Forside

Eksamensinformation
NDAB12006E - Bachelorprojekt i datalogi, Datalogisk Institut - ID:rwv746 (Louis Krog Kaufmann)

Besvarelsen afleveres af
Louis Krog Kaufmann
rwv746@alumni.ku.dk

Eksamensadministratorer
DIKU Eksamen
uddannelse@diku.dk

Bedømmere
Martin Elsman
Eksaminator
mael@di.ku.dk
+4535335683

Mads Rosendahl
Censor
madsr@ruc.dk

Besvarelsesinformationer

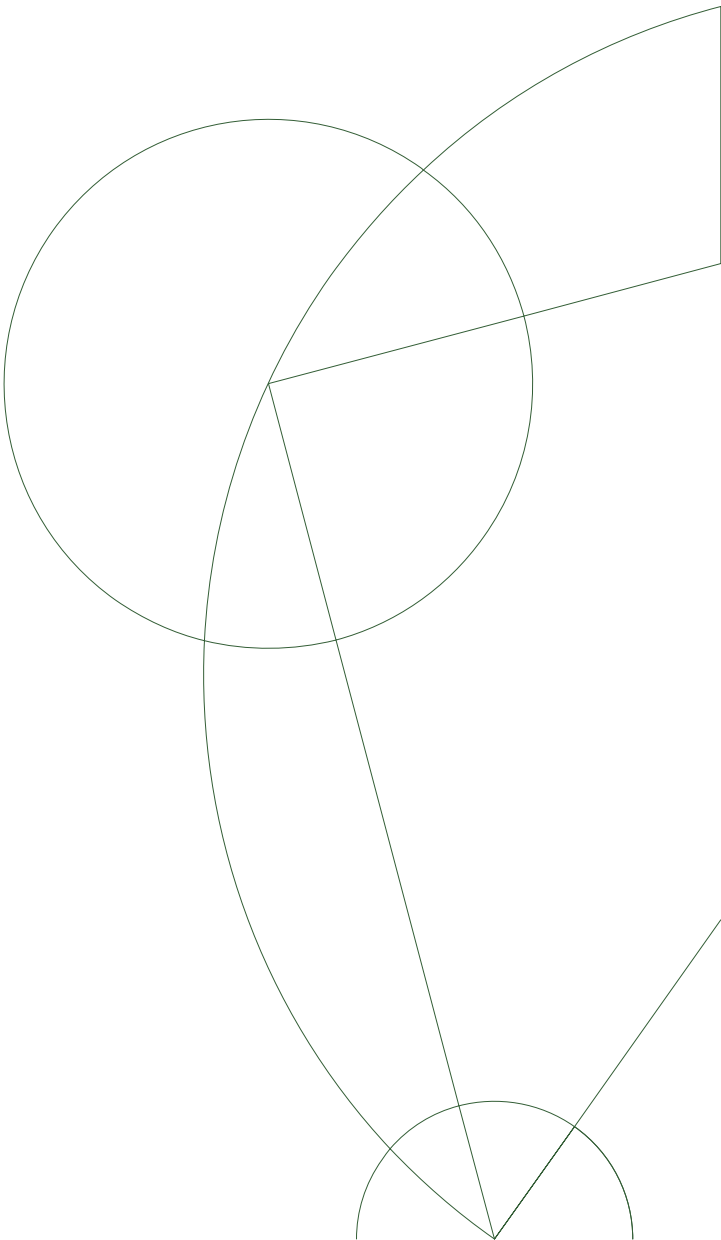
Titel, engelsk: Acceleration of CVA calculations using Futhark
Tro og love-erklæring: Ja

FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

BSc Thesis in Computer Science

Louis Krog Kaufmann <rwv746>

Acceleration of CVA calculations using Futhark



Supervisor: Martin Elsman

November 12, 2020

Abstract

Credit Value Adjustment (CVA) is the expected value of losses that could occur as a result of the default of a counterparty to a portfolio of financial derivatives. Calculating CVA requires a very computationally intensive simulation framework, motivating the need for code that can benefit from the increased performance provided by parallel execution on General-Purpose Graphics Processing Units (GPGPUS). Futhark is a functional programming language designed to be compiled to efficient GPGPU code.

This thesis describes a parallelization strategy for a simulation framework for calculating CVA for a portfolio of interest rate swap derivatives. Two parallel approaches to a CVA framework are proposed and implemented, one using Futhark library array functions, and one using the novel approach of expansion, which allows for more complex derivatives to be used in the framework.

Finally, a number of experiments are used to determine the performance of the framework when executed in parallel, compared to a baseline sequential execution. On average, the Futhark library approach yields a speedup by a factor of 4000 when executed in parallel on a GPGPU, compared to sequential CPU execution. GPGPU execution of the expansion based approach yields an average speedup factor of 400.

Contents

1 Introduction

Modern corporations use financial derivatives to reduce risks arising from their exposure to the financial markets. However, while derivatives can mitigate some risks, they can introduce the risk of the counterpart of the derivative contract defaulting prior to the expiry, causing the counterparty to fail to fulfil the contractual obligations.

Credit risk can be quantified and measured through the Credit Value Adjustment (CVA) framework, which is the adjustment to the value of a portfolio to account for the risk that the counterpart may default. Calculation of CVA consists of simulating paths of future exposure, leading to a large computational requirement. For example, industry standard calculation of CVA for a portfolio consisting of 50 instruments, would involve 100,000 market scenarios and 100 time steps, requiring 500,000,000 individual instrument valuation calculations [1, pg. 12]. Additionally, CVA needs to be calculated on a frequently basis as market risks are continuously evolving, which means that speed is critical for CVA calculations.

Traditionally, programs have relied on the exponential increases in processor speed that have been observed in the past, but in recent years the growth in traditional processing speed has stagnated [2]. This has shifted the focus in the high performance computing community from execution on traditional Central Processing Units (CPUs) to General Purpose Graphical Processing Units (GPGPUs), which have many more computational cores than CPUs. If programs are designed such that computation is executed in parallel across all the GPGPU cores, they can reap massive performance benefits when compared to typical sequential execution. Thus, enabling parallel execution for the CVA framework is relevant as to perform faster calculations, or allow for more complex risk scenarios to be simulated.

Futhark is a functional programming language, with a focus on compilation to high performance parallel code that can run on GPGPUs [2]. Futhark aims to have enough expressiveness to allow for complex programs to be created, while still being restricted enough to allow for programs to be aggressively optimized for parallel execution.

This thesis explores the design and implementation of the calculation of a CVA framework for a portfolio of interest rate swaps in Futhark. The performance of the program executed in parallel will be compared to that of sequential execution through experiments.

The motivation of this thesis is to show that the CVA framework can benefit from parallel execution in terms of speed without sacrificing functionality and retaining enough expressiveness to specify complex derivatives.

1.1 Problem statement

The following problem statement encapsulates the problems that will be investigated in this work. Is the Contract Value Adjustment (CVA) framework suited for parallel execution on GPGPUs through the Futhark programming language? Will parallel execution of CVA yield any significant performance improvements? Will an implementation

of the CVA framework in Futhark have limited expressiveness compared to a traditional sequential implementation?

1.2 Readers guide

Chapter 2 introduces the financial domain of the thesis, including interest rate models, interest rate swaps and CVA, assuming no prior financial knowledge. Chapter 3 introduces the parallel programming paradigm, as well as introducing the Futhark programming language and the specific constructs used in this thesis. Chapter 4 describes the design of a CVA framework with pseudocode, as well as two different implementations of the framework in Futhark. Chapter 5 describes experiments that were used to benchmark the performance of the two Futhark implementations when they are executed in parallel and sequentially, and describes the results of these experiments. Chapter 6 evaluates the results of the experiments and discusses potential improvements to address shortfalls. Finally, chapter 7 draws a conclusion in the context of the problem statement and discusses future work that could be done beyond the scope of this thesis.

1.3 Source code

The source code is available at `https://github.com/LouisKaufmann/Futhark-CVA`, along with instructions on how to replicate the benchmarks that were done in this thesis.

2 Financial background

The following chapter will introduce the knowledge of the financial domain needed to evaluate Credit Value Adjustment (CVA) for interest rate swaps, assuming no prior knowledge of finance. First the concept of credit risk and discounting is introduced, leading to the pricing of derivatives as discounted expected future cashflows. Then bonds and swaps are introduced as derivatives dependent upon interest rates, and the Vasicek model of of the short rate is introduced. Finally, CVA is introduced, along with a simulation framework for evaluating CVA for a portfolio of derivatives.

2.1 Derivatives and counterparty credit risk

A derivative is a financial instrument that have a value that depends on another financial instrument. The financial instrument upon which the value of the derivative depends upon is called the underlying.

Derivatives can either be bilateral, meaning the derivatives transaction consists of an agreement directly between two different parties, or centrally cleared, meaning there is an institution acting as a middleman for the derivatives transaction between two parties. Centrally cleared derivatives are guaranteed by a third party, meaning if one party defaults on the agreement, the third party will step in and fulfill the contractual obligations of the counterparty that is in default.

Bilateral derivative transactions have a possibility of a loss occurring for a party if the opposite party (the counterparty) defaults and cannot meet the contractual obligations of the derivatives transaction, which is described as credit risk. Credit value adjustment (CVA) provides a framework for quantification of counterparty credit risk and a market price for the credit risk of a derivative transaction. This thesis will be limited to CVA for vanilla interest rate swaps, which are derivatives which have values that are dependent on interest rates.

2.2 Interest rates and zero rates

Interest rates are the rates that borrowers pay to borrow money for a given amount of time. When lending money there is an aspect of credit risk, that the borrower will be unable to pay back some or all of the amount borrowed. When this credit risk of a borrower is high, the interest rates that the borrower will have to pay to borrow will also be high [3, pg. 77].
Interest rates are measured annually, but the compounding rate, which is the rate at which the interest is reinvested can vary. If an amount N is invested at an annual rate of r for a period of n years, with annual compounding, the return of the investment at the end of the period will be

$$N(1+r)^n$$

However, if the interest is instead realized and then reinvested at the same rate m times a year, the return of the same initial investment at the end of the period would be

$$N(1 + \frac{r}{m})^n$$

For example, investing \$10 at an annual rate of 10% with annual compounding for 1 year would yield a return of

$$\$10(1 + 0.1) = \$11$$

Whereas the same investment, but compounded quarterly (4 times a year) would yield a return of.

$$\$10(1 + \frac{0.1}{4})^4 = \$11.038$$

And the same investment compounded daily (365 times a year) would yield a return of

$$\$10(1 + \frac{0.1}{365})^{365} = \$11.052$$

So, increasing the frequency of compounding improves the returns of an investment, but the marginal returns are diminishing, and eventually, as the compounding periods become more frequent, the return of an investment with a return of r over n years approaches the following value:

$$\lim_{m \to \infty} A(1 + \frac{r}{m})^{mn} = e^{rt} \tag{1}$$

This limit of an investment as the compounding rate approaches infinity, is called continuous compounding, and throughout this thesis, only continuous compounding will be used.

2.3 Risk-free rates and discounting

The risk-free rate is the return of a risk-free portfolio [3, pg. 79]. This risk-free rate is used to quantify the time value of money, since money received today could be invested in a risk-free portfolio and provide the risk-free rate of return, whereas the same amount of money received in 1 year would not be able to be invested. Thus money has an inherent time value, and the value today of an amount M received at a later point in time must be discounted to reflect the opportunity cost.
For example, \$100 received in 1 year must be worth less, and the precise value today of the amount can be found by discounting at the risk free rate r. If the risk-free rate is 2%, then the value today of \$100 in 1 year is found by discounting:

$$\$100e^{-0.02} = \$98.01$$

In general, the discounted value of a notional amount M at time t at a discount rate of r is given by

$$M(e^{-tr}) \tag{2}$$

Discounting can also be used to price financial derivatives in general. The risk-free value of a derivative is equal to the expected value of the the future cashflows, discounted to the present time. For example, a derivative that has a payoff represented as a random variable X at time t, dependent on the distribution of an underlying, then the derivative has the value P at time 0:

$$P = e^{-rt}E^{Q}[X] \tag{3}$$

2.4 Zero coupon bonds

A bond is a financial instrument that can be used by an institution or a government to issue debt.

The issuer of a bond will pay the holders of the bond periodic interest payments on the amount loaned, called the principal, until the maturity date of the bond, where the principal is paid back. These intermediary interest payments are called coupons, and may be paid at different intervals, or not at all.

A zero coupon bond is a bond that pays no coupons, and the interest on the principal, along with the principal is all realized at once at the maturity of the bond. This makes calculating the value simple, as only one discount rate needs to be used. The n-year zero rate is the return earned on a zero-coupon bond with a maturity in n years. [3, pg. 82] For example, if the 2 year zero rate is 2%, \$100 invested for 2 years would grow to

$$\$100(e^{2\cdot 0.02}) = \$104.08$$

and conversely, if the 2 year zero rate is 2%, then a zero coupon bond that provides a payoff of \$100 after 2 years must be worth

$$\$100(e^{-2\cdot 0.02}) = \$98.02$$

The price $P(t,T)$ per unit notional at time t of a zero-coupon bond with maturity T with a yield of r is given by discounting a notional amount of 1 from time T to t :

$$P(t,T) = e^{-(T-t)r} \tag{4}$$

2.5 Interest rate swaps

An interest rate swap is a derivative that involves an exchange of future interest rate payments between two parties on a notional amount.

While there are many different possible specifications for an interest rate swap, this thesis will focus on the most basic type of interest rate swap, often called a vanilla swap,

which includes an exchange of a fixed interest rate to a floating interest rate, which is based upon a benchmark interest rate, such as the London Inter-bank Offered Rate (LIBOR).

A swap is parameterized by specifying a fixed rate, a benchmark floating rate (e.g LIBOR), the amount of payments, time between payment dates and the notional amount of the contract. At every payment date, the floating rate is received on the notional amount, and the fixed rate is paid on the notional amount.

Consider the below example of cash flows incurred by a swap on a notional amount of $2 mm.

Date	1 year LIBOR floating rate (%)	Floating cash flow received ($ mm)	Fixed cash flow paid ($ mm)	Net cash flow ($ mm)
2020/01/01	2	4	-4	0.0
2021/01/01	2.3	4.6	-4	0.6
2022/01/01	1.6	3.2	-4	-0.8
Total		11.8	-12	-0.2

Table 1: Cash flows for a swap where the 1 year LIBOR floating rate is received every year, and a fixed rate of 2% is paid on a notional amount of $2 mm annually

The value per unit of notional of a swap with fixed rate δ and time k between transaction is given by the following equation [4][pg. 100]:

$$V(t) = P(t, T_j) - P(t, T_N) - \sum_{i=j}^{N} \delta k P(t, T_i) \tag{5}$$

$$\begin{aligned}
\delta &= \text{fixed rate} \\
k &= \text{time between each payment in years} \\
N &= \text{total number of payment dates} \\
T &= \text{Payment dates: } T_i = ki, i = 0, ..., N-1 \\
T_j &= \text{Smallest value in } T \text{ that satisfies } T_j \geq t
\end{aligned}$$

For the case $t = T_0$, then $P(t, T_0) = 1$, as the price of a bond with an instant maturity must be one, simplifying the above to:

$$V = 1 - P(t, T_n) - \sum_{i=1}^{N} \delta k P(t, T_i) \tag{6}$$

A property of vanilla swaps is that they are only traded for one fixed rate, which is the rate that gives the swap a value of 0 at time $t = 0$. The fixed rate that ensures the

value of the swap is 0 can be found by setting $V = 0$ in (4) and solving for the fixed rate δ:

$$\delta = \frac{1 - P(t, T_N)}{\sum_{i=1}^{N} \delta k P(t, T_i)} \tag{7}$$

2.6 The Vasicek model of the short rate

When modelling the evolution of interest rates, it is impractical to model the evolution of every single interest rate for every maturity. Instead, short rate models allow for simulation of a single rate, which can be used to calculate zero coupon bond prices for many maturities.

2.6.1 The short rate

The risk-neutral short rate, r, is the interest rate at time t that applies to an infinitesimally short period of time [3, pg. 706]

The short rate can be expressed using the following differential equation describing the instantaneous change of a continuously updated bank account $\beta(t)$ as dependent on the instantaneous short rate $r(t)$ [5, pg. 427]

$$d\beta(t) = r(t)\beta(t), \quad \beta(0) = 1$$

which leads to the following analytical solution $\beta(T)$ for the value of the bank account at time T.

$$\beta(T) = \beta(0)e^{\int_0^T r(s)ds}$$

From equation (3), we know that derivatives can be priced by discounting the expected value of future cashflows. The short rate can also be used for discounting cash flows. The arbitrage free value at time t of a derivative with a payout X, at time T, with the risk neutral measure Q defined by $\beta(t)$ is given by:

$$P(t) = E_t^Q[e^{-\int_t^T r(s)ds} X] \tag{8}$$

From which it follows that for a zero coupon bond, paying $X = 1$ at time T, the value at time t is given by:

$$P(t, T) = E_t^Q[e^{-\int_t^T r(s)ds}]$$

2.6.2 The Vasicek model

The Vasicek model is a one-factor model for the short rate, meaning there is only one source of uncertainty in the model. The change in the short rate r is given by the differential equation:

$$dr = a(b - r)dt + \sigma dz \tag{9}$$

With the stochastic term dz being a Wiener process which follows a standard normal distribution, given by

$$dz \sim \mathcal{N}(0, 1)$$

Vasicek's model incorporates mean reversion, meaning the short rate will trend towards the long-term mean level b at rate a. The parameters a, b, σ are usually defined by calibrating the model to historical interest rates.

In order to numerically evaluate the evolution of the short rate, the continuous differential equation (9) needs to be discretized. This discretization can be done using the Euler–Maruyama method [6, pg. 35]. A Stochastic Differential Equation (SDE) of the form

$$dX_t = a(X_t, t)dt + b(X_t, t)dW_t$$

can be approximated for a discrete time interval Δt with the following equation

$$\Delta X_t = a(X_t, t)\Delta t + b(X_t, t)\Delta W_t \tag{10}$$

So for the Vasicek model, the change Δr of the short rate during a small period of time Δt can be approximated by the equations

$$\Delta r = a(b - r)\Delta t + \sigma \Delta z \tag{11}$$
$$\Delta z \sim \mathcal{N}(0, \Delta t) \tag{12}$$

Using the above, we can simulate a path of approximations y_i to r_i for the short rate with the following algorithm:

- Start: $t_0,\ y_0 = r_0, \delta t = \frac{T}{m},\ W_0 = 0$
- loop $j = 0, 1, ..., m$
  - $t_{j+1} = t_j + \Delta t$
  - $\Delta W = Z\sqrt{\Delta T}, Z \sim \mathcal{N}(0, 1)$
  - $y_{j+1} = y_j + a(b - y_j)\Delta t + \sigma \Delta W$

As we will need to know the bond prices at time t for a bond with maturity T, to value interest rate swaps, we can derive these from the short rate in a risk-free world. For the Vasicek model, the analytical solution to the expected value of the following claim, representing a zero coupon bond

$$E_t^Q[e^{-\int_t^T r(s)ds}]$$

is given by [3, pg. 709]:

$$P(t,T) = A(t,T)\exp\left(-B(t,T)r(t)\right) \tag{13}$$

$$A(t,T) = \frac{1-\exp\left(-a(T-t)\right)}{a} \tag{14}$$

$$B(t,T) = \exp\left[\frac{(B(t,T)-T+t)(a^2b-\sigma^2/2)}{a^2} - \frac{\sigma^2 B(t,T)^2}{4a}\right] \tag{15}$$

These closed form solutions to the bond price can be used to price interest rate swaps using equation (5) with a single short rate.

2.7 Monte Carlo methods

From 3, we know that the risk-neutral price P of a derivative with a future cash flow at time t, defined by a random variable X is given by the discounted expectation:

$$P = e^{-rt}E^Q[X]$$

with the mean $E^Q[X] = \mu$. The price P can be calculated by directly evaluating the integral arising from the expectation $E^Q[X]$. However, in many cases there is no closed form solution to this integral.

Thus, Monte Carlo methods can be used to approximate the expectation, by drawing samples from the underlying distribution X, as $\bar{X_n}$ and calculating the sample mean as:

$$\hat{\mu} \approx \frac{1}{N}\sum_{1}^{N} \bar{X_n}$$

By the law of large numbers, as n approaches infinity, the sample mean $\hat{\mu}$ will converge to μ.

2.8 CVA Framework

CVA is the adjustment to the value of a portfolio of derivative contracts to take into account the risk of counterparty default. Another way to interpret CVA is that it is the expected value of derivative losses resulting from the default of a counterparty. In the case of the default of a counterparty that a firm has an outstanding derivative contract with, there are two scenarios, depending on the market value of the contract at the time of default V_t is positive or negative [7, pg. 17].

If $V_t > 0$, then the firm closes the position with the counterparty and does not receive the payment, and has to pay another counterparty to enter the same contract, so the loss to the firm is V_t.

If $V_T < 0$, then the firm has to pay the value of the contract to the counterpart in order to close the position, and enter the derivatives transaction with another counterparty, receiving the market value of the contract, resulting in a loss of 0.

Thus, the exposure to a counterparty at time t for a single derivative contract is given by

$$E(t) = \max(0, V_t) \tag{16}$$

In this thesis, it is assumed that portfolios of derivatives are covered by a netting agreement, meaning that in the case of a counterparty default, the market value of all outstanding contracts are aggregated, such that contracts with negative values can offset contracts with positive values. With these netting rules, the exposure to a counterparty at time t for an entire portfolio is given by

$$E(t) = \max(0, \sum_{i \in PO} V_i(t)) \tag{17}$$

where $V_i, i \in PO$ describes the value of the i'th derivative contract in a portfolio containing one or more derivatives. Thus, the expected value of losses caused by the default of a counterparty can be calculated by multiplying the expected value of exposure to the counterparty, by the probability that the counterparty will default, over the entire lifetime of the outstanding derivatives. The formula for CVA in continuous time is given by [7, pg. 21]

$$\begin{aligned} \text{CVA} &= (1-R)\int_0^T \text{EE}^*(t) dPD(0,t) \\ \text{EE}^*(t) &= \text{Discounted Expected exposure to counterparty at time } t \\ R &= \text{Recovery rate} \\ dPD(0,t) &= \text{Risk-neutral probability of default between time 0 and t} \end{aligned} \tag{18}$$

where the Recovery Rate R describes the percentage of outstanding exposure that will be recovered if the counterparty defaults. The discounted expected exposure $\text{EE}^*(t) = B(0,t)E^Q(\max(0, E(t)))$ is the expected value of the positive exposure defined by (17) to the counterparty at time t.

For some simple derivatives, this value can be calculated analytically by integration, however there are many derivatives for which there exists no analytical solution, and thus the value of the expected exposure must be approximated by simulation using Monte Carlo methods as described in section 2.7

To approximate the expected exposure, simulation can be done in a framework as described.

1. **Scenario generation**
   For a number of paths n, generate n simulations of paths of market scenarios, each consisting of k discrete time steps t_k. Scenario generation is done using a model of the evolution of one or more market variables, for example the Vasicek model of the short rate.

2. **Portfolio evaluation at each scenario**
   For each time step t_k in each path, calculate the value of the portfolio for the

corresponding market scenario, as $V_n(t_k)$ for a path n. Then calculate the positive exposure, as $E_n(t_k) = \max(0, V_n(t_k))$.

3. **Aggregate exposures to CVA**
   Calculate the Monte Carlo estimation of the expected exposure at time t_k as the sample mean of the potential future exposures through all paths, such that: $EE^*(t_k) = \frac{1}{N}\sum_{n=1}^{N} PFE_n(t_k)$.
   The plot of the curve of the expected exposure with time is called the *Exposure profile*.

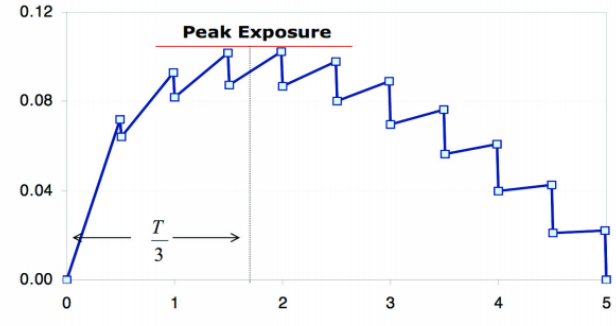


Figure 1: Exposure profile of an interest rate rate swap [7][pg. 20]

Now that the expected exposure $EE^*(t)$ has been calculated for the discrete time steps, 18 can be approximated numerical integration as following sum

$$\text{CVA} = (1 - R)\sum_{k=1}^{N} \text{EE}^*(t_k) PD(t_{k-1}, t_k) \tag{19}$$

where $PD(t_{k-1}, t_k)$ is the probability of counterparty default between time t_{k-1} and t_k. For the sake of this thesis, the probability of counterparty default is assumed to remain constant throughout the entire period. In practice, the probability of counterparty default can be derived from the value of Credit Default Swaps (CDS), which are derivatives that provide a payoff in the event of a default of a counterparty.

3 Parallelism

In this section the parallel programming paradigm is introduced as well as the motivation for developing programs that can be executed in parallel. Then the Futhark programming language is introduced, including the functionality and constructs needed to implement a CVA framework in Futhark.

3.1 Parallel programming

Traditionally, computer programs are written in a sequential fashion, where execution of the program leads to sequence of function calls, done on a central processing unit (CPU). For many years, sequential programs were getting faster simply due to the exponential growth of CPU speed. However, recently the evolution of CPU speeds has slowed, and advances in high performance computing have been made instead by focusing on other architectures, such as General Purpose Graphical Processing Units (GPGPUs)[2]. CPUs typically have up to 12 cores, whereas GPGPUs have thousands of cores. Thus, as the development of faster CPUs stagnates, designing programs that can fully utilize the high amount of cores in modern GPGPUs seems to be the logical next step towards improving program performance.

Parallelism refers to the design of programs to allow segments to be executed independently on multiple processing units. This thesis will specifically focus on data parallelism, which is the design of algorithms to allow computation of large amounts of data on multiple processing units.

Sequential program design assumes that all computations are dependent on each other, and thus instructions are executed one after another. However, it is often the case that two instructions are not interdependent, and therefore these instructions can be executed in parallel without affecting the result of the program. Parallel program design takes advantage of this lack of interdependence between most instructions to execute as many independent instructions as possible in parallel, allowing for significant increases in performance when executed on GPGPUs.

3.2 Futhark

Futhark is a high-level programming language developed for data-parallel array programming on the GPU. Futhark is purely functional meaning that functions must map an input to an output one-to-one, and have no side effects, like altering global variables. The syntax and conceptual basis is most similar to the Haskell and Standard ML programming languages. In contrast to these languages, Futhark makes a trade off of having a more constrained and less expressive syntax, to instead allow for more aggressive optimization and compilation to parallel code [2].

An advantage of Futhark as opposed to other GPGPU programming languages is that it provides a high-level interface where code can be written with sequential semantics. This allows for parallel code to be written without requiring knowledge of details of the

target architecture, as the Futhark compiler takes care of compiling to low-level CUDA or OpenCL code, that can run efficiently on GPGPUS.

Futhark programs should be written as combinations of Second-Order Array Combinators (SOACS), which are bulk operations taking arrays as inputs. The use of SOACs allows for the Futhark compiler to effectively create programs that can run in parallel on GPGPUS. The following subsections introduce the futhark functionality and constructs that will be used in the implementation of the CVA framework.

3.2.1 SOACs

The `map` function in Futhark is an array transformer, meaning that it returns a transformed version of an input array, and has the following generic type [8, pg. 15]:

$$\textbf{map} :: (\alpha \to \beta) \to []\alpha \to []\beta$$

Thus, `map` takes an input consisting of a function mapping an element of type α to an element of type β, and an array containing n elements of type α, and returns an array with n elements of type β which is the result of applying the input function to each element of the input array. In the following example, `map` is used in conjunction with a lambda function to increment each array element by 2.

```
> map (\x -> x + 2) [1,2,3]
[3i32, 4i32, 5i32]
```

The `reduce` function is an array aggregator, meaning it aggregates the elements of an array into a scalar, and has the following type [8, pg. 15]:

$$\textbf{reduce} :: (\alpha \to \alpha \to \alpha) \to \alpha \to []\alpha \to \alpha$$

`reduce` takes 3 inputs, an associative function of type $(\alpha -> \alpha -> \alpha)$, describing how to combine two elements of type α. `reduce` also takes a neutral element of type α as input, and an array of elements of type α to apply the reduction on.
In the following example, the addition operator $(+)$ is used as the associative function, along with 0 as the neutral element, to compute the sum of the input array [1,2,3].

```
> reduce (+) 0 [1,2,3]
6i32
```

3.2.2 Type system

A record can be defined with named fields using the following syntax:

```
type Date = {year:i32, month:i32, day:i32}
```

A variable with a record type can be initialized with the following expression:

```
let newyearsday : Date = {year=2020, month=1, day=1}
```

Where the fields can be accessed using dot notation: `newyearsday.month`,

3.2.3 Sequential loops

Futhark does not support recursive functions, but does provided syntax for expressing sequential loops. Consider the following function to calculate the factorial of a number `n`.

```
1 let factorial(n: i32): [] i32 =
2   let x = loop x = 1
3           for i < n do x*i
4   in x
```

Additionally, Futhark allows for in-place updates to be done on arrays, using the `with [i] = x` syntax, element `i` of an array is set to `x`. See below a sequential implementation of the `iota` function, using in place updates in combination with the looping syntax.

```
1 let seq_iota(n: i32): [n] i32 =
2   let x =
3     loop x = (replicate 100 0) for i < n
4     with [i+1] = x[i] + 1
5   in x
```

This allows for sequential loops to be used with outer parrallelism, as the array returned by the loop has a known size, and thus outer-parallelism can surround sequential code.

3.2.4 Random sampling

Pseudo-random numbers in Futhark can be generated using the `cpprandom` module, designed to mimic the pseudorandom generators present in C++. Consider the following code for generating a realisation from the standard normal distribution:

```
1     module norm_dist = normal_distribution f32 minstd_rand
2
3     let rng = minstd_rand.rng_from_seed [1]
4     let (rng, y) = norm_dist.rand {mean=0, stddev=1} rng
```

`norm_dist` is defined as a `normal_distribution` module with single precision floats (f32), using `minstd_rand` as the RNG engine. A seed to be supplied to the normal distribution module is created using `minstd_rand.rng_from_seed`, which is then supplied to the `minstd_rand.rand` function to generate a tuple containing the new seed for the RNG engine `rng`, and a sample from the normal distribution, `y`.

3.2.5 Fusion

To perform parallel operations using SOACS, Futhark splits an array into chunks, of which each chunk is processed individually on a computing unit, and finally the processed chunks are joined into the result array.

Futhark implements the concept of fusion in order to reduce memory and computational overhead that comes along with chaining multiple SOACs together and having to store each intermediary result. The first rule of fusion states that the result of composing the result of two maps involving two different functions is equivalent to the result of a single map of the composition of those two functions. This fusion rule can be expressed as:

$$(\text{map}f) \circ (\text{map}g) \equiv \text{map}(f \circ g)$$

To illustrate the first rule of fusion, consider the below example:

```
1 let unfused [n] (a : [n]i32) : [n]i32 =
2     let f = map (\x -> x + 2) a
3     let g = map (\x -> x ** 2) f
4     in g
5
6 let fused [n] (a : [n]i32) : [n]i32 =
7     map(\x -> (x+2) ** 2) a
```

The two functions `unfused` and `fused`, both consisting of maps will compile to be equivalent following the first rule of fusion, as the lambda function for `fused` is simply just the function composition of the two lambda functions of the maps in `unfused`. Fusion allows for parallel algorithms to be expressed by using sequential application of SOACs.

Applying the first rule of fusion as well as the subsequent fusion rules, nested applications of map constructs are transformed into a single map construct by flattening the input array, applying the fused function on the flattened array, and finally unflattening the resulting array so it is of the desired shape. This process can be repeated for multiple nested map constructs.

3.2.6 Regular and irregular flattening

In order to convert nested applications of SOACs to kernels that can be efficiently run in parallel, Futhark uses map fusion, and other techniques, in a process referred to as regular flattening [9]. Futhark has the limitation that it only supports regular arrays, meaning that inner arrays must have the same shape. This has the implication that in order for nested maps to be flattened, each function in a nested map must only use intermediate arrays of the same size, and also only return arrays that have the same, known size.

This limitation to regular arrays can constrain expression of certain problems in parallel, such as early stopping problems. These require simulation of a random variable until a criteria is met, which results in irregular amounts of computation being required. The Futhark compiler can handle some basic irregular problems, by letting some GPU threads run longer than others, but most irregular problems have to be addressed by using some of the following techniques in order to compile to parallel code. These

techniques are called irregular flattening, as they allow for irregular code to be flattened to parallel code.

A simple way to deal with irregular data is to pad the intermediate arrays with some additional elements. However, the computational resources and memory needed to pad arrays will often overshadow the benefits of enabling parallelisation.

Alternatively, a technique called expansion may be used, in which a function `expand` transforms each an array of elements into a flattened, long target array, which can be processed in parallel, overcoming the irregularity that would otherwise be present if the array was not flattened, as the resulting array is one-dimensional [10].

The `expand` function from the segmented library [1] has the following type [10, pg. 15]:

```
val expand 'a 'b : (sz: a -> i32) -> (get: a -> i32 -> b) -> []a -> []b
```

This type expression can be interpreted as the function taking two functions and an array of type 'a as an input, and returning an array of type 'b. The first parameter is a size function, which for each element in the input array, determines how many elements there should be in the target expanded array. The second array is a get function, which determines for a given input element, and index i, the element in the target array at index i. As a simple example, consider the expression

```
expand (\x -> x) (+) [1,2,3]
```

here the size function `(x -> x)` is the identity function such that each target array has the same size as the value of the input element. The get function is the expression $(+)$, which denotes addition, such that entry i in an array expanded from the value x has the value $x + i$. Thus, for the above expression, each input element is expanded into the following arrays:

```
1 -> [1 + 0] = [1]
2 -> [2 + 0, 2+1] = [2,3]
3 -> [3 + 0, 3+1, 3+2] = [3,4,5]
```

Thus the statement returns the concatenated results: [1,2,3,3,4,5].

The segmented library also includes the function `expand_outer_reduce`, with the following generic type [10]:

```
val expand_outer_reduce 'a 'b [n] : (sz: a -> i32 )
    -> (get: a -> i32 -> b)
    -> (b -> b -> b) -> b -> [n] a -> [n] b
```

`expand_outer_reduce` takes the same size and get functions as `expand`, along with an associative function defining how to combine the elements in the input array, a neutral element and the array to perform the expansion on.

The `expand_outer_reduce` function first performs the expansion that calling `expand sz get a`, and then calling `reduce` on each resulting array that is expanded from the input elements. Consider the following example:

[1] https://futhark-lang.org/pkgs/github.com/diku-dk/segmented/0.2.0/doc/lib/github.com/diku-dk/segmented/segmented.html

```
expand_outer_reduce (\x->x) (+) (+) 0 [1,2,3]
```

First, the target elements are expanded using the identity function `(x -> x)` as the size function and the expression (+) as the get function, resulting in the same expansion as in the previous example. Then, `reduce` is called on each expanded array, using the given associative function (+) and neutral element 0. So each element is expanded and reduced in the following way:

```
   1 -> [1] -> reduce (+) 0 [1] -> 1
2 -> [2,3] -> reduce (+) 0 [2,3] -> 5
3 -> [3,4,5 -> reduce (+) 0 [3, 4, 5] -> 12
```

Finally, the arrays are concatenated, and the statement returns [1,5,12].

Unfortunately, it is currently not possible to wrap an outer map around an `expand_outer_reduce` computation to work with arrays of higher dimensions, as computing the result of the inner expand will involve irregular amount of computation. Instead, arrays with 2 or more dimensions can be flattened to a one dimensional array, which can be used with `expand_outer_reduce`, such that it does not need to be placed inside a map. The result of `expand_outer_reduce` can then be unflattened, so it has the desired dimensions.

4 Design and implementation

This chapter builds upon the general CVA framework, to describe the design of a program that can calculate CVA for vanilla interest rate swaps under the Vasicek model, as well as the implementation of the framework in Futhark.

4.1 Design

Using the framework described in section 2.8 for calculation of CVA, the specific case for a portfolio containing an interest rate swap, with market scenarios modelled by the Vasicek model of the short rate can be described by the following pseudocode:

```
algorithm Swap-CVA
input:  Vasicek model with parameters a,b, sigma, delta t, r0
        Specification of portfolio containing n>1 swaps, each specified by
            fixed rate, notional value, number of payments, and payment term
        number of paths to simulate, n,
        number of steps to simulate for each path m
output: CVA value C approximating the credit value adjustment
        for the portfolio consisting of the swap
Main algorithm:
0. Generate n paths of the short rate, with m steps each, using formula (11)
    for the discretized evolution short rate following the Vasicek model.
1. For each path n and time t, the portfolio value by calculating and summing
    the price of all swaps using formula (5) as V(n,t)
2. Calculate the positive future exposure for each path n and time t
    as exp(n,t) = max(0,V(n,t))
3. Calculate the expected exposure at time t as EE(t) = avg(exp(t))
4. Apply formula (19) to calculate the CVA using the expected exposure
```

4.2 Parallel implementation

This chapter will deal with the translation of this sequential pseudocode presented previously, into functional Futhark code that is suitable for parallel execution. Two approaches to the portfolio valuation step described in section 2.8 will be presented, arising from the need to deal with irregularity present in pricing of swaps and other derivatives.

4.2.1 Scenario generation

Using the type system, the record types for instances of swaps and the Vasicek model are defined as follows. The type abbreviations allow for values of these record types to be used as parameters for functions and improves the readability of the code.

```
type Swap = {
    term: f32,
    payments : i32,
    notional : f32
}
type Vasicek = { a : f32, b : f32, sigma : f32 }
```

These record types can then be used to generate variables detailing the portfolio and parameterized Vasicek model.

```
let vasicek : Vasicek = {a=a, b=b, sigma=sigma, deltat=dt,r0 = r0}
let swaps [n] Swap = map (\x : Swap ->
    {term=swap_term[x],
    payments=payments[x],
    notional=notional[x],
    fixed=(set_fixed_rate swap_term[x] payments[x] vasicek)})
    (indices swap_term)
```

Listing 1: Instantiation of Vasicek model and portfolio variables using record types

To do the scenario generation step, following the algorithm presented in section 2.6.2, the evolution of the short rate following the Vasicek process for a discrete time step can be done with the following formula:

$$r_{i+1} = r_i + a(b - r_i)\Delta t + \sigma \Delta t \Delta z \tag{20}$$

$$\delta z \sim \mathcal{N}(0, 1) \tag{21}$$

Thus, for each step in the Vasicek process we need a sample from the standard normal distribution $\mathcal{N}(0, 1)$.
The Futhark support for pseudo-random number generation is introduced in section 3.2.4. The reliance of random number generation on a seed, presents a challenge for the parallel design, as pseudorandom numbers are generated by a sequential process in which a seed is provided as an input, which is used to generate a random number, and a new seed for generating the next pseudorandom number in the series. Futhark provides a workaround with the `minstd_rand.split_rng` function, which splits a single RNG state into an array of n states, which can then be used in combination with SOACs to generate pseudorandom numbers.
For our scenario generation step, we need to generate n paths, with each path consisting of m discrete time steps in the Vasicek model. Thus, we need mn samples from the standard normal distribution, which we can generate in parallel using the below code.

```
let generate_rands (paths: i32) (steps: i32) : [paths][steps]f32 =
    let rng = minstd_rand.rng_from_seed [1]
    let rng_vec = minstd_rand.split_rng paths rng
    let rands = map (\r ->
        let rng_mat = minstd_rand.split_rng steps r
        let row = map (\x ->
            let (_, v) = dist.rand {mean = 0.0, stddev = 1.0} x
            in v) rng_mat
        in row) rng_vec
    in rands
```

Listing 2: Function to generate a matrix of standard normal realizations using the `split_rng` function.

Intially, we define a single seed for the RNG engine as `rng`, and use `split_rng` to create an array of RNG engine seeds `rng_vec` with type `[paths]rng`. We use the map SOAC on this array to split each seed in the array another time, such that each array element is transformed to an an array of type `[steps]rng`. Each array is then passed to another map function, which uses each seed to generate a realization from the standard normal distribution. The result is that `rands` is an array of standard normal realizations with type `[paths][steps]f32`.

Now that we have the desired amount of realizations of the standard normal, we can simulate the short rate following a Vasicek process. Below is a function that given a Vasicek model, initial short rate `r0`, and array of standard normal realizations, generates a path. The mean-reverting property of the Vasicek means that each value of the short rate depends on the previous calculated value, so thus the short rate simulation must be done sequentially, which can be accomplished by using the Futhark syntax for expressing loops with in-place updates introduced in section 3.2.3.

```
let shortstep (vasicek : Vasicek) (r:f32) (dt:f32)
(rand:f32): f32 =
    let delta_r = vasicek.a*(vasicek.b-r) * dt
    let delta_z = (f32.sqrt dt) * rand * vasicek.sigma
    in r + delta_r + delta_z


let mc_shortrate (vasicek: Vasicek) (r0:f32) (steps: i32)
(rands: [steps] f32): [steps] f32=
    let xs =
        loop x = (replicate steps r0) for i < steps-1 do
            x with [i+1] = (shortstep vasicek x[i] vasicek.delta_t rands[i])
    in xs :> [steps] f32
```

Listing 3: Functions used to generate simulated path of a short rate following the Vasicek dynamics

The `shortstep` function is a scalar function, which takes a vasicek type, a value r_i, a value Δt, and a random float as an input and returns the increment of the short rate r_{i+1}.
As the shortrate function is done using the in-place array update syntax, the function is regular and no dynamic memory is used, so the function can be used in combination with the map SOACs to generate simulations of all the paths in parallel.

```
1     let shortrates = map(\x -> mc_shortrate vasicek r0 steps delta_t x) rands
```

The above code uses map with a lambda function detailing the parameters for the short-rate simulation function, with the `rands` array, generated earlier, containing $[n][m]$ realizations of standard normal realizations.

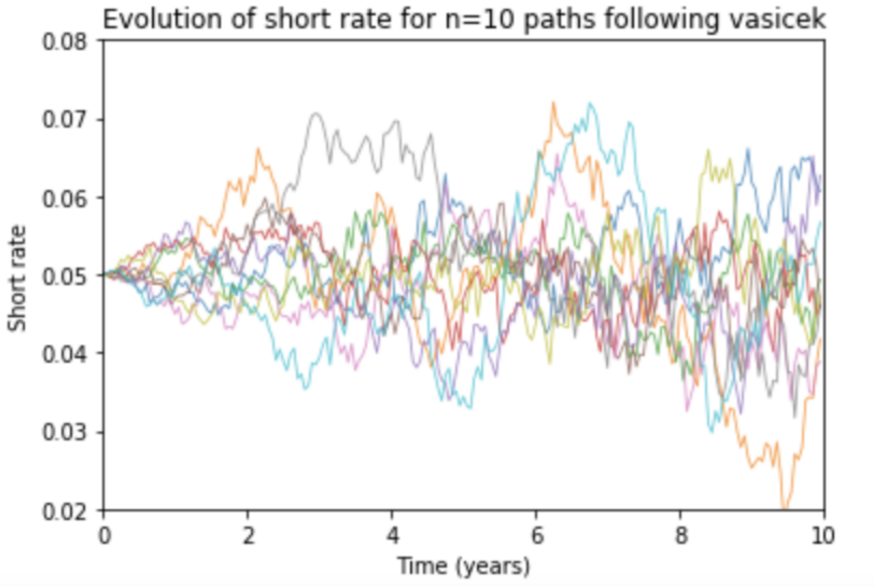


Figure 2: Evolution of the short rate for $n = 10$ paths following the vasicek model generated using Futhark

4.2.2 Portfolio valuation with nested maps

Now that the scenario generation step described in 2.1 has been completed, the portfolio needs to be evaluated for each generated market scenario. Recall that an interest rate swap has a value given by equation (5):

$$V(t) = P(t, T_j) - P(t, T_N) - \sum_{i=j}^{N} \delta k P(t, T_i)$$

This can be implemented in futhark with the following function utilizing iota to create an array containing payment dates T_j to T_N:

```
1 let swapprice (swap : Swap) (vasicek : Vasicek) (r:f32) (t:f32) =
2     let remaining = swap.payments - i64.f32 (f32.ceil (t/swap.term))
3     let nextpayment = (f32.ceil(t/swap.term))*swap.term
4     let remaining_dates = map(\x ->
5             nextpayment + ((f32.i64 x + 1)*swap.term)) (iota remaining)
6     let leg1 = bondprice vasicek r t nextpayment
7     let leg2 = bondprice vasicek r t (remaining_dates[::-1])[0]
8     let leg3 = reduce (+) 0 (map (bondprice vasicek r t) remaining_dates)
9     in swap.notional * (leg1 - leg2 - swap.fixed*swap.term*leg3)
```

Listing 4: Function `swapprice` to return the price of a swap given a scenario

This implementation has irregularity when used across different times t and different swaps, as the intermediate array used in the calculation generated with `iota remaining` will have a varying size across different scenario dates.

As t increases, there will be fewer remaining payments, and as such, the array detailing the remaining fixed flows will be of varying size for different times.

For example, for a swap with 10 payment dates, at time $t = 0$, would involve a summation of 10 different terms so `remaining_dates` would be of size 10, whereas at time $t = T_{n-1}$, `remaining_dates` would be of size 1.

This irregularity can be problematic for parallelism, as described in section 3.2.6. Luckily, the Futhark compiler can flatten the irregularity present in the `swap_price` function, allowing for the function to be used in conjunction with other SOACs.

The `swapprice` function can be used inside of nested maps to price every swap for every generated scenario, and aggregate the prices of the portfolio for every scenario, to return an array of Potential Future Exposures (PFEs).

```
let exp =
map(\x ->
    map2 (\y z ->
    let exp = map (\swap ->
        if z > swap.term *f32.i64(swap.payments - 1) then 0
        else swap.notional * (swapprice swap vasicek y z)) swaps
    let netted = reduce (+) 0 exp
    let exp = f32.max 0 netted
    in exp
) x times) shortrates
```

Listing 5: Nested maps used in conjunction with the swapprice method to calculate PFE over the generated scenarios

4.2.3 Portfolio valuation by expansion

While the Futhark compiler was able to overcome the irregularity present in the `swapprice` function, more complex pricing functions for other derivatives, such as path dependent options will not be able to be flattened by the Futhark compiler due to the irregularity present in the valuation of the instruments.

Thus, to improve the modularity of the framework to allow pricing of more advanced derivatives of swaps, the irregular flattening technique of expansion discussed in section 3.2.6 can be used. The following subsection will discuss the implementation of the flattening approach specifically for valuing vanilla interest rate swaps, however the implementation is easily transferable to other derivative pricing functions.

As valuation of swaps involves calculating and then summing the future cash flows, the `expand_reduce` function as introduced in section 3.2.6, can be used for swap pricing. In order to use the `expand_reduce` function, two helper functions to be defined and some changes to the representation of scenarios needs to be done.

To assist with portfolio evaluation, we introduce a pricing record, which represents a single pricing scenario for a single instrument in the portfolio, in this case a swap.

```
type Pricing = {
    swap : Swap,
    vasicek: Vasicek,
    t : f32,
    r : f32
}
```

Nested maps can be used then to generate a 3 dimensional array containing a pricing object corresponding to each swap for each scenario.

```
let pricings =
map(\x -> map2(\y z ->
    map(\swap ->
        let pricing : Pricing = {swap = swap, vasicek=vasicek,t = z, r = y}
        in pricing
    ) swaps) x times ) shortrates
```

In order to use the `expand_reduce` function, we first need to define the size function, which for each pricing object, returns the length of the corresponding swap pricing array. 5 shows that the amount of terms required to price a swap is the amount of remaining payment dates at time t.

```
let pricing_size (pricing:Pricing) : i64=
    let payments = f32.ceil (pricing.t/pricing.swap.term - 1)
    in i64.max (pricing.swap.payments - i64.f32 payments) 0
```

The second function to be passed to `expand` is more involved, which returns the price of the bond for payment time T_i for a given pricing object and an index i. Below function `pricing_get` does this using a scalar function `bond_price`, which calculates the closed form solution for a bond price under the Vasicek model, given by (13). A helper function `coef_get` is also defined, that returns the coefficient for the bond price, depending on if it is the first, last or middle swap payment date.

```
let coef_get (pricing:Pricing) (i : i64) : f32=
    let size = pricing_size(pricing) - 1
    let res = 0
    let res = if i == 0 then res + 1 else res
    let res = if i >0 then res -pricing.swap.fixed*pricing.swap.term else res
    let res = if i == size then res - 1 else res
    in pricing.swap.notional*res

let pricing_get (pricing:Pricing) (i : i64) : f32=
    let start = f32.ceil (pricing.t/pricing.swap.term) * pricing.swap.term
    let coef = (coef_get pricing i)
    let price = bondprice pricing.vasicek
           pricing.r pricing.t (start + (f32.i64 i)*pricing.swap.term)
    in coef * price
```

Listing 6: Expansion helper functions used to expand swap object to array of floats detailing future cashflows

To provide some intuition for the `pricing_get` and `coef_get` functions, consider the swap pricing formula introduced in section 2.5:

$$V(t) = P(t, T_j) - P(t, T_N) - \sum_{i=j}^{N} \delta k P(t, T_i)$$

and recognize that the formula simply consists of the summation of bond prices $P(t, T_n), n \in k, ..., N$, with different coefficients, representing the future payment dates. The coefficient for the bond with the shortest maturity $P(t, T_0)$ will always be 1, and the coefficient for the bonds with maturity $n \in [T_{k+1}, T_{n-1}$ will be δk, and the coefficient for the bond with the last maturity, representing the last payment date will have a coefficient of $\delta k - 1$. This logic is implemented in the `coef_get` function, which is used in the `pricing_get` function, where `start` represents T_j, which is the next payment date for a given time t, to return the bond price for a payment date, multiplied by the corresponding coefficient.

As the price of a swap consists of the aggregated prices of the individual cash flows, the summation can be done by the `expand_outer_reduce` function, as introduced in section 3.2.6. As such, an array of pricing records can be priced using `expand_outer_reduce`, with the two functions `coef_get` and `pricing_get` introduced earlier.

As explained in section 3.2.6, the amount of computation done by `expand_outer_reduce` is irregular, so the function cannot be used inside of a `map` SOAC.

To overcome this limitation, the array of pricings is flattened fully, then `expand_outer_reduce` is used on the array to generate prices for each pricing record. Then the array is unflattened such that it has the desired shape. When the prices for each scenario and swap have been generated, a second nested map is used to aggregate the prices of the swaps in the portfolio, and return the positive exposures in the array `exp`.

```
1 let flattened = flatten_3d pricings
2 let prices = expand_outer_reduce pricing_size pricing_get (+) 0 flattened
3 let unflattened : [paths] [steps] [n] f32 = unflatten_3d paths steps n prices
4 let transposed : [steps] [paths] [n] f32 = transpose unflattened
5 let exp = map (\(xs : [paths] [n] f32) : f32->
6               let netted : [paths] f32 = map(\x -> reduce (+) 0 x) (xs)
7               let exposures = map (\x -> f32.max 0 x) netted
8               in exposures
9           ) (transposed)
```

Listing 7: Using `expand_reduce` with additional flattening to price swap

4.2.4 CVA calculation

When the portfolio exposure has been calculated for each scenario, either by the nested map approach or the expansion approach, the expected exposure can be calculated using a map and a reduce, to average the exposure for each path.
Calculation of the CVA can then be done using equation (19), implemented in Futhark by using a map SOAC to discount future exposures, and reduction to aggregate the exposures into the CVA.

```
1 let EE = map(\xs -> in (reduce(+) 0 xs)/(f32.i64 paths)) exp
2 let dexp = map2 (\y z ->
3         y * (bondprice vasicek 0.05 0 z) ) avgexp times
4 let CVA = (1-0.4) * defaultprb * (reduce (+) 0 (dexp))
```

Listing 8: Calculation of discounted expected exposure and CVA

The expected exposure (EE) can also be plotted to visualize the exposure profile of a portfolio.

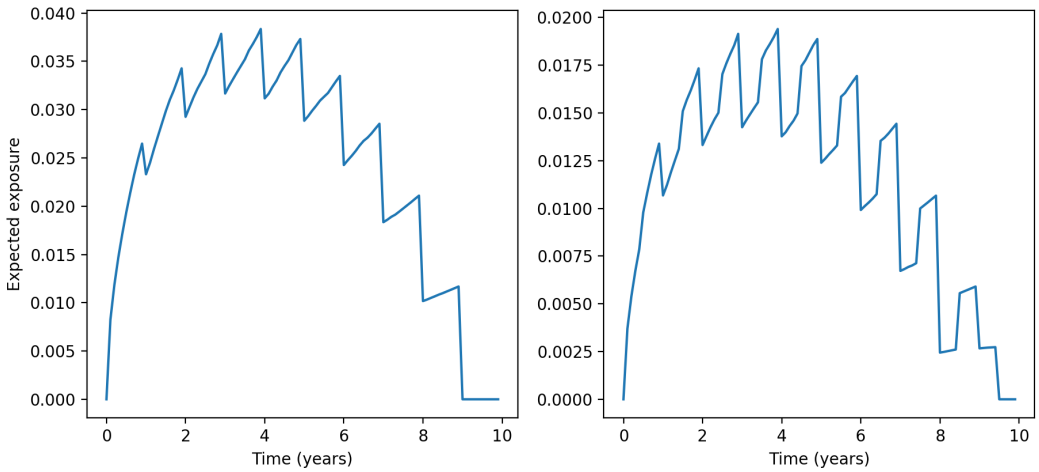


Figure 3: Exposure profile of a portfolio containing a single swap (left), and a netting set of swaps with opposite exposures (right)

5 Experiments

This chapter seeks to explore the performance of the two implementations of the CVA framework described section 4.2.2 and section 4.2.3. Both implementations are benchmarked using the futhark-bench tool, such that the performance of the program compiled to sequential C code, can be compared to the same program compiled to code invoking parallel OpenCL cores.

Two Futhark implementations of the CVA framework were created following the implementations described in sections 4.2. Both implementations have identical scenario generation and CVA calculation steps, but differ in the approach to the portfolio valuation step. The first implementation uses the nested map approach described in section 4.2.2, and is saved in the file `cva-map.fut`. The second implementation uses the expansion approach detailed in section 4.2.3, and is saved in the file `cva-expand.fut`.

The experiments are conducted with an Intel Xeon CPU E5-2650CPU with 8 cores, 128GB RAM and an GTX 2080 Ti GPU with 11 GB VRAM. All CVA implementations use 32-bit floating point numbers and 64 bit integers.

All experiments used a Vasicek model with the parameters $a = 0.01, b = 0.05, \sigma = 0.001$.

An experiment was first conducted to determine the performance of the two implementations when compiled using `futhark-opencl` and `futhark-c`. A portfolio was created containing 5 swaps with different specifications, and 4 different combinations of paths and steps were used to benchmark performance across different amounts of scenarios.

Paths	Timesteps	Total Scenarios
1000	100	10^5
10000	100	10^6
100000	100	10^7
1000000	100	10^8

Table 2: Input scenarios used for benchmarking the two CVA implementations

These various number of scenarios were then used to benchmark `cva-map.fut` and `cva-expand.fut` and record runtimes for C and OpenCL backends.

	Time (S)		
Scenarios	C	OpenCL	Speedup
10^5	0.357238	0.000411	869x
10^6	3.562312	0.001153	3089x
10^7	32.71632	0.008466	3864x
10^8	320.5624	0.070394	4553x

Table 3: Average runtime of 10 runs along with relative speedup of `CVA-map.fut` for C and OpenCL compilations.

	Time (S)		
Scenarios	C	OpenCL	Speedup
10^5	0.454182	0.001531	297x
10^6	4.708219	0.011679	406x
10^7	51.07664	0.106415	480x
10^8	481.0748	1.075012	448x

Table 4: Average runtime of 10 runs along with relative speedup of `CVA-expand.fut` for C and OpenCL compilations.

To investigate the relative performance of the compiled OpenCL versions of `cva-expand` and `cva-map`, the performance of the two versions is shown below in the same table, along with the relative speedup of `cva-map` over `cva-expand`.

	Time (S)		
Scenarios	CVA-expand	CVA-map	Speedup
10^5	0.00153	0.00041	3.72x
10^6	0.01160	0.00115	10.06x
10^7	0.10642	0.00847	12.57x
10^8	1.07501	0.07039	15.27x

Table 5: Relative runtimes of `CVA-map.fut` and `CVA-expand.fut` for OpenCL compilations

6 Evaluation

This section discusses and interprets the results achieved in the experiments, and discusses possible improvements that could be made to address performance and other shortfalls.

The experiments conducted showed that both implementation had significant speedups when compiled to parallel OpenCL code as opposed to compilation to sequential C code. Calculating CVA for a portfolio containing 10^8 resulted in a speedup of 4553x for parallel execution compared to sequential execution for the nested map approach, and the expansion approach resulted in a speedup of 448x. These speedups for GPGPU execution are similar to results seen in other financial applications of Futhark [11].

Across the different number of scenarios tested, the implementation of the nested map approach `cva-map` was on average an order of magnitude faster than the implementation of the expansion approach `cva-expand`, with the performance diverging more on larger datasets, with the nested map approach becoming relatively faster.

An explanation for the deviation in the performance of the two approaches is that the expansion approach uses the `expand` function, which has a higher computational burden than the more simple nested map approach for pricing of vanilla interest rate swaps. The benefit of using the expansion approach is that it is more versatile and can handle more complicated derivatives that would introduce more irregularity and thus not be able to be computed by the nested map approach. Thus, the nested map approach has faster performance, but more limited expressiveness than the expansion approach.

One limitation of the current implementation of the expansion approach is that the Futhark compiler stores the entire flattened array in memory, rather than chunking it as the nested map approach does. This leads to the expansion approach currently having memory errors when handling more than 10^8 scenarios, whereas the nested map approach overcomes this by chunking the arrays. However, this is an area that will be addressed by the developers of Futhark, meaning in the future, the expansion approach will not have the same memory limitations.

The CVA framework is also very well suited for execution on parallel hardware, as the framework is very computationally intensive, and not very memory intensive, with no required interdependency between computations. Futhark is in continuous development, with run time on benchmarks gradually getting faster as the development cycle continues, it would be reasonable to assume that the GPGPU performance will be even faster in the future [2].

It is important to note that the experiments were performed on systems with top of the line hardware, so the performance speedup for GPGPU execution may be less significant on a system with average hardware.

Both implementations had no calculation accuracy issues, and there was no difference in outputted CVA values across all the scenarios tested.

[2] https://futhark-lang.org/blog/2020-07-01-is-futhark-getting-faster-or-slower.html

7 Conclusion and future work

The aim of this thesis was to explore the potential of the Futhark programming language for performing financial simulations, particularly the CVA framework.

Functional programming constructs such as second-order array combinators (SOACS) and concepts such as irregular flattening were used to translate the CVA framework from the financial domain to a parallel Futhark implementation. Two methods for implementing a CVA framework in Futhark were proposed. The nested map approach utilizes Futhark's SOACS to value portfolios of vanilla interest rate swaps. The expansion approach uses the novel parallel programming construct of expansion to value portfolios, which allows for the framework to be able to price more complicated derivatives that would not be able to be handled by the nested map approach.

Experiments were conducted to measure the performance of the two implementations. Across various benchmarks, the nested map implementation was on average an order of magnitude faster than the expansion implementation. In general, the Futhark implementation was shown to yield a performance increase of more than 3 orders of magnitude for GPGPU execution, when compared to sequential CPU execution.

7.1 Future work

Futhark has recently added support for 64-bit precision floating point numbers, which could be implemented in order to further increase the accuracy of the CVA calculations. This could be relevant in situations where high volatility environments need to be simulated, where numeric precision is important to avoid round-off errors compounding, and for pricing derivatives requiring high amounts of precision.

Some modularity is present in the current solution, such as the use of the record system to define portfolios with several interest rate swaps, however the current implementation is limited to only a specifically definition of vanilla interest rate swap and a Vasicek model of the short rate. To become a viable CVA framework, the implementation could utilize Futharks higher order module system to make the framework more general and add support for portfolios consisting of generic instruments, detailed by valuation or payoff functions functions. Additionally, the scenario generation framework could also be made more generic, to add support for different models of market scenarios, as well as models of multiple market variables.

As mentioned in section 4.2.3, the current implementation is for vanilla swaps, which have a pricing function that is irregular, but is simple enough to be optimized and flattened by the Futhark compiler. More complicated derivatives will require pricing functions which are more irregular, like path dependent options or barrier swaps, requiring simulation for an unknown future period. These more complicated derivatives would not be able to be flattened by the current Futhark compiler when using the nested map implementation. The expansion implementation would be able to flatten the irregularity however, so programming more advanced derivatives would be a good way of demonstrating the benefits of the expansion approach.

References

[1] Basel Committee on Banking Supervision. Review of the credit valuation adjustment risk framework. *Bank for International Settlements 2015.*, 2015.

[2] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark.* 2018.

[3] John C. Hull. *Options, Futures, and Other Derivatives, 9th edition.* Pearson, 2015.

[4] David Lando and Rolf Poulsen. Lecture notes, finance 1. *University of Copenhagen, Mathematical Institute.*

[5] Tomas Bjork. *Arbitrage Theory in Continuous Time.* Oxford University Press, 3 edition, 2009.

[6] Rüdiger Seydel. *Tools for Computational Finance.* Pearson, 2009.

[7] Steven H. Zhu and Michael Pykhtin. A guide to modeling counterparty credit risk. *GARP Risk Review, July/August 2008.*

[8] Troels Henriksen. Design and implementation of the futhark programming language. *University of Copenhagen*, 2017.

[9] 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.

[10] Martin Elsman, Troels Henriksen, and Niels G. W. Serup. Data-parallel flattening by expansion. *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '19).*

[11] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Troels Henriksen Fritz Henglein, Maj-Britt Nordfang, and Cosmin E. Oancea. Finpar: A parallel financial benchmark. *ACM Transactions on Architecture and Code Optimization (TACO). Volume 13, Issue 2, Article 18. June.*, 2016.