we run into memory overflows on **GPU1**. It is especially evident in *RN* generation, American Option Portfolio pricing and the full risk workflow runs described in the next sections. In the result presentation, we mainly put emphasis on comparing speedups between *CUDA* execution over *Multicore*. All our experiments in this section are performed using *FP64*numbers. The missing data in Tables and Figures are due to the fact that our *CUDA* version runs out of memory during its execution on **GPU1**.

5.5.1 Validation Test

We are unable to find a matching implementation in a different technology to compare our whole setup with in a structured manner. However, as our calculations are built on top of the pricing components from the other chapters, we have a proof that they are correct on the fundamental level. For instance, **LSMC** was validated in 4.7.2. Our risk workflow comprise revaluation of portfolio holdings in various *MS*s, which in practical terms means that we call the validated pricing functions with different inputs.

5.5.2 Random Number Generation

Efficient *RN* generation is fundamental to our **MCVaR** implementation, because we need to generate large matrices of *RN*s for both external and internal simulations. In the experiments in this section, we try to verify if we can generate Sobol number sequences in parallel fast enough for risk analysis purposes, and investigate how they compare in performance with ordinary *PRNG*s like `minstd_rand`. We need to asses (i) which *RNG* is more suitable, that is, gives us a satisfactory approximation to a uniform distribution, with less samples required, and (ii) which one of them generates *RN*s faster. We asses the performance of *RNG*s in isolation, that is, the result of our experiments is a matrix of size `pathCount` $\times$ `stepCount`. As the returned *RN*s are generic, the matrix can then be passed as input for further processing to the consecutive steps in our risk workflow, that is, portfolio generation, *MS* generation or pricing parts for further processing.

We run experiments for two different types of random generators that we describe in Section 5.4.2, *PRNG* and *QRNG*. As mentioned in Section 5.4.2, in current *Futhark* `sobol` module we need to specify the dimension of *QRNG* at compile time. Based on the input datasets in our Full Workflow experiments, described in the next sections, we make an assumption that we need three different dimensions `D` – 10, 100, and 1000. We initialise three different *QRNG*s with the mentioned `D` and denote them Q^{10}, Q^{100}, and Q^{1000}, respectively, when we present the results. In our tests, we generate all the *RN*s in a single run. Moreover, we denote *PRNG* as P in the presentation of the results.

We use the same seeds to initialize both types of *RNG*s. However, for *QRNG* their function is different than for *PRNG*, because it is used to determine, which dimensional vector to start with. The low-numbered dimensional vectors do not exhibit sufficient variability of *RN*s for practical purposes. For *PRNG*, it is a start of a sequence of *RN*s. For purpose of parallel execution, each thread needs to get its own seed for sequences to be unique. The `split_rng` functions from `random` *Futhark* library resolves this issue for us.

Execution along `pathCount` dimension is fully parallelised for both *RNG*s, because the paths for our purposes can be generated in isolation. With *PRNG* we can choose an arbitrary `stepCount`, because we draw *RN*s along the `stepCount` dimensions one at a time. With *QRNG* we can also choose an arbitrary `stepCount`, but *QRNG* first draws `D` *RN*s. Afterwards, if `stepCount` is less than `D` of an initialised *QRNG*, we trim *RN*s in each generated vector `D` to fit to `stepCount` and we execute this process in parallel for all vectors along the `pathCount`. We use a **`map`** operation on the (flattened) matrix to filter out the *RN*s. This procedure is necessary, because the current implementation of `sobol` module forces us to specify the dimensionality at the module initialisation. `stepCount` is an obvious dimension to choose not only because we need independent *RN*s in this dimension to simulate a propagation of a stochastic process, but also because `stepCount >> pathCount` for any practical use case that we considered. In addition, based on the domain knowledge we can expect this dimension to change less often, because in risk analysis time horizons, which are expressed in days equal to `stepCount` for daily reporting, are usually set to particular low fixed values such as 1 day, 10 days, 14 days or 1 month for reporting purposes. Effectively, it is mostly `pathCount` that determines the accuracy of the simulation.

The main motivation behind use of high-dimensional *QRNG* is a flexibility it gives in runtime. The range of steps that we can specify is higher, because we can draw any number that we need for `stepCount`. However, such approach has two negative consequences. Firstly, keeping redundant *RN*s in global memory is a wasteful use of scarce resources, especially because they are never going to be used. We have to remember that *RN* is *FP64* kept in *GPU* global memory. Secondly, the trim step introduces an additional memory copy, because the operation cannot be done in-place to sustain memory alignment for coalesced memory accesses, when *RN*s is going to be consumed in consecutive workflow steps. On top of that, adding the trim step to *QRNG* allows us to generalise the Random Number Generation process, because the *RN* matrix dimensions returned by *PRNG* and *QRNG* are the same.

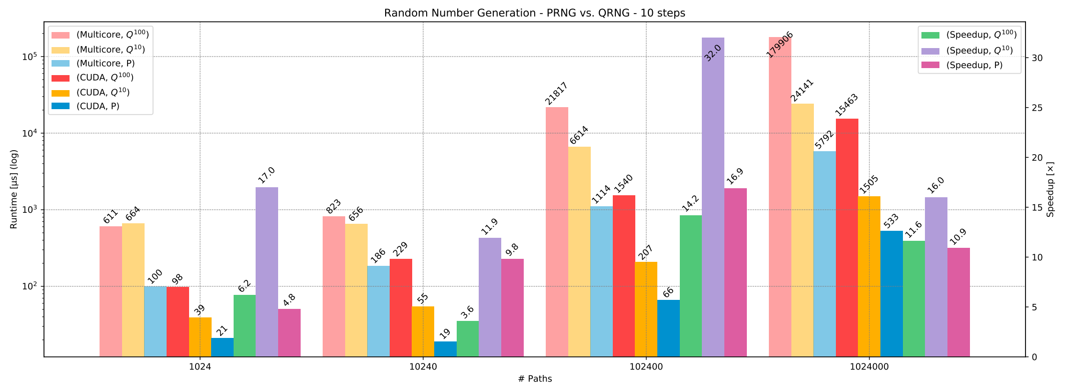

Figure 5.2: Runtimes [μs] and Speedups of *PRNG* and *QRNG* for Random Number Generation using 10 steps and variable number of paths executed on **CPU3** and **GPU1**. The last three bars in each # Paths group represent a speedup of *CUDA* over *Multicore* execution for Q^{100}, Q^{10}, and P, respectively.

Figure 5.2 and Table 5.1 shows runtime results for different executions of *RN* generation, when the requested `stepCount` is equal to 10. We compare P, Q^{10}, and Q^{100} on 4 different `pathCount`s. First of all, we verify our hypothesis that *QRNG*, which generates too many redundant *RN*s that are subsequently trimmed, is a high cost to pay for runtime flexibility that it provides. If we consider the difference between Q^{10} and Q^{100}, we can

Runtime [μs]	*Multicore*			*CUDA*			$\Delta(\times)$		
RNG	Q^{100}	Q^{10}	P	Q^{100}	Q^{10}	P	Q^{100}	Q^{10}	P
# Paths									
1024	611	664	100	98	39	21	6.2	16.8	4.7
10240	823	656	186	229	55	19	3.6	11.8	9.4
102400	21817	6614	1114	1540	207	66	14.2	31.9	16.7
1024000	179906	24141	5792	15463	1505	533	11.6	16.0	10.9

Table 5.1: Runtimes [μs] of *PRNG* and *QRNG* for Random Number Generation using 10 steps and variable number of paths executed on **CPU3** and **GPU1**. The last three columns present a speedup of *CUDA* over *Multicore* execution for Q^{100}, Q^{10}, and P, respectively.

see that generating 100 instead of just 10 severely degrades the overall performance, especially for large `pathCount` such as 102 400 (7.4× speedup on *CUDA*) or 1 024 000 (10.3× speedup on *CUDA*). Furthermore, same behaviour is observed on *Multicore* and *CUDA* runs, but on a *GPU* it is much more evident, because it is always faster to draw an exact number of required Sobol numbers (Q^{10} for 10 steps case).

In the risk workflow, *RN*s, both in their pure, but also in a processed form, for instance, as normally distributed *RN*s in path generation or Stock prices `S` in *MS* generation, are the primary data kept in the memory. In general, on *CPU*s in comparison to *GPU*s we have more RAM memory at our disposal to store many *RN*s for reuse, which definitely helps when executing larger cases. On *GPU*s, we have to be more prudent and manage how many *RN*s we store at any given time, otherwise we risk a memory overflow. This is why, code versions that generate *RN* matrices in one step and put them in memory for further reuse, are preferred on *CPU*s, while drawing *RN*s on demand, usually next to the code, that consequently processes them, is a preferred approach on *GPU*s. In other words, ratio of instructions per *RN* matrix to the memory transfer time of the matrix needs to be low on a *GPU*.

In case of performance comparison of *CUDA* and *Multicore* runs, we note that *CUDA* consistently outperforms *Multicore* on all benchmarks, both all *RNG*s, `pathCount`s, and `stepCount`s. We note the highest observed speedups of 32× on 102 400 paths and 10 steps for Q^{10} (Table 5.1) and 22.3× on 10 240 paths and 10 steps for Q^{100} (Table 5.2). Moreover, *CUDA* speedups are on average larger for *QRNG*s (19.1× for Q^{10}, 12.6× for Q^{100}, and 12.9× for Q^{1000}) than for *PRNG* (7.4×). *QRNG*s seem to benefit more from larger parallelism on a *GPU*.

Figure 5.3 and Table 5.2 shows runtime results for different executions of *RN* generation, when the requested `stepCount` is equal to 100. When we increase the number of `stepCount`, we continue to observe the same correlations. First of all, we are unable to generate 1024000 × 1000 matrix on **GPU1**, with our current code implementation, because it means we are out of available memory. **GPU1** has 16 GB of global memory, and this matrix takes 1024000 × 1000 × $8B$, that is 8 GB. When we try to trim it, we copy it to a new matrix of the same size and effectively run out of memory. This exemplifies an issue

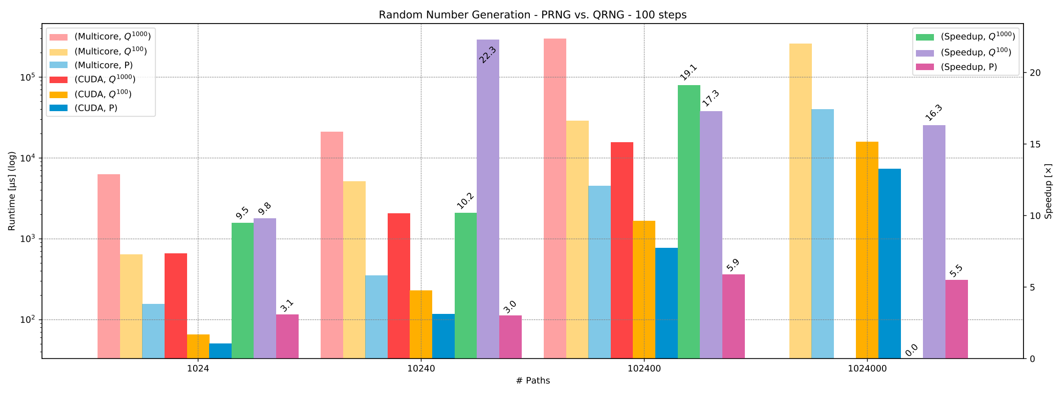

Figure 5.3: Runtimes [μs] and speedups of *PRNG* and *QRNG* for Random Number Generation using 100 steps and variable number of paths executed on **CPU3** and **GPU1**. The last three bars in each # Paths group represent a speedup of *CUDA* over *Multicore* execution for Q^{1000}, Q^{100}, and P, respectively.

Runtime [μs]	*Multicore*			*CUDA*			$\Delta(\times)$		
RNG	Q^{1000}	Q^{100}	P	Q^{1000}	Q^{100}	P	Q^{1000}	Q^{100}	P
# Paths									
1024	6294	644	156	662	66	51	9.5	9.8	3.1
10240	21100	5159	353	2066	231	118	10.2	22.3	3.0
102400	297785	28717	4540	15631	1663	771	19.1	17.3	5.9
1024000	2181173	260589	40289	0	15968	7329	0.0	16.3	5.5

Table 5.2: Runtimes [μs] of *PRNG* and *QRNG* for Random Number Generation using 100 steps and variable number of paths executed on **CPU3** and **GPU1**. The last three columns present a speedup of *CUDA* over *Multicore* execution for Q^{1000}, Q^{100}, and P, respectively. We cannot execute Q^{1000} case for 1 024 000 paths on **GPU1**, because we run out of device memory.

Version	*Multicore*			*CUDA*		
$\Delta(\times)$	Q^{10}/P	Q^{100}/P	Q^{1000}/P	Q^{10}/P	Q^{100}/P	Q^{1000}/P
# Paths						
1024	6.6	4.1	40.3	1.9	1.3	13.0
10240	3.5	14.6	59.8	2.9	2.0	17.5
102400	5.9	6.3	65.6	3.1	2.2	20.3
1024000	4.2	6.5	54.1	2.8	2.2	0.0

Table 5.3: Speedups of *PRNG* over *QRNG*s for Random Number Generation using 10 and 100 steps and variable number of paths executed on **CPU3** and **GPU1**. We cannot execute Q^{1000} case for 1 024 000 paths on **GPU1**, because we run out of device memory.

of keeping *RN*s in memory all at one time.

Table 5.3 shows a comparison of speedups of *PRNG* over 3 *QRNG* variants, Q^{10} with `D = 10`, Q^{100} with `D = 100`, and Q^{1000} with `D = 1000`, that determine available `stepCount`. Q^{10} is used to get 10 steps for each path, while Q^{100} and Q^{1000} are used to get 100 steps. They are matched with P on 10 steps and 100 steps, respectively. In general, *QRNG*s with less direction vectors `D` are faster, but they can never match *PRNG* on the same number of `pathCount`. Even executed in parallel per path and on a *GPU* they are still going to be slower to generate and take more space in memory, because the generation procedure is more sophisticated and they depend on sequences of *RN*s generated already (sequential process).

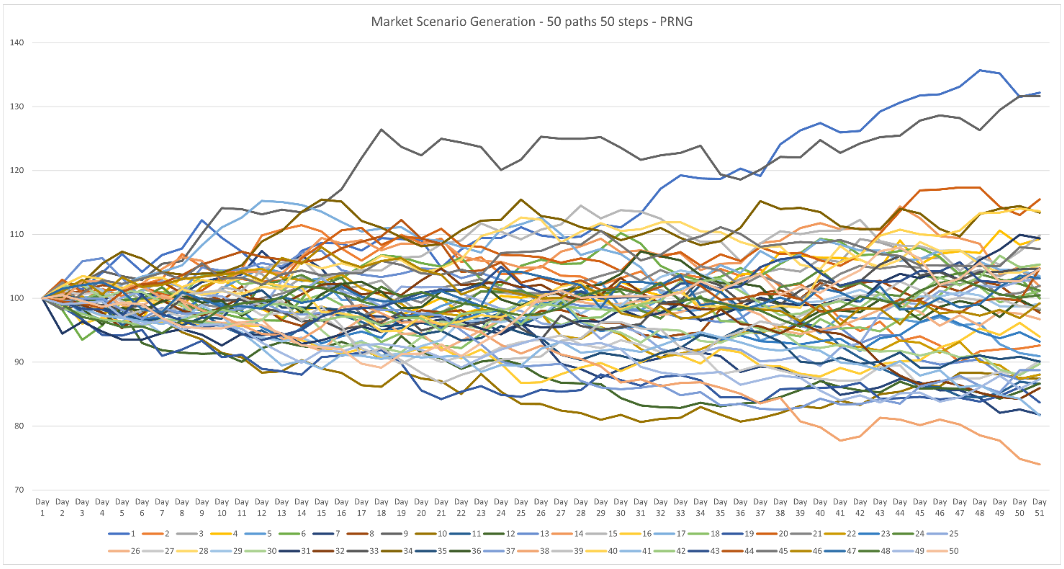

Figure 5.4: Stochastic simulation paths that are produced by external simulation for Market Scenario Generation using a *PRNG*. *RF* is spot price `S0`. 50 paths (different *MS*) and 50 steps (time horizon in days).

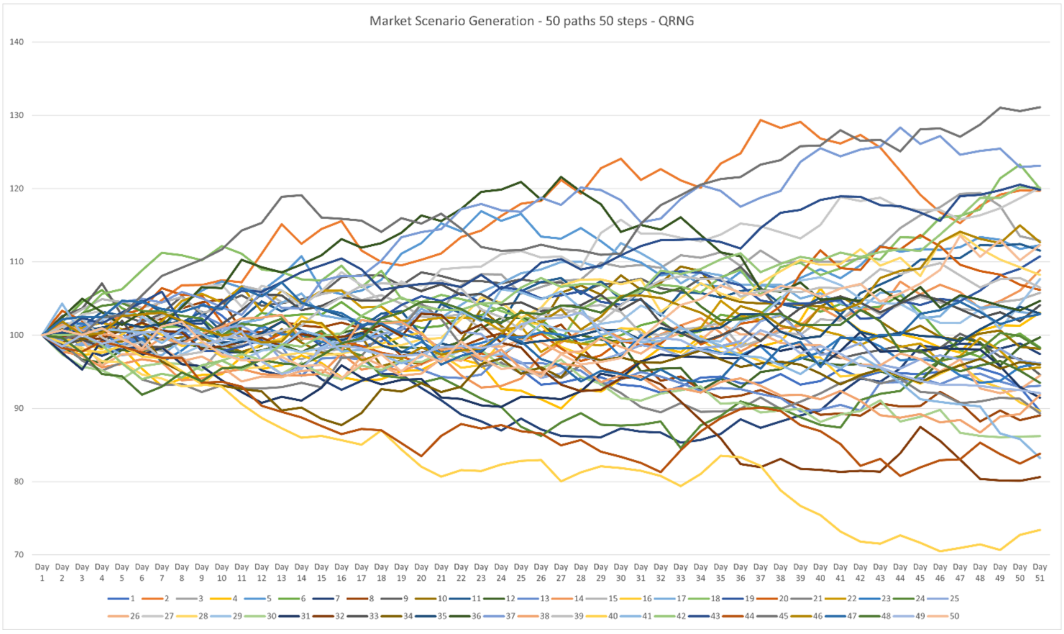

Figure 5.5: Stochastic simulation paths that are produced by external simulation for Market Scenario Generation using a *QRNG*. *RF* is spot price $S0$. 50 paths (different *MS*) and 50 steps (time horizon in days).

Figures 5.4 and 5.5 visualise the outcome of putting the *RN*s in practice. In particular, they are used here to obtain the results of an external simulation for market scenario generation, but can equally well be perceived as a result of any general simulation, for instance, for pricing. A visualisation of an internal simulation looks exactly the same, but paths and steps are different and they vary among products. Conceptually, each internal simulation starts in the last step of each external simulation path finished and generates a new set of internal simulation paths. The simple example compares performance of *PRNG* and *QRNG* on just 50 paths and 50 steps. However, it is sufficient to observe that *QRNG* is noticeably better than *PRNG* in filling the same sample space on each time step (day) in a uniform fashion.

Experiment Proposal to Validate *QRNG*s for Complete Risk Workflow

The results of our experiments conducted so far are inconclusive in terms of answering the question of validity and preference of using *QRNG*s over *PRNG*s. We have verified so far that we need to (i) be careful about the dimensionality `D` of Sobol number generators not to draw numbers that are redundant and instead match `D` exactly to out runtime needs as well as (ii) reduce the number of `pathCount` for *QRNG*s to match the runtime performance of *PRNG*s. On top of that, the result of our experiments agrees with the fundamental notions behind a difference between Quasi- and Pseudo-*RN*s, where former take longer to generate, but instead cover the distribution space in a more uniform manner. As a result, *QRNG*s are solely used in practice to *reduce the number of samples* that are required to approximate the same distribution *PRNG*s. To motivate the use of *QRNG*s, we need to first asses how many Quasi-*RN*s we need to math the approximation quality of an arbitrary of Pseudo-*RN* and build a mapping between them. In our case, as we cannot change the dimensionality of the required *QRNG*, because of the application requirements (`stepCount` is fixed), we can only focus on reducing `pathCount` that we use. In other words, the paths are our *samples* for reduction. We propose an experiment to be conducted in future to verify the theory behind the *QRNG*'s $O(\mathtt{N}^{-1})$ convergence rate and build an automatic mapping function that returns a minimal required number of Quasi-*RN*s to match the approximation quality of a standard *PRNG* for given number of N. The result would give us insights into the effective performance of *QRNG*s in comparison to *PRNG*s from not only a parallelisation angle, but also algorithmic efficiency. An experiment would involve running *QRNG*s and *PRNG*s side by side with different seeds on a series of increasing `pathCount` and measuring the standard deviation of the resultant approximate variable, for instance, a price, when the *RN*s would be feeded to a pricing function. So far, we have not assessed if using *QRNG*s with their specific requirements are a better fit for our risk workloads.

Experiment Proposal to Reduce the Number of Generated Quasi-*RN*s

Another experiment that we would plan to conduct is to base the sample generation on a single dimensional vector instead of generating `stepCount` independent vectors for each step. This would severely reduce the number of Sobol *RN*s kept in the memory in one time. To fill up the required *RN* matrix space of `pathCount` × `stepCount`, we would use shuffling techniques based on *PRNG* to rearrange the generated vector in random permutations. The expressibility of *QRNG*s is expected not to be reduced for practical use cases like ours. They would not be directly expressed in the memory, but rather drawn on demand. The current code would need to be adapted for such operation, because in this approach the independent paths cannot be generated in parallel across threads any more as the propagation is performed across the steps dimension.

5.5.3 Equity Option Portfolio Generation

We start our complete risk workflow with generation of the equity portfolio. The portfolio, which we analyse, is generated during runtime and represents a possible distribution of derivatives in a realistic portfolio. This is not a compute-intensive step, as the results in Table 5.4 demonstrate, because the portfolio sizes in our experiments do not allow for a lot of parallelism in this part. In general, we observe that the more portfolio holdings (`pfHCount` or # PfH) we generate, the more speedup we get on *CUDA* execution time, with the largest observed being $6\times$ for 1000 portfolio holdings. Because all options in the portfolio, no matter their type or style, are dependent on changes over time in a single underlying *RF* and all remaining model parameters are constant (they do not change over time), the number of variables that define every portfolio holding is the same and is equal in our implementation to 11 different variables in total. This changes when the generated portfolio is more complex and each of the holdings depends on a different number of parameters. In our experiments, we use only *PRNG* as a source of *RN*s from which we generate parameters to define portfolio holdings. We simulate three different portfolio holding distributions: a portfolio that consists of only European options, that are priced with the *BSMC* algorithm, a portfolio that consists of only American options, which are priced with the **LSMC** algorithm, and mixed random portfolio that comprise both mentioned option styles. The holding distribution in the mixed portfolio follows uniform distribution. The generated options in portfolios have, among others, different maturities, which lead to different number of time steps, that are used in pricing. The differences in maturities between portfolio holdings lead to workload divergence if the holdings are priced in parallel.

Runtime [μs]	*Multicore*			*CUDA*			$\Delta(\times)$		
Pf Dist.	Eu	Rand	Us	Eu	Rand	Us	Eu	Rand	Us
# PfH									
5	12	28	2	31	32	32	0.4	0.9	0.1
25	22	28	19	33	34	34	0.7	0.8	0.6
50	25	24	22	34	34	34	0.7	0.7	0.6
100	35	40	34	40	34	33	0.9	1.2	1.0
500	136	145	127	35	38	35	3.9	3.8	3.6
1000	203	206	205	36	35	34	5.6	5.9	6.0

Table 5.4: Runtimes [μs] and speedups of Equity Option Portfolio Generation for all portfolio holdings and distributions used in our experiments executed on **CPU3** and **GPU1**.

5.5.4 Outer Parallelism Level: Market Scenario Generation

In our experiments, we use only *PRNG* as a source of *RN*s, which we process further to turn them into *RF* values for *MS*s. For all our experiments that generate Market Scenarios, we use a term unit that corresponds to 1 business week or 5 business days. We fix time horizon (number of external steps - # Ext Steps) to 10 that corresponds to 50 business days. The choice of different numbers of *MS*s (# Ext Paths) for the benchmark setup is motivated by the domain knowledge that. We try to imitate a workload for different types of investment manager use cases A small asset manager corresponds to a workload case with a range between 1000 and 6000 *RF*s (that map 1:1 to portfolio holdings in our setup based on our assumptions) and between 1000 and 10 000 *MS*s. A large asset manager corresponds to a workload case with $>$ 20000 *RF*s and $>$ 100000 *MS*s. As we see in the next experiments due to a limited device memory available, we are restricted to pricing one *MS* at a time, that is, one external path is used at any time. We consider it a realistic assumption, because in a practical scenario with many *RF*s impacting the value of the portfolio holding, the memory footprint of one *MS*s is in any case prohibitively large for parallel pricing on one *GPU* device. However, solely for *MS* generation, which precedes the portfolio pricing part, the parallel execution can be treated as **`segmap`** with a **`reduce`** on the innermost level because its both dimensions (# Ext Paths $\times$ # Ext Steps) are regular for all # PfH portfolio holdings. **`reduce`** operation across # Ext Steps is small (10 steps) in comparison to # Ext Paths for all cases.

We focus on comparison of the speedups between *Multicore* and *CUDA* runs as presented in Figure 5.6 and Table 5.7. Table 5.5 presents the runtimes of *Multicore* runs, while Table 5.6 – the runtimes of *CUDA* runs. From the speedup results, we can clearly deduce that the increased parallelism in **`map`** operation across the level of *MS*s benefits from the *GPU* architecture the most when the portfolio is small (# PfH = 1, 10, 100), with the largest 56.4$\times$ speedup on a single instrument portfolio. For larger portfolio with 1000, 6000, or 20 000 holdings, the difference diminishes to 17.6$\times$ on average for 100 000 *MS*s, and is stable with increasing # Ext Paths (number of *MS*s). It means that the nested parallelism on the inner **`map`** *MS* level does not fully exhaust the *GPU* device capabilities, and as such is a candidate for further tuning. Finally, we are unable to check the behaviour on all large asset manger cases, because we run out of device memory on **GPU1**.

5.5.5 Inner (Nested) Parallelism Level: Derivative Portfolio Pricing

We do not measure Portfolio Pricing in isolation. Portfolio Generation and Market Scenario Generation timings are included in these measurements, because they are required in the process. However, in our experiments we observe that the portfolio pricing takes significantly more time than any other part of the complete risk workflow. These are the largest cases that we mange to benchmark across all three differentiate As mentioned in Section 5.5.3, we consider three different portfolio distributions in our experiments. We do

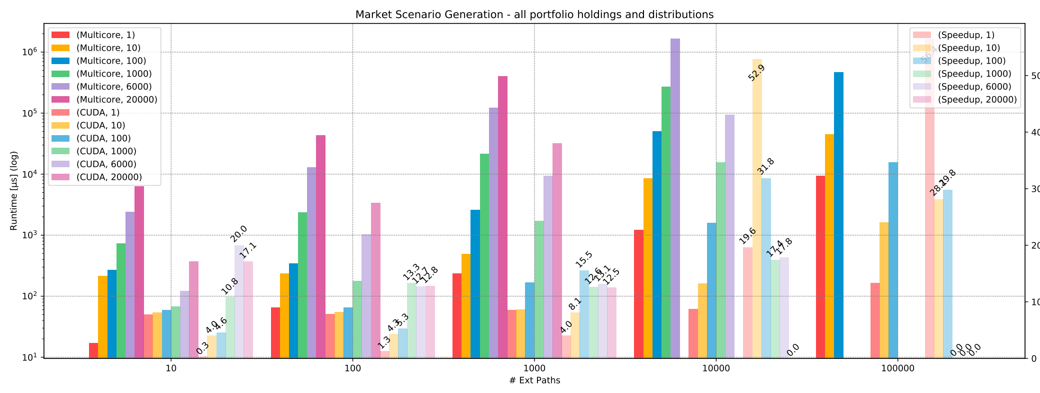

Figure 5.6: Runtimes [μs] and speedups of Market Scenario Generation using 10 steps for a 10 day horizon and variable number of external paths for *MS*s executed on **CPU3** and **GPU1**. *PRNG* is used for *RN* generation and the portfolio distribution comprise both European and American Equity options. The last three bars in each # Ext Paths group represent a speedup of *CUDA* over *Multicore* execution.

# PfH # Ext Paths	1	10	100	1000	6000	20000
10	17	214	271	737	2419	6338
100	65	237	344	2355	13074	43185
1000	237	495	2585	21507	122978	405393
10000	1218	8523	50238	272630	1666341	0
100000	9361	45149	470151	0	0	0

Table 5.5: Runtimes [μs] of Market Scenario Generation of *Multicore* runs on **CPU3** for all *MS* numbers and portfolio holding numbers.

# PfH # Ext Paths	1	10	100	1000	6000	20000
10	50	54	59	68	121	371
100	51	55	65	177	1028	3384
1000	60	61	167	1704	9356	32374
10000	62	161	1580	15626	93386	0
100000	166	1609	15753	0	0	0

Table 5.6: Runtimes [μs] of Market Scenario Generation of *CUDA* runs on **GPU1** for all *MS* numbers and portfolio holding numbers.

# PfH # Ext Paths	1	10	100	1000	6000	20000
10	0.3	4.0	4.6	10.8	20.0	17.1
100	1.3	4.3	5.3	13.3	12.7	12.8
1000	4.0	8.1	15.5	12.6	13.1	12.5
10000	19.6	52.9	31.8	17.4	17.8	0.0
100000	56.4	28.1	29.8	0.0	0.0	0.0

Table 5.7: Speedups *CUDA* / *Multicore* for Market Scenario Generation for all *MS* numbers and portfolio holding numbers.

not manage to test all combinations of (# PfH, # Ext Paths, # Int Paths), on all three portfolio distributions, so we only report results on these, on which we managed to collect all three measurements.

European Options Portfolio Pricing

The results are collected in Figure 5.7. Runtimes are presented in Table 5.8, while speedups in Table 5.9. In case of the European option portfolio, we observe large speedups of *CUDA* version over *Multicore* of up to $44.5\times$ on a single portfolio holding, 10 000 *MS*s and 102 400 internal paths used in *BSMC* pricing algorithm, as well as $39.1\times$ on 10 portfolio holdings, 1000 *MS*s and 102 400 internal pricing paths. These are also the largest cases that we mange to benchmark across all three portfolio cases.

The general high performance in this European option case is mainly due to all three simulation nested levels being processed in parallel. Portfolio pricing runtime is in order of 100 ms for *CUDA* runs. It is possible, because, in contrast to American, European options (i) require only a forward propagation step in pricing (*BSMC* algorithm), which processes each path in isolation, so it can be easily mapped to *GPU* threads, as well as (ii) are expressed in memory in a form of a flat *FP64* array of # Int Paths size that consist of option payoffs at their maturity.

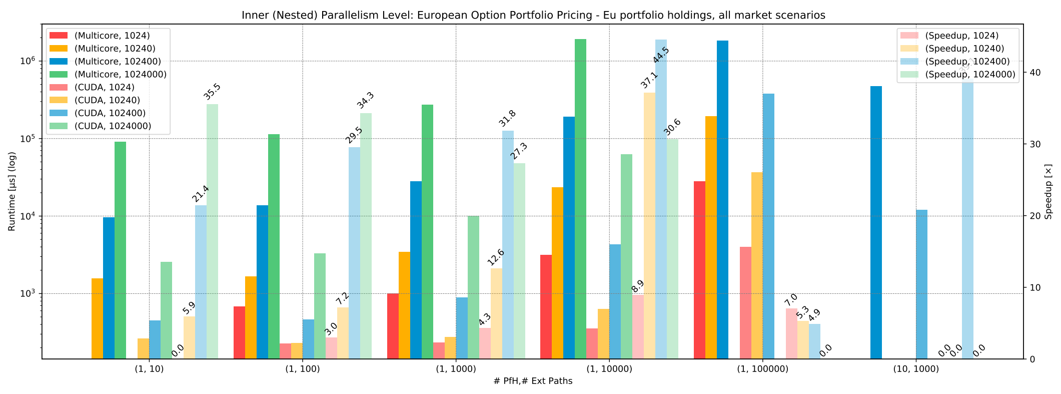

Figure 5.7: Runtimes [μs] and speedups of European Option Portfolio Pricing using *BSMC* algorithm. We use 10 steps for time horizon and variable number of external and internal paths for *MS*s executed on **CPU3** and **GPU1**. *PRNG* is used for *RN* generation. The last four bars in each # Ext Paths group represent a speedup of *CUDA* over *Multicore* execution.

Pricing Portfolio of American Options Portfolio Pricing

The results are collected in Figure 5.8. Runtimes are presented in Table 5.10, while speedups in Table 5.11. In case of the American option portfolio, we observe largest speedups of *CUDA* version over *Multicore* of up to $19.7\times$ on a single portfolio holding, only 10 *MS*s and significant 1 024 000 internal paths used in **LSMC** pricing algorithm, but observe only a modest $4.2\times$ speedup on a case with 10 portfolio holdings, 1000 *MS*s and 102 400 internal pricing paths.

	Runtime [μs]	*Multicore*				*CUDA*			
	# Int Paths	1024	10240	102400	1024000	1024	10240	102400	1024000
# PfH	# Ext Paths								
1	10	0	1567	9632	90973	0	264	450	2562
	100	678	1660	13645	113100	225	231	462	3293
	1000	1008	3447	28153	272777	234	273	885	9975
	10000	3159	23587	191172	1912371	353	635	4296	62548
	100000	28122	193146	1835794	0	4010	36619	378129	0
10	1000	0	0	470469	0	0	0	12038	0

Table 5.8: Runtimes [μs] of European Option Portfolio Pricing for all *MS* numbers and 2 different small portfolio holding numbers. *Multicore* on **CPU3** and *CUDA* on **GPU1** runs for different number of external and internal paths.

		$\Delta(\times)$			
	# Int Paths	1024	10240	102400	1024000
# PfH	# Ext Paths				
1	10	0.0	5.9	21.4	35.5
	100	3.0	7.2	29.5	34.3
	1000	4.3	12.6	31.8	27.3
	10000	8.9	37.1	44.5	30.6
	100000	7.0	5.3	4.9	0.0
10	1000	0.0	0.0	39.1	0.0

Table 5.9: Speedups of European Option Portfolio Pricing for all *MS* numbers and 2 different small portfolio holding numbers. *CUDA* / *Multicore* for different number of external and internal paths.

In general, in American option portfolio case the speedups are on average $3\times$ smaller than for European options. Here, only the innermost parallel level of # Int Paths is extracted. With current implementation, it is impossible to price on higher level due to memory requirements of **LSMC** algorithm that efficiently maps to all available parallelism on the *GPU*. is mainly due to all three simulation nested levels being processed in parallel. This time, portfolio pricing is still in order of hundreds of ms for *CUDA* runs, but *Multicore* version is significantly faster on same cases. Firstly, American option pricing require both a parallel forward, but also sequential backward propagation step in pricing (**LSMC** algorithm), which still processes each path in isolation, at least in forward step. Secondly, in contrast to European options each computation is are expressed in memory in a form of a two-dimensional *FP64* array of $\#IntPaths \times \#IntSteps$, because this time we need to keep the whole history through the option lifetime for the backward step.

Random (Mixed) Options Portfolio Pricing

The results are collected in Figure 5.9. Runtimes are presented in Table 5.12, while speedups in Table 5.13. In case of the Random option portfolio, we observe largest speedups of *CUDA* version over *Multicore* of up to $19.4\times$ on a single portfolio holding, only 10 *MS*s and significant $1\,024\,000$ internal paths used in **LSMC** pricing algorithm, but observe only a modest

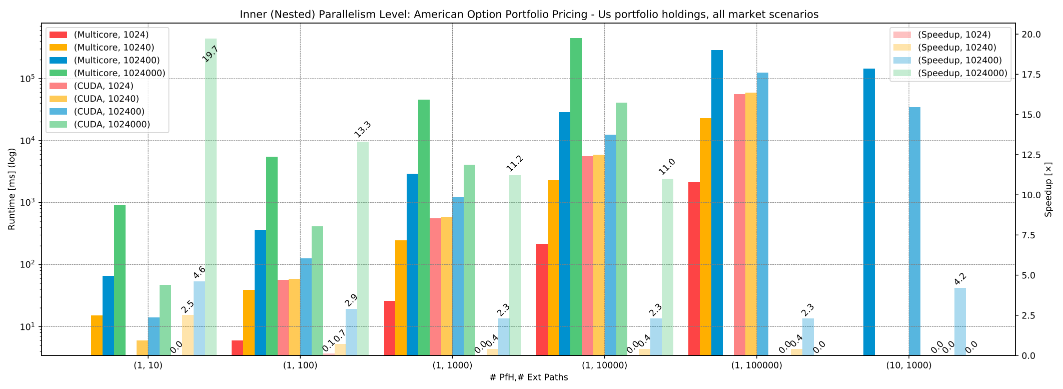

Figure 5.8: Runtimes [μs] and speedups of American Option Portfolio Pricing using **LSMC** algorithm. We use 10 steps for time horizon and variable number of external and internal paths for *MS*s executed on **CPU3** and **GPU1**. *PRNG* is used for *RN* generation. The last four bars in each # Ext Paths group represent a speedup of *CUDA* over *Multicore* execution.

	Runtime [ms]	*Multicore*				*CUDA*			
	# Int Paths	1024	10240	102400	1024000	1024	10240	102400	1024000
# PfH	# Ext Paths								
1	10	0	15	65	925	0	6	14	47
	100	6	39	362	5454	56	59	125	411
	1000	26	244	2916	45659	558	590	1242	4062
	10000	217	2298	28502	447222	5581	5870	12390	40609
	100000	2133	23042	288019	0	55457	58538	124467	0
10	1000	0	0	145305	0	0	0	34329	0

Table 5.10: Runtimes [ms] for American Option Portfolio Pricing for all *MS* numbers and 2 different small portfolio holding numbers. *Multicore* on **CPU3** and *CUDA* on **GPU1** runs for different number of external and internal paths.

		$\Delta(\times)$			
	# Int Paths	1024	10240	102400	1024000
# PfH	# Ext Paths				
1	10	0.0	2.5	4.6	19.7
	100	0.1	0.7	2.9	13.3
	1000	0.0	0.4	2.3	11.2
	10000	0.0	0.4	2.3	11.0
	100000	0.0	0.4	2.3	0.0
10	1000	0.0	0.0	4.2	0.0

Table 5.11: Speedups of American Option Portfolio Pricing for all *MS* numbers and 2 different small portfolio holding numbers. *CUDA* / *Multicore* for different number of external and internal paths.

3.8× speedup on a case with 10 portfolio holdings, 1000 *MS*s and 102 400 internal pricing paths. In general, these results follow the pattern of the American option portfolio results. The reason is that the single portfolio holding consists of an American option. In case of 10 portfolio holdings, the bottleneck is still on pricing American option, as **LSMC** algorithm is significantly more complex than *BSMC* algorithm. In conclusion, the number of American options in Random portfolio determines the overall workflow performance.

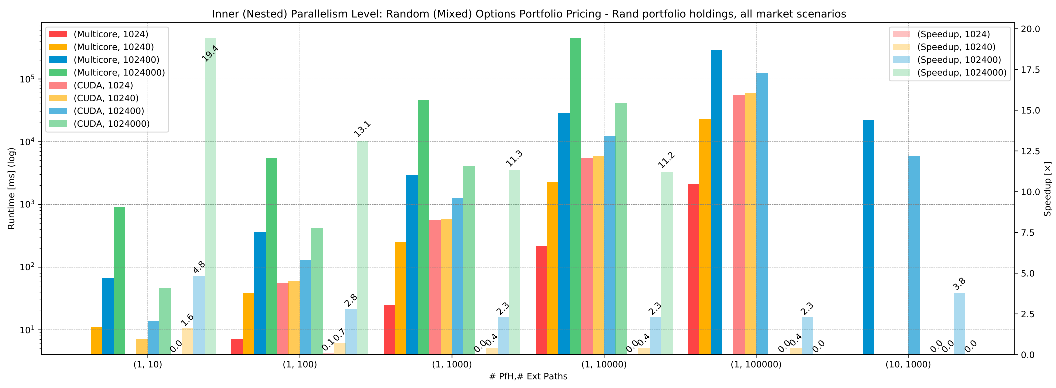

Figure 5.9: Runtimes [μs] and speedups of Random (Mixed) Options Portfolio Pricing using *BSMC* and **LSMC** algorithms. We use 10 steps for time horizon and variable number of external and internal paths for *MS*s executed on **CPU3** and **GPU1**. *PRNG* is used for *RN* generation. The last four bars in each # Ext Paths group represent a speedup of *CUDA* over *Multicore* execution.

	Runtime [ms]	*Multicore*				*CUDA*			
	# Int Paths	1024	10240	102400	1024000	1024	10240	102400	1024000
# PfH	# Ext Paths								
1	10	0	11	67	914	0	7	14	47
	100	7	39	364	5400	56	59	128	412
	1000	25	248	2923	45890	561	583	1244	4070
	10000	215	2305	28547	454042	5564	5845	12380	40612
	100000	2123	22731	284797	0	55683	58717	124399	0
10	1000	0	0	22278	0	0	0	5921	0

Table 5.12: Runtimes [ms] for Random (Mixed) Options Portfolio Pricing for all *MS* numbers and 2 different small portfolio holding numbers. *Multicore* on **CPU3** and *CUDA* on **GPU1** runs for different number of external and internal paths.

5.5.6 Complete Risk Workflow using PRNG

For the measurements of Complete Risk Workflow, we select a Random Options Portfolio from the last section. The results are collected in Figure 5.10. Runtimes are presented in Table 5.14, while speedups in Table 5.15. In case of the Random option portfolio, we again observe largest speedups of *CUDA* version over *Multicore* of up to 18.7× on a single portfolio holding, only 10 *MS*s and significant 1 024 000 internal paths used in **LSMC** pricing algorithm, but observe only a modest 3.7× speedup on a case with 10 portfolio holdings, 1000 *MS*s and 102 400 internal pricing paths. Both runtime and speedup results follow the

		$\Delta(\times)$			
	# Int Paths	1024	10240	102400	1024000
# PfH	# Ext Paths				
1	10	0.0	1.6	4.8	19.4
	100	0.1	0.7	2.8	13.1
	1000	0.0	0.4	2.3	11.3
	10000	0.0	0.4	2.3	11.2
	100000	0.0	0.4	2.3	0.0
10	1000	0.0	0.0	3.8	0.0

Table 5.13: Speedups of Random (Mixed) Options Portfolio Pricing for all *MS* numbers and 2 different small portfolio holding numbers. *CUDA* / *Multicore* for different number of external and internal paths.

pricing results from the previous section. This means that most of the time is spent in portfolio repricing part. In fact, when compared with the second most expensive part based on the collected runtimes in Table 5.6, which is *MS* generation, the portfolio repricing in all *MS*s takes up to 30× longer on *CPU* and 747× longer on *GPU* than that part. This fact leads us to conclusion that the most important optimisation that we can introduce at the current stage is to reduce the memory footprint of the internal simulation, so more *MS*s can be priced in parallel as it is a case for European Option Portfolio. We propose to

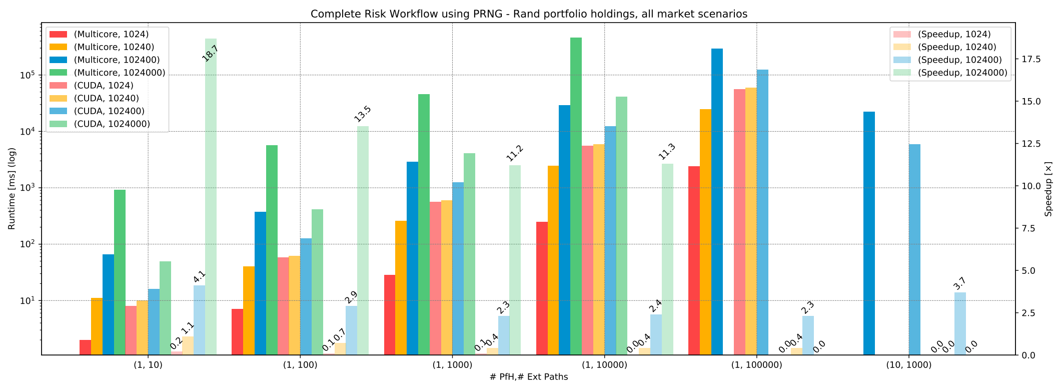

Figure 5.10: Comparison of measurements for Complete Risk Workflow that uses Random (Mixed) Options Portfolio Pricing using *BSMC* and **LSMC** algorithms. We use 10 steps for time horizon and variable number of external and internal paths for *MS*s executed on **CPU3** and **GPU1**. *PRNG* is used for *RN* generation. The last four bars in each # Ext Paths group represent a speedup of *CUDA* over *Multicore* execution.

GPU Memory Requirements

For Complete Risk Workflow, in addition to the runtimes, we also measure the maximum memory footprint that is recorded during its execution on a *GPU*. The results are presented in Table 5.16. In case of the Random option portfolio, we observe the largest memory requirement for *CUDA* version for cases with the largest number of # Int Paths, that is, 1 024 000, that is equal to around 668 MB for a single American Option. The number of

	Runtime [ms]	*Multicore*				*CUDA*			
	# Int Paths	1024	10240	102400	1024000	1024	10240	102400	1024000
# PfH	# Ext Paths								
1	10	2	11	65	917	8	10	16	49
	100	7	40	368	5623	58	61	127	415
	1000	28	258	2898	45353	558	592	1241	4061
	10000	246	2445	29182	457097	5571	5884	12375	40573
	100000	2387	24488	289275	0	55497	58997	123872	0
10	1000	0	0	21998	0	0	0	5902	0

Table 5.14: Runtimes [ms] for Complete Risk Workflow that uses Random (Mixed) Options Portfolio Pricing and *PRNG* for all *MS* numbers and 2 different small portfolio holding numbers. *Multicore* on **CPU3** and *CUDA* on **GPU1** runs for different number of internal paths.

	# Int Paths	1024	10240	102400	1024000
# PfH	# Ext Paths				
1	10	0.2	1.1	4.1	18.7
	100	0.1	0.7	2.9	13.5
	1000	0.1	0.4	2.3	11.2
	10000	0.0	0.4	2.4	11.3
	100000	0.0	0.4	2.3	0.0
10	1000	0.0	0.0	3.7	0.0

Table 5.15: Speedups of Complete Risk Workflow that uses Random (Mixed) Options Portfolio Pricing and *PRNG* for all *MS* numbers and 2 different small portfolio holding numbers. *CUDA* / *Multicore* for different number of internal paths.

portfolio holdings and the portfolio distribution make a difference here when we deal with more holdings. As we can see for a case with 10 portfolio holdings, 1000 *MS*s and 102 400 internal pricing paths, we record a smaller memory footprint of 205MB, although there are 10 different options considered.

	Device Memory [KB]	*CUDA*			
	# Int Paths	1024	10240	102400	1024000
# PfH	# Ext Paths				
1	10	1737	11197	68556	684156
	100	1739	11199	68558	684158
	1000	1760	11220	68579	684179
	10000	2710	11431	68790	684390
	100000	25550	25658	70899	0
10	1000	0	0	210298	0

Table 5.16: Device memory requirements for Complete Risk Workflow that uses Random (Mixed) Options Portfolio Pricing, *PRNG* for *RN*s, and *BSMC* and **LSMC** algorithms for all *MS* numbers and 2 different small portfolio holding numbers. *Multicore* on **CPU3** and *CUDA* on **GPU1** runs for different number of internal paths. We use 10 steps for time horizon and variable number of external and internal paths.

5.5.7 Complete Risk Workflow with BSMC and QRNG

The execution times of the whole algorithm are gathered in Table 5.17. We can observe significant speedups as we increase the number of *MS*s. This increases the parallelism on the outer level. We test pricing on the increasing number of simulation paths, which exposes more parallelism on the inner levels. For larger combinations of paths and steps, we run out of device memory, because we keep all the generated Sobol *RN*s at all times. We need to investigate the approach where we recompute the Sobol numbers when necessary. It can yield better performance and allows to approximate the risk with a larger number of paths and steps. We also consider running these benchmark cases on *Multicore* and *CUDA* backends.

# Ext Paths	# Int Paths	*Seq*	*CUDA*	$\Delta(\times)$
100	100	120	13	9,23
100	1000	1171	53	22,09
100	10000	11705	398	29,41
100	20000	23406	788	29,70
100	30000	35173	1182	29,76
100	40000	47067	1570	29,98
1000	100	734	25	29,36
1000	1000	6872	83	82,80
1000	10000	68151	633	107,66
1000	20000	136100	1242	109,58
1000	30000	204070	1833	111,33
10000	100	6864	126	54,48
10000	1000	63702	360	176,95
10000	10000	634892	2730	232,56
10000	20000	1282457	5383	238,24
10000	30000	1905676	8035	237,17

Table 5.17: Comparison of runtimes between sequential **Seq** and parallel *CUDA* versions using *QRNG* (*Sobol RN*s). Portfolio comprise only European options priced with *BSMC* algorithm. The runtime consists of the whole algorithm execution. The runtimes are given in milliseconds. Single-precision floating-point numbers are used. Portfolio consists of 100 positions. The path-dependent version of *BSMC* is used. The number of steps for *BSMC* pricing is fixed to 50 steps.

5.6 Related Work

We identify two distinct lines of research work that we consider comparable to our efforts in terms of the addressed risk problem and use of *GPU*s for accelerated implementation.

Dixon et al. [DCK09; DCK12; Dix+12] implement two accelerated versions of a Delta-Gamma approximation method to calculate *VaR*. The method that they describe differs from a *Monte Carlo* method that we apply mainly in how the loss function is evaluated. They apply the partial revaluation approach mentioned in Section 5.2 instead of a full revaluation approach like we do. The *RF*s are mathematically reduced to a few of them that represent most of the loss function distribution. Delta-Gamma approach uses the “Greeks”, in particular Δ, the first moment and Γ, the second distribution moments of the time series that represent the *RF*s log returns. This technique applies a Taylor expansion to approximate the price changes over the course of time. This method works well for the vanilla derivative instruments as they can be approximated by a quadratic function. However, for all other highly non-linear, exotic derivative instruments (such as bonds with call/put options, swaptions, credit derivatives, etc.) and instruments that are highly volatile (or where volatility impacts are not linear) and illiquid, full revaluation method is recommended as the approximation methods yield less accurate values. They achieve $127\times$ speedup with their parallel *CPU* implementation and $538\times$ with their *GPU* implementation. They compare the speedups against naive baseline implementations on *CPU* and *GPU*, respectively. They do so to emphasise the impact of the 3 types of optimisations that they apply. Firstly, they propose to reformulate the loss function approximations with a set of linear algebra transformations. A matrix-matrix multiplication is replaced with matrix-vector multiplication to reduce the amount of necessary computation. This approach is specific to the approximation that they use, so we cannot consider such a change to the algorithm based on the underlying mathematics. Secondly, they make use of Sobol sequences to generate uniformly-distributed *RN* to enable faster numerical convergence for the loss function approximation. As described in Section 5.4.2, we agree with the superiority of quasi-*RN*. However, in the current implementation of *Futhark* library, we identify a problem with using a fixed number of dimensions that needs to be known upfront at the compile time. The number of dimensions is equal to the number of time steps in the simulations. This fact limits the application functionality, because the parameter is not specified in the input to the application. Then they use Box-Muller sampling method to transform the uniform *RN*s to standard Gaussian (normally-distributed) *RN*s. They claim that the standard error converges twice as fast for single-precision floating-point numbers in comparison to the Beasley, Springer, and Moro Inverse Normal *CDF* approximation that we choose. However, in our experiments our focus is on the *FP64* floats. Thirdly, they conclude that merging data-parallel kernels is beneficial for the overall performance. They do it manually as they use the low-level *C++*, *CUDA* and *cuBLAS* library in their implementation. This step is automated for us as *Futhark* kernel fuses the *SOAC*s that implement the kernels in an efficient manner to reduce redundant memory operations.

Varela et al. [Var+15b; Var+15a; VW17; Var+17; VW18] implement a full Nested Simulation for **MCVaR**. Just like us, they consider the estimation of *VaR* and component *VaR* using nested *Monte Carlo* simulations. They use *OpenCL* for portability and investigate the performance of their implementation across *CPU*, *GPU* and *FPGA* architectures. They also scale the size of their simulations to evaluate the runtime variations of different parallelisation techniques. The main algorithmic optimisation that they propose is to generate a set of

*RN*s once in a separate step and then reuse them for external simulations. The set of *RN*s is then reused across simulations for *MS* generation, that is, for each type of the instrument we use the same set of *RN*s. They apply the same optimisation step for internal simulation to price the derivative instruments in the portfolio, but here a separate set of *RN*s is necessary for each individual *MS*. We adapt the same optimisation in our work. We also price the different types of instruments (European, American, etc.) separately to minimise the impact of work divergence across different pricing models. In [Var+17], they present their implementation in Python, a general-purpose high-level programming language that is extensively used in the financial industry today. It is well suited for rapid prototyping and has a rich library ecosystem, but at the same time lacks the runtime performance. They compensate this by utilising *PyOpenCL* library to enable high-performance computations and target 4 different heterogeneous platforms. A *Futhark* program likewise can be translated to Python code that uses *PyOpenCL* to invoke *OpenCL* kernels.

5.7 Conclusion

The **MCVaR** algorithm is based on a generation of hundreds of thousands of *MS*s to simulate the future portfolio changes, which are necessary to build an accurately the *P/L* distribution of a portfolio. The distribution gives a solid foundation for calculation of risk measures like *VaR* or *ES* that are obtained from combinations and aggregations of the portfolio prices in the simulated *MS*s. In our case, *Monte Carlo* simulations are used to both generate the *MS*s and to price the derivative portfolio. In particular, as we deal with a portfolio of complex exotic derivatives, we end up with a nested simulation problem, as revaluation of the portfolio involves itself a series of simulations. In other words, for each scenario, the portfolio needs to be fully revaluated using simulated input parameters. Clearly, this nesting significantly impacts the device memory requirements.

We conclude that the high-level parallel structure of the code is determined by the complexity of the instruments that the portfolio consists of. It is not possible to revaluate instruments with the algorithms that have vastly different computational requirements. We are thus forced to group the instruments based on the computational intensity of their pricing algorithm, identify parts that have to be executed in a sequential fashion and execute them in common batches.

We discover that the significant memory requirements of a **MCVaR** algorithm, even in a standard case, still exceed the memory capabilities of the modern powerful *GPU*s. The high-level implementations need to be thus amended with guideline expressions that instruct the compiler to use specific optimisations or abandon parallel execution completely in certain parts of code. Compiler is unable to deduce an efficient if the algorithm itself is not optimally handling the large number of intermediate data that it produces to compute the final risk measure results.

Chapter 6

Conclusion

6.1 Evaluation and Discussion

At it core, this thesis investigates the problem of providing an efficient acceleration to a set of different derivative pricing and risk management applications that are commonly used in investment management practice. We mainly target the *GPU* platform and use *Futhark* and *CUDA* programming technologies and focus on extracting parallelism from the chosen algorithms. With this goal in mind, we conclude by summarising the general observations and outcomes and taking a general look ath impact of our investigation results. We implement two derivative pricing models, one for fixed-income (**HW1F** in Chapter 3) and another for equity markets (**LSMC** in Chapter 4). To solve these models, we use two distinct numerical algorithms that we parallelise to a high degree. Subsequently, we attempt to combine them and build a risk measurement workflow that can be used to and identify the inherent portfolio risks (**MCVaR** in Chapter 5).

For **HW1F**, we identify that multiple versions of code are required to cover a whole spectrum of different input data parameters. The problem is characterised by a high variance in dimensions of the data, which leads to variable sizes of data structures that result in high thread divergence and irregular memory accesses. These challenges combined with the intrinsic nested parallelism on two levels of the numerical method make it non-trivial to map the method to a *GPU* architecture, which is based on regular thread blocks. In particular, in the first GPU-OUTER version, each thread works on a valuation of a single instrument in isolation. We show that this implementation can be refined by a set of generic optimisations. However, there still exist more parallelism available to exploit inside the valuation of the instrument. We demonstrate this in the second GPU-FLAT version, which instead assigns each thread to work with one tree node. This requires that we distribute the instruments among thread blocks in a way that minimises the number of idle threads at any given time. We use low-level *CUDA* programming framework and compare it with the low-level code that is auto-generated from high-level *Futhark* code. *CUDA* gives us the verbosity and the ability to fine-tune the relevant optimisations that help us achieve increased performance. *Futhark* compiler in the current version is not able to apply such optimisations automatically.

For **LSMC**, we deal with an algorithm, that consists of two parts. The first part is embarrassingly parallel, because it can be divided into chunks of simulation paths, that can be executed in isolation and mapped directly to threads. On the contrary, the second part is inherently a sequential problem. Extracting parallelism and mapping it efficiently to a parallel architecture involves non-trivial linear algebra transformations and fundamental changes to the underlying numerical algorithm. Our investigation exemplifies the fact that without thorough understanding of the algorithm and a combination of concepts from finance, mathematics, and computer science, we are not able to optimise numerical implementations through parallelisation, because the generic optimisations might not be suitable for the existing algorithmic structure. Therefore, to reduce the software development burden, and focus instead on exploring different algorithms, we choose to use a high-level functional approach for this implementation. We express the algorithm using data-parallel constructs and let the optimising compiler auto-generate an efficient parallel code that targets massively parallel *GPU* hardware. As a result, the performance matches the state-of-the-art *CUDA* implementation, which is hand-tuned by *NVIDIA* experts, on the majority of cases and manages to outperform the benchmark on particular small cases. This promising finding motivates further work on the optimisation compiler and the algorithm. Based on our work, we consider the high-level functional specification as being more suitable and accessible for the financial-domain experts than the low-level dedicated code, that is usually implemented and optimised by expert software developers. Its expressibility and modularity enables code maintainability, hides the implementation details that target particular parallel architecture, and turns our focus to algorithmic and programming-specific considerations. The high-level approach facilitates future algorithmic changes, which are common in constantly changing financial industry, for instance, it needs to be simple to adapt to a multitude of new and custom financial instruments introduced in the global markets.

In the last case **MCVaR** we demonstrate a potential of integrating the modules with fast pricing capabilities in a broader risk management system that is aimed at large and diverse derivative investment portfolios. We implement classical risk measure computations for a portfolio of exotic options using a *Monte Carlo* simulation, which is a method deemed to be overly compute-intensive for a practical use on complex derivative portfolios. Using *Monte Carlo* simulations in risk workflows leads to a large nested simulation with multiple levels of parallelism that cannot be simply mapped to *GPU* architecture to be executed in parallel. However, as we turn again to high-level programming, it is easy to modularise and adapt the code for fast computation. On top of that, we introduce algorithmic optimisations like grouping instruments of common computational complexity and reusing simulation paths where it is statistically valid to reduce the amount of data processed. Moreover, we introduce implementation optimisations like sequentialising the outer levels of parallelism, so we do not run out of limited *GPU* device memory, while still performing computation in parallel.

All these contributions, when combined together and backed by the experimental results, demonstrate that it is a worthwhile and viable commitment to accelerate different financial algorithms using modern parallel programming languages and frameworks to prepare them for execution on highly parallel architectures.

6.2 Future Work

During work on the project, we identified various opportunities and improvements that we consider valuable and worth further investigation.

For **HW1F**, one valuable improvement would be to build on top of the *multiversioning* approach and introduce an Inspector-Executor module that determines which kernel version is used based on the input dataset. Moreover, it would be valuable to check the impact of the applied optimisations on a bounded grid to solve the same pricing problem with a *PDE* approach and a *FDM* numerical method. Such an approach would be more generic and could be extended beyond finance to other scientific and engineering fields. In addition, we would need to specify a set of generic rewrite rules to integrate the proposed optimisation techniques in an aggressively-optimising compiler like the one for *Futhark*.

The advantages of **LSMC** approach would be even more vivid if we would test the performance on a derivative instrument dependent on many multiple risk factors. We would need to increase the necessary number of paths and hence more computation would be done per each time step. In such a multi-dimensional case, the *Monte Carlo* simulation would become even more compute-bound and hence even better suited for execution on massively parallel *GPU*s. Another option would be to use a more sophisticated pricing model like *Heston* model that introduces a second stochastic risk factor. A benchmark like STAC-A2 [STA20] from Securities Technology Analysis Center is a test benchmark that we could use to compare the achieved performance.

Besides the immediate potential algorithmic improvements to *Monte Carlo* simulations proposed in Section 5.5.2, there are at least three directions that we could consider in the future. The first improvement for **MCVaR** would be to adapt it fully for **HW1F** pricing and trinomial trees, because current *Futhark* implementation is not using all the introduced optimisation and thus is significantly slower than *CUDA* implementations. The second improvement would prepare the implementation for realistic use cases extend the number of simulated risk factors in the market scenario generation step, as well as add correlation between different risk factors. The third improvement would be to connect our research with previous work done on financial workloads implemented in *Futhark* like, for instance, [And+16]. Finally, we mention that the *EOM* workflows for reporting purposes add yet another scaling factor, because they provide one more outermost parallel level in form of assessing risk measures across multiple portfolios that comprise holdings in different instruments.

All experiments would also benefit from tests on different types of accelerators, most importantly *FPGA*s that are growing in popularity in finance, especially in high-frequency trading. We would able to achieve this, because the high-level portable implementation in *Futhark* is compiled to a portable compute backend *OpenCL* code, which is supported on any accelerator platform.

Another natural next step to extend the risk experiments in this work would be to introduce Algorithmic Adjoint Differentiation (*AAD*) for the calculation of the risk factor sensitivities. As demonstrated in this work, the main goal of quantitative research and derivative pricing libraries is measuring portfolio risk, not the valuation itself. To take informed investment decisions, we want to asses the contributions and portfolio changes that we need to introduce to hedge the portfolio market risk or perform a regulatory adjustment for our derivative portfolio. One of the main achievements of the financial theory is the discovery that hedge ratios correspond to differential sensitivities. The goal is therefore to produce fast and accurate differentials of the valuation function. The determination and implementation of the valuation function is merely a step on the path towards this, because a valuation function must be defined before it can be differentiated. The work of [GG06; CJM17; Cap11] introduced the concepts of *AAD* to the field of computational finance. In addition, *AAD* can also be performed on *GPU*s [TLN13; Gre+16]. Standard references on the topics of *AAD* are [GW08; Nau12; Sav19]. The natural step would be to introduce an automatic differential approach to a high-level programming language like *Futhark*.

Finally, the thriving field of Machine Learning [Bis06; HTF09], in particular Deep Learning [GBC16], drives most of the current research in software acceleration and parallel hardware architectures. In the financial domain, these two fields has driven most of the original research in the recent two years, especially in the areas of derivative pricing and risk measurement [Pra18]. It is common today to propose approaches that use the output from simulations, which are presented in this work, to train Deep Neural Networks (*DNN*s) on the synthetic data. Once trained, a *DNN* can be used to infer prices in real-time and on demand. An acceleration of the whole workload is achieved, because the *DNN* is trained offline in batches on precomputed data. However, when used online on a previously unseen data, the *DNN* can infer price and risk results at a speed of a closed-form formula, which is far beyond the capabilities of any simulation-based system, which is built on top of classical numerical methods. These approaches work with an arbitrarily sophisticated financial models and are now use for real-time exotic derivative pricing [FG18; McG18; HMT19; LOB19], market data generation and model calibration [Liu+19]. On top of that, the performance is further impacted, because these techniques usually use the reduced floating-point precision and tensor cores deployed in the latest *GPU* generations.

References

[Abb+14] Lokman A. Abbas-Turki et al. "Pricing Derivatives on Graphics Processing Units Using Monte Carlo Simulation". In: *Concurrency and Computation: Practice and Experience* 26.9 (June 25, 2014), pp. 1679–1697. ISSN: 15320626. DOI: 10.1002/cpe.2862. URL: http://doi.wiley.com/10.1002/cpe.2862 (visited on January 31, 2021).

[AE18] Danil Annenkov and Martin Elsman. "Certified Compilation of Financial Contracts". In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. PPDP '18. New York, NY, USA: ACM, 2018, 5:1–5:13. ISBN: 978-1-4503-6441-6. DOI: 10.1145/3236950.3236955. URL: http://doi.acm.org/10.1145/3236950.3236955.

[Alb07] Claudio Albanese. *Callable Swaps, Snowballs and Videogames*. SSRN Scholarly Paper ID 1018468. Rochester, NY: Social Science Research Network, September 1, 2007. URL: https://papers.ssrn.com/abstract=1018468 (visited on January 31, 2021).

[And+06] Jesper Andersen et al. "Compositional Specification of Commercial Contracts". In: *International Journal on Software Tools for Technology Transfer* 8.6 (2006), pp. 485–516.

[And+16] Christian Andreetta et al. "FinPar: A Parallel Financial Benchmark". In: *ACM Transactions on Architecture and Code Optimization* 13.2 (June 27, 2016), pp. 1–27. ISSN: 15443566. DOI: 10.1145/2898354. URL: http://dl.acm.org/citation.cfm?doid=2952301.2898354 (visited on January 31, 2021).

[AP10] Leif B. G. Andersen and Vladimir V. Piterbarg. *Interest Rate Modeling. Volume 1: Foundations and Vanilla Models*. London ; New York: Atlantic Financial Press, February 6, 2010. 492 pp. ISBN: 978-0-9844221-0-4.

[Bal14] Luigi Ballabio. *Implementing QuantLib*. Leanpub, June 8, 2014. URL: https://leanpub.com/implementingquantlib (visited on January 31, 2021).

[Bal21] Luigi Ballabio. *QuantLib, A free/open-source library for quantitative finance*. https://www.quantlib.org/. January 31, 2021.

[BBE15] Patrick Bahr, Jost Berthold, and Martin Elsman. “Certified Symbolic Management of Financial Multi-Party Contracts”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 20th ACM SIGPLAN International Conference on Functional Programming. ICFP ’15. August 2015, pp. 315–327. DOI: 10.1145/2784731.2784747. URL: https://dl.acm.org/doi/abs/10.1145/2784731.2784747.

[Bil12] Patrick Billingsley. *Probability and Measure*. John Wiley & Sons, January 20, 2012. 586 pp. ISBN: 978-1-118-34191-9. Google Books: a3gavZbxyJcC.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. New York: Springer, 2006. 738 pp. ISBN: 978-0-387-31073-2.

[Ble+94] Guy E Blelloch et al. “Implementation of a Portable Nested Data-Parallel Language”. In: *Journal of parallel and distributed computing* 21.1 (1994), pp. 4–14.

[Ble89] Guy E. Blelloch. “Scans as Primitive Parallel Operations”. In: *Computers, IEEE Transactions* 38.11 (1989), pp. 1526–1538.

[Ble90] Guy E Blelloch. *Vector Models for Data-Parallel Computing*. Vol. 75. MIT press Cambridge, 1990.

[Ble96] Guy E. Blelloch. “Programming Parallel Algorithms”. In: *Communications of the ACM (CACM)* 39.3 (1996), pp. 85–97.

[BM06] Damiano Brigo and Fabio Mercurio. *Interest Rate Models - Theory and Practice: With Smile, Inflation and Credit*. 2nd edition. Berlin ; New York: Springer, August 2, 2006. 982 pp. ISBN: 978-3-540-22149-4.

[Bon+08] Uday Bondhugula et al. “A Practical Automatic Polyhedral Parallelizer and Locality Optimizer”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’08. New York, NY, USA: ACM, 2008, pp. 101–113. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375595. URL: http://doi.acm.org/10.1145/1375581.1375595.

[Boy86] Phelim P. Boyle. “Option Valuation Using a Three Jump Process”. In: 1986.

[BR12] Lars Bergstrom and John Reppy. “Nested Data-Parallelism on the GPU”. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’12. New York, NY, USA: ACM, 2012, pp. 247–258. ISBN: 978-1-4503-1054-3. DOI: 10.1145/2364527.2364563. URL: http://doi.acm.org/10.1145/2364527.2364563.

[Bru+15] Christian Brugger et al. “Reverse Longstaff-Schwartz American Option Pricing on Hybrid CPU/FPGA Systems”. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015 Design, Automation Test in Europe Conference Exhibition (DATE). March 2015, pp. 1599–1602. DOI: 10.7873/DATE.2015.0688.

[BSS10] André Bernemann, Ralph Schreyer, and Klaus Spanderen. “Pricing Structured Equity Products on GPUs”. In: *2010 IEEE Workshop on High Performance Computational Finance*. 2010 IEEE Workshop on High Performance Computational Finance. November 2010, pp. 1–7. DOI: 10.1109/WHPCF.2010.5671821.

[BV18] Stephen Boyd and Lieven Vandenberghe. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. 1 edition. Cambridge University Press, June 30, 2018. 457 pp.

[Cap11] Luca Capriotti. “Fast Greeks by Algorithmic Differentiation”. In: *The Journal of Computational Finance* 14.3 (2011), p. 34.

[Car96] Jacques F. Carriere. “Valuation of the Early-Exercise Price for Options Using Simulations and Nonparametric Regression”. In: *Insurance: Mathematics and Economics* 19.1 (December 1, 1996), pp. 19–30. ISSN: 0167-6687. DOI: 10.1016/S0167-6687(96)00004-2. URL: http://www.sciencedirect.com/science/article/pii/S0167668796000042 (visited on January 31, 2021).

[Cha+07] Manuel M. T. Chakravarty et al. “Data Parallel Haskell: A Status Report”. In: *Int. Work. on Decl. Aspects of Multicore Prog. (DAMP)*. 2007, pp. 10–18.

[Cha+11] Manuel MT Chakravarty et al. “Accelerating Haskell Array Codes with Multicore GPUs”. In: *Proc. of the Sixth Workshop on Declarative Aspects of Multicore Programming*. ACM. 2011, pp. 3–14.

[Chi+04] Y. Chicha et al. “Parametric Polymorphism for Computer Algebra Software Components”. In: *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Comput.* Mirton Publishing House, 2004, pp. 119–130.

[CHL15] Ching-Wen Chen, Kuan-Lin Huang, and Yuh-Dauh Lyuu. “Accelerating the Least-Square Monte Carlo Method with Parallel Computing”. In: *The Journal of Supercomputing* 71.9 (September 1, 2015), pp. 3593–3608. ISSN: 1573-0484. DOI: 10.1007/s11227-015-1451-7. URL: https://doi.org/10.1007/s11227-015-1451-7 (visited on January 31, 2021).

[Cho+08] A. R. Choudhury et al. “Optimizations in Financial Engineering: The Least-Squares Monte Carlo Method of Longstaff and Schwartz”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008 IEEE International Symposium on Parallel and Distributed Processing. April 2008, pp. 1–11. DOI: 10.1109/IPDPS.2008.4536290.

[CJM17] Luca Capriotti, Yupeng Jiang, and Andrea Macrina. “AAD and Least-Square Monte Carlo: Fast Bermudan-Style Options and XVA Greeks”. In: *Algorithmic Finance* 6.1-2 (October 21, 2017), pp. 35–49. ISSN: 21585571, 21576203. DOI: 10.3233/AF-170201. URL: http://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/AF-170201 (visited on January 31, 2021).

[CLP02] Emmanuelle Clément, Damien Lamberton, and Philip Protter. “An Analysis of a Least Squares Regression Method for American Option Pricing”. In: *Finance and Stochastics* 6.4 (October 1, 2002), pp. 449–471. ISSN: 0949-2984. DOI: 10.1007/s007800200071. URL: https://doi.org/10.1007/s007800200071 (visited on January 31, 2021).

[CPT18] Gérard Cornuéjols, Javier Peña, and Reha Tütüncü. *Optimization Methods in Finance*. 2 edition. Cambridge, United Kingdom ; New York, NY: Cambridge University Press, September 20, 2018. 348 pp. ISBN: 978-1-107-05674-9. DOI: 10.1017/9781107297340. URL: /core/books/optimization-methods-in-finance/8A4996C5DB2006224E4D983B5BC95E3B (visited on January 31, 2021).

[CSS12] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. “Expressive Array Constructs in an Embedded GPU Kernel Programming Language”. In: *Work. on Decl. Aspects of Multicore Prog DAMP*. 2012, pp. 21–30.

[CSS15] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. “Polyhedral Optimizations of Explicitly Parallel Programs”. In: *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*. PACT ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 213–226. ISBN: 978-1-4673-9524-3. DOI: 10.1109/PACT.2015.44. URL: https://doi.org/10.1109/PACT.2015.44.

[DCJ12] Duy Minh Dang, Christina C. Christara, and Kenneth R. Jackson. “An Efficient Graphics Processing Unit-Based Parallel Algorithm for Pricing Multi-Asset American Options”. In: *Concurrency and Computation: Practice and Experience* 24.8 (June 10, 2012), pp. 849–866. ISSN: 1532-0626. DOI: 10.1002/cpe.1784. URL: http://onlinelibrary-wiley-com/doi/full/10.1002/cpe.1784 (visited on January 31, 2021).

[DCJ13] Duy Minh Dang, Christina C. Christara, and Kenneth R. Jackson. “A Highly Efficient Implementation on GPU Clusters of PDE-Based Pricing Methods for Path-Dependent Foreign Exchange Interest Rate Derivatives”. In: *Computational Science and Its Applications ICCSA 2013*. International Conference on Computational Science and Its Applications. Springer, Berlin, Heidelberg, June 24, 2013, pp. 107–126. DOI: 10.1007/978-3-642-39640-3_8. URL: http://link-springer-com/chapter/10.1007/978-3-642-39640-3_8 (visited on January 31, 2021).

[DCJ14] Duy Minh Dang, Christina C. Christara, and Kenneth R. Jackson. “Graphics Processing Unit Pricing of Exotic Cross-Currency Interest Rate Derivatives with a Foreign Exchange Volatility Skew Model”. In: *Concurrency and Computation: Practice and Experience* 26.9 (June 25, 2014), pp. 1609–1625. ISSN: 1532-0626. DOI: 10.1002/cpe.2824. URL: http://onlinelibrary-wiley-com/doi/10.1002/cpe.2824 (visited on January 31, 2021).

[DCK09] Matthew Dixon, Jike Chong, and Kurt Keutzer. “Acceleration of Market Value-at-Risk Estimation”. In: *Proceedings of the 2nd Workshop on High Performance Computational Finance* (Portland, Oregon). WHPCF ’09. New York, NY, USA: ACM, 2009, 5:1–5:8. ISBN: 978-1-60558-716-5. DOI: 10.1145/1645413.1645418. URL: http://doi.acm.org/10.1145/1645413.1645418 (visited on January 31, 2021).

[DCK12] Matthew Dixon, Jike Chong, and Kurt Keutzer. “Accelerating Value-at-Risk Estimation on Highly Parallel Architectures”. In: *Concurrency and Computation: Practice and Experience* 24.8 (June 10, 2012), pp. 895–907. ISSN: 15320626. DOI: 10.1002/cpe.1790. URL: http://doi.wiley.com/10.1002/cpe.1790 (visited on January 31, 2021).

[De 15] Christian De Schryver. *FPGA Based Accelerators for Financial Applications*. Springer, 2015.

[Dem14] Julien Demouth. *Monte-Carlo Simulation of American Options with GPUs*. http://on-demand.gputechconf.com/gtc/2014/presentations/S4784-monte-carlo-sim-american-options-gpus.pdf. Presentation at NVIDIA GPU Technology Conference. 2014.

[Dix+12] Matthew Dixon et al. “Chapter 25 - Monte CarloBased Financial Market Value-at-Risk Estimation on GPUs”. In: *GPU Computing Gems Jade Edition*. Ed. by Wen-mei W. Hwu. Applications of GPU Computing Series. Boston: Morgan Kaufmann, January 1, 2012, pp. 337–353. ISBN: 978-0-12-385963-1. DOI: 10.1016/B978-0-12-385963-1.00025-3. URL: http://www.sciencedirect.com/science/article/pii/B9780123859631000253 (visited on January 31, 2021).

[DK99] Chen Ding and Ken Kennedy. “Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time”. In: *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. PLDI ’99. New York, NY, USA: ACM, 1999, pp. 229–241. ISBN: 1-58113-094-5. DOI: 10.1145/301618.301670. URL: http://doi.acm.org/10.1145/301618.301670.

[Ege+17] Benjamin Egelund-Müller et al. “Automated Execution of Financial Contracts on Blockchains”. In: *Business & Information Systems Engineering* 59.6 (2017), pp. 457–467.

[EHO18] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark*. Department of Computer Science, University of Copenhagen, November 2018. URL: https://futhark-book.readthedocs.io.

[Els+18] Martin Elsman et al. “Static Interpretation of Higher-Order Modules in Futhark: Functional GPU Programming in the Large”. In: *Proceedings of the ACM on Programming Languages* 2 (ICFP July 2018), 97:1–97:30. ISSN: 2475-1421. DOI: 10.1145/3236792. URL: http://dl.acm.org/citation.cfm?doid=3243631.3236792.

[FG18] Ryan Ferguson and Andrew Green. *Deeply Learning Derivatives*. September 6, 2018. arXiv: 1809.02233 [cs, q-fin]. URL: http://arxiv.org/abs/1809.02233 (visited on January 31, 2021).

[FP13] Massimiliano Fatica and Everett Phillips. “Pricing American Options with Least Squares Monte Carlo on GPUs”. In: *Proceedings of the 6th Workshop on High Performance Computational Finance* (Denver, Colorado). WHPCF ’13. New York, NY, USA: ACM, 2013, 5:1–5:6. ISBN: 978-1-4503-2507-3. DOI: 10.1145/2535557.2535564. URL: http://doi.acm.org/10.1145/2535557.2535564 (visited on January 31, 2021).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. The MIT Press, 2016. ISBN: 978-0262035613.

[Ger03] Alexandros V. Gerbessiotis. “Trinomial-Tree Based Parallel Option Price Valuations”. In: *Parallel Algorithms and Applications* 18.4 (December 1, 2003), pp. 181–196. ISSN: 1063-7192. DOI: 10.1080/10637190310001633655. URL: https://doi.org/10.1080/10637190310001633655 (visited on January 31, 2021).

[Ger04] Alexandros V. Gerbessiotis. “Architecture Independent Parallel Binomial Tree Option Price Valuations”. In: *Parallel Computing* 30.2 (February 1, 2004), pp. 301–316. ISSN: 0167-8191. DOI: 10.1016/j.parco.2003.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0167819103001753 (visited on January 31, 2021).

[GG06] Michael B. Giles and Paul Glasserman. “Smoking Adjoints: Fast Monte Carlo Greeks”. In: *Risk* (January 2006), pp. 88–92. URL: http://web.comlab.ox.ac.uk/oucl/work/mike.giles/finance.html.

[Gie+20] Fabian Gieseke et al. “Massively-Parallel Change Detection for Satellite Time Series Data with Missing Values”. In: *Procs. of 36th IEEE International Conference on Data Engineering*. ICDE’20. IEEE, 2020.

[Gil08] Michael B. Giles. “Multilevel Monte Carlo Path Simulation”. In: *Operations Research* 56.3 (June 1, 2008), pp. 607–617. ISSN: 0030-364X. DOI: 10.1287/opre.1070.0496. URL: https://pubsonline.informs.org/doi/10.1287/opre.1070.0496 (visited on January 31, 2021).

[Gla04] Paul Glasserman. *Monte Carlo Methods in Financial Engineering*. New York: Springer, 2004. ISBN: 978-0-387-00451-8.

[GMS19] Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. 2 edition. Cambridge: Academic Press, August 30, 2019. 638 pp. ISBN: 978-0-12-815065-8.

[Gra+13] Scott Grauer-Gray et al. “Accelerating Financial Applications on the GPU”. In: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units. ACM, March 16, 2013, pp. 127–136. ISBN: 978-1-4503-2017-7. DOI: 10.1145/2458523.2458536. URL: http://dl.acm.org/citation.cfm?id=2458523.2458536 (visited on January 31, 2021).

[Gre+16] Felix Gremse et al. “GPU-Accelerated Adjoint Algorithmic Differentiation”. In: *Computer Physics Communications* 200 (March 2016), pp. 300–311. ISSN: 00104655. DOI: 10.1016/j.cpc.2015.10.027. URL: https://linkinghub.elsevier.com/retrieve/pii/S0010465515004099 (visited on January 31, 2021).

[GS03] Clemens Grelck and Sven-Bodo Scholz. “SaC - From High-Level Programming with Arrays to Efficient Parallel Execution”. In: *Parallel Processing Letters* 13 (September 2003), pp. 401–412. DOI: 10.1142/S0129626403001379.

[GS06] Clemens Grelck and Sven-Bodo Scholz. “SAC - A Functional Array Language for Efficient Multi-Threaded Execution”. In: *International Journal of Parallel Programming* 34.4 (2006), pp. 383–427.

[GTS11] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. “Breaking the GPU Programming Barrier with the Auto-Parallelising SAC Compiler”. In: *Procs. Workshop Decl. Aspects of Multicore Prog. (DAMP)*. ACM, 2011, pp. 15–24.

[Gui+13] Yechen Gui et al. “High Performance Implementation of Binomial Option Pricing Using CUDA”. In: *GPU Solutions to Multi-scale Problems in Science and Engineering* (2013), pp. 201–214. DOI: 10.1007/978-3-642-16405-7_12. URL: http://link-springer-com/chapter/10.1007/978-3-642-16405-7_12 (visited on January 31, 2021).

[GW08] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Second. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, January 1, 2008. 448 pp. ISBN: 978-0-89871-659-7. DOI: 10.1137/1.9780898717761. URL: https://epubs.siam.org/doi/book/10.1137/1.9780898717761 (visited on January 31, 2021).

[Hau07] Espen Gaarder Haug. *The Complete Guide to Option Pricing Formulas*. 2 edition. New York: McGraw-Hill Education, January 8, 2007. 492 pp. ISBN: 978-0-07-138997-6.

[Hen+16] Troels Henriksen et al. “APL on GPUs: A TAIL from the Past, Scribbled in Futhark”. In: *Proceedings of the 5th International Workshop on Functional High-Performance Computing - FHPC 2016*. The 5th International Workshop. Nara, Japan: ACM Press, 2016, pp. 38–43. ISBN: 978-1-4503-4433-3. DOI: 10.1145/2975991.2975997. URL: http://dl.acm.org/citation.cfm?doid=2975991.2975997 (visited on January 31, 2021).

[Hen+17] Troels Henriksen et al. “Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. URL: http://doi.acm.org/10.1145/3062341.3062354.

[Hen+19] Troels Henriksen et al. “Incremental Flattening for Nested Data Parallelism”. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP ’19. New York, NY, USA: ACM, 2019, pp. 53–67. ISBN: 978-1-4503-6225-2. DOI: 10.1145/3293883.3295707. URL: http://doi.acm.org/10.1145/3293883.3295707.

[HEO18] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. “Modular Acceleration: Tricky Cases of Functional High-Performance Computing”. In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing - FHPC 2018*. The 7th ACM SIGPLAN International Workshop. St. Louis, MO, USA: ACM Press, 2018, pp. 10–21. ISBN: 978-1-4503-5813-2. DOI: 10.1145/3264738.3264740. URL: http://dl.acm.org/citation.cfm?doid=3264738.3264740 (visited on January 31, 2021).

[HHE18] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. “High-Performance Defunctionalization in Futhark”. In: *Symposium on Trends in Functional Programming (TFP’18)*. September 2018.

[HLO16] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. “Design and GPGPU Performance of Futhark’s Redomap Construct”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY 2016*. The 3rd ACM SIGPLAN International Workshop. Santa Barbara, CA, USA: ACM Press, 2016, pp. 17–24. ISBN: 978-1-4503-4384-8. DOI: 10.1145/2935323.2935326. URL: http://dl.acm.org/citation.cfm?doid=2935323.2935326 (visited on January 31, 2021).

[HMT19] Blanka Horvath, Aitor Muguruza, and Mehdi Tomas. *Deep Learning Volatility*. SSRN Scholarly Paper ID 3322085. Rochester, NY: Social Science Research Network, January 24, 2019. URL: https://papers.ssrn.com/abstract=3322085 (visited on January 31, 2021).

[HO13] Troels Henriksen and Cosmin Eugen Oancea. “A T2 Graph-Reduction Approach to Fusion”. In: *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing - FHPC ’13*. The 2nd ACM SIGPLAN Workshop. Boston, Massachusetts, USA: ACM Press, 2013, p. 47. ISBN: 978-1-4503-2381-9. DOI: 10.1145/2502323.2502328. URL: http://dl.acm.org/citation.cfm?doid=2502323.2502328 (visited on January 31, 2021).

[HT05] Kai Huang and Ruppa K. Thulasiram. “Parallel Algorithm for Pricing American Asian Options with Multi-Dimensional Assets”. In: *19th International Symposium on High Performance Computing Systems and Applications (HPCS’05)*. 19th International Symposium on High Performance Computing Systems and Applications (HPCS’05). May 2005, pp. 177–185. DOI: 10.1109/HPCS.2005.38.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. 2nd ed. Springer Series in Statistics. New York: Springer-Verlag, 2009. ISBN: 978-0-387-84857-0. URL: //www.springer.com/gp/book/9780387848570 (visited on January 31, 2021).

[Hul17] John Hull. *Options, Futures, and Other Derivatives*. 10th ed. Pearson Education, 2017.

[Hul18] John Hull. *Risk Management and Financial Institutions*. Fifth edition. Hoboken, NewJersey: John Wiley & Sons, Inc, 2018. ISBN: 978-1-119-44811-2.

[HW90] John Hull and Alan White. “Valuing Derivative Securities Using the Explicit Finite Difference Method”. In: *The Journal of Financial and Quantitative Analysis* 25.1 (March 1990), p. 87. ISSN: 00221090. DOI: 10.2307/2330889. JSTOR: 2330889.

[HW93] John Hull and Alan White. “One-Factor Interest-Rate Models and the Valuation of Interest-Rate Derivative Securities”. In: *The Journal of Financial and Quantitative Analysis* 28.2 (June 1993), p. 235. ISSN: 00221090. DOI: 10.2307/2331288. JSTOR: 2331288.

[HW94] John Hull and Alan White. “Numerical Procedures for Implementing Term Structure Models II: Two-Factor Models”. In: *The Journal of Derivatives* 2.2 (November 30, 1994), pp. 37–48. ISSN: 1074-1240, 2168-8524. DOI: 10.3905/jod.1994.407908. URL: https://jod.pm-research.com/content/2/2/37 (visited on January 31, 2021).

[HW96] John Hull and Alan White. “Using Hull-White Interest Rate Trees”. In: *The Journal of Derivatives* 3.3 (February 29, 1996), pp. 26–36. ISSN: 1074-1240, 2168-8524. DOI: 10.3905/jod.1996.407949. URL: http://jod.iijournals.com/lookup/doi/10.3905/jod.1996.407949 (visited on January 31, 2021).

[Int] Bank for International Settlements. *OTC derivatives outstanding*. https://www.bis.org/statistics/derstats.htm. (Visited on January 31, 2021).

[Jäc02] Peter Jäckel. *Monte Carlo Methods in Finance*. Wiley Finance Series. Chichester, West Sussex, England: J. Wiley, 2002. 222 pp. ISBN: 978-0-471-49741-7.

[Jia+18] Zhe Jia et al. *Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking*. April 18, 2018. arXiv: 1804.06826 [cs]. URL: http://arxiv.org/abs/1804.06826 (visited on January 31, 2021).

[JK03] Stephen Joe and Frances Y. Kuo. “Remark on Algorithm 659: Implementing Sobol’s Quasirandom Sequence Generator: ACM Transactions on Mathematical Software: Vol 29, No 1”. In: *ACM Transactions on Mathematical Software (TOMS)* (2003). URL: https://dl.acm.org/doi/abs/10.1145/641876.641879 (visited on January 31, 2021).

[JK08] Stephen Joe and Frances Y. Kuo. “Constructing Sobol’ Sequences with Better Two-Dimensional Projections”. In: *SIAM Journal on Scientific Computing* 30.5 (July 2008), pp. 2635–2654. ISSN: 10648275. DOI: 10.1137/070709359. URL: https://search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=34835527&site=ehost-live (visited on January 31, 2021).

[Lee+10] A. Lee et al. “On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods”. In: *J. Comp. Graph. Stat* 19.4 (2010), pp. 769–789.

[LH17] Rasmus Wriedt Larsen and Troels Henriksen. “Strategies for Regular Segmented Reductions on GPU”. In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. FHPC 2017. New York, NY, USA: ACM, 2017, pp. 42–52. ISBN: 978-1-4503-5181-2. DOI: 10.1145/3122948.3122952. URL: http://doi.acm.org/10.1145/3122948.3122952.

[Liu+19] Shuaiqiang Liu et al. “A Neural Network-Based Framework for Financial Model Calibration”. In: *Journal of Mathematics in Industry* 9.1 (September 5, 2019), p. 9. ISSN: 2190-5983. DOI: 10.1186/s13362-019-0066-7. URL: https://doi.org/10.1186/s13362-019-0066-7 (visited on January 31, 2021).

[LOB19] Shuaiqiang Liu, Cornelis W. Oosterlee, and Sander M. Bohte. *Pricing Options and Computing Implied Volatilities Using Neural Networks*. January 25, 2019. arXiv: 1901.08943 [cs, math, q-fin]. URL: http://arxiv.org/abs/1901.08943 (visited on January 31, 2021).

[LS01] Francis A. Longstaff and Eduardo S. Schwartz. “Valuing American Options by Simulation: A Simple Least-Squares Approach”. In: *The Review of Financial Studies* 14.1 (January 1, 2001), pp. 113–147. ISSN: 0893-9454. DOI: 10.1093/rfs/14.1.113. URL: https://academic.oup.com/rfs/article/14/1/113/1587472 (visited on January 31, 2021).

[McG18] William A. McGhee. *An Artificial Neural Network Representation of the SABR Stochastic Volatility Model*. SSRN Scholarly Paper ID 3288882. Rochester, NY: Social Science Research Network, November 21, 2018. URL: https://papers.ssrn.com/abstract=3288882 (visited on January 31, 2021).

[Meh+12] Amit Mehta et al. “Managing Market Risk: Today and Tomorrow”. In: *McKinsey & Company McKinsey Working Papers on Risk* 32 (2012), p. 24.

[MFE15] Alexander J McNeil, Rüdiger Frey, and Paul Embrechts. *Quantitative Risk Management: Concepts, Techniques and Tools*. Revised. Princeton University Press, 2015. 720 pp. ISBN: 978-0-691-16627-8. URL: https://press.princeton.edu/books/hardcover/9780691166278/quantitative-risk-management.

[MG16] Duane Merrill and Michael Garland. “Single-Pass Parallel Prefix Scan with Decoupled Lookback”. In: 2016.

[MH99] Sungdo Moon and Mary W. Hall. “Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization”. In: *Int. Symp. Princ. and Practice of Par. Prog. (PPoPP)*. 1999, pp. 84–95.

[Nau12] Uwe Naumann. *The Art of Differentiating Computer Programs*. 1 edition. Philadelphia: Society for Industrial and Applied Mathematics, January 26, 2012. 356 pp. ISBN: 978-1-61197-206-1.

[NL11] Fredrik Nord and Erwin Laure. “Monte Carlo Option Pricing with Graphics Processing Units”. In: *Int. Conf. ParCo*. 2011.

[NVI] NVIDIA. *NVIDIA Developer Blog Code Samples repository at GitHub*. `https://github.com/NVIDIA-developer-blog/code-samples/tree/master/posts/american-options`. (Visited on January 31, 2021).

[NVI17] NVIDIA. *NVIDIA Tesla V100 GPU Architecture*. NVIDIA Corporation, 2017. URL: `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf` (visited on January 31, 2021).

[Oan+05] C. E. Oancea et al. “Distributed Models of Thread-Level Speculation”. In: *Proceedings of the PDPTA’05*. 2005, pp. 920–927. URL: `http://www.csd.uwo.ca/~coancea/Publications`.

[Oan+12] Cosmin E. Oancea et al. “Financial Software on GPUs: Between Haskell and Fortran”. In: *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-Performance Computing*. FHPC ’12. New York, NY, USA: ACM, 2012, pp. 61–72. ISBN: 978-1-4503-1577-7. DOI: `10.1145/2364474.2364484`. URL: `http://doi.acm.org/10.1145/2364474.2364484`.

[OM08] Cosmin E. Oancea and Alan Mycroft. “Set-Congruence Dynamic Analysis for Software Thread-Level Speculation”. In: *Procs. Langs. Comp. Parallel Computing*. 2008, pp. 156–171.

[OR11] Cosmin E. Oancea and Lawrence Rauchwerger. “A Hybrid Approach to Proving Memory Reference Monotonicity”. In: *Languages and Compilers for Parallel Computing*. International Workshop on Languages and Compilers for Parallel Computing. Springer, Berlin, Heidelberg, September 8, 2011, pp. 61–75. DOI: `10.1007/978-3-642-36036-7_5`. URL: `https://link-springer-com/chapter/10.1007/978-3-642-36036-7_5` (visited on January 31, 2021).

[OR15] Cosmin E. Oancea and Lawrence Rauchwerger. “Scalable Conditional Induction Variables (CIV) Analysis”. In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 213–224. ISBN:

978-1-4799-8161-8. URL: http://dl.acm.org/citation.cfm?id=2738600.2738627.

[OW05a] C. E. Oancea and S. M. Watt. “Domains and Expressions: An Interface between Two Approaches to Computer Algebra”. In: *Proceedings of the ACM ISSAC 2005*. 2005, pp. 261–269. URL: http://www.csd.uwo.ca/~coancea/Publications.

[OW05b] Cosmin E. Oancea and Stephen M. Watt. “Parametric Polymorphism for Software Component Architectures”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 05. San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 147–166. ISBN: 1595930310. DOI: 10.1145/1094811.1094823. URL: https://doi.org/10.1145/1094811.1094823.

[PES00] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. “Composing Contracts: An Adventure in Financial Engineering (Functional Pearl)”. In: *Int. Conf. on Funct. Prog. (ICFP)*. 2000, pp. 280–292.

[Pou+11] Louis-Noël Pouchet et al. “Loop Transformations: Convexity, Pruning and Optimization”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’11. New York, NY, USA: ACM, 2011, pp. 549–562. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926449. URL: http://doi.acm.org/10.1145/1926385.1926449.

[Pra18] Marcos Lopez de Prado. *Advances in Financial Machine Learning*. 1 edition. New Jersey: Wiley, February 21, 2018. 400 pp. ISBN: 978-1-119-48208-6.

[PW12] Gilles Pagès and Benedikt Wilbertz. “GPGPUs in Computational Finance: Massive Parallel Computing for American Style Options”. In: *Concurrency and Computation: Practice and Experience* 24.8 (2012), pp. 837–848. ISSN: 1532-0634. DOI: 10.1002/cpe.1774. URL: http://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1774 (visited on January 31, 2021).

[RAP95] Lawrence Rauchwerger, Nancy Amato, and David Padua. “A Scalable Method for Run Time Loop Parallelization”. In: *Int. Journal of Par. Prog* 26 (1995), pp. 26–6.

[Rav+14] Mahesh Ravishankar et al. “Automatic Parallelization of a Class of Irregular Loops for Distributed Memory Systems”. In: *ACM Transactions on Parallel Computing* 1.1 (October 2014), 7:1–7:37. ISSN: 2329-4949. DOI: 10.1145/2660251. URL: http://doi.acm.org/10.1145/2660251.

[RKC16] Chandan Reddy, Michael Kruse, and Albert Cohen. “Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU”. In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT ’16. New York, NY, USA: ACM, 2016, pp. 87–97. ISBN: 978-1-4503-4121-9. DOI: 10.1145/2967938.2967950. URL: http://doi.acm.org/10.1145/2967938.2967950.

[RS15] John Reppy and Nora Sandler. “Nessie: A NESL to CUDA Compiler”. In: (January 2015).

[Sav19] Antoine Savine. *Modern Computational Finance: AAD and Parallel Simulations*. Hoboken, New Jersey: John Wiley & Sons, Inc, 2019. 1 p. ISBN: 978-1-119-53954-4 978-1-119-53952-0.

[SCF03] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. “Compile-Time Composition of Run-Time Data and Iteration Reorderings”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI ’03. New York, NY, USA: ACM, 2003, pp. 91–102. ISBN: 1-58113-662-5. DOI: 10.1145/781131.781142. URL: http://doi.acm.org/10.1145/781131.781142.

[Shi16] Albert N. Shiryaev. *Probability-1*. 3rd ed. Graduate Texts in Mathematics, Probability. New York: Springer-Verlag, 2016. ISBN: 978-0-387-72205-4. URL: https://www.springer.com/gp/book/9780387722054 (visited on January 31, 2021).

[SHO18] Michelle Strout, Mary Hall, and Catherine Olschanowsky. “The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code”. In: *Proceedings of the IEEE* PP (August 2018), pp. 1–15. DOI: 10.1109/JPROC.2018.2857721.

[SHP08] Hannes Schabauer, Ronald Hochreiter, and Georg Ch Pflug. “Parallelization of Pricing Path-Dependent Financial Instruments on Bounded Trinomial Lattices”. In: *Computational Science ICCS 2008*. International Conference on Computational Science. Springer, Berlin, Heidelberg, June 23, 2008, pp. 408–415. DOI: 10.1007/978-3-540-69387-1_46. URL: http://link-springer-com/chapter/10.1007/978-3-540-69387-1_46 (visited on January 31, 2021).

[SSC11] Joel Svensson, Mary Sheeran, and Koen Claessen. “Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors”. In: *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*. IFL’08. Hatfield, UK: Springer-Verlag, 2011, pp. 156–173. ISBN: 978-3-642-24451-3. URL: http://dl.acm.org/citation.cfm?id=2044476.2044485.

[STA20] Securities Technology Analysis Center STAC. *STAC-A2 Benchmark Suite*. https://stacresearch.com/a2. 2020. (Visited on January 31, 2021).

[Suo+15] Simon Suo et al. “GPU Option Pricing”. In: Proceedings of the 8th Workshop on High Performance Computational Finance. ACM, November 15, 2015, p. 8. ISBN: 978-1-4503-4015-1. DOI: 10.1145/2830556.2830564. URL: http://dl.acm.org/citation.cfm?id=2830556.2830564 (visited on January 31, 2021).

[Sve11] Joel Svensson. “Obsidian: GPU Kernel Programming in Haskell”. Ph.D. Thesis. Chalmers University of Technology, 2011.

[The+15] Alexios Theiakos et al. *Ultra-Fast Scenario Analysis of Mortgage Prepayment Risk*. SSRN Scholarly Paper ID 2799643. Rochester, NY: Social Science Research Network, February 9, 2015. URL: https://papers.ssrn.com/abstract=2799643 (visited on January 31, 2021).

[TLN13] Jacques du Toit, Johannes Lotz, and Uwe Naumann. “Adjoint Algorithmic Differentiation of a GPU Accelerated Application”. In: 2013.

[TOP20] TOP500. *List of 500 fastest supercomputers in the world*. https://www.top500.org/lists/. 2020.

[Tre08] Lloyd N. Trefethen. “Numerical Analysis”. In: *The Princeton Companion to Mathematics*. Princeton University Press, 2008, pp. 604–615.

[TV01] John N. Tsitsiklis and Benjamin Van Roy. “Regression Methods for Pricing Complex American-Style Options”. In: *IEEE Transactions on Neural Networks* 12.4 (July 2001), pp. 694–703. ISSN: 1045-9227. DOI: 10.1109/72.935083.

[Var+15a] Javier Alejandro Varela et al. “Optimization Strategies for Portable Code for Monte Carlo-Based Value-at-Risk Systems”. In: *Proceedings of the 8th Workshop on High Performance Computational Finance - WHPCF ’15*. The 8th Workshop. Austin, Texas: ACM Press, 2015, pp. 1–8. ISBN: 978-1-4503-4015-1. DOI: 10.1145/2830556.2830559. URL: http://dl.acm.org/citation.cfm?doid=2830556.2830559 (visited on January 31, 2021).

[Var+15b] Javier Alejandro Varela et al. “Pricing High-Dimensional American Options on Hybrid CPU/FPGA Systems”. In: *FPGA Based Accelerators for Financial Applications*. Ed. by Christian De Schryver. Cham: Springer International Publishing, 2015, pp. 143–166. ISBN: 978-3-319-15407-7. DOI: 10.1007/978-3-319-15407-7_7. URL: https://doi.org/10.1007/978-3-319-15407-7_7 (visited on January 31, 2021).

[Var+17] Javier Alejandro Varela et al. “Real-Time Financial Risk Measurement of Dynamic Complex Portfolios with Python and PyOpenCL”. In: *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing* (Denver, CO, USA). PyHPC'17. New York, NY, USA: ACM, 2017, 3:1–3:10. ISBN: 978-1-4503-5124-9. DOI: 10.1145/3149869.3149872. URL: http://doi.acm.org/10.1145/3149869.3149872 (visited on January 31, 2021).

[VW17] Javier Alejandro Varela and Norbert Wehn. “Near Real-Time Risk Simulation of Complex Portfolios on Heterogeneous Computing Systems with OpenCL”. In: *Proceedings of the 5th International Workshop on OpenCL* (Toronto, Canada). IWOCL 2017. New York, NY, USA: ACM, 2017, 2:1–2:10. ISBN: 978-1-4503-5214-7. DOI: 10.1145/3078155.3078161. URL: http://doi.acm.org/10.1145/3078155.3078161 (visited on January 31, 2021).

[VW18] Javier Alejandro Varela and Norbert Wehn. “Running Financial Risk Management Applications on FPGA in the Amazon Cloud”. In: 2018.

[Zha+17] Shuai Zhang et al. “Mapping of Option Pricing Algorithms onto Heterogeneous Many-Core Architectures”. In: *The Journal of Supercomputing* 73.9 (September 1, 2017), pp. 3715–3737. ISSN: 1573-0484. DOI: 10.1007/s11227-017-1968-z. URL: https://doi.org/10.1007/s11227-017-1968-z (visited on January 31, 2021).

[ZLM12] Nan Zhang, Chi-Un Lei, and Ka Lok Man. “Binomial American Option Pricing on CPU-GPU Hetergenous System”. In: *Engineering Letters* 20.3 (September 2012), pp. 279–285. ISSN: 1816093X. URL: https://search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=82189139&site=ehost-live (visited on January 31, 2021).

[ZM08] Mohammad Zubair and Ravi Mukkamala. “High Performance Implementation of Binomial Option Pricing”. In: *Computational Science and Its Applications ICCSA 2008*. International Conference on Computational Science and Its Applications. Springer, Berlin, Heidelberg, June 30, 2008, pp. 852–866. DOI: 10.1007/978-3-540-69839-5_64. URL: http://link-springer-com/chapter/10.1007/978-3-540-69839-5_64 (visited on January 31, 2021).

[ZM12] Yongpeng Zhang and Frank Mueller. “CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures”. In: *Proceedings of the 2012 41st International Conference on Parallel Processing*. ICPP'12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 340–349. ISBN: 978-0-7695-4796-1.

Appendix A

Domain of Quantitative Finance

At present, the financial industry is one of the key drivers for developments in computer science, in particular *HPC*. This domain deals with an efficient implementation of numerical algorithms that extract the computational power of parallel or distributed hardware to solve complex scientific and engineering models in a fast and accurate manner. To achieve performance in computational finance, we need to combine knowledge from different fields. The theoretical background encompasses concepts from economy (financial market theories, flow of money in society) theoretical mathematics (measure theory), and statistics (probability, stochastic calculus). The practical application of the theory involves applied mathematics (numerical methods), data science (data engineering) as well as the computational and computer science techniques, that this work focuses on.

Finance is a subfield of the economics social sciences and studies quantitative aspects of the flow of money in society. Finance can be further divided into multiple areas that vary from purely theoretical to more applied ones. To cast a light on the diversity of the financial domain, we start with a high-level description of these subareas as well as list the main techniques and applications they are concerned with. Even though a transparent division is not a trivial task, we benefit from developing a clear understanding of their multiple differences. Furthermore, this process is highly relevant for our research, as we want to focus our investigations on the fundamental areas with the largest impact as well as approach the most frequent issues of the financial applications from the algorithmic and computational perspective. Therefore, this section proceeds to define and outline the delicate distinctions of the three quantitative areas of finance, *Mathematical Finance*, *Computational Finance* and *Financial Engineering*. The concepts and techniques from each of the above fields are used to a varying degree throughout this work to establish an application background for our software implementation work.

A.1 Mathematical Finance

This area (also known as *Financial Mathematics* or *Quantitative Analysis*) is concerned with the application of mathematics to finance and is mainly focused on building the theoretical basis for mathematical models used to describe financial phenomena that are observed in the markets. The area is mainly based on methods from statistics and econometrics. and deals with analytical modelling of evolution of financial systems over time. The theory is mainly based on *Stochastic Calculus*, where stochastic and partial differential equations are formulated to model behaviour of financial assets. These quantitative models are then applied on extremely large datasets to analyse financial markets and instruments traded in them.

A.2 Computational Finance

This area is an applied field of computer or computational science, which deals with handling large volumes of the financial data and developing suitable algorithms that are applied to mathematical models to analyse financial markets and securities. This field most often requires massive computational effort to extract knowledge from raw data. The majority of research in this field is focused on the techniques for finding accurate solutions to theoretical models. We achieve it through efficient numerical approximation methods that can be computed under feasible and finite time constrains. For instance, we always apply numerical methods when no analytical solution exists represented by a sufficiently accurate closed-form formula, that can be calculated immediately. In fact, most of the mathematical models that are used currently in finance and engineering are too sophisticated to be solved analytically, and thus can only be solved using a numerical method. This greatly emphasises the importance of this field.

Computational Finance studies various applications such as:

- financial data engineering, which involves gathering, processing, and storing data in large amounts originating from various sources,
- validating data consistency and completeness, so it can be used in production,
- back-testing investment and trading strategies,
- optimising computational algorithms for fast performance and efficient resource use.

A.3 Financial Engineering

In broad sense, this field puts the theoretical models and scientific methods to a test in practice by applying them to realistic workflows in the financial industry. In this sense, Mathematical and Computational Finance can be considered as subfields of Financial Engineering. This field explores how to best match the proposed scientific and engineering tools with the requirements and needs of the practitioners. Obviously, one size does not fit all, as use cases and the necessary depth of sophistication in modelling varies across the investment managers. The techniques need to be adapted to suit their needs.

Some of the common financial applications are:

- financial instrument trading and speculation,
- pricing financial assets and their derivatives,
- hedging the investment portfolio risks,
- portfolio risk management,
- portfolio performance analysis and reporting,
- building custom portfolios for clients based on instruments available in the market,
- development of algorithmic trading strategies,
- financial restructuring for the need of corporate finance,
- underwriting liability insurance,
- or creation of new bespoke financial instruments through scripting techniques.

All three fields are strongly entangled and involve a multi-disciplinary mix of skills in finance, mathematics, and computer science. In addition, financial engineers need to understand not only a quantitative, but also qualitative side of the problem at hand. They have to understand holistically what is the role and the impact of their actions on the financial system. Here, we identify two main forces driving the development of financial modelling, which are areas of an active practitioners' and research work today. The first is a thriving field of *Behavioural Finance*, that studies the influence of psychology on behaviour of financial investors and analysts, often used to asses investor's risk appetite or analyse a general market sentiment that is then fed into algorithmic trading strategies. The second is the highly involved regulation process that governs the current markets and is described in more detail in the next paragraph.

Appendix B

Business Concepts

B.1 Financial Industry

The efficiency requirements of financial applications are determined by the specific use cases and data complexity of the market participants that use them. In this work, we take a perspective of the *buy-side*, the large investment managers that invest their multi-billion $ assets under management in all types of financial markets. Such a group of investors is primarily concerned with sustaining their assets. Thus, they are mainly concerned with risk management of their investment portfolios at any given time. In contrast, the *sell-side*, with banks being a primary example, operate creating and selling new financial instruments for the *buy-side* investors to buy. Banks are thus primarily concerned with accurate prices for the financial instruments, that they offer to their clients. Thereby they adjust the margins that they charge and yield profits.

B.2 Investment Managers

There are three main groups of investment managers: asset owners, asset managers and asset service providers. In general, they need common software functionalities, but they differ fundamentally in the nature of their business, which is resembled in the ratio between portfolios and number of instruments they deal with. For computational point of view, it has a direct impact on the scale and complexity of the algorithms. In the context of investment management, asset owners are financial organisations who own assets under management and their business is dependent on how they manage them. Example organisations are life and pension funds, sovereign wealth funds, or insurance companies. They usually manage a small number of well-diversified portfolios, but each of them comprise thousands of different financial instruments. In contrast, asset managers are financial organisations that, as part of their business, manage assets and investments on behalf of their clients, who themselves are asset owners. Example organisations are mutual funds for private investors, asset management departments for institutional investors, wealth management for retail investors with large assets, and national treasury management. They usually manage thousands of different portfolios, but each of them is more focused and comprise significantly smaller number of different instruments. Moreover, the instruments tend to be much more complex, because they are custom-tailored for clients of the asset managers. Asset service providers, such as custodians, provide different financial services to their clients to support the operational and administrative aspects of investment management process.

B.3 Regulation

Investment managers around the world need to adhere to the rules laid out by different regulatory authorities if they want to be involved in trading and managing investments in the public markets. We differentiate a set of global rules that every financial organisation needs to follow as well sets of local (national or regional) market rules. Regulation continues to broaden and deepen as public sentiment becomes less and less tolerant for any appearance of preventable errors and inappropriate business practices. Some examples of regulatory frameworks that in the recent time came into force in the European Union are:

Basel I–IV A constantly evolving four-pillar legislation for the banking industry that sets out rules governing minimum capital banks must hold against market and credit risk exposures. The most recent is the Basel Pillar IV that introduces Fundamental Review of the Trading Book (FRTB).

Solvency I–III A three-pillar directive for the insurance industry that concerns the amount of capital they must hold to reduce the risk of insolvency.

The regulations oblige the investment managers to introduce the risk models that are a subject of this work. As a result, they experience an increased reliance on computational resources as they forced to periodically report their assets and risk measures to abide with the regulation.