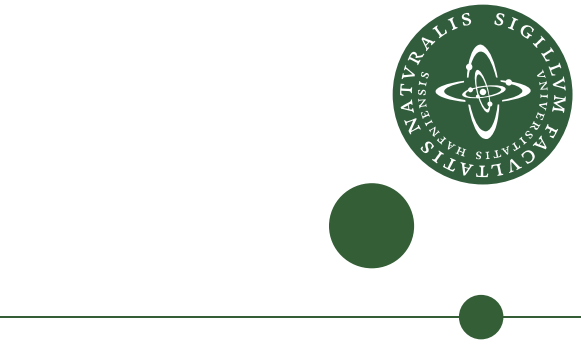
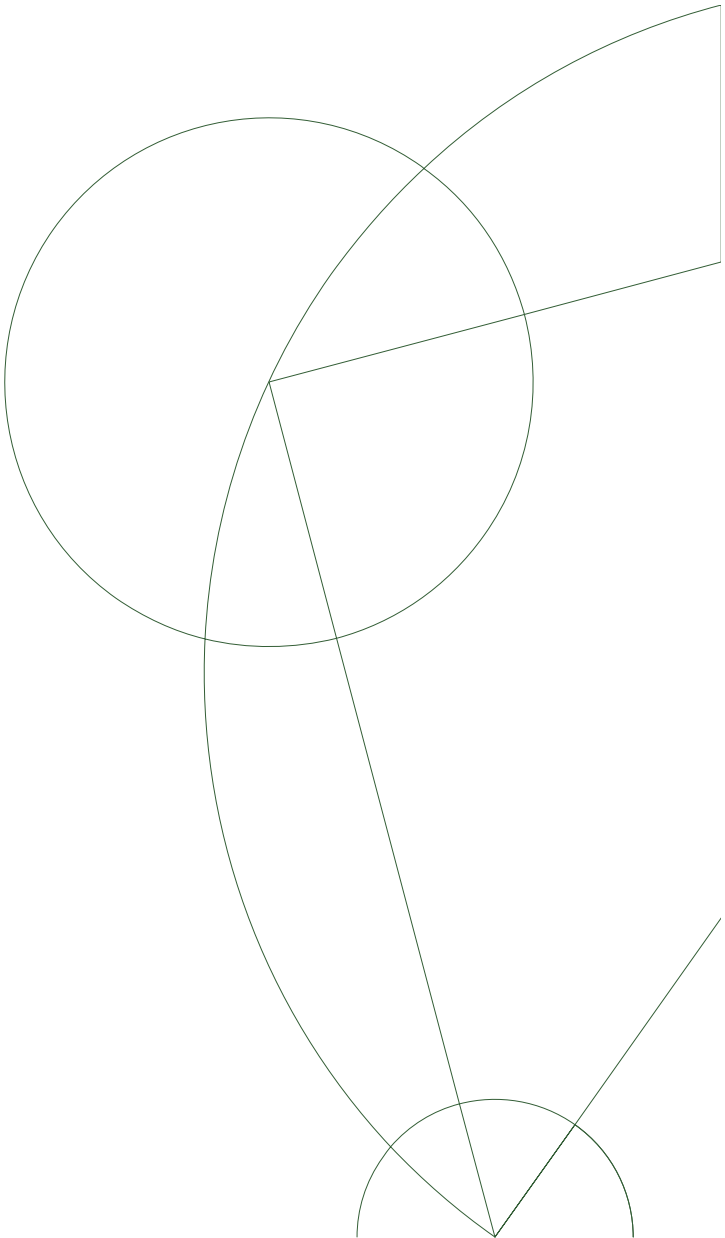
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN



Data-Parallel implementation of Monte Carlo Based Financial Risk Calculations

Master's Thesis

Frederik Bøgelund Larsen - mgv295



Department of Computer Science
University of Copenhagen

Supervised by - Martin Elsman

August 6, 2018

Abstract

Exploiting the increasing amounts of computational power, that is available on modern parallel hardware, is an ongoing research topic. The issues is to make parallel programming more accessible, maintainable and efficient for use in in the real world. Many solutions have been proposed, ranging from library support for existing mainstream programming languages [14] [16], while outer solutions involve entirely new programming languages. In this thesis the Futhark programming language [12] [9] is examined in the context of writing financial applications, that contain many compute-intensive problems that are inherently parallel, and can therefore benefit enormously in terms of performance, when executed in parallel [1]. One such compute-intensive task is simulation of the financial market, in order to price products and calculate risk factors, in particular Monte Carlo simulation. This thesis describes how a small Value-at-Risk (VaR) engine is creating, using Futhark for Monte Carlo simulation, for pricing of financial products. Benchmarks for the resulting VaR-engine shows significant performance improvements when executed on parallel hardware, compared to sequential execution. Furthermore this thesis argues that while Futhark shows promising results in terms of the complier technology, there are still some limitation in terms of expressiveness, and the programming model associated with the language and parallelism, that requires the user to have certain expertise and knowledge about parallel algorithms.

Contents

Chapter 1

Introduction

Modern financial institutions use large systems for business critical applications, such as risk calculation engines for VaR (Value at Risk). These engines run thousands of calculations that are highly compute-intensive, use large amounts of computational resources, and require the financial institutions to acquire and administrate large amounts of physical machines, which use large amounts of energy in terms of actual power consumption and require specialists to manage.

These business critical systems are built from different components, many of which are what would be called legacy systems, in that they have been a part of the operational system for typically several decades, and were often built with technologies that were not designed to work efficiently on modern hardware. For further development of these systems, other technologies have been introduced along side the old technologies, requiring complicated exchange of data between different programming environments, resulting in wasted resources and potential utilization of the available modern hardware, such as GPUs (Graphical Processing Unit).

Since the early 2000s, utilizing GPUs to perform non-graphical calculations has been given more attention in research and practical use under the term GPGPU (General-purpose computing on Graphical Processing Unit). Since GPUs have a different architecture designed towards highly parallel computations, in particular image generation for graphical applications, traditional algorithms and programming models do not map directly. Research on parallel programming on GPUs in general has seen increasing interest. There are many examples of algorithms designed to take advantage of the architecture of GPUs, which sometimes show an increased performance by an order of magnitude or more. Application of GPGPU, is mostly found in

areas such as Financial programming, Machine Learning, and other scientific areas with demanding numerical calculations.

At the Computer Science Department of the University of Copenhagen, the research center HIPERFIT (High Performance Programming for Financial IT), which includes several areas of research, conducts research in computation on highly parallel hardware. Futhark is one of the products of the research at HIPERFIT. Futhark is a Purely Functional Data-Parallel programming language, which compiles down to efficient code to be run on GPUs. Futhark has a highly optimizing compiler that transforms the program in such a way that it takes advantage of the parallel capabilities of the hardware, while still emphasising a simple purely functional programming model.

The motivation for this thesis is to leverage the capabilities of Futhark, and to create a proof of concept risk calculation engine, for a business critical area in financial institutions such as calculation of Value at Risk (VaR). The goal is to show that the model provided by Futhark is viable in the context of a financial institution described in the first section, such that these institutions can use the results and eventually move towards such a parallel model for their systems. In particular the computations of VaR has been identified by the HIPERFIT partner SimCorp as a challenging computation, which without the proper data-parallel processing can take hours to compute for customers with large portfolios [5].

1.1 Problem statement

Will programming financial systems with high performance demands, benefit from implementations in a language such as Futhark? How can such a system be structured to take advantage of capabilities of Futhark in regards to GPGPU programming and a Parallel Functional Programming model in general? Specifically how can risk calculations such as VaR (Value at Risk) be mapped to efficient algorithms that takes advantage of the highly parallel modern hardware available. Efficient calculations of VaR involves Monte Carlo methods for simulation, commonly used in finance. Monte Carlo methods for finance in particular involves simulation of many different scenarios or paths. The nature of such a method involves many nested computations with an irregular structure, which classifies the problem as an irregular nested data parallel problem, an area of research most noticeably explored in the NESL programming language, as with a general model for dealing with nested parallelism. Compared to NESL, Futhark has a real

implementation on GPUs, with a restricted but often sufficient model for dealing with nested parallelism. Is the nested parallelism support in Futhark viable for financial computations specifically VaR, using Monte Carlo methods.

1.2 Course of action

To examine the above problem statement, a VaR engine in Futhark will be created, capable of calculating the VaR of a small portfolio of financial products. To review the performance and the amount of parallelism achieved, benchmarks will run on several platforms examining various parts of the program, using sequential execution of the same programs as baseline. To review the correctness of the VaR engine, test comparing simulated prices with actual prices will be provided. The structure of the Futhark program will be reviewed i terms of how the different constructs provided by Futhark are sufficient for programming financial applications.

1.3 Related work

The question of how to make GPGPU programming more accessible and maintainable for real-world scenarios is an ongoing research area, resulting in various approaches. Futhark is one such approach by creating a new programming language with many of the constructs found in general purpose languages, and letting the user define the structure of the programs resembling the problem domain [1]. The focus of Futhark is more towards the technology with a highly optimizing compiler that other languages can take advantage such as in [10] where APL is transpiled into Futhark. Another example of targeting Futhark as a backend can be found in [2], where a DSL for pricing financial contracts provides a declarative approach of GPGPU programming.

1.4 Readers guide

This thesis is comprised of three chapters. The first chapter introduces the theory behind the financial domain used including options, VaR and Monte Carlo simulation. The second chapter introduces the concepts of parallelism including paradigms, cost model and an introduction to the Futhark programming language. The third chapter present a VaR engine created using

Futhark, with a deatiled overview of design, results of experiments and test results.

Chapter 2

Financial Domain

The following chapter will introduce the theoretical knowledge needed in order to create a program to calculate VaR. First options are introduced, in particular barrier options, that will serve as the financial instrument on which VaR calculations will be based. Hereafter an introduction to VaR is given. Lastly Monto Carlo simulation is introduced, in the context of VaR.

2.1 Vanilla Options

An *option* is a financial instrument that gives the holder of the option the right, but not the obligation to exercise the option at a given maturity date. There are two types of what are known as *vanilla options* or *European options*, namely call and put options. A call option gives the holder the right to buy the underlying asset at what is known as the *strike price* at a given date. A put option gives the holder the right to sell at a given strike price on a give maturity date [13, p. 7].

2.1.1 Valuation

Valuation of vanilla options is mostly based on the Black-Scholes model, which provides a formula that enables a fair pricing of call and put options respectively. The Black-Scholes model is based on some key assumptions.

- Markets are efficient meaning they cannot be predicted
- Volatility of the underlying asset and interest rates are constant
- Returns are normally distributed

The formulas for option pricing based on the Black-Scholes model, are derived from the following formula

$$dS = \mu S dt + \sigma S dz \tag{2.1}$$

assuming the price of the underlying asset S follows a *Geometric Brownian motion*, where σ is the volatility of the rate of return, μ is the expected return rate of the underlying asset, and dz is a *Wiener process* The following formulas are for pricing European call (C) and put options (P) [13, p. 313].

$$C = S_0 N(d_1) - X e^{-rT} N(d_2) \tag{2.2}$$

$$P = X e^{-rT} N(-d_2) - S_0 N(-d_1) \tag{2.3}$$

where

$$d1 = \frac{ln(S_0/X) + (r - \sigma^2/2)T}{\sigma\sqrt{T}}$$
$$d2 = \frac{ln(S_0/X) + (r - \sigma^2/2)T}{\sigma\sqrt{T}} = d1 - \sigma\sqrt{T}$$

S_0 = Asset price a start time t
X = *Strike price*
r = *Risk-free interest rate*
σ = *Volatility* of the relative price change for the underlying asset
T = Time to the options *Maturity* date in years
$N(\cdot)$= *Cumulative normal distribution function*

The payoff for European options can be expressed as $max(0, S - X)$ for call options, and $max(0, X - S)$ for put options. If we have a payoff that is greater than zero, we say the option is *in the money*, and if the payoff is zero or below the option is *out of the money*. In practice however, options come with a premium that is not accounted for in the payoff functions, therefore making a profit requires a higher payoff.

2.2 Barrier Options

Barrier options belong to a set of instruments commonly known as *Exotic options*, that share the characteristic that they depend on the behaviour of the price of the underlying asset during the option's lifetime.

Barrier options are different from vanilla options, in that they have an additional parameter called the *barrier*, denoted S_b. The barrier acts as an activation point, based on the underlying asset price, deciding if there is any payoff or not. There are two types of barriers, *knock-in* and *knock-out*. Knock-in means that the option becomes active when a certain price for the underlying is reached, conversely knock-out means the option becomes worthless if a certain price is reached. We also use the terms *down* and *up* to categorize Barrier options where down means that $S_b < S_0$, and for up we have that $S_b > S_0$. A *down-and-out call* options, is a call option that becomes worthless when the underlying asset price goes below the barrier, therefore it would be a knock-out option. Another example is an *up-and-in put* option, which is a regular put option that only becomes active once the barrier is hit [13, p. 579].

2.2.1 Valuation

Barrier options are path-dependent, meaning that the valuation is dependent on all monitored asset prices for given a path from t_0 to T, since the asset price might hit the barrier at any point. Each type of barrier option features its own payoff function. The payoff function for a barrier option is defined based on the type of option (i.e., in/out, down/up and call/put). If we consider a *down-and-out call* option, we have the payoff function $max(0, S_T - X)$, where X is the strike price, and S_T is the last price on the path. Since we are dealing with a call option, we are interested in having as high an S_T as possible. Closed formulas for calculating barrier options are also derived from the Black-Scholes model. There are 8 different formulas, one for each type of barrier option. The following formula is used to value a *down-and-out* put option.

$$\begin{aligned} p_{do} = &-Se^{-q(T-t)}(N(d_3) - N(d_1) - b(1 - N(d_6))) \\ &+ Xe^{-r(T-t)}(N(d_4) - N(d_2) - a(N(d_7) - N(d_5))) \end{aligned} \tag{2.4}$$

where

$$d_1 = \frac{log(S_0/X) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}, \quad d_2 = \frac{log(S_0/X) + (r - \sigma^2/2)T}{\sigma\sqrt{T}},$$
$$d_3 = \frac{log(S_0/S_b) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}, \quad d_4 = \frac{log(S_0/S_b) + (r - \sigma^2/2)T}{\sigma\sqrt{T}},$$
$$d_5 = \frac{log(S_0/S_b) - (r - \sigma^2/2)T}{\sigma\sqrt{T}}, \quad d_6 = \frac{log(S_0/S_b) - (r + \sigma^2/2)T}{\sigma\sqrt{T}},$$
$$d_7 = \frac{log(S_0X/S_b^2) - (r - \sigma^2/2)T}{\sigma\sqrt{T}}, \quad d_8 = \frac{log(S_0X/S_b^2) - (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

and where

$$a = \left(\frac{S_b}{S_0}\right)^{-1+\frac{2(r-q)}{\sigma^2}}, \quad b = \left(\frac{S_b}{S_0}\right)^{1+\frac{2(r-q)}{\sigma^2}}$$

The above formula assumes continuous time monitoring. In practice, however, such financial instruments are monitored in discrete time, such as once a day on the time of market closing. This discretisation will make the price of the option cheaper, since there is a lower risk of the option hitting the barrier. To apply discrete time monitoring, however, we need a correction value for the valuation formulas, to approximate the lower risk. An approximation has been derived in part from the Riemann zeta function [3, p. 534] in order to correct the barrier level, resulting in the following formula

$$S_b = S_b e^{\pm 0.5826 \cdot \sigma\sqrt{\delta T}} \tag{2.5}$$

2.3 Value at Risk

2.3.1 Risk

Value at Risk (VaR) is a method for determining the overall risk exposure of a trading book, in the context of a bank, portfolio or other financial entities. The overall goal is to discover the largest amount of loss that can be expected, with a certain level of confidence. More formally we define VaR as follows

> **VaR is a measure of market risk. It is the maximum loss which can occur with X% confidence over a holding period of t days** [4, p. 30]

Assuming we have portfolio with a VaR of $100,000, where we have set a confidence level of 95%, then there is a 5% chance that the loss on one trading day will be greater than $100,000 [4, p. 30]. In other words, VaR aims to provide an upper-bound on the loss on a portfolio, that can be expected for given period of time, with a certain level of confidence.

2.3.2 Calculating VaR

Although there are different approaches for valuation of a portfolio used in calculating VaR, they follow the same general steps [4, p. 36].

1. Value the portfolio at t_0 giving the current value.
2. Revalue the portfolio at time T, given the value of the portfolio for the time horizon the VaR calculation seeks to predict.
3. Revalue the portfolio with a number of alternate market factors, resulting in a distribution of possible value changes. This enables VaR to be expressed in terms of a confidence level.
4. Calculate the maximum that can be lost based on the confidence level and the given time horizon.

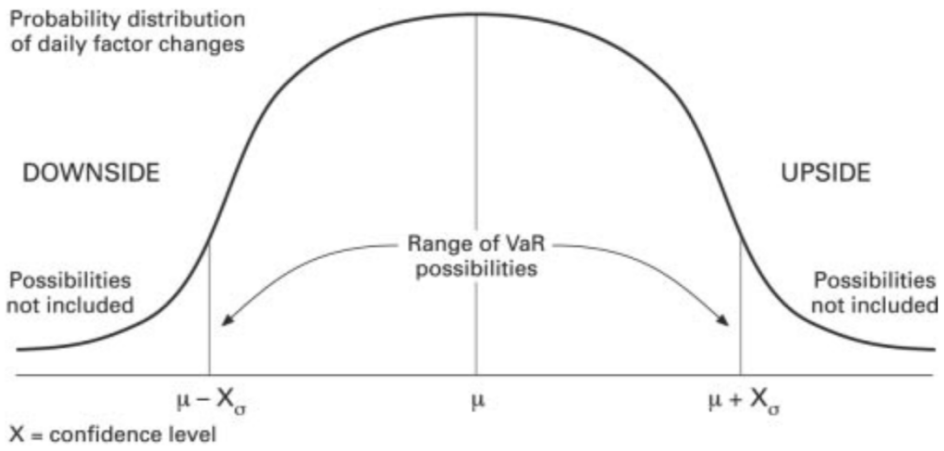


Figure 2.1: VaR in the context of the normal distribution [4, p. 36]

In order to calculate VaR, we need a method to value the current portfolio, and afterwards revalue the same portfolio with different market variables.

There are servaral different approaches, such as the *historical method*, that as the name implies use previous market changes to estimate the future value of the portfolio. A different and more flexible method is to use Monte Carlo simulation, where we simply simulate the market up until time T, and value the portfolio at time T.

As an example assume a portfolio of simple assets with a value of \$100,000, and we assume that returns are normally distributed. The volatility of the portfolio is 3% and we wish to calculate VaR with a 95% confidence level, which is equivalent to 1.6449 standard deviations, we get the following calculation

$$\$100,000 \cdot 0.03 \cdot 1.6449 = \$4934,7 \tag{2.6}$$

If we wish a confidence level of 99%, which is equivalent to 2.3263, the VaR will increase accordingly.

$$\$100,000 \cdot 0.03 \cdot 2.3263 = \$6978,9 \tag{2.7}$$

2.4 Monte Carlo Simulation

In *Monte Carlo* simulation, we aim to approximate a given deterministic quantity, using randomness to sample the problem space. In practice it means that we are approximating an expected value $E(X)$, by generation a large amount of samples, and taking the average of the results. The method is largely supported by *the law of large numbers*, a central result in probability theory, that states that the average of a large sample size, will be close to the actual value.

Suppose we wish to estimate a value μ then we have

$$\mu = E(g(X)) \tag{2.8}$$

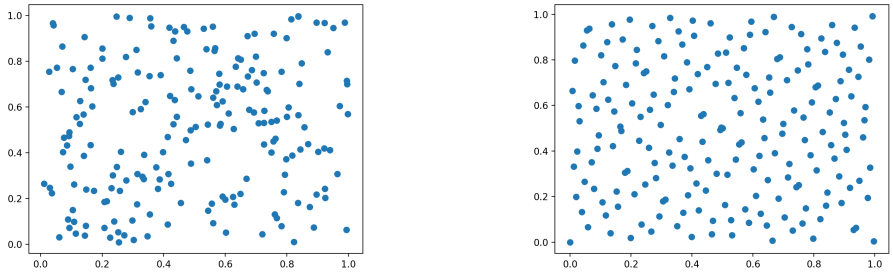
Where g is some function. Now we generate n independent random samples $X_1, X_2, .., X_n$, and we provide the follwing Monte Carlo estimator

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} g(X_i) \tag{2.9}$$

By the law of large numbers we have that when $n \to \infty$, $\hat{\mu} \to \mu$

2.4.1 Low-discrepancy sequences

One of the core problems with Monte Carlo simulations is convergence, in regards to how many simulations needs to be performed before it results



(a) Random numbers

(b) Sobol sequence

Figure 2.2: Comparison of 200 points generated from random numbers and sobol sequences

in an estimation close to the expected value. It has high importance also in regards to performance, if the number of paths needed to be simulated can be reduced. The standard naive approach is to use pseudo-random numbers for sampling, but this approach often requires a high number of simulations, since pseudo-random numbers tends to create local clusters, and therefore cover a smaller area of the distribution. An alternative approach is to use a *low-discrepancy sequence*. Low-discrepancy sequences has the property of having values that are almost perfectly distributed in a grid-like pattern, without clustering or having duplicate values. Figure 2.2 clearly shows how 200 points generated from a low-discrepancy sequence are more evenly distributed. They are not random in nature, and will produce the same values if the same sequence is requested. Several low-discrepancy sequences exist such as *Halton* and *Sobol*. but the focus in this thesis will be entirely on *Sobol sequences*. Both Sobol and Halton are based on the generalization of the *Van der Corput one-dimensional* sequence. The idea behind Van der Corput is illustrated in the following [3, p. 382].

We represent the number n in the base b

$$n = (...d_3d_2d_1d_0)_b \tag{2.10}$$

We then represent the base b version of n in decimal, by prefixing "0." and preserving the base representation

$$h = (0.d_0d_1d_2d_3...)_b \tag{2.11}$$

The formal equation for finding the n'th number in a Van der Corput sequence is given by

$$V(n,b) = \sum_{k=0}^{m} d_k b^{-(k+1)} \tag{2.12}$$

As an example, if we wish to find the number 2 in a Van der Corput sequence in base 2 (i.e. binary), we have the binary representation for 2 $(...0010)_2$, and the reversed decimal representation $(0.0100...)_2$, and, using formula 2.12 we have $V(2,2) = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 0.25$.

Sobol sequences

A Sobol sequence is similar to a Halton sequence and based on the Van der Corput sequence, but with the key difference of only using base 2 as its integer expansion, which has the effects of making the Sobol sequence more regular in higher dimensions, and also performing better on computers because of their binary structure.

A Sobol sequence is generated using so called *direction numbers*, which is a set of numbers less than 1.

First we consider the representation of the integer n in a Sobol sequence, which is similar to that of the Van der Corput sequence, but strictly in base 2(i.e. binary) [3, p. 390].

$$n = (...b_3 b_2 b_1 b_0)_2 \tag{2.13}$$

Retrieving the n'th Sobol number is done by computing the bitwise exclusive OR with a direction number corresponding to each index in the binary representation.

$$x^{(n)} = b_1 v_1 \oplus b_2 v_2 \oplus ... \tag{2.14}$$

The direction numbers need to be chosen properly in order for a low-discrepancy sequence to be generated correctly. All direction number are initialized with odd integers such that $0 < m_i < 2^i, i = 1...b$. Generating the direction numbers, involves exploiting the properties of *primitive polynomials*, meaning polynomials that cannot be reduced further, and the fact that they have binary coefficients.

$$P = x^d + a_1 x^{d-1} + \ldots + a_{d-1} x + 1 \tag{2.15}$$

where an example of such a polynomial could be

$$x^3 + x + 1$$

The direction numbers are generated using the following formula.

$$m_i = 2a_1 m_{i-1} \oplus 2^2 a_2 m_{i-2} \oplus \ldots \oplus 2^{d-1} a_{d-1} m_{i-d+1} \oplus 2^d m_{i-d} \oplus m_{i-d} \quad (2.16)$$

where $m_1 \ldots m_d$ are numbers chosen arbitrarily to initialize the recurrence given they are odd and $m_i < 2^i$. Direction numbers can also be expressed as the following binary fraction.

$$v_i = \frac{m_i}{2^i} \quad (2.17)$$

After generating the direction numbers, a Sobol sequence can be generated. An optimization using a Gray Code representation of n, is often used. The benefit of using a Gray Code representation is that the generation of a Sobol sequence becomes simpler in that given the n'th Sobol number in a sequence x^n, we have that $x^{n+1} = x^n \oplus v_c$, where c is defined as the index of the rightmost zero bit b_c in the binary representation of n [3, p. 393].

2.5 VaR using Monto Carlo simulation

Estimation of VaR as outlined in Section 2.3.2 can relatively simply be applied using Monto Carlo simulation. However, since the aim of the VaR engine is to enable estimation of VaR on a portfolio containing barrier options, there are some additional concerns to address. The first step is to value the portfolio at time t. We Denote the value of the portfolio as $V(S(t), t)$, where S is a vector of risk factors such as interest rates or asset prices. As part of the simulation the risk factors are changed after time steps of δt , and the portfolio is priced again with the new risk factor after a shock has been applied giving us

$$\delta S_i \equiv S_i(t + \delta t) - S_i(t), i = 1, ..., n. \quad (2.18)$$

Where S_i denotes the current value of risk factor at time $t + \delta t$ such that $S_i = S_i(t), i = 1, ..., n$, and δ_i denoting the shock such that $\delta_i = \delta S_i$ [3, p. 601]. In this thesis we will focus on asset prices as the only risk factor that will change for each simulation of the portfolio value. For VaR we wish to calculate the value of the portfolio at δt time steps ,where δt could be one day or twenty days in the future. We will generate n number of market scenarios based on the asset price S_0, giving us n number of paths going from t to $t + \delta t$. For each path we will take the last price and use this a the asset price for further pricing, again using Monte Carlo simulation, where the change in asset price acts as the shock. Since we are working

with path dependent options, in particular barrier options, we have to look at the entire path including the path from t to $t+\delta t$, and not only the path from t to $t+T$. This is because the price might hit the barrier at any given point on the path, therefore if the barrier is hit between t and $t+\delta t$ it has already become worthless or active. The following list summarises the steps involved.

1. Value the portfolio at time t giving us $V(S,t)$
2. Simulate paths for S from t to $t+\delta t$
3. For each simulated path ending at $t+\delta t$, use the asset price at $t+\delta t$, and again simulating paths using Monto Carlo simulation from $t+\delta t$ to $t+T$, but pricing the entire path from t to $t+T$ returning S_i.
4. From each result in S_i, calculate the loss L_i using

$$L_i = V(S_i, t+\delta t) - V(S,t) \tag{2.19}$$

5. Sort L_i and return the value at the index corresponding to the required percentile, giving the VaR

Chapter 3

Parallelism

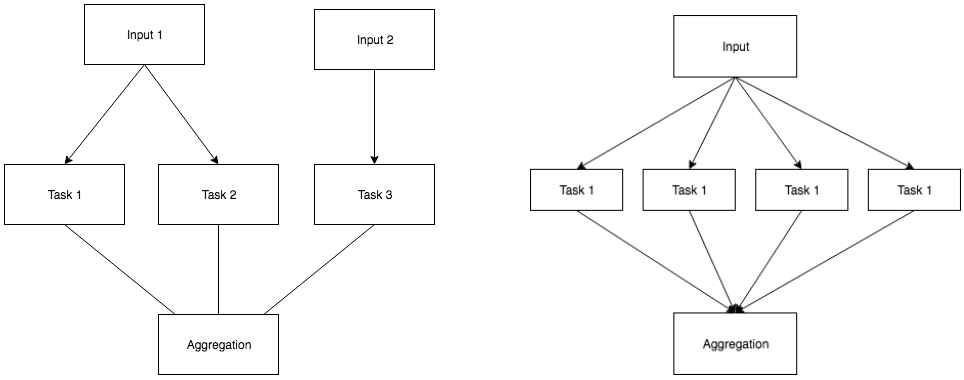
The following section presents an introduction to parallelism and parallel programming in general. A cost model for how to reason about parallel algorithms is introduced in detail. This is follow by a brief introduction to GPUs, since it is the only parallel hardware target in this thesis. Lastly a detailed introduction to the Futhark programming language is given, including an overview of language constructs and tools.

3.1 Parallel Programming

Parallel programming is the concept of *performing at set of computations simultaneously*, such that a program performs a given computation using multiple computing resources, in contrast to *sequential programming* where execution is done on a single computational resource. If broken down into steps, a parallel program execution has the following steps

1. Break the data into equally sized chunks, typically one chunk per computational unit.
2. Execute the program concurrently one each computational unit
3. Collect the results with a given reduction
4. Return the result to the main thread of execution

Parallelism is often confused with *Concurrency*, since the concepts are related. Concurrency is the coordination of multiple *processes*. By a *process* we mean some unit of a program, such as a function. Another aspect of concurrency is communication between the processes to coordinate program



(a) Task-parallelism (b) data-parallelism

Figure 3.1: Illustration of task-parallelism vs data-parallelism

execution. It is important to note that a concurrent program does not need multiple CPUs, since concurrency is purely about program structure, therefore execution of a concurrent program on a single CPU will interleave the execution, such that longer running processes will be handled by secondary threads, where the main program can continue on the main thread of execution.

Parallel programming is by definition deterministic, in that a parallel computation will result in the same output on every execution, also if the program is executed sequentially on one CPU. This aspect is in contrast to concurrency that by definition is non-deterministic, since we cannot necessarily expect the same result, or any result at all on each execution. A typical example of non-determinism in the context of concurrency is calling a web-service, since we cannot expect it to return a result on every request, since the web service might not respond.

Within the domain of parallel programming, there are two main paradigms, *task-parallelism* and *data-parallelism* that are defined as follows.

- **Task-parallelism** - simultaneous execution of multiple parts of a program typically a function, on multiple datasets, on multiple computational units.
- **Data-parallelism** - simultaneous execution of a single part of a program on multiple computational units, with each unit processing a separate part of the data.

The different paradigms vary in the applications in which they are ap-

plied where Task-parallelism is typically applied in a scenario where we have multiple different tasks, say $task_1, task_2, \cdots, task_n$, that we wish to process in parallel. All these tasks might take different datasets as parameters, and produce different datasets as results. Data-parallelism is typically applied in applications with high data-processing demands, such as scientific computing or other domains where there are a large amounts of numerical calculations.

3.1.1 Cost model

When analysing the theoretical efficiency of a parallel algorithm, we use the measures *work* and *span* and the number of processors. When reasoning about the running time of parallel algorithms, we assume an ideal parallel computer with P processors. We denote the total running time of a parallel algorithm on P processors as T_p. Work is the total running time for the execution of the parallel algorithm on a single processor denoted by T_1. Span is the length of the longest execution path, also called the *critical path* of the parallel algorithm denoted by T_∞. Work and span provides a way to reason about the lower-bound of the running time of a parallel algorithm executed on P processors. On an ideal computer with P processors, we can only perform P work per processor, that is we can perform at most $P \cdot T_p$ work. Because we have total work of T_1 on one processor, we must have $P \cdot T_p \geq T_1$, and then by dividing with P we get what is called the *work law* in 3.1.

$$T_p \geq T_1/P \tag{3.1}$$

An ideal computer with P processors cannot be faster than an ideal computer with an infinite number of processors. Therefore the running time T_p is bounded by T_∞. This is called the *span law*.

$$T_p \geq T_\infty \tag{3.2}$$

For measuring the actual speed-up of a parallel algorithm, we have the ratio T_1/T_p of the running time on one processor, and the running time on P processors. The speed-up is bound by the number of processors available such that $T_1/T_p \leq P$, meaning we cannot achieve a theoretical speed-up greater than the number of processors available on an ideal parallel computer. If we look at a simple example such as the `map` function that maps over a sequence of size n and applies a function f to each value x. We assume here the programming language providing `map` is purely functional, and no side effects can occur. To make a parallel version of `map` we would split the input

into P equally sized chunks, and execute each chunk on each processor, since we do not have any data dependencies. Assuming f has $O(1)$ running time, the work of the algorithm would be $O(n)$ since we would map through a sequence of length n on one processor. The span however would be $O(1)$ since the algorithm only splits the sequence once, making the length of the longest execution path 1. Therefore we have the speed-up $T_1/T_p = \theta(P)$, thus we have *linear speed-up* [6]. However, theoretically possible, linear speed-up as the number processor increase is rarely achievable, because of the overhead associated with orchestrating execution on multiple cores.

3.1.2 GPUs

GPUs (*Graphical Processing Units*), have since the 90s becomes more and more commonplace in PCs, because of the increase in applications with high graphical processing demands, in particular video games, and are now available in almost all personal computing devices from mobile phone to tablets and low-end laptops. Over the last couple of decades the processing power of GPU has increased and their inherit architecture of a highly parallel piece of hardware, have made them very well suited for parallel programming applications, beyond those found in the graphical processing domain, giving rise to the term *general-purpose GPU programming* (GPGPU) [9, p.10].

A Program on a GPU is called a *kernel*, and is launched by the host system, which is the a CPU-controlled computer in the conventional sense. A kernel contains some amount of threads, that are independent where each thread executes the same sequential program. Since each thread performs the same sequential program, the number of threads is the quantity that determines the amount of parallelism expressed in the program. The limiting factor of execution speed however is memory access, therefore access patterns are very important. Modern GPUs have several hierarchies of memory

- **Registers** - Small memory local to each thread for holding scalars and small arrays
- **Local memory** - Small memory shared among a group of threads.
- **Global memory** - Large GiB sizes memory accessible for all thread groups

The nature of the program whether it being a memory- or computational-bound problem is therefore critical in the amount a speed-up achievable by solving the problem in parallel [9, p.60].

Programmering interface

Writing programs to target parallel hardware also referred to as *accelerators*, are commonly achieved using specialized frameworks, that impose a specific programming model. There are two such framework that are most commonly used, namely *CUDA* and *OpenCL*. CUDA is a proprietary framework, developed by the GPU manufacturer Nvidia, targeting their line of products. CUDA is specific to Nvidia hardware and various software requirements. OpenCL is an open standard for a programming interface for executing programs not only on GPUs, but also on other heterogeneous computing platforms. In contrast to CUDA , OpenCL supported on more hardware and by more compilers, however, OpenCL still requires driver support. They both have in common that they provide a C like imperative programming language, on top of regular C/C++, that is hard to program.

3.2 Futhark

There are various issues regarding provided the programming interfaces such as CUDA and OpenCL, that are a large source of complexity which makes it challenging to write programs that take advantage of the available parallel hardware.

- **Programming languages** - As mentioned, imperative C like programming languages are required to write parallel programs. This of course has the same problems associated with imperative programming in a sequential context, but with the added complexity of expressing the problem in a way that can be maintained, be data-race and bug free, while also performing well on the targeted hardware.
- **Hardware** - Parallel hardware has very different characteristics. Therefore the programmer needs a certain amount of expertise and knowledge about the hardware.
- **Parallel algorithms** - Algorithm design has to be done from the perspective of the parallel cost model. This means many standard algorithms has to be rewritten in order to be efficient and take advantage of the hardware.

The Futhark programming language [12] [9] aims to tackle these challenges by providing a different model of programming, than the ones provided by OpenCL and CUDA at a programmer level. This is done by providing

the programmer with high level constructs to express parallel programs, while an optimizing compiler maps the final executable down to an efficient representation for parallel hardware. Futhark is a purely functional array programming language, based mainly on the syntax and concepts of the Standard ML programming language. Futhark provides all basic constructs for expressing programming problems, such as value declarations, functions and records, but also special parallel programming constructs, that will presented in Section 3.2.1. The feature set does restrict Futhark to work as a special purpose language that is highly optimized at targeted for high performance parallel programming tasks.

3.2.1 Parallel programming constructs

Futhark supports data-parallelism, explicit in the constructs provided, but with sequential semantics enabling programs to be understood entirely as a sequential series of steps. This is not unique to Futhark, but an overall property of being a purely functional programming language, which enables transformations of the resulting executed code to make it operationally parallel [9]. As mentioned Futhark provides a set of *second-order array combinators* (SOACs), which gives the programmer high level functions for programming with arrays. The most commonly used SOAC is `map` which is available in most functional programming languages, that takes an array and applies a pure function to each element, producing a new array. It is trivial to see how `map` can be run in parallel over multiple execution units, by splitting an array into a number of chunks corresponding to the number of CPUs, for example. Futhark also provides other core SOACs such as `filter` selection and `reduce` for reduction. It is important to note however, that some SOAC has certain constraints in order to guarantee parallel execution in a deterministic manner, such as `reduce`, which requires the reduction function to be associative, and a neutral element need to be supplied. This falls upon the programmer to insure that constraints are met, since the properties are not proved by the compiler. Futhark has also support for *nested parallelism*, where a parallel function executed in parallel itself can contains parallel code such as a `map`. This nesting is achieved through a flattering algorithm, that in essence transforms a nested parallel program into a flat data-parallel problem. Flattering has been pioneered by the NESL programming language in particular.

3.2.2 Code transformations

As mentioned, Futhark can perform certain optimisations on the resulting code. The primary of such transformation is *fusion*. Assume we have a Futhark program using two `map` operations on an input array shown in Listing 1. It has been shown that applying one `map` as input to a second `map` is equivalent to applying the two mapping functions to a single `map`, or more formally

$$map\, f \circ map\, g = map\, (f \circ g) \tag{3.3}$$

In Listing 1, a conceptual example of fusion of two map operations is shown. Since Futhark is purely functional, it can take advantage of such equivalences

```
1 -- add n and square each element
2 let func(arr: []i32, n: i32) =
3     let adds = map (\x -> x + n) arr
4     let sqs = map (\x -> x**2) adds
5     in sqs
6
7 -- manual fused version
8 let func_fus(arr: []i32, x: i32) = map (\x -> (x + n)**2) arr
```

Listing 1: Conceptual example of fusion

and uses fusion in such cases. In imperative languages, such transformations would require non-trivial analysis by the compiler, or have the programmer signal to the compiler that a certain part of the program is free of side effects, typically through annotations. Again this is an area where Futhark removes complexity away from the programmer and hides it through technical solutions, that are made possible by the features of the language.

3.2.3 Modules

In order to support maintainability, Futhark provides a *module* [7] mechanism, reminiscent of what is found in most programming languages, in order to organize source code. The modules system in Futhark also provides the language with a mechanism in which generic code can be expressed. In Listing 2 a *module type* for a monoid is defined.

```
module type Monoid = {
  type t
  val add : t -> t -> t
  val zero : t
}
```

Listing 2: Module type declaration [17]

Other modules can now take another module that implements the monoid module as parameter, which are called a *parametric module*. In Listing 3, a module `Sum` is defined, that requires the parameter to be an implementation of the `Monoid` module.

```
module Sum(M: Monoid) = {
  let sum (a: []M.t): M.t = reduce M.add M.zero a
}
```

Listing 3: Parametric module declaration [17]

Similar mechanism are found in other languages, however, they often result in extra runtime overhead. Modules in Futhark are compiled away in the resulting program, an therefore incurs no runtime overhead.

3.2.4 Compilers and tools

Compilers

Futhark features two main compiler, `futhark-c` and `futhark-opencl`. `futhark-c` is the C backend, that produces a binary executable, that run on a single CPU, on a single thread. `futhark-opencl` is the backend targeting the OpenCL framework, that produces a binary that runs on the parallel hardware components of the system. The compilers are used through a simple command-line interface such as

```
$ futhark-c program.fut
./program

$ futhark-opencl
./program.fut -d GeForce
```

Note that it might be necessary to provide to flag `-d` telling the program which device to use, since there might be several on a single machine.

Tools

As well as the compiler backends, Futhark comes bundled with a toolchain supporting testing, test data generation, and benchmarking, amount other things [8]. This makes for a simple , but efficient environment to develop small experimental applications.

Chapter 4

VaR engine using Futhark

The following section presents an implementation of a small VaR engine, written in Futhark. A description of the method and implementation of the market simulation needed to price the underlying asset is presented. A detailed description of the structure and implementation of the Monte Carlo simulations is presented. This is followed by a presentation of the actual VaR calculation itself, using Monte Carlo simulation. Lastly sections on experiments and testing of the various parts of the engine is presented.

4.1 Financial programming in Futhark

Many domains have performance as a key concern to solve the problems in that particular domain, whether it is performance in terms of resource utilisation in constrained environments, or performance in terms of throughput(results/time). Financial programming is a numerically heavy domain, where simulating market movements, estimating risk, and pricing products, are just some of the very common applications that demands programs that can deliver high throughput and fast answers. Many problems found in financial programming are a matter of performing a single calculation, on a set of parameters, and on hundreds of thousands or even millions of products, which makes these types of problems inherently parallel. Futhark is therefore a good candidate for writing Financial applications.

When starting a project in any programming language, one has be to aware of the constraints, and Futhark is no exception. Futhark is designed to support high level parallel programming, using a purely functional programming model, therefore we not only have the constraint of being purely functional, but also from the parallel programming point of view. Common

features such as *Sum types*, *polymorphic functions* and side-effects are not supported, even though other purely functional languages such as Haskell supports these features. Although not support many common features, Futhark still provides plenty of constructs to enable the sorts of applications it targets to optimize for. Looking at the task of calculating VaR for a portfolio, we want some way to represent the products, such as barrier options. We could use an array and have a convention on which field is at what index in the array, but that would be both tedious and error prone. Instead Futhark provides a `record` construct, reminiscent of structures found in almost all programming languages, used to organise associated data into a named type. A `record` in Futhark is declared using `{ fields, .. }`, but in most cases we want to associate a name with a `record`, therefore we use the `type` keyword following a name and then the actual declaration of `{ fields, .. }` such as in Listing 4.

```
1     type Option = {
2       S  : f64, -- asset price
3       X  : f64, -- strike price
4       v  : f64, -- volitility
5       r  : f64, -- risk-free interest rate
6       T  : f64, -- time to expire
7       ds : i32, -- days
8       Sb : f64, -- barrier
9       ot : i32  -- option type
10    }
```

Listing 4: Representation of a 'generic' option in Futhark

The `type` keyword is used to associate a name with a type, such as `type Int = i32`, but also essential for defining other abstraction such as functions. Such an abstraction used in the context of option pricing, can be seen in the payoff functions for each barrier option type shown in Listing 8, where we define a payoff-function as taking the *strike price*, the *barrier* and a *path*, to calculate the payoff. Using function abstraction in this way,

```
type payoff_function = (X: f64) -> (Sb: f64) -> (path: []f64) -> f64
```

Listing 5: Payoff function abstraction

combined with *high-order functions* also supported by Futhark, lets us extract many parts of the application into small reusable components, and simply parameterizing larger parts of the application, to achieve a modular structure at the function level.

4.1.1 Market Simulation

In order to perform valuations of barrier options, we need to simulate the price movement of the underlying assets. We will limit the application in this thesis to stocks as the underlying asset. The price movement of an asset, such as a stock, typically follows a pattern of increases and decreases in the price of a time horizon T. A common method of simulating stock prices is using what is known as *Geometric Brownian Motion* (GBM). GBM is a stochastic process that takes the logarithm of the random sample value, where the results follow a Brownian motion. A Brownian motion, also called a Wiener process denotes by W, has certain properties that are useful in path generation, where in each time interval the motion has an independent increase such that

$$W_{t+s} - W_t \ N(0,1) \tag{4.1}$$

meaning the increments follow a normal distribution. As mentioned, GBM is a stochastic-process, that derives from a differential equation, meaning an equation representing some rate of change, in this case a stock [3, p. 292]. The differential equation is defined as

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{4.2}$$

where S_t is the stochastic process, μ denotes a drift component, σ denotes the volatility component and W_t is a *Brownian motion*. By solving the differential equation with the Itô lemma, we get

$$S_{t+\delta t} = e^{(r-\frac{1}{2}\sigma^2)\cdot\delta t+\sigma\cdot(\sqrt{\delta t})\cdot\epsilon} \tag{4.3}$$

that provides the formula for generating sample paths.

There will be two methods of generation the GBM, one using pseudo-random numbers and the second using Sobol sequences. Two GBM generation methods will provide a baseline for measuring convergence, performance and speed-up on parallel hardware.

GBM Pseudo random number

Futhark has build-in support for generation of random numbers using the standard library. Random numbers are generated using a `rng_engine`,

where there are several to choose from. For path generation with pseudo-random numbers, we will use the standard *Linear congruential generator* (LCG), even though LCG is not the most efficient or modern approach, the behaviour and implementation is simple [3, p.256]. Since generation of random numbers is inherently sequential, Futhark also generates random numbers in a sequential fashion. Since LCG is initialized with the same seed, it will return the same sequence of numbers.

```
1  let sample_path(o: Option, seed: i32, n: i32, m: i32) =
2    let rng  = minstd_rand.rng_from_seed [seed]
3    let col_rngs = minstd_rand.split_rng n rng
4    let dt = o.T / f64.i32(m)
5    let d1 = (o.r - 0.5 * o.v**2)  * dt
6    let d2 = o.v * f64.sqrt(dt)
7    let mat = map (\r ->
8      let row_col_rngs = minstd_rand.split_rng m r
9      let row = map (\x ->
10         let (_, v) = normal_dist.rand {mean = 0.0, stddev = 1.0} x
11         let w = f64.exp(v * d2 + d1)
12         in w) row_col_rngs
13     let sc = scan (*) 1.0 row
14     let rs = map (*o.S) sc
15     in rs) col_rngs
16   in mat
```

Listing 6: Function for sample paths using pseudo random numbers in Futhark

We cannot execute pseudo-random number generation directly in parallel using SOACs such as `map`, since each function f would be applied using the same `rng_engine`, thus applying the same random number sequence to the input. Futhark instead supplies a `split_rng` function for splitting a `rng_engine` into n `rng_engine`, each at a different state, and each generating a different sequence as shown in Listing 6 lines 3 and 8. Every `rng_engine` needs to be initialised with a distribution, where, in this case, a `normal_distribution` is used. To sample a path, we use a sequence of normally distributed random numbers, and apply equation 4.3, and thereby return a sequence of increments, representing the price changes for the underlying asset as shown in Listing 6. Afterwards, we need to apply S_0 to get

the simulated path. We use the `scan` SOAC provided by Futhark, in this case an inclusive `scan`, to apply the S_0 at to each increment in the generated path shown in Listing 6 line 13. The result of a use of the path sampling function can be seen in Figure 4.1, where a GBM has been generated using a option with $S_0 = 50$, and a time horizon of 100 days.

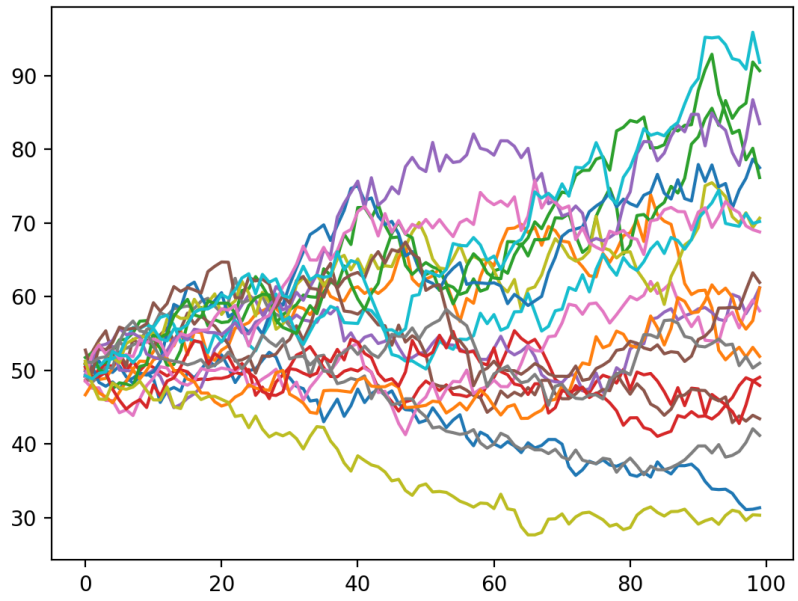


Figure 4.1: GBM generated using pseudo random numbers in Futhark

4.1.2 Monte Carlo simulations

In Listing 7 an implementation of a Monte Carlo simulation runner for the VaR simulation is shown, taking an option, a seed for the random number generation, number of paths to generate, number steps in each path, and a fixed future array of number. The fixed future array is used, since we simulate the new paths from time $t + \delta t$ to $t + T$, and therefore the fixed future array contains the path from t to $t + \delta t$ for the barrier option. As described in section 2.5, we have to price the entire path, since we might have hit the barrier a any point on the path. As shown in Listing 7 line 20, we concat the simulated path on to the fixed future before calculating the

payoff.

```
let mc_runner_var (o           : Option,
                   seed        : i32,
                   num_paths   : i32,
                   num_steps   : i32,
                   fixed_future: []f64) =
  let df  = f64.exp(-o.r * o.T)
  let rng = minstd_rand.rng_from_seed [seed]
  let col_rngs = minstd_rand.split_rng num_paths rng
  let dt = o.T / f64.i32(num_steps)
  let d1 = (o.r - 0.5 * o.v**2)  * dt
  let d2 = o.v * f64.sqrt(dt)
  let mat = map (\r ->
    let row_col_rngs = minstd_rand.split_rng num_steps r
    let row = map (\x ->
        let (_, v) = normal_dist.rand {mean = 0.0, stddev = 1.0} x
        let w = f64.exp(v * d2 + d1)
        in w) row_col_rngs
    let sc  = scan (*) 1.0 row
    let rs  = map (*o.S) sc
    let fix = concat fixed_future rs
    let pay = barrier_payoff(o, fix, df)
    in pay) col_rngs
  in mean(mat)
```

Listing 7: Monte-Carlo *runner* function

Since we have several different types of barrier options, where each has its own payoff function, we need to have a way of dispatching the parameters to the correct payoff function. In Listing 8 is shown the function that dispatches the parameters to the correct payoff function. Each option has an *option type* tag `ot`, represented by an integer. There is defined such a tag for each barrier option type. The reason for this design, is that Futhark does not support various polymorphic constructs found in other functional programming languages, such as *sum types* and pattern matching, found in a language such as Haskell. This is an area where Futhark lacks in expressive power.

```
let barrier_payoff (o: Option, path: []f64, df: f64) =
  if      o.ot == DI_CALL then (payoff_di_call o.X o.Sb path) * df
  else if o.ot == DO_CALL then (payoff_do_call o.X o.Sb path) * df
  else if o.ot == DI_PUT  then (payoff_di_put o.X o.Sb path) * df
  else if o.ot == DO_PUT  then (payoff_do_put o.X o.Sb path) * df
  else if o.ot == UI_CALL then (payoff_ui_call o.X o.Sb path) * df
  else if o.ot == UO_CALL then (payoff_uo_call o.X o.Sb path) * df
  else if o.ot == UI_PUT  then (payoff_ui_put o.X o.Sb path) * df
  else if o.ot == UO_PUT  then (payoff_uo_put o.X o.Sb path) * df
  else 0.0
```

Listing 8: Dispatching payoff function

4.1.3 Sobol

Futhark has built-in library support for working with Sobol sequences in an efficient manner. The Sobol module provides several function for working with Sobol sequences shown in Listing 9, but the most frequently used is the `sobol` function [11]. The Sobol module also support creation of user-defined reduction functions by implementing the `Reduce` module.

```
module type sobol = {
  val D : i32
  val sobol       : (n: i32) -> [n][D]f64
  val independent : i32 -> [D]u32
}
```

Listing 9: Sobol module

We will focus on the `sobol` function. To use the Sobol module we have to specify the dimensionality of the sequences we wish to generate with the parameter `D`, and also choosing one of the modules containing precomputed direction numbers, based on the needed dimensionality in our application. In Listing 10 the instantiation of all the needed Sobol modules are shown. We specify an instantiation for each number of paths we wish to simulate for, i.e. we need a Sobol sequence of 50 numbers, when we wish to simulate paths with 50 days each. For the direction numbers we provide the module that contains 1000 direction numbers, since we need to simulate paths with

up to 500 days in each. This is a limitation of Futharks modules system, since we cannot instantiate the Sobol module at runtime.

```
1 module Sobol_20  = Sobol sobol_dir { let D = 20 }
2 module Sobol_50  = Sobol sobol_dir { let D = 50 }
3 module Sobol_252 = Sobol sobol_dir { let D = 252 }
4 module Sobol_365 = Sobol sobol_dir { let D = 356 }
5 module Sobol_500 = Sobol sobol_dir { let D = 500 }
```

Listing 10: Sobol module instantiation for the VaR engine

4.2 VaR calculation in Futhark

Shown in Listing 11 is the function calculating VaR. A detail to note is that the `loop` construct in Futhark is used. `loop` is a functional way to approximate the semantics of an imperative for loop, since some algorithms are more conveniently expressed in an imperative style. When using `loop`, the programmer is telling the Futhark compiler that each iteration of `loop`, should be executed sequentially, however the code inside the body can contain arbitrary parallelism. `loop` is also the only construct in Futhark where recursion can be expressed, since recursion is not directly supported by the language. `loop` is also commonly used in Futhark for *outer-loops*, where the assumption is that the amount of parallelism in the body will be enough too keep the hardware busy with computations. In Listing 11 in lines 8-7, we first value the portfolio at time t to $t + \delta t$. Afterwards in lines 9-19 we do the simulations on the new market risk factors. In lines 10-16 we prepare the input for the `mc_runner_var`. We calculate $T - t + \delta t$ on line 10, which is how many days for `mc_runner_var` to simulate. We then on line 13 find the new S_0 for the option, which is last value of the current path being simulated. `S` is also what we use as the `fixed_future`, since it is the path from t to $t + \delta t$. Afterwards we simply sort the resulting array, and use the calculated `cut_off` that represents the percentile of the confidence level, which in this case is a confidence level of 95%, and then we can return the estimated VaR.

```
let var(o        : Option,
        seed     : i32,
        var_sims : i32,
        var_days : i32,
        num_paths: i32) =
   let df = f64.exp(-o.r * o.T)
   let S  = sample_path(o, seed, var_sims, var_days)
   let V  = mean(map (\i -> (payoff_di_call o.X o.Sb i) * df) S)
   let Si = loop Si = (replicate var_sims 0.0) for i in 0...(var_sims-1) do
               let dt  = o.ds - var_days
               let dT  = (r64(dt)/252.0)
               let x   = S[i]
               let s   = x[var_days - 1]
               let opt = { S=s, X=o.X, v=o.v,
                           r=o.r, T=dT, Sb = o.Sb,
                           ot = o.ot , ds = dt}
               let v   = mc_runner_var(opt, seed, num_paths, dt, S[i])
               let vi  = V - v
               in Si with [i] <- vi
   let ordered = merge_sort (f64.<=) Si
   let cut_off = var_sims / 100 * 95 - 1
   in ordered[cut_off]
```

Listing 11: VaR computation function

A similar version for calculating VaR is also made for using Sobol sequences for pricing, that has the same basic structure.

4.3 Experiments

The following section will present experiments conducted on two platforms in different configurations, benchmarking several areas of the VaR-engine. The aim is to show the difference in execution speed between the sequential and parallel versions of the program, verifying that some amount of parallelism has been achieved. The sequential execution is done using the `futhark-c` backend, where the resulting program is running on a single core, on a single thread. The sequential execution will serve as the baseline for the performance comparison. The sequential baseline will be compared against the program compiled with the `futhark-opencl` backend, where the resulting

program will be run on graphics cards on each platform. The experiments has been conducted on the following systems

- MacBook Pro (laptop) 2,3-GHz quad-core Intel Core i7-processor, 16 GB 1600-MHz DDR3 RAM, NVIDIA GeForce GT 750M with 2 GB GDDR5 RAM
- Acer Predator G3-710 (desktop), 2.7GHz Intel Core i5-6400 processor, 8GB DDR4 RAM, NVIDIA GeForce GTX 970 4GB RAM

The benchmarks for the VaR engine has been split in two, since the two methods, pseudo random numbers and Sobol sequences, does not require the same amount of work. Both benchmarks are performed by pricing a portfolio of 1 barrier option, where the initial price a time t is priced using 10000 simulated paths, and where each new market risk factor has been priced with 10000 paths simulated. The maturity of the options is 50 days, and the VaR horizon is 20 days.

4.3.1 Acer Predator G3-710 core i5

In Tables 4.1 and 4.2 are shown the benchmarks performed using pseudo random numbers on the Acer Predator G3-710 platform, using the i5 processor as the baseline, compared to the GTX 970 Nvidia graphics card.

Version	Runtime	Speedup ´
Intel i5 2.7 GHz (baseline)	482.66s	X1.0
NVIDIA GeForce GTX 970	8.62s	X55.99

Table 4.1: VaR on portfolio of 1 barrier options using pseudo random numbers with i5 as baseline

Looking at the results in Table 4.1, we see that the calculation of VaR using pseudo random number is 55.99 times faster than the same program executed on a single i5 processor. We can conclude from the benchmark that even if `futhark-c` compiled the program in such a way where it can utilize all of the available cores, which in this particular case is 4, the `futhark-opencl` version of the program would still run considerably faster. Table 4.2 shows the benchmarks performed now using Sobol sequences on the Acer Predator G3-710 platform, using the i5 processor as the baseline, compared to the GTX 970 Nvidia graphics card.

Version	Runtime	Speedup ´
Intel i5 2.7 GHz (baseline) Sobol	347.42s	X1.0
NVIDIA GeForce GTX 970 Sobol	8.48s	X40.95

Table 4.2: VaR on portfolio of 1 barrier options using Sobol sequences with i5 as baseline

Looking at the results in Table 4.2, we again see a considerable speedup. The GTX 970 graphics card is 40.95 times faster in this case.

4.3.2 MacBook Pro i7

In Tables 4.3 and 4.4, the benchmarks of the running two version of the VaR engine on the MacBook Pro with an i7 processor. Also, the GTX 970 performance from the Acer Predator G3-710 benchmarks is also included for extra comparison.

Version	Runtime	Speedup ´
Intel i7 2,3 GHz (baseline)	173.43s	X1.0
NVIDIA GeForce GT 750M	59.02s	X2.94
NVIDIA GeForce GTX 970	8.62s	X20.12

Table 4.3: VaR on portfolio of 1 barrier options using Sobol sequences with i7 as baseline

In Table 4.3 we have a similar result comparing the i7 processor on a laptop to the graphics card found on the desktop. In this case the `futhark-opencl` version is 20.12 faster. The result is expected since the i7 is a more powerful processor. If we look at the on-board graphics cards on the laptop, the NVIDIA GeForce GT 750M, the performance increase is still considerable, by being 2.94 faster.

Version	Runtime	Speedup ´
Intel i7 2,3 GHz (baseline)	153.21s	X1.0
NVIDIA GeForce GT 750M	89.74s	X1.71
NVIDIA GeForce GTX 970	8.48s	X18.06

Table 4.4: VaR on portfolio of 1 barrier options using Sobol sequences with i7 as baseline

Looking at the results in Table 4.4, we again see a considerable speedup. In this case the GTX 970 graphics card is 18.95 times faster.

4.3.3 Path generation benchmarks

In the following Tables 4.5, 4.6 and 4.7, benchmarks of path generation performance on the Acer Predator G3-710 core i5 are displayed. Each benchmarks generated paths for an increasing number of options. The benchmarks in Tables 4.5 and 4.6 generates paths with 50 days each, where the benchmark in Table 4.5 generates 10000 paths, the benchmarks in table Table 4.6 generates 100000 paths. The last benchmark shown in Table 4.7 generates 10000 paths for each option, with a length of 50 days. For each benchmark we see large speedups when the benchmark is executed on the graphics card, where for 100000 paths generated, shown in Table 4.6, we see the execution being 85 times faster. When we increase the number of days in a path the performance declines as shown in Table 4.7.

n	futhark-c	futhark-opencl	speedup
10	828.45ms	17.19ms	X48.19
20	1568.11ms	36.98ms	X42.40
30	3456.62ms	55.56ms	X62.21
40	3997.03ms	72.67ms	X55.00
50	4271.66ms	95.80ms	X44.59

Table 4.5: Performance comparison for generation of 10000 paths, with 50 days, for each option

n	futhark-c	futhark-opencl	speedup
10	8361.11ms	99.24ms	X84.25
20	15679.42ms	185.58ms	X84.49
30	23521.30ms	275.53ms	X85.37
40	31252.11ms	365.85ms	X85.42
50	39262.53ms	454.82ms	X86.33

Table 4.6: Performance comparison for generation of 100000 paths, with 50 days, for each option

n	futhark-c	futhark-opencl	speedup
10	3.96s	0.06s	X66.00
20	7.98s	0.11s	X72.55
30	11.87s	0.15s	X79.13
40	15.82s	0.21s	X75.33
50	19.77s	0.27s	X73.22

Table 4.7: Performance comparison for generation of 10000 paths, with 252 days, for each option

4.4 Testing

To verify the Monto Carlo simulations are pricing the options correctly, we can compare sample options priced using the Monto Carlo simulations, against the same options priced through the analytical formula. The simulations are inherently approximations, therefore we cannot expect the exact results same as the ones calculated from the analytical version. However we known by the *Law of large numbers* that as $n \to \infty$, $\hat{\mu} \to \mu$, therefore we expect that the results of the Monte carlo simulations will converge with the analytical formulas, when the number of simulations n increases. A visual example of the convergence can be seen in figure 4.2 where the number of simulations are increased along the x-axis, going from 1000 to 100000 with steps of 1000. As figure 4.2 show, the sobol generated sequence is faster to converge compared to the one generated using random number that still has high fluctuations even as the number of paths approach 100000.

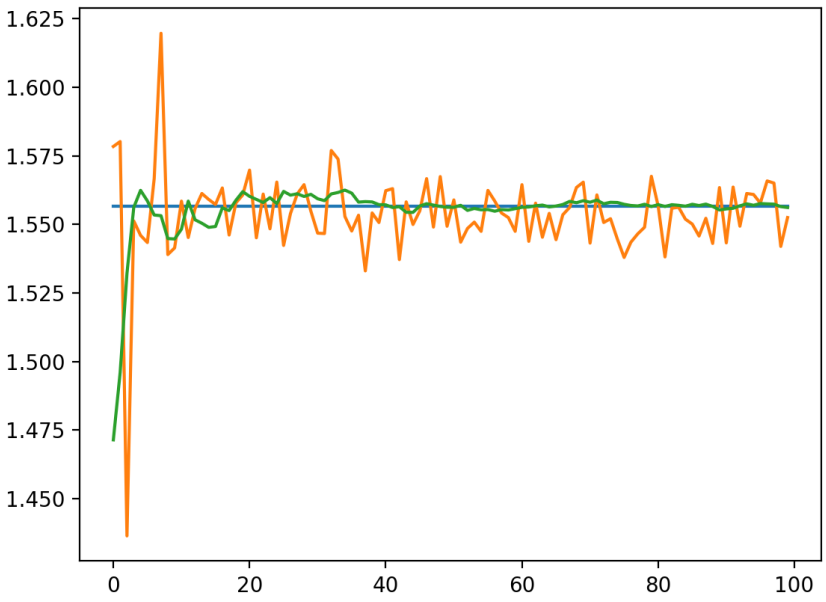


Figure 4.2: Convergence of dont-and-out-put option pricing, actual price in blue, random numbers in orange and sobol sequence in green

To test the accuracy of the barrier option pricing, several experiments have been conducted, adjusting the parameters that specifies the length of paths, number of iteration, type of option and if pseudo random numbers or sobol sequences were used. In table 4.8 and 4.9 are shown the calculated value of the option using the analytical formula, along with a the simulated price for 100000 and 1000000 paths, and the root-mean-squared deviation, for all option types, with a time horizon of 50 days. The accuracy can vary, particularly between types of options.

Days = 50, Simulations = 100					
Option	Actual	Paths = 100000		Paths = 1000000	
		Simulated	Error	Simulated	Error
down-and-out-call	0.000403	0.000383	0.000116	0.000375	0.000049
down-and-in-call	3.157109	3.152943	0.013901	3.153575	0.005054
down-and-out-put	0.690228	0.694567	0.008928	0.694044	0.004351
down-and-in-put	1.484998	1.481998	0.008758	1.481430	0.004363
up-and-out-call	2.139341	1.815430	0.324239	1.816818	0.322536
up-and-in-call	1.018171	1.337897	0.319804	1.337132	0.318970
up-and-out-put	0.009832	0.003354	0.006487	0.003290	0.006542
up-and-in-put	2.165394	2.173211	0.012385	2.172185	0.007117

Table 4.8: Simulation error for each options type, using pseudo random numbers for pricing

Days = 50, Simulations = 100					
Option	Actual	Paths = 100000		Paths = 1000000	
		Simulated	Error	Simulated	Error
down-and-out-call	0.000403	0.000423	0.000020	0.000417	0.000014
down-and-in-call	3.157109	3.127064	0.030045	3.132564	0.024545
down-and-out-put	0.690228	0.669478	0.020750	0.673738	0.016490
down-and-in-put	1.484998	1.485021	0.000023	1.484345	0.000653
up-and-out-call	2.139341	1.787560	0.351781	1.786439	0.352902
up-and-in-call	1.018171	1.339927	0.321756	1.346542	0.328371
up-and-out-put	0.009832	0.002796	0.007036	0.003170	0.006661
up-and-in-put	2.165394	2.151703	0.013691	2.154912	0.010482

Table 4.9: Simulation error for each option type, using sobol sequence for pricing

4.5 Discussion

In previous sections a review of the performance and programming model in the context of financial programming, using the Futhark programming language has been provided. The problem of data-parallel programs that Futhark aim to provide solutions for, is nothing new, and there are many existing solutions. For the mainstream programming languages such as Java and C#, there exist library support for data-parallel programming, where both are using the functional style facilities found in those languages [16] [14]. The approach of these languages is to provide the data-parallel constructs, that can be used when needed for certain computationally expensive areas of the application, but still in an imperative environment. In both cases it is up to the user to make sure, that the parts of the program using these facilities, use them in an efficient manner, since the compiler cannot make assumption of properties such as purity, in order to perform certain optimizations. Other purely functional programming languages such as Haskell has rich library support both for data-parallel programming, but also concurrency in a purely functional model [15]. For data-parallel programming on GPUs in particular, a library for Haskell called Accelerate exists, providing a DSL for array programming, that is translated in to code, that is executed on the GPUs. This is similar to Futhark, but having an entire compiler that optimises for parallel execution on GPUs has proven to provide promising results, such as in [9, p. 150], presenting benchmarks of programs written in both languages, where Futhark shows significant speedups in some cases. Futhark fits well in the area it targets, namely small but highly compute-intensive part of an application, possibly invoked from another language, where most of the application is programmed in. The functional model fits well with the target domains, such as financial programming, where expressing models for financial computations is straightforward, while still providing enough constructs to support modularity and maintainability. However, Futhark still lacks in expressiveness in certain areas, in particular a polymorphism construct such as sum types, and a way to instantiate parametric modules at runtime. In term of developments environment, one can not expect much from a research language, however, Futhark already provides a useful set of tools, that aids in development. Debugging is more difficult, since there cannot be any side effects, therefore we cant just print out the state of our program to fix bugs.

4.6 Conclusion

The aim of this thesis was to explorer the Futhark programming language, in the financial programming domain. The task was to create a small financial application, in particular a program that can calculate the VaR for a limited set of relatively advanced derivatives. Afterwards examination of the program structure, performance and correctness was performed. The results showed a considerable amount of performance was achieved using a programming language that targets parallel hardware.

As highlighted Futhark enforces a strict purely functional programming model, enabling the highly optimising compiler to have certain assumption. The programming model did limit the expressiveness of the programming language slightly, while still provided construct enough, to make application development possible and scalable in terms of maintainability. Even though Futhark has many common programming constructs, there is still room for improvement in regards to expressiveness, in particular when it comes to polymorphism.

Futhark is of course still a research language, therefore lack of highly sophisticated tools is expected, but are required in real-world scenarios. However Futhark still provides many useful tools, such as benchmarking, testing and integration with other languages, python in particular [8].

Performance is the major concern Futhark aim to aid developers in achieving, with parallel execution of compute-intensive parts of an application. The highly optimizing compiler produces programs that are able to take advantage of parallel hardware, such a GPUs, without the user having knowledge of all the intrinsics of that particular piece of hardware. However, certain knowledge and expertise is still needed to design efficient parallel algorithms, that takes advantage of the available hardware.

Bibliography

[1] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. "FinPar: A Parallel Financial Benchmark". In: vol. 13. 2. New York, NY, USA: ACM, June 2016, 18:1–18:27. DOI: 10.1145/2898354. URL: http://doi.acm.org/10.1145/2898354.

[2] Danil Annenkov and Martin Elsman. "Certified Compilation of Financial Contracts". In: *In Proceedings of the 2018 ACM SIGPLAN International Conference on Functional Programming (ICFP'18)*. St. Louis, Missouri, USA, September 2018.

[3] P Brandimarte. "Handbook in Monte Carlo Simulation: Applications in Financial Engineering, Risk Management, and Economics". In: (June 2014).

[4] Moorad Choudhry. *An Introduction to Value-at-Risk - 5th edition*. 2013.

[5] Carl Balslev Clausen. *MC^2 challenge in Risk*. HIPERFIT workshop. Dec 10, 2014. URL: http://hiperfit.dk/pdf/HIPERFIT_Dec2014_Clausen.pdf.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[7] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. "Static Interpretation of Higher-order Modules in Futhark: Functional GPU Programming in the Large". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018), 97:1–97:30. ISSN: 2475-1421. DOI: 10.1145/3236792. URL: http://doi.acm.org/10.1145/3236792.

[8] *Futhark User's Guide*. URL: https://futhark.readthedocs.io/en/latest/.

[9] Troels Henriksen. *Design and implementation of the Futhark programming language*. eng. Copenhagen: University of Copenhagen, Faculty of Science, [Department of Computer Science], 2017.

[10] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. "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. Nara, Japan: ACM, 2016, pp. 38–43. ISBN: 978-1-4503-4433-3. DOI: 10.1145/2975991.2975997. URL: http://doi.acm.org/10.1145/2975991.2975997.

[11] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. "Tricky Cases of Functional High-Performance Computing". In: *In Proceedings of the 6th ACM SIGPLAN workshop on Functional High-Performance Computing (FHPC '18)*. St. Louis, Missouri, USA, September 2018.

[12] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. "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. Barcelona, Spain: 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.

[13] John C. Hull. *Options, Futures, And Other Derivatives - 8th edition*. 2012.

[14] *Java Parallelism*. URL: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html.

[15] Simon Marlow. "Parallel and Concurrent Programming in Haskell". In: 2013. URL: https://simonmar.github.io/pages/pcph.html.

[16] *.NET Task Parallel Library*. URL: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/data-parallelism-task-parallel-library.

[17] *Parallel Programming in Futhark*. URL: https://futhark-book.readthedocs.io/en/latest/index.html.

List of Tables

List of Figures

List of source code listings

Appendix A

Appendix

A.0.1 Source code

The source code for the thesis is contained in zip file, provided with the thesis.

A.0.2 Benchmarks

Benchmarks for path generation, conducted on the MacBook Pro i7, 10000 and 100000 paths have been generated.

n	futhark-c	futhark-opencl
10	2716726.00us	474974.00us
20	5538029.00us	817448.00us
30	8076611.00us	1211053.00us
40	10710204.00us	1621415.00us
50	13519063.00us	2034338.00us

Table A.1: 10000 paths generated

n	futhark-c	futhark-opencl
10	266627.00us	110860.00us
20	535896.00us	241015.00us
30	804436.00us	406549.00us
40	1074067.00us	481142.00us
50	1333906.00us	706491.00us

caption100000 paths generated