Fast Calculations of Portfolio Credit Losses and sensitivities

Arvind Nayak

ING

TU Delft

Preface

My Master's program was far easier to start than to finish. For helping me cross the line, I am thankful to all my advisors for their infinite patience and tolerance with my weakly convergent behavior. Dr. Fang and Dr Xiaoyu Shen for introducing me to field of Computational Finance. Their regular and enthusiastic mentoring have been highly instrumental in my thesis work. I thank Prof. Kees Vuik for his constant guidance, kindness and for his valuable time. Finally, I owe a huge thank you to Prof. Drona Kandhai and the Quantitative Analytics team at ING for allowing me to conduct my master thesis with them. It was an huge learning experience.

Finally, I owe my existence, uniqueness and (hopefully) well posedness to my parents. I cannot thank them enough for their constant love and support throughout my life.

Abstract

Computing portfolio credit losses and associated risk sensitivities is crucial for the financial industry to help guard against unexpected events. Quantitative models play an instrumental role to this end. As a direct consequence of their probabilistic nature, portfolio losses are usually simulated using Monte Carlo copula models, which in turn play a decisive role in their measurement of risk metrics such as the Value-at-Risk (VaR). Semi-analytical numerical methods are alternatives to the Monte Carlo simulations to compute the distribution of the portfolio credit losses, the VaR and the VaR sensitivities. We find that numerical approaches such as the COS method, based on a Fourier cosine series expansion are superior to the Monte Carlo based computations in terms of both, the computational speed and the accuracy. Several studies have demonstrated these results, using the examples of various copula models in a single threaded environment. In this study, we extend that scope and critically examine modelling approaches for improving the computing efficiency of the COS method by investigating and validating, a multi-threaded GPU based algorithm for the COS method. In this process, we demonstrate the suitability of COS algorithm for parallelization on the GPU and highlight the performance improvements over existing methods.

Contents

List of Figures

List of Tables

1

Introduction

1.1. Context

Financial institutions are engaged in borrowing and lending of monetary products, which means they face two principal risks, namely market risk and credit risk. While market risk is associated with unexpected changes in the prices of market ingredients, credit risk corresponds to losses resulting from the failure of an obligor to make a payment. The source of risk for a bank from credit risk is so significant that market risk is analogised to be *'a pimple on the back of the elephant that is credit risk'*.[20] In order to reduce exposure, banks are involved in trading activities of such credit risky instruments. Thus, decision makers seek for methods and technologies that are practically useful for managing their credit portfolios and for enhancing their profits from trading these instruments.In the wake of the Global Financial Crisis of 2007-08, financial authorities have become committed to improve the existing directives for risk management processes and the related capital buffers. This is linked to the banks' obligation to protect against unexpected losses that may arise from widespread defaults throughout their portfolios. The Basel Committee on Banking Supervision (BCBS from now on) updated large parts of the regulations for financial markets. The biggest change in decades for market risk, is the Fundamental Review of the Trading Book (FRTB), to be in effect from 2022, but has already pushed banks to be compliant with it [cf. 4, 30].The main goal of the FRTB is to put appropriate capital charges on risks in the trading book. Previous regulation gave the opportunity to gain regulatory arbitrage by shifting credit related products from the banking book to the trading book and vice versa. The FRTB requires a different treatment of credit, and a sharper defined boundary between the trading and banking books. The FRTB regulation has not only consequences for capital calculation, but as well effects the granularity of reporting. Previously, reporting took place at company-level, under FRTB it has to be done at trading-desk level. A milestone example is represented by the competition between the Value at Risk (VaR) and Expected Shortfall (ES). The VaR was adopted as the official market risk measure in 1996 by BCBS, [see 36]. Then,the academic community pointed out its drawbacks, such as the lack of sub-additivity property.[24]

Historically, the Asymptotic Single Factor (ASRF) model described by Gordy [14] has been used for determining capital charges for credit risk. This single factor model has been built on the foundations of the work by Merton [28] and Vasicek [40], and by modifying the model it can also be applied to default risk in trading portfolios. The use of factor models is a popular tool to model correlations in large portfolios. The work of Vasicek [40] led to the widespread use of the Large Homogeneous Pool (LHP) model, which at first was used to model defaults in loan portfolios. In the LHP model lies the implicit assumption that defaults are correlated through a bivariate Gaussian copula. The use of the Gaussian copula method became conventional after the publication of the work of Li [26]. Research objective, questions and model xthe correlation of defaults. This led to a widespread application of the Gaussian copula in the world of finance. However, after the 2008 global financial crisis the bivariate Gaussian copula received heavy criticism. The main critique was on the lack of tail dependence implied by the Gaussian copula. Other copulas like the Student-t copula and the Clayton copula exist, which are known to imply fatter tails. Burtschell et al compare some of the factor copula models used in the context of CDO pricing. The BCBS, 2016a [cf 4, Pg.20] does give financial institutions the freedom of developing their own default models, as long as they are compliant to the BCBS' requirements for internal models on market risk. This in combination with the global introduction of FRTB regulation brings momen-

tum to (re-)develop default models.

The BCBS also sets new regulations on the calculation of capital resulting from the risk of default: the Default Risk Charge (DRC). The size of the capital buffer is intended to reflect the level of credit quality that the bank affords its clients. It is determined in terms of a confidence interval, up to which the bank claims it can absorb losses and remain solvent. As a result highly rated banks maintain large capital buffers to justify their rating. DRC under the internal model approach of FRTB is defined as the 99.9% Value-at-Risk (VaR) of the total loss distribution of the trading book positions that are subject to issuer risk. A review of the entire FRTB regulation for the DRC can be found in [5]. The recognized industry standard for measuring a portfolio's credit risk is the Monte Carlo (MC) simulation method. However, it is worth noting that a very high quantile level, 99.9% for DRC, MC computed empirical quantile can suffer from high variance estimation error. A huge number of heavy simulations are then required to get convergence. This implies a time intensive task which can be difficult to set up, especially for a large trading portfolio. The volume of banks data calculation is increasing each year with scale and complexity of different financial products. Subsequently, there is a need for fast and more importantly, accurate numerical methods to reliably price and quantify risks in the banks' portfolios.

In [34], MC simulation is replaced by a semi-analytical method to compute the distribution of the portfolio credit losses, the VaR, and the VaR sensitivities in factor copula models. In essence, it directly recovers the cumulative distribution function of the portfolio loss via the COS method, a method based on the Fourier-cosine series expansion. Fourier coefficients are extracted from the characteristic function of the portfolio loss which can be pre-calculated by means of numerical integration. Both the computational speed and the accuracy of this method are observed to be superior to MC simulation.

1.2. Research Questions & Contributions

Traditional methods still used in this field use Monte Carlo approaches that are computationally slow and inefficient. The semi-analytical COS method is an attractive alternative that demonstrates superior computational speed as well as accuracy. However, a drawback of the method is the impact on computational speed when extending the higher dimensional spaces, such as with the multifactor copula models. However, due to the "embarrassingly parallel" nature of the algorithm, the COS method can be optimized further with the help of data parallelism concepts such as the GPU device. The current thesis aims to contribute to the above outlined goal. The work investigates the combination of the factor-copula based COS method, its application to portfolio loss distributions and its sensitivities, multi-threaded throughput based programming models, all in the search for a optimal implementation of the baseline algorithm. Therefore, our two main research questions are:

> *Are there any suitable multi-threaded programming approaches for the COS method and if so, what are they?*

We tackle this question by answering the following subquestions:

- Literature has noted the computational complexities of the "workhorse" Monte Carlo methods and the COS method in computing large portfolios with high dimensional correlated structures. A GPU based throughput programming models are a much suggested alternative for this particular kind of workload: can we find the optimal multithreaded GPU based implementation for our baseline model?

- A parallel version of our algorithm can have multiple potential solutions. Can we propose a novel approach? What can be understood about the performance as well as in the accuracy in contrast to the existing serial implementation?

- Since our work is performed with the cooperation of ING's Quantitative Risk Analytics division within the Structured Products team. Our final aim is to compete with the existing models with the literature. Hence, an important question is: do the performance metrics improve upon the current industry standard models like the Monte Carlo method?

1.3. Thesis Organization

Chapter 1 provided the necessary information regarding the broader context, the main research questions and contributions of this work. We start with a brief overview of High performance computing relevant in the

case of numerical simulations and computational finance is explained. In chapter 2, we lay the groundwork of formal terms and definitions in portfolio credit risk management along with its associated risk measures and attributions. We explain the application of risk measures such as VaR in important modeling approaches used by the financial industry for instance, the Default Risk Charge(DRC) introduced by BCBS. Finally, we briefly explain the important algorithms present in parallel computing, that are relevant to accelerating portfolio loss calculations. In Chapter 3 we introduce the mathematical representation of different Credit Risk models, i.e. the mathematical definitions of portfolio default loss, measures under different multifactor copulas. We review the COS method and show how risk measures like VaR and ES are derived from it. Finally, with the help of test examples, we list different results and convergence characteristics of the COS method, that serve as a benchmark to our GPU implementations. Chapter 4, will detail the three GPU algorithms for the COS method that we have chosen to explore in this study along with several numerical experiments detailing performance characteristics and improvements. Through these examples, we expect that the associated challenges, specifically the computational complexity due to *"curse of dimensionality"* will become apparent with the existing Monte Carlo and COS methods. Finally, performance results in contrast with a few existing CUDA libraries are analyzed, motivating our approach to the problem. In closure, Chapter 5 summarizes the main findings of this study and describe the scope for this work, which allows us to suggest potential areas for future work.

2

Preliminaries

In this study, we briefly present the main concepts needed to set our understanding of our study. First, we review the common high performance computing architectures, including a brief overview of GPU based computing and the CUDA programming framework. We then shift our focus on specific trends of High performance computing prevalent in the computational finance sector, including existing work on option pricing and risk management. This leads us to introduce our topic of focus in this thesis, Portfolio Credit Risk Losses and accelerated algorithms for its simulation.

2.1. High Performance Computing

In the context of financial calculations, the Monte Carlo method has a relatively slow rate of convergence, and thus becomes a trade off between number of simulations and the required error tolerance. A number of optimizations exist to increase the rate of convergence, some of them being improved numerical algorithms such as the COS method which we later discuss. The purpose of this section is to introduce the reader to High Performance Computing as an alternative. This type of optimization alongside the COS method forms one of the focuses of this work. Eventually, we would like to explore the advantages of both higher fidelity and fast algorithms.

An upgraded hardware may result in a higher CPU rate for instance or the use of advanced compiler can result in better code optimization. A less straightforward but promising approach consists of parallelizing the application. If utilized to a certain potential, parallelization can ensure higher acceleration. It may be that the code be required to be refactored, (e.g. CUDA, openCL). In other cases, few added lines could be enough to enable a parallel execution of a portion of a code, in general for/while-loops (e.g. OpenMP, OpenACC). Nonetheless, parallelizing with the introduced methods requires a know-how of our used hardware and the specifics of our application.
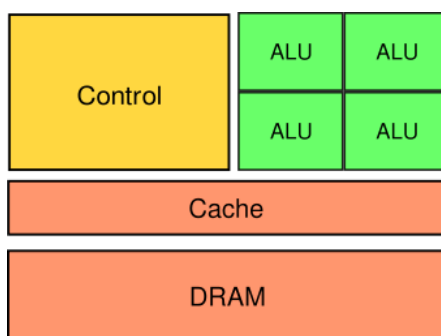
Figure 2.1: Shared Memory Architecture

High performance systems can be classified following the memory architecture. The two main architectures are the shared-memory and the distributed-memory systems. For the shared-memory architecture,

a set of processors shares the same memory contingent. Current popular architectures such as multi-core CPUs belong to this group. OpenMP can handle parallelization tasks in such systems. With the help of few simple compiler directives (#pragmas) surrounding sequential for-loops can divide automatically the work between available cores and every core processes a part of the loop. The communication among processors is very simple since they all share the same memory contingent. The maximum available number of cores and the memory capacities for this system are, however, too low to cover large-scale problems.
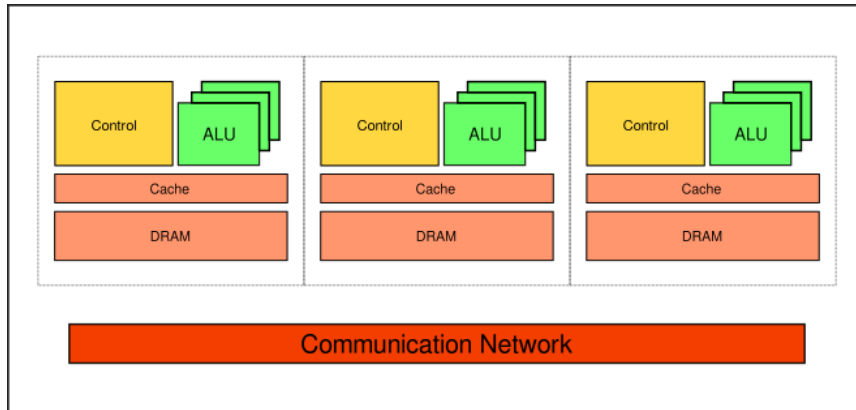
Figure 2.2: Distributed Memory Architecture

In the distributed-memory configuration, better known as clusters, every processor has its own memory. The communication between processors occurs through the Message Passing Interface (MPI). A decomposition of the computational domain is essential for the parallelization on distributed-memory systems. Every processor contributes to the solution of the simulation by solving a part of the computational domain. A high number of cores (e.g. cluster of CPUs) could speed up the whole process significantly. But the parallelization increases also the programming burden, since the designer has to distribute the computational work among the available CPU processors and regulate the communication. For a realistic application, a large number of cores is essential and MPI is the most implemented paradigm on today HPC systems. Hybrid systems, consisting of a cluster of shared-memory systems, are getting more and more popular. The most powerful HPC systems nowadays are hybrid, where the first benchmark measurement of exaflops performance was made [refer 37]. Hybrid systems use in every cluster node not only standard CPUs but also accelerators such as Field Programmable Gate Arrays (FPGAs) and Graphics-Processing Units (GPUs).
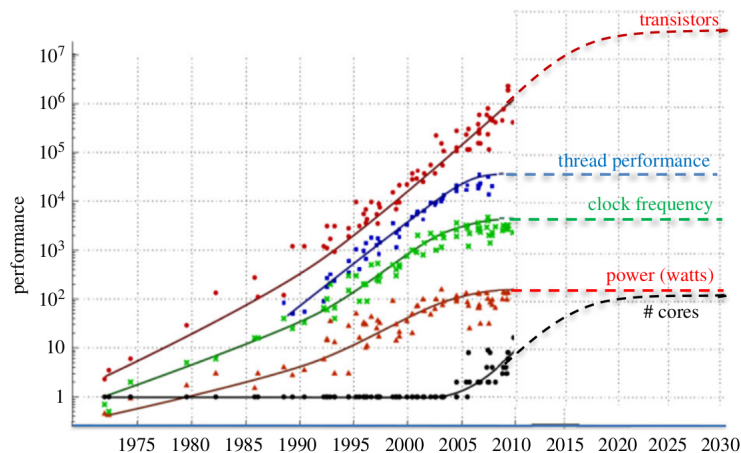
Figure 2.3: Different characteristic trends for general purpose processor technologies

Source: Figure from Kunle Olukotun, Lance Hammond, Herb Sutter, Mark Horowitz
extended by John Shalf [see 33]. (Online version in colour.)

2.1.1. GPU vs. CPU
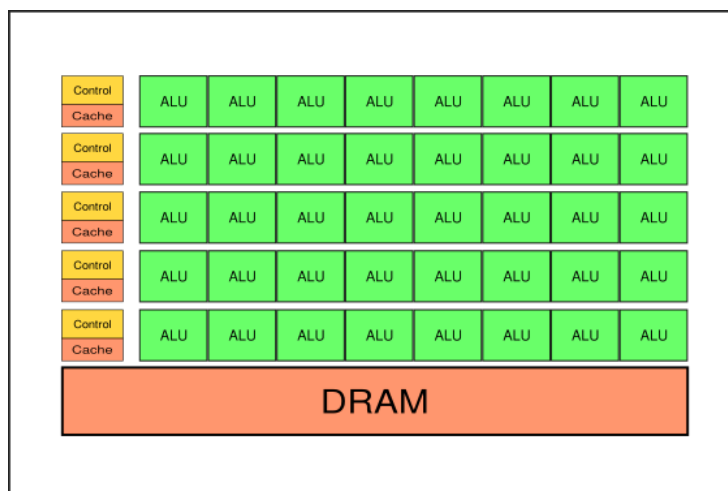
Figure 2.4: Massively parallel architecture (GPU)

Before investigating the details of GPU computing, it is perhaps worthwhile to discuss the drivers behind the recent shift towards GPU computing specifically as well as towards massive parallelism in general. Moore's Law and Dennard Scaling have guided the development of the semiconductor industry. Traditionally, the performance gains realised in computing have been attributed to increasing amounts of transistors on integrated circuits (ICs) as well as the frequency with which these transistors could switch. Hence, Moore's law states that the number of transistors doubles every 1.5-2 years, and thus increasing the logic and computing power has been true since its inception circa 1965. Dennard scaling described that the energy consumption of a chip would stay in proportion to the size of the chip. Shrinking transistor sizes (i.e., reducing the process size) allowed us to increase the computing capabilities of the device without consuming more energy. Moore's Law and Dennard Scaling charted a promising future for computers. As we reduce the transistor size, the performance of computing devices would exponentially increase over time. But it has been predicted that within a decade (see Figure 2.3), the technological underpinnings for the process that Gordon Moore described will come to an end, as lithography gets down to atomic scale.[33] As the transistor sizes is approaching the practical limitations of physics, reducing transistor sizes can hardly improve performance. To guide future semiconductor research, we need to revisit these historical development trends and find new directions that can drive performance improvement. GPUs offer a way to continue accelerating applications as it has been witnessed already in fields such as graphics and supercomputing[see 17, chap 4].

GPUs can be considered as a shared-memory system since a number of multi-processors are connected to the same device memory. Notice the architectural differences between a traditional CPU and a GPU as depicted in Figures 2.1 & 2.4. A typical GPU contains hundreds of cores almost entirely dedicated to single and double-precision arithmetic operations, called Arithmetic Logic Units (ALUs). Compared to a CPU core, a GPU core is less powerful. The conventional multi-core CPU relies mainly on a large and fast memory cache, which serves a small number of cores, in order to reduce the instructions execution time (latency) to a minimum. A significant portion of the transistors is dedicated to the instruction flow control (instruction pipelining, branch prediction and similar tasks). The CPU cache has a spacial and a temporal locality as data is more likely to be kept in the cache if it is often used (temporal) and if it is close to an often used data (spatial). For a CPU, the computational power is provided by a high-speed processor optimized for serial operations and low latency. A GPU includes, however, a large number of these cores, which combined result in a higher computational power. Hence, the second advantage is the specialization of the GPU cores. The GPU cache is much smaller and disposes of spatial locality. The memory bandwidth in CPUs can be hoarded by requirements from the operating system and I/O devices, which can make it more difficult for it to increase. This is reflected in a performance gap between CPUs and GPUs in in terms of arithmetic throughput (FLOPS: FLoating OPerations per Second) and memory bandwidth (amount of data transferred per second). [see 21, p.10]

2.1.2. GPU Computing

Today's GPUs evolved from graphics cards installed in most computers starting from the eighties [cite some source]. A graphics card is a complex electrical circuit that processes graphical data sent from the CPU to render and visualize it on the monitor with increasing quality and refreshing rates. The primary idea was to offload the CPU from rendering images on the display and use a dedicated hardware instead. A high pressure on graphics cards for fast refreshing of pictures (mainly for video-gaming) caused the spectacular increase of the computational power reflected by the large number of cores packed in one card. The high computational power attracted scientists and engineers looking for low-cost high-performance alternatives to speed up their numerical calculations in different applications. However, to use these first graphics cards, scientists had to adapt their problems to the partially programmable GPU, which implied a change to the data storage and the programming language. The term GPGPU, which stands for General Purpose Graphics Processing Unit, was established for this type of use of the graphics card. In response to this emerging demand, NVIDIA released in 2007 the first fully programmable open graphics processing units in a C-based programming language called CUDA. At the same time, AMD released its programmable GPU with OpenCL. The programming model CUDA is specialized for NVIDIA GPUs whereas OpenCL is preferred on AMD GPUs.

2.1.3. CUDA C programming

This subsection explains briefly some concepts of GPU programming implemented in the CUDA C or CUDA language. An extensive and detailed treatment of the topic can be found in the CUDA user manual [1]. CUDA is based on the C programming language with a minimal set of extensions to handle the parallel execution and the memory organization. The main computing systems involved in CUDA programming are the *host*, which is the traditional CPU in a personal computer and the the *devices*, which are the massively parallel processing devices, typically the GPUs. Many software programs have sections that may exhibit *data parallelism*, a term used for a case that allows arithmetic operations to be safely performed on different parts of the data structure in parallel. A CUDA program can help accelerate these sections of programs by running them on CUDA compatible GPUs, which as we discussed in 2.1.1 have massive number of arithmetic units. Hence a CUDA program contains both host code and device code. The functions running on the GPU are called kernels and are executed by threads, which are organized in grids of blocks distributed among the multiprocessors. CUDA kernels are launched as follows:

```
kernel_name <<<nB,nT>>>(args);
```

where, nB he number of thread blocks launched and nT the number of threads per block. A kernel is usually executed by $nB \times nT$ threads, also known as a grid; though a run with one block of one thread is equivalent to a serial run on a CPU. Each thread is executed by a core of the GPU and a thread block can be computed on a streaming multiprocessor. Several concurrent blocks can reside on one streaming multiprocessor depending on the blocks' memory requirements and the processor's memory resources. The GPU starts threads always as a multiple of the warp size, in which the input blocks are divided into thread wide units. The division is implementation specific, though currently the warp size is 32 contiguous threads per block. Therefore, the number of requested threads per block is recommended to be a multiple of 32 otherwise, the last warp is not fully used and its extra threads are inactive but consume, nevertheless, registers and shared memory space. When all threads of a kernel complete their execution, the corresponding grid eliminates, and the execution continues on the host until another kernel is launched. Multiple kernels can execute on a device at one time. Let us illustrate a typical CUDA programming structure using the simple example of scalar vector addition (SAXPY), a popular benchmark in computing, $\mathbf{y} \leftarrow \mathbf{y} + \alpha \mathbf{x}$ where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ and $\alpha \in \mathbb{R}$. First it is helpful to review the CPU-only implementation. (see Listing 2.1)

Listing 2.1: sequential SAXPY in C

```c
// compute saxpy
void saxpy(int N, float a, float *h_x, float *h_y)
{
  for (i=0; i<N; i++) h_y[i] = h_y[i] + a*h_x[i];
}

int main(void)
{
  // memory allocation for a , h_y and h_x
  // I/O to read h_x and h_y, N elements each and scalar a
```

[1]https://docs.nvidia.com/cuda/index.html, retrieved December 2020

```
   ...

   saxpy(N,a,h_x,h_y);

}
```

We prefix the variable names that are mainly processed by the host with `h_` and that by the device with `d_`. The implementation is straightforward. The `saxpy()` function uses a for loop to iterate through the `N` vector elements and the ith iteration `h_y[i]` receives the sum of `a*h_x[i]` and `h_y[i]`. Since the variables are passed by reference, when the function `saxpy()` returns, the new contents of `h_y` can be accessed.

A standard way to execute vector addition in parallel is to replace the sequential loop with a grid of threads (see Listing 2.2).

Listing 2.2: Revised parallel SAXPY for CUDA

```
#include <cuda.h>
...
__global__
void saxpy(int N, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if(i<N) y[i] = a*x[i] + y[i];
}

 int main(void)
{
  // I/O for host variables x, y and N
  // Allocate d_x, d_y pointers to device memory

  ...

  //copy x,y to device memory
  cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

  // Perform SAXPY , warp size is 256
  saxpy<<<ceil(N/256.0),256>>>(N, a, d_x, d_y);

  //copy back result to host memory
  cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);

  //free device pointers
  cudaFree(d_x); cudaFree(d_y);

  ...
}
```

Notice the use of keyword `__global__` before the function declaration. This instructs the compiler to generate a kernel function, executed on the device but callable from the host. Other keywords such as `__device__` and `__host__` define purely device and host functions respectively, callable only inside the particular system. `saxpy()` kernel is launched using `N/256.0` thread blocks and `256` threads per block. There exists a set of predefined variables `threadIdx`, `blockIdx`, `blockDim` that assign each thread with a unique global identifier. Every thread treats a small subset of elements, and the programmer must ensure the proper use of an index inside the kernel. Kernel calls are asynchronous since the control is returned to the host immediately after the call, which can then proceed with the control flow. A synchronization or a memory copy can force the CPU to wait for the completion of the last called kernel. Finally, CUDA supports a set of runtime API functions, for managing data in the device memory. In the above example, `cudaMemcpy()` function is used for data transfer between device and host memories. `cudaFRee()` is then used to free the object from device global memory.

It should be emphasised that the scalar vector addition example used above is trivial. Infact, the overhead of allocating device memory, bandwidth between host and device, and final memory deallocation will likely make the resulting code slower than the its sequential counterpart. Relative to the floating point operations performed in this example, data processing requires more work. Hence an important point to note is that, GPU code optimization would not always accelerate every algorithm. A careful understanding of the use

case and the device architecture is therefore, a must to reach good performance. We will discuss this in later sections for our excercise. Several popular numerical libraries now exist in CUDA optimized for production, such as CUBLAS [2], CUSPARSE [3], CUFFT [4]. For brevity, we omit talking about other available functionalities here. An in depth discussion for other CUDA features can be found in the official CUDA manual and in textbooks such as [21].

2.2. High Performance Computing in Finance

Almost 10% of the TOP500 supercomputers, are dedicated for computational finance purposes[31]. The optimization of diverse mathematical algorithms used in financial mathematics marked with the use of high performance computing has been promising. In this domain, computationally intensive financial problems attempt to utilize computing systems to improve models and computations. This is achieved by practical numerical methods and techniques for analyzing economical trade-offs. This approach leads to a rather broad spectrum of applications, all requiring HPC capabilities. The increased performance as well as the scalability provided by GPUs means that institutions are able to increase business at the same time as having tighter control on their exposure. We observe the pervasive use of GPU computing in several areas of quantitative finance. Core research in computational finance covers three main areas: option pricing that targets organizations such as banks, high-frequency algorithmic trading for hedge funds and actuarial science aimed towards the insurance industry.

2.2.1. Option Pricing

Options are means which can be used to harvest information emerging from stock movements. Options are formed from the underlying assets. These underlying assets usually take the form of a stock. Thus, the main difference with traditional investment approaches as stocks, is that stocks inherit their values. These instruments i.e., the options, can be obtained by different means. Usually, these option contracts, provide purchasers or holders with a number of rights. These rights give permission but not the obligation to them for buying stocks with a given value for specific time period. The given value is called strike value while the time period is called expiration date. In the same manner, holders can trade their options in the pre-negotiated value up to a specific date. One of the many reasons why option pricing is such a popular candidate for accelerations is the nature of the underlying models that are used[11]. For example, typical option pricing applications are Monte-Carlo, making them embarrassingly parallel and therefore suitable for GPU acceleration. Other approaches are PDE based, so GPUs are again a very convenient choice due to the amount of parallelism that can be exploited. We can see the application of GPUs in option pricing in many works [see 35, 38, 41]. High Frequency Algorithmic trading is another popular application for performing buy and sell operations. Stock market exchanges are highlighted by the tremendous demand for high speed calculations. The introduction of data feeds almost thirty years ago, initiated a new era of high frequency trading. Data feeds are constantly updated, while the stock market is open. These updates are regarding information based on the rates of ordered, canceled and executed stocks. Traders have live access to this information and they are responsible on take action on buy or sell. Also, responses to market events like currency exchange values, buy/sell of big companies or any other significant event, needs to be done in microseconds in order to achieve the best prices. With today's explosion of data, sophisticated computing systems are used to perform these actions. [10]

2.2.2. Risk Management

Usually, financial decisions are accompanied by a specific strategy. These strategies are optimized with risk-return trade-off in mind. One needs to ensure that implementation, following the strategy and all its processes can be effectively tracked. Thus, as cNeil and Embrechts stated in [27], "Risk Management is defined as the process of tracking risks that are related to the above-mentioned process, taking control actions at times when expected or unexpected events appear." Thus, for individual investors or larger organizations, effective risk management is in particular demand, in order to achieve their goals and minimize the exposure to un-tolerated risks. A typical application in this field is the management of risk in large credit portfolios of banks.

---

[2]https://docs.nvidia.com/cuda/cublas/index.html

[3]https://docs.nvidia.com/cuda/cusparse/index.html

[4]https://docs.nvidia.com/cuda/cufft/index.html

2.2.3. NVIDIA GPUs in Computational Finance

Computational finance tends to use NVIDIA GPUs for accelerating its applica- tions. According to NVIDIA[5], 85% of the clients that use accelerators, prefer NVDIA GPUs. In addition, for the remaining market share, 4% use Intel Xeon Phi co-processors and 11% prefer other methods. An increasingly large group of companies in the financial industry that chose to use NVIDIA GPUs as accelerating platforms. Examples are JPMorgan[29], nag[40], jedox.[30] and SUNGARD. JPmorgan moved from the traditional approach of using GPGPUs for their risk management calculation to deploy NVDIA GPUs for that purpose. Risk management computations include equity-derivative calculations. Thus, by moving to Tesla GPUs, they managed to accelerate the performance of their application by a factor of 40x, thus calculating the required risks from hours to just few minutes. This upgrade led to additional power efficiency benefits on their data centers[32]. Another important contribution in that field is from NAG, a supplier for HPC computational software. Usually, their products offer optimizations, parallelization and restriction of numerical applications in order to be able to follow cutting edge trends. Therefore, as they offer products related to algorithmic differentiation for mathematical derivatives aiming to the financial industry, HPC techniques consist their main choice for acceleration. The NAG Numerical Routines for NVIDIA GPUs resulted in a 10x improvement for Monte-Carlo based index pricing.[32]

2.3. Credit portfolio losses

The primary objective of Credit Risk Management is to improve the ability of banks to identify optimal credit portfolios. It is thus, important for financial institutions to possess tools for pricing and managing their trading portfolios. Let us now see how this can be quantified. On an obligor level, the three basic components of credit risk are:

- exposure at default (ϵ_n), the total amount that the bank is exposed at the time of default of the n-th obligor. The bank can calculate ϵ_n at any specified time.

- loss given default (v_n), this factor quantifies the losses incurred by the bank when there is a default of the n-th obligor. v_n is calculated separately from the ϵ_n because the bank is unlikely to lose all the value that they may be exposed to. There can be some recovery from each obligor based on a certain Recovery rate (R_n). So the loss given default then becomes,

$$v_n = A_n(1 - R_n) \tag{2.1}$$

  where A_n is the notional amount of the n-th obligor.

- probability of default of the n-th obligor (PD_n) within a fixed time horizon.

ϵ_n is usually assumed to be deterministic. v_n is a random variable and its uncertainty arises from PD_n. Suppose the random variable x_n represents the creditworthiness of the n-th obligor. The n-th obligor is said to have defaulted if x_n falls below a certain default threshold $\xi_n(T)$, where T can be taken as fixed time horizon for bond maturity. $\xi_n(T)$ is based on the Merton's structural model of default [28]. Suppose the asset value of the borrower V follows a geometric Brownian motion with initial value V_0, drift μ and volatility σ, so that $dV_t = \mu V_t dt + \sigma V_t dx_t$, where x_t is a Brownian motion process representing the creditworthiness of an obligor. The asset value at horizon T can be given as

$$V_T = V_0 exp(V_0 + \mu T - \frac{1}{2}\sigma^2 T + \sigma\sqrt{T}x_T)$$

. The obligor defaults if $V_T < B$. Hence $\xi_n(T)$ is formulated as,

$$\xi_n(T) = \frac{\ln B_n - \ln V_n - \mu_n T + 0.5\sigma_n^2 T}{\sigma_n\sqrt{T}}$$

The probability of default for obligor n at time T can then be formalized as

$$PD_n(T) = P(\tau_n \le T) = P(x_n \le \xi_n(T)) \tag{2.2}$$

---

[5]NVIDIA. Introducing nvidia tesla gpus for computational finance, jul 2016.

Let us now consider a credit portfolio containing N obligors. We consider the risk/loss of the $n-th$ obligor as a random variable $L_n \colon \Omega \to \mathbb{R}$, where we work on a probability space $(\Omega, \mathscr{F}, \mathbb{P})$. The dependence on T can be suppressed for simplicity, since typically T is chosen to be one unit. The loss incurred by obligor n is given by,

$$L_n = \epsilon_n v_n \mathbb{1}_{x_n \leq \xi_n(T)} \tag{2.3}$$

It follows that the portfolio loss is given by

$$L = \sum_{n=1}^{N} L_n = \sum_{n=1}^{N} \epsilon_n v_n \mathbb{1}_{x_n \leq \xi_n(T)} \tag{2.4}$$

where $\mathbb{1}_A$ is the indicator function which equals 1 when A is true and zero otherwise.

2.4. Quantitative Risk Measures

In order to manage credit risk it is imperative to quantify credit risk on a portfolio level. Financial institutions calculate the required capital to shield against possible extreme losses and hence can determine whether they are adequately compensated for the risk incurred. These are often measured by Value at Risk (VaR) which is the quantile of the portfolio loss distribution at a given confidence level. Aggregation of credit risk from individual obligors to a portfolio level involves specification of the correlations among obligors. Factor models are utilized in common practice , in which the obligors itself are independent but are dependent on some common factors, e.g systemic risks.(c.f Chapter 3) We discuss this at length in Chapter 3. VaR for a given confidence level α is the α-quantile of the portfolio loss distribution L and therefore is given by,

$$VaR_\alpha = \inf\{x : P(L \leq x) \geq \alpha\} \tag{2.5}$$

VaR has faced criticism to not being *statistically coherent*[cf. 1]. In particular VaR does not satisfy subadditivity,i.e., the VaR of a portfolio can be larger than the sum of the VaRs of its subportfolios.

One of the fundamental areas of applications of such quantitative risk measures like the VaR are the Risk charge regulations, introduced by the Fundamental Review of the Trading Book (FRTB)[4], a document published by the Basel Committee for supervising financial institutions with their risk modeling activities. Risk charge regulations, for example the Default Risk Charge "*captures default risk of credit and equity trading book exposures with no diversification effects allowed with other market risks*" (BCBS, 2016a). The model is based on a large number of Monte Carlo simulations of portfolio default loss scenarios, and the DRC figure corresponds to a VaR at 99.9% confidence level. The aim is to calculate a total financial outcome of different scenarios that a bank might face, which may depend on several internal and external factors. As a consequence, the DRC uses a multifactor Gaussian copula model (which we will describe in greater detail in the next section). The set of factors can represent different characteristics like for instance, the geographical regions or the sectors of the economy. For each of these factors, an index is chosen that encodes their performance. The correlations between these data, corresponding to these factors are then measured along with the creditworthiness of the obligor present in the portfolio. Based on matching a threshold obtained from the PD_n of issuer and L_n, we find whether the financial impact of the default of the issuer should be added to a total portfolio loss L. In cases were no default occurs the contribution of issuer is set to zero. By going through all the issuers and all the scenarios, the vector with the cumulative financial impact across all issuers per scenario, L, is created and subsequently sorted. From this vector a 99.9% quantile is taken, which is the DRC value. Such models and metrics help financial institutions manage their risk while taking into account different factors.

2.4.1. Risk Allocation

After computing the overall risk distribution of a portfolio like for example, the above mentioned case of the DRC calculations, an equally important task as to quantify the portfolio level credit risk is to measure how much each obligor in a portfolio contributes to the total risk, i.e., the risk contributions of single exposures. The Euler principle of risk allocation has been shown (e.g. in [8]) to be the only compatible principle to the subadditivity of a risk measure. First, the Euler decomposition is homogeneous, meaning that the risk attribution of a portfolio scales in proportion to the size of the portfolio. Second, the decomposed risk across obligors or sub-portfolios sums up to the risk measure of the portfolio.

Decomposition of ES or VaR boils down to solving conditional expectations of losses of obligors or sub-portfolios, conditioned on the tail event that characterizes the risk measure. We consider the following definition for allocating ES by the Euler principle:

$$CES_n = \mathbb{E}\left[\mathbb{1}_{x_n} \leq \xi_n L_n | L \geq VaR_\alpha\right] \tag{2.6}$$

Both ES and CES_n scale in proportion to the size of the portfolio.

$$ES = \mathbb{E}\left[\sum_n \mathbb{1}_{x_n} \leq \xi_n L_n | L \geq VaR_\alpha\right] = \sum_n CES_n \tag{2.7}$$

Similar to Conditional ES, Conditional VaR decomposes VaR by the Euler principle for risk allocation. Consider the definition,

$$CVaR_\alpha = \mathbb{E}\left[\mathbb{1}_{x_n \leq \xi_n} \cdot L_n | L = VaR_\alpha\right] \tag{2.8}$$

such that

$$VaR = \mathbb{E}\left[\sum_n \mathbb{1}_{x_n \leq \xi_n} \cdot L_n | L = VaR_\alpha\right] = \sum_n CVaR_n \tag{2.9}$$

2.4.2. Accelerating Portfolio Credit Risk Calculations

In simulating credit risk losses of large portfolios, which is the focus of this work GPU algorithms have implemented Monte Carlo methods. Monte Carlo simulations are very computationally expensive owing to the slow convergence properties of the tails of the loss distribution; they require no less than a CPU server farm consisting of several hundred cores running over several hours. The motivation of GPU acceleration is to investigate the extent to which a single GPU can replicate the performance of a typcial multicore CPU and, if so, what new capabilities does a GPU implementation offer. Monte Carlo analysis are embarrassingly parallel processes and hence an good fit for GPU both algorithmically and architecturally, Moreover, the onboard memory provides both the capacity and bandwidth to complement the frequent processing of large portfolio data sets. We now look at a few core algorithms that are useful for implementing the Monte Carlo simulation. [11, 32]

2.4.3. Random Number Generation

A core feature of the Monte Carlo simulation is the random number generator requiring almost $O(10^{15}) \sim O(2^{50})$ samples with good statistical quality. The GPU architecture also places additional requirements on the generator: the ability to generate different substreams on parallel nodes with no correlations between substreams on different nodes. If we use the same generator in all parallel processes, the effective generator over the whole process will produce stretches of identical values. A master-worker generator, that generates random numbers on one thread and distributes them to other processes, causes considerable overhead or may again cause correlation between processes. An alternative is to set up independent generators with different parameter choices. One thread may generate the starting point for the other thread's generated sequence. A popular random number generator is the *Mersenne twister*(MT19937). It has been adapted to allow parallel streams of uncorrelated numbers but may require lots of memory. As a result, the former may not be ideal for the GPUs.

Two different GPU optimized generators, the pseudo random and the quasi-random number generators have been benchmarked on the GPU and the CPU. The first is a pseudo-random number generator which uses an implementation of L'Ecuyer's multiple recursive random number generator (mrg32k3a) described in [25]. This generator ports well onto the GPU because of its inherent structure of long streams and substreams, providing a substantially large period of 2^{191}. The latter is a quasi-random number generator which uses a Sobol sequence implementation described in [6]. The block and grid structures on the GPU complement the Sobol implementation. Using pseudo-random numbers, offers a rate of convergence $O(N^{-\frac{1}{2}})$, whereas with quasi-random numbers this rate of convergence can improve to as much as $O(N^{-1})$. Making use of the latter scheme would require fewer simulations to achieve the same convergence properties and effectively offer a speed-up.

2.4.4. Sorting Loss Distribution

Calculating the capital buffer based on the analysis of the portfolio losses requires the loss vector to be sorted, a process which is particularly efficient. Performance benefits can be gained from the careful ordering of the

portfolio data, which results in better arrangement in memory and hence permitting efficient data requests. Sort algorithms are supplied within the Thrust library [6]. This library has several performance optimized algorithms- merge or radix sort.

2.4.5. Numerical Integration

Integration is one of many types of numerical computations that is highly suitable for parallel processing. Due to the linearity of the integration operator, no communications among the processors are required during computation, one can achieve high parallel efficiency. In addition, it scales well with many workers. At the end of computation, a many-to-one, collective, communication is required to collect the integral sums from all the processors and compute the final integral sum. In context of GPUs, this approach implies that the points or the subregions of discretized domain processed at once is equal to the number of threads. A GPU has to hold the state of each thread, hence the number of threads executed simultaneously maybe limited by insufficient GPU resources. Popular approaches with summation operations can be applied to numerical integration tasks easily. These include the parallel sum reduction which uses a tree-based approach within each thread block to compute individual sums. A schematic Fig. 2.5 shows this hierarchical approach. Let us consider without loss of generality, a scalar valued function $h(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}$ to be integrated over a domain D partitioned as $\bigcup_i^n D_i$:

$$I = \int_D h(\mathbf{x})\,d\mathbf{x} \approx \Sigma_{i=1}^n \overbrace{\int_{D_k} h(\mathbf{x})\,d\mathbf{x}}^{H_k}$$
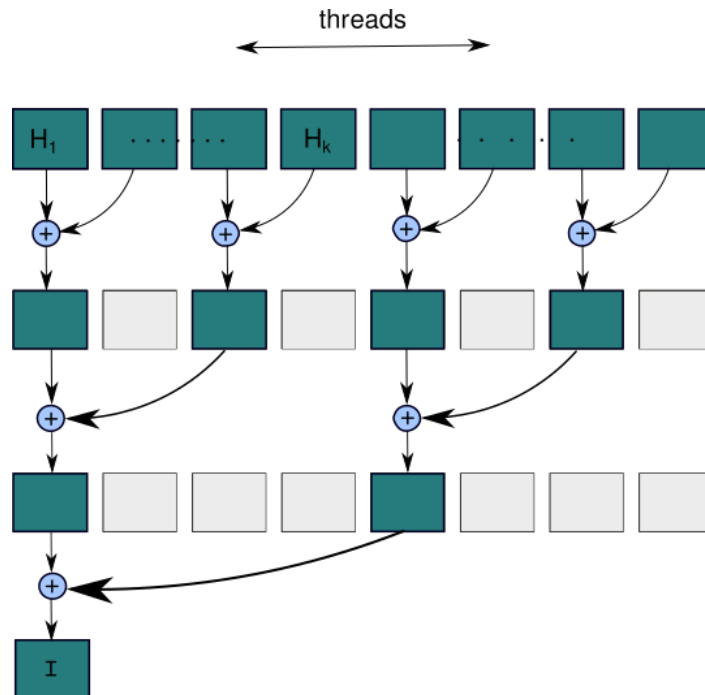
Figure 2.5: With each operator +, the parallel reduction algorithm repeatedly groups each computation of a subintegral, H_k in pairs. Each pair is computed in parallel with others, halving the overall array size in one step. The process is repeated until a single element exists

---

[6]http://code.google.com/p/thrust/

2.5. Summary

In this section, we survey all the related concepts that is relevant to this project. We analyze computational finance areas that broadly use HPC techniques. Such a domain is option pricing and risk management for which we note that GPUs are the traditional way to tackle this problem. As CUDA based multithreaded programming model is suggested to be attractive for increasing the computational performance of financial applications, we identify our research aim that discusses this in our focus of portfolio credit losses, and note potential algorithms that could be used for its optimized simulation. In conclusion, from our review of related work, we couldn't discover any other project that uses GPU-based computing for accelerating Portfolio Credit Losses simulations using the COS method. Therefore, our work in the following chapters proposes a novel approach in such simulations with CUDA.

3

Modeling Portfolio Loss Distributions

In section 2.3 we introduced the concept of credit portfolio losses and various defined measures that are beneficial for determining the optimal portfolio. We now outline a few mathematical approaches to model these portfolio losses. Models for estimating credit risk has progressed for a long time. There are now three main approaches to modelling portfolio loss distributions of debt instruments: the "Merton-style" approach, the purely empirical econometric approach, and the actuarial approach. Each of these approaches has, in turn, produced several models. As described in 2.3, our concentration in this thesis will be on models based on structural default approach of Merton[28] in a *static* context. A key challenge in all such models always has been, to explain the relationship between default event, including on the dependence between defaults, Loss given defaults and Obligor default probabilities. Since direct modelling of the pairwise dependencies becomes unfeasible as credit portfolios become larger. Models follow different approaches to reduce this computational complexity by utilizing several kinds of approximations. The earliest credit risk measure models that were published, are popular industrial examples such as CreditMetrics by J.P. Morgan, KMV's Portfolio Manager, CreditRisk+ by Credit Suisse. These models, though are slight variants of each other, essentially use a factor based approach to introduce default dependence via a few common market variables and obligor dependent idiosyncratic variables. We refer to [9] for a detailed description of the above. Before understanding what factor based models are, let us give a brief overview of an important tool used for representing correlations.

3.1. Copulas

Copulas are tools to describe the interrelation of several random variables. Let us consider an explanatory example: what can be a simple way to map a one dimensional random variable, X, to a one dimensional standard normal, Y? The answer is built on a transformation that takes either one to a standard uniform random variable, U. Standard uniform means that the probability density of U is $h(u) = 1$ if $0 \leq u \leq 1$, and $h(u) = 0$ otherwise. The copula construction is based on the above mapping. The cumulative distribution function (CDF), $F(x) = Pr(X \leq x)$ maps any random variable, X to a standard uniform. If X is a random variable whose CDF is F , then $U = F(X)$ has the standard uniform distribution. This goes both ways. If U is standard uniformly distributed then, X can be found by solving $F = U(X)$. An N-dimensional copula is a distribution function on $[0,1]^N$ with standard uniform distributions:

$$C(u) = C(u_1, u_2, \ldots, u_N) \tag{3.1}$$

Theorem 3.1.1 (**Sklar's theorem (1959)**)**.** *Consider an N-dimensional CDF F with marginals $F_1, \ldots, F_N$. There exists a copula $C : [0,1]^N \rightarrow [0,1]$, such that*

$$F(x_1, \ldots, x_N) = C(F(x_1), \ldots F(x_N)) \tag{3.2}$$

$\forall x_i \in [-\infty, \infty]$, $i = 1, \ldots, d$. *If F_i is continuous $\forall i = 1, \ldots, d$ then C is uniquely determined; otherwise C is uniquely determined only on $RanF_1 \times \ldots \times RanF_N$, where $RanF_i$ denotes the range of CDF F.*

According to Theorem 3.1.1, we can extract a unique representation of copula C explicitly, in terms of F and its continuous margins by calculating

$$C(u_1, u_2, \ldots, u_N) = F(F_1^{-1}(u_1), F_2^{-1}(u_2), \ldots, F_N^{-1}(u_N)) \tag{3.3}$$

where $F_1^{-1}, \ldots, F_N^{-1}$ are (generalised) inverses of $F_1, \ldots, F_N$.

Li [26] used the copula function conversely. The copula function links univariate marginals to their full multivariate distribution. For N given univariate marginal distribution functions $F_1(x_1), F_2(x_2), \ldots, F_N(x_N)$ with $x_i = F_i^{-1}(u_i)$ the joint distribution function C is defined as

$$C(F_1(x_1), F_2(x_2), \ldots, F_N(x_N), \Sigma) = Pr[U_1 \le F_1, U_2 \le F_2, \ldots, U_N \le F_N]$$
$$= F(x_1, x_2, \ldots, x_N) \tag{3.4}$$

where Σ is correlation matrix of $U_1, U_2, \ldots, U_N$ With given marginal functions, we can construct the joint distribution through copulas accordingly. Thus, with given individual distribution (e.g., creditworthiness over a 1-year horizon) of each credit asset within a portfolio, we can obtain the joint distribution and default correlation of this portfolio through copula function. In our methodology, we do not use copula function directly. We describe the concept of factor copula for further improvement to form the default correlation. If there are substantial number of instruments (for instance, N > 1,000) in our portfolio, to capture obligor-obligor interactions, we may be required to store a $N \times N$ correlation matrix, hence scalability can be a challenge. Factor copulas can be used to avoid constructing a high-dimensional correlation matrix. We can attempt to explain a large part of interactions in terms of a smaller set of market variables.

3.1.1. Linear Factor Copulas

Factor copula describes dependence structure between random variables not from the perspective of a certain copula form but from the factors model. Factor copula models have been broadly used for derivative pricing as well as computing banks' capital charge requirements. The main concept of factor copula model is that under a certain macro environment, credit default events are independent to each other. And the main causes that affect default events come from potential market economic conditions. Therefore a variation in the N-dimensional creditworthiness vector $\mathbf{X} = (x_1, \ldots, x_n, \ldots, x_N)$, can be explained in terms of the variation of a smaller set of market economic conditions or systematic risk factors, say $\mathbf{Z} = (Z_1, \ldots, Z_d)$. Formally we define a linear factor model as follows.

Definition 3.1.1 (**Linear Factor Copula model**). The random vector $\mathbf{X}$ is said to follow a $d-$factor model if it can be decomposed as
$$\mathbf{X} = \beta^T \mathbf{Z} + \mathbf{b} \circ \epsilon$$

where, $\circ$ represents the Hadamard product of vectors and

- $\mathbf{Z} \in \mathbb{R}^d$ with $d < N$ and a covariance matrix that is positive definite.

- $\epsilon \in \mathbb{R}^N$ is a random vector of idiosyncratic risk factors, which are uncorrelated.

- $\beta \in \mathbb{R}^{d \times N}$ & $\mathbf{b} \in \mathbb{R}^N$ are the constant *factor loadings* matrices.

3.1.2. Examples

Gaussian copula A $d-$factor Gaussian copula model assumes that the correlations among x_n are imposed by linking them to $d \times 1$ $\mathbf{Z}$ vector of independent standard normal variables, for $n = 1, \ldots, N$.

$$x_n = \beta_n^T \mathbf{Z} + b_n \epsilon_n, \tag{3.5}$$

where $\mathbf{Z} = (Z_1, \ldots, Z_d)^T$ with $Z_n \sim N(0,1)$, $\beta_n = (\beta_{n,1}, \ldots, \beta_{n,d})^T$, $b_n = \sqrt{1 - \sum_{i=1}^d \beta_{n,i}^2}$ and $\epsilon_n \sim N(0,1)$. Using the result of the above Gaussian assumptions in the context of modeling credit portfolio losses as described in Section 2.3, the default threshold of x_n is given by

$$\xi_n = N^{-1}(p_n)$$

T Copula As an alternative to the Gaussian copula, the correlations of defaults can be modelled by a Student-t copula. Banking regulations published by the Basel group [4] suggested two ways of modelling for the correlations via a Student-t copula for credit risk, either using a Student-t copula for all issuer risk factors, or using a Student-t copula for the systematic risk factors. Here we discuss the second case. Particularly, we model the systematic risk factors as a multivariate-t random vector and the idiosyncratic factors as Gaussian random

(a) bivariate Gaussian copula with correlation=0.5 (b) bivariate T-copula with correlation=0.5 and $\gamma = 1$
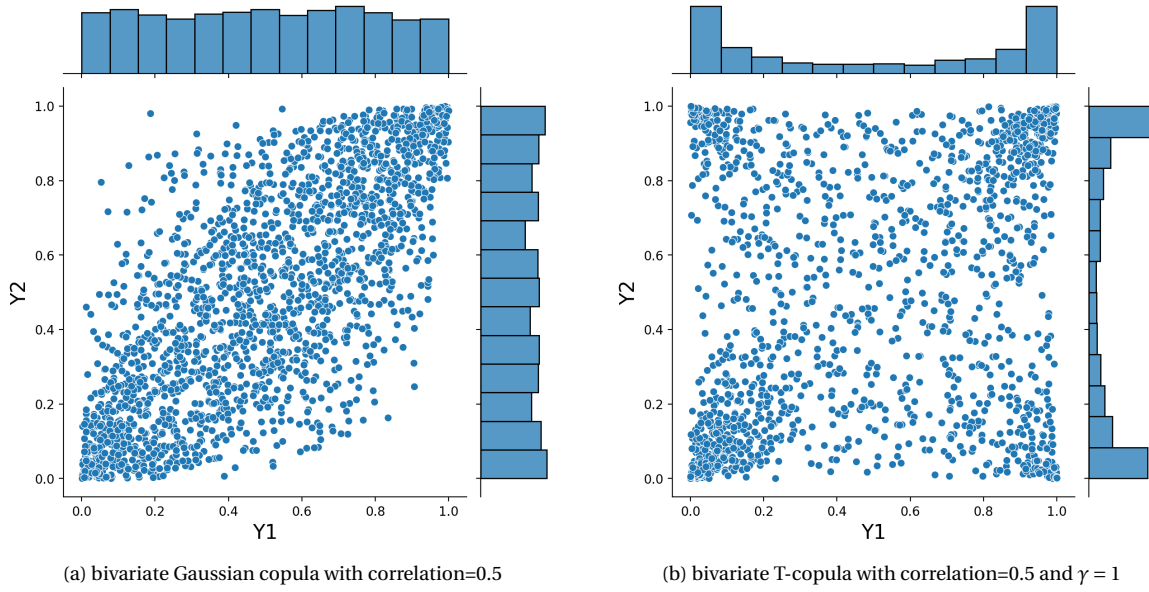
Figure 3.1: Visualization of bivariate copula on a unit cube $[0,1]^2$ with corresponding marginal distributions. Note the differences of the distributions in the tail.

variables. This model choice gives another layer of difficulty since the default thresholds of the obligors do not have analytical solutions. The standard Student-t copula $C_{\gamma,\Sigma}^t$ given by

$$C_{\gamma,\Sigma}^t(u_1,\ldots,u_d) = \int_{-\infty}^{t_\gamma^{-1}(u_1)} \cdots \int_{-\infty}^{t_\gamma^{-1}(u_d)} \frac{\Gamma\left(\frac{\gamma+d}{2}\right)}{\Gamma\left(\gamma/2\right)\sqrt{\left(\pi\gamma\right)^d |\Sigma|}} \left(1 + \frac{\mathbf{X}^T\Sigma^{-1}\mathbf{X}}{\gamma}\right)^{-\frac{\gamma+d}{2}} d\mathbf{X}$$

If a random vector $\mathbf{X}$ has the t copula $C_{\gamma,\Sigma}^t$ as the component-wise probability transformed random vector and the student-t distribution as the component-wise marginal distribution with the same degree of freedom γ, then it follows a multivariate student-t distribution with γ degrees of freedom. Specifically, the d-dimensional random vector $\mathbf{X}$ is said to have a multivariate t distribution with γ degrees of freedom, the mean vector μ and a positive-definite dispersion matrix Σ, denoted by $\mathbf{X} \sim t_d(\gamma,\mu,\Sigma)$. Another convenient representation for simulation is

$$\mathbf{X} = \sqrt{W}\mathbf{Z} \tag{3.6}$$

where $\mathbf{Z} \sim N_d(0,\Sigma)$, and W is independent of Z and has an inverse gamma distribution i.e,

$$W \sim I_g(\gamma/2,\gamma/2) \tag{3.7}$$

Hence incorporating the dependence of the (3.6) to modify the systematic factors of x_n we find,

$$x_n = \sqrt{W}\beta_n^T\mathbf{Z} + b_n\epsilon_n \tag{3.8}$$

The marginal distribution of x_n is a convolution of a Student t distribution and a Gaussian distribution, while the dependence among x_n solely depends on the common factors $\mathbf{W}$, $\mathbf{Z}$ and the idiosyncratic factors ϵ_n remain independent.

3.1.3. The Vasicek model
The first copula model for portfolio credit risk was given by Li [26]; his model is based on the Gaussian copula. The one-factor Gaussian copula model offers analytic tractability by the assumption that the underlying portfolio of assets is large and homogeneous, hence a special case of the examples described above. This approach by [40] is also referred to as the Large Homogeneous Pool (LHP) Model. Although it relies on the Gaussian distribution and is often criticized for being simplistic, there have been extensions to this model.[cite] For now, we follow the standard assumptions. Without loss of generality, we set loss value at default ($\epsilon_n v_n$)

(Ref. eqn 2.3) constant for all obligors. The Vasicek model evaluates the default of an obligor in terms of the evolution of its asset value over a fixed time horizon $T > 0$, based on the firm value model of Merton (Ref. section 2.3). With β constant for all obligors, Equation (3.5) for a single factor Gaussian copula for the n-th obligor reduces to,

$$x_n = \beta Z + \sqrt{1 - \beta^2}\epsilon_n, \tag{3.9}$$

where Z and ϵ_n are i.i.d standard normal random variables. Other assumptions of the Vasicek model are that all PD_n are equal and ϵ_n equals 1. The probability of default conditional on the common factor $Z = z$ can be given by,

$$p_n(z) = \mathbb{P}\left[\epsilon_n < \frac{\xi_n - \beta Z}{\sqrt{1 - \beta^2}} | Z = z\right] = N\left(\frac{N^{-1}(PD_n) - z\beta}{\sqrt{1 - \beta^2}}\right) \tag{3.10}$$

The resulting loss distribution is $L = \sum_{n=1}^{N} \mathbb{1}_{x_n \leq \xi_n} \sim Bin(N, p_n)$. As a consequence of the Central Limit theorem,

$$\lim_{n \to \infty} L/n = p(Z) = N\left(\frac{N^{-1}(PD_n) - Z\beta}{\sqrt{1 - \beta^2}}\right) \quad a.s.$$

The analytical asymptotic Vasicek formula is helpful for calibrating our results which we will see in Chapter 4.

The methodology for measuring credit risks in debt instruments is in a state of rapid development. Here, we have only presented only a few of them. We refer to Laurent and Gregory [23] and Burtschell, Gregory and Laurent [7], that provide comparative analyses of various copula-based factor models. Typically factor copulas are simulated using the Monte Carlo method. An alternative approach are the whole branch of semi-analytic methods, which are based on on a combination of numerical integration and analytic methods. These models are significantly faster than Monte Carlo calculations. Hull and White introduced a bucketing approach[], Andersen and Sidenius [2] and Jackson [19] proposed a recursive method valid in more general cases. Saddle-point approximations were analyzed by Huang and Oosterlee [18]. Fourier transform methods were analyzed by Laurent and Gregory [23] and Grundke [16]. These methods suggest using Fourier transforms of densities , that is characteristic functions of the random variables, to compute the distribution of the sum of the independent assets. This alternative class of methods based on Fourier transforms, leads us to our main approach in this thesis for our models, the COS method which we describe in Section 3.4. A recent numerical approach proposed by Colldeforns-Papiol et al [8] is based on the Haar wavelets basis instead of a Fourier basis to compute the inversion of the characteristic function.

3.2. Monte Carlo methods

For computing a general cumulative loss distribution function, denoted by F_L and its corresponding risk measures, the industry standard is using Monte Carlo methods. First we need to draw independent and identically distributed replicates of the random variable L, and then given a set of n i.i.d samples $L^{(1)}, \ldots, L^{(n)}$, we require a method to estimate the risk measures.

The *Crude Monte Carlo* estimator of VaR comes from the quantile estimator explained in traditional statistics. From our definition of VaR (Ref. eqn 2.5) the analytical distribution is replaced with the equivalent empirical distribution function $\hat{F}_L$,

$$VaR_\alpha = \inf\{x : \hat{F}_L(x) \geq \alpha\}$$

where, $\hat{F}_L(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{(L^{(i)} \leq x)}$ is the empirical distribution function of the iid sorted samples $L^{(i)}$ such that $L^{(1)} \leq, \cdots \leq L^{(n)}$. For Expected Shortfall (eqn. 2.7), the estimator is just the an expectation.

$$ES = \frac{1}{N(1 - \alpha)}\sum_{i=1}^{N} L^{(i)} \mathbb{1}_{L^{(i)} \geq VaR_\alpha}$$

By the virtue of central limit theorem, the VaR and ES estimators have asymptotically normal distributions. An efficient variant of the Monte Carlo based methods is the Importance Sampling (IS) approach. Since the canonical Monte Carlo estimators generally require a very large sample size for desired lower tolerances, IS approach in practice can give better performance with lower sample sizes. However, we do not discuss this approach in our study. Refer to [See 27, Chapter 8] for a detailed treatment.

Algorithm 1: Monte Carlo method for multifactor Gaussian copula

---

Data: number of simulations n, factor loadings $\beta \in \mathbb{R}^{d \times N}$, obligor specific properties **PD** & **l** $\in \mathbb{R}^N$
Result: Portfolio Loss vector **L** $\in \mathbb{R}^n$
for $i=1,..,n$ **do**
    $L^{(i)} = 0$;
    for $D=1,...,d$ **do**
        Generate $z_D \sim \mathcal{N}(0,1)$;
    end
    for $j=1,...,N$ **do**
        Generate $\epsilon \sim \mathcal{N}(0,1)$;
        for $D=1,...,d$ **do**
            $x_j = \sum_{D=1}^{d} \beta_{D,j} z_D + \sqrt{1 - \sum_{D=1}^{d} \beta_{D,j}^2}\, \epsilon$;
        end
        if $x_j < N^{-1}(PD_j)$ **then**
            $L^{(i)} = L^{(i)} + l_j$;
        end
    end
end

3.3. Fourier Transform Techniques

The essence of the Fourier transform is that it represents a function as a continuous superposition of periodic functions. Often we will obtain an analytic expression for the Fourier transform but not be able to analytically find the original function, and instead numerical schemes are used to invert the transformation.However , the amount of computational effort is small compared to that required to compute option prices via other means such as Monte Carlo simulation. A suggested approach is using Fourier transforms of densities ,that is characteristic functions of the random variables, to compute the distribution. This method relies on the fact that if x_n are independent random variables then

$$\mathbb{E}\left(e^{i\xi \sum_n x_n}\right) = \prod_n \mathbb{E}\left(e^{i\xi x_n}\right) \tag{3.11}$$

This corresponds to the fact that the Fourier transform of a convolution is the product of the Fourier transforms.We therefore compute the characteristic function of the random variable expressing the contribution of each asset to the loss distribution , take their product, and take the inverse Fourier transform to compute the loss distribution. Given the individual characteristic functions , it is simple to take the product. The loss distribution is then calculable via an inverse Fourier transform. A detailed analysis is given in the paper by Laurent and Gregory. [23]

3.4. The COS method

The previous section briefly outlined the general Fourier transform approach. Our work concentrates on the related idea of the *COS method* [12]. The algorithm is based on the usage of a cosine series on a truncated finite interval. In contrast to the complete Fourier transform, the cosine series converges rapidly for smooth functions on a bounded interval. Thus one needs to compute only a small number of terms. We will also see that the cosine coefficients of the density can be easily computed from its characteristic function , and so whenever there is a closed-form , they are simple to find.

3.4.1. Characteristic Function of Portfolio Loss

We use ϕ_L to denote the characteristic function of the portfolio loss L described in Section 2.3. It then follows that

$$\phi_L(\omega) = \mathbb{E}\left[\mathbb{E}\left[e^{i\omega \sum_{n=1}^{N} l_n \cdot \mathbb{1}_{x_n \le \xi_n}} \,|\, \mathbf{Z} = \mathbf{z}\right]\right]$$

$$= \mathbb{E}\left[\prod_{n=1}^{N} \mathbb{E}\left[e^{i\omega l_n \cdot \mathbb{1}_{\epsilon_n \le \alpha_n(z_n)}} \,|\, \mathbf{Z} = \mathbf{z}\right]\right] \tag{3.12}$$

and, $\alpha_n(z) = \frac{\xi_n - \beta_n^T \mathbf{z}}{b_n}$

$$\phi_L(\omega) = \mathbb{E}\left[\prod_{n=1}^{N}\left[1 + P(\epsilon_n \leq \alpha_n(\mathbf{z}))(e^{i\omega l_n} - 1)\right]\right]$$

$$= \int_{\Omega^d} f_Z(z) \prod_{n=1}^{N}\left[1 + P(\epsilon_n \leq \alpha_n(z_n))(e^{i\omega l_n} - 1)\right]dz.$$

$$= \int_{\Omega^d} \Phi(\mathbf{z};\omega)d\mathbf{z}. \tag{3.13}$$

$f_{\mathbf{Z}}(\mathbf{z})$ denotes the joint probability density function of the systematic factors and $\Omega^d \in \mathbb{R}^d$ denotes the d-dimensional hyperrectangular domain over which the integration is performed.

3.4.2. Recover the CDF
The main idea of the COS method is that the density of a continuous random variable can be re-constructed from a Fourier cosine series. If the portfolio loss is a continuous function over a real domain, we could fix a range $[l_a, l_b]$ that is sufficiently broad to cover a large enough probability level, and consider the finite K terms Fourier cosine expansion of the density f_L of the portfolio loss L within the range as

$$f_L \approx \tilde{f}_K(x) \equiv \frac{A_0}{2} + \sum_{k=1}^{K} A_k \cos\left(k\pi \frac{x - l_a}{l_b - l_a}\right) \tag{3.14}$$

Whilst we have the Fourier cosine series, the Fourier cosine coefficients can be directly extracted from the characteristic function.

$$A_k = \frac{2}{l_b - l_a}\Re\left\{\phi_L(\frac{k\pi}{l_b - l_a})\cdot e^{-i\frac{k\pi l_a}{l_b - l_a}}\right\} \tag{3.15}$$

For computing the CDF F_L, it follows from (3.14) that yields,

$$F_L(x) \approx \tilde{F}_K(x) \equiv \frac{A_0}{2}(x - l_a) + \sum_{k=1}^{K} A_k \frac{l_b - l_a}{k\pi}\sin\left(k\pi \frac{x - l_a}{l_b - l_a}\right) \tag{3.16}$$

Here in the integration of the density function we again truncate the integration range and the last equation is obtained by interchanging the order of summation and integration. It can be seen that the CDF is expressed in the form of sine series.

Algorithm 2: COS Method for multifactor Gaussian copula

Data: number of COS terms K, factor loadings $\beta \in \mathbb{R}^{d \times N}$, obligor specific properties **PD** & $\mathbf{l} \in \mathbb{R}^N$
Result: CDF of Loss vector $\mathbf{L} \in \mathbb{R}^n$ at any point x
Set $\Delta l = \sum(\mathbf{l}) - \min(\mathbf{l})$;
for *k=0,..,K* **do**
    Initialize $\omega = \frac{k\pi}{\Delta l}$;
    Set $\Omega^d \in [-\text{erf}^{-1}(10^{-7}), \text{erf}^{-1}(10^{-7})]^d$;
    Compute $\phi_L(\omega) = \int_{\mathbb{R}^d} \Phi(\mathbf{z};\omega)d\mathbf{z}$;
    Compute $A_k = \frac{2}{\Delta l}\Re\left\{\phi_L(\frac{k\pi}{\Delta l})\cdot e^{-i\omega\min(\mathbf{l})}\right\}$;
end
Evaluate $\tilde{F}_K(x) = \frac{A_0}{2}(x - \min(\mathbf{l}) + \sum_k A_k \frac{\Delta l}{k\pi}\sin(k\pi\frac{x - \min(\mathbf{l})}{\Delta l})$

3.4.3. Gibbs phenomenon
Since $f_L(x)$ in our case will always be a probability mass function, the corresponding Fourier cosine density expansion $\tilde{f}_K(x)$ will not converge to it in cases of highly non-smooth functions for example, step functions. The rate of convergence drops to first order and spurious oscillations develop near the discontinuities known as *Gibbs phenomenon.* The spurious effects do not die out as the number of the Fourier terms increases and also causes a slow convergence rate of the Fourier series at other continuous locations. Several techniques

have been developed to mitigate or remove the Gibbs phenomenen, which includes applying spectral filters, change of basis for example using wavelets series expansion and edge detection algorithms [13]. In our study, we follow the approach of filtering. The main idea of applying spectral filters is to let the obtained series coefficients decay faster. Modifying eqn. (3.14), we obtain the COS density function after applying a spectral filter as,

$$\tilde{f}_K^\sigma(x) = \frac{A_0}{2} + \sum_{k=1}^{K} A_k \sigma(k/K) \cos\left(k\pi \frac{x - l_a}{l_b - l_a}\right) \tag{3.17}$$

where the spectral filter $\sigma(\eta) \in \mathscr{C}^{q-1}[-\infty,\infty]$ with the following properties,

- $\sigma(\eta) = 0$ for $|\eta| > 1$

- $\sigma(0) = 1$ and $\sigma(1) = 0$

- $\sigma^m(0) = \sigma^m(1) = 0 \; \forall\, m \in [1,\ldots,q-1]$

Analogously we get the COS CDF approximation using the now modified filtered density distribution eqn. (3.17),

$$\tilde{F}_K^\sigma(x) = \frac{A_0}{2}(x - l_a) + \sum_{k=1}^{K} A_k \sigma(k/K) \frac{l_b - l_a}{k\pi} \sin\left(k\pi \frac{x - l_a}{l_b - l_a}\right) \tag{3.18}$$

Error convergence of Fourier series expansion with spectral filters has been studied in literature, such as [39]. The analysis in shows that, to restore convergence in the Fourier series after applying a spectral filter, the function itself has to be at least piecewise continuous. However, it is already proven in [34] that, though f_L is not piecewise continuous, the recovered CDF function in the eqn. (3.18) still converges to the true CDF. The proof is based on the observation that the recovered expression is nothing but a finite linear combination of the Fourier series expansions of some piecewise constant functions. Let us see this now with the help of two theorems in the next section.

3.4.4. Error Analysis

We now explain briefly the error convergence behaviour of the COS CDF approximation as derived in Section 3.4.2. We introduce three main sources of error in the approximation formula in eqns. (3.14) and (3.15). Observe that the integration range in eqn. (3.13) is covers the d-dimensional hyperrectangle spanning $\mathbb{R}^d$. Computationally, we need to truncate this range within a reasonable tolerance. The second approximation inaccuracy stems from the numerical integration scheme itself used to resolve eqn. (3.13). We refer to both these errors as the numerical integration error, which has been studied extensively in the literature. Our main focus will be the error behaviour COS approximation, i.e. the error from approximating the density function of the portfolio loss distibution by the filtered cosine-series expansion.

As mentioned earlier, it was demonstrated in [39] that, for piecewise smooth functions, applying spectral filters can restore the convergence of the Fourier series and the speed of convergence can even be exponential for functions with known discontinuities. This rules out the case we studied here, whereby we attempt to apply the Fourier series expansion to the probability mass function of a discrete random variable. Here we prove that the recovered CDF does converge to the true CDF of the discrete random variable L, if spectral filters are applied. The convergence speed is determined by the order of the filter. Moreover, we demonstrate that the numerical integration error is also bounded. We first re-phrase one of the findings in [39] into the following theorem, which states that the convergence speed of the filtered Fourier series of a piecewise continuous function is determined by the order of the spectral filter:

Theorem 3.4.1. *Let $f(y)$ be a piecewise $C^p([-\pi,\pi])$ function with one point of discontinuity ξ. Let $\sigma(k/K)$ be a filter of order p. Then if $y \neq \xi$, it holds that*

$$|f_K^\sigma(y) - f(y)| \sim \mathcal{O}(K^{1-p}). \tag{3.19}$$

Using Theorem 3.4.1, by applying the spectral filter to the COS CDF of the portfolio loss in the way we proceed in the previous sections, the approximation error is bounded as in the lemma below:

Lemma 3.4.1.1. *The absolute difference between $\tilde{F}_L(x)$ as given in equation (3.16) and $\tilde{F}_K^\sigma(x)$ as defined in equation (3.18) converges as follows:*

$$|F_K^\sigma(x) - \tilde{F}_L(x)| \sim \mathcal{O}(K^{1-p}) \tag{3.20}$$

where K is the number of Fourier cosine terms and p is the order of the spectral filter.

Proof. Notice that the loss L can be transformed to a random variable L' bounded in $[0, \pi]$, by a bijective linear mapping $L' = \pi(L - l_a)/(l_b - l_a)$. Therefore, it is sufficient to analyze the case $0 \le L \le \pi$. Particularly, we denote the possible realizations of L by a finite set, i.e, $0 = \mathscr{L}_0 < \mathscr{L}_1 \ldots \mathscr{L}_m \cdots < \mathscr{L}_M = \pi$. Applying the COS expansion to the density of L yields,

$$f_K^\sigma(x) = \Sigma_{k=0}^{'K} \sigma(k/K) A_k \cos(kx)$$

where,

$$A_k = \frac{2}{\pi} \Re \phi(k) = \frac{2}{\pi} \Re \left\{ \Sigma_{m=0}^M e^{ik\mathscr{L}_m} p_m \right\} = \frac{2}{\pi} \Sigma_{m=0}^M \cos(k\mathscr{L}_m) p_m$$

p_m is the probability of $L = \mathscr{L}_m$ Substituting and interchanging the order of the summations, we have

$$f_K^\sigma(x) = \Sigma_{m=0}^M p_m g_m^\sigma(x)$$

with,

$$g_m^\sigma(x) = \frac{2}{\pi} \Sigma_{k=0}^{'K} \Sigma(k/K) \cos(k\mathscr{L}_{\hat{\jmath}}) \cos(kx)$$

Integrating $f_K^\Sigma(x)$ from 0 gives the COS-CDF approximation of L,

$$F_K^\Sigma(x) = \Sigma_{m=0}^M p_m g_m^\sigma(x)$$

with

$$G_m^\sigma(x) = \frac{1}{\pi} x + \Sigma_{k=1}^K \frac{2}{k\pi} \sigma(k/K) \cos(k\mathscr{L}_{\hat{\jmath}}) sin(kx)$$

On $[-\pi, \pi]$, $G_m(x)$ is the Fourier series expansion of the function

$$H_m(x) = \begin{cases} 1 & \text{if} \quad \mathscr{L}_m \le x \le \pi \\ 0 & \text{if} \quad -\mathscr{L}_m < x < \mathscr{L}_m \\ -1 & \text{if} \quad -\pi \le x < \mathscr{L}_m \end{cases}$$

for $m = 0, 1, \ldots, M$. Note that for $\mathscr{L}_0 = 0$, $H_0(x)$ reduces into the Heaviside step function

$$H_0(x) = \begin{cases} 1 & \text{if} \quad 0 \le x \le \pi \\ -1 & \text{if} \quad -\pi \le x < 0 \end{cases}$$

and that for $\mathscr{L}_M = \pi$, $H_M(x)$ turns into,

$$H_M(x) = \begin{cases} 1 & \text{if} \quad x = \pi \\ 0 & \text{if} \quad -\pi < x < \pi \\ -1 & \text{if} \quad x = -\pi \end{cases}$$

H_m corresponds to the CDF of a loss distribution with the loss centered on L_M. Since they are piecewise continuous, we apply Theorem 3.4.1,

$$|G_m^\sigma(y) - H_m(y)| \sim \mathcal{O}(K^{1-p})$$

Given that the CDF of L is a finite linear combination of H_m, each of them weighted by p_m, it follows that $F_K^\sigma(x)$ converges to $\tilde{F}_L(x)$ at the speed as described above. $\qquad\square$

Recall that there is one extra layer of approximation: the cosine coefficients A_k is obtained via numerical integration scheme, after we truncate the integration range of the factors with a truncation error at the level of TOL. Here we base our analysis on the Clenshaw-Curtis quadrature. Thus, the discretized version of the characteristic function obtained in (3.13) is given by,

$$\phi_L(\omega) \approx \tilde{\phi}_Q(\omega) = \Sigma_{q_1=0}^Q h_{q_1} \ldots \Sigma_{q_d=0}^Q h_{j_d} \Phi_{\mathbf{z}}(z_{q1}, \ldots, z_{q_d}; \omega) \tag{3.21}$$

The above integral can be computed by using any standard numerical integration scheme. Clenshaw-Curtis quadrature rule is preferred because of its exponential convergence on a smooth integrand as well as fast

computational performance - the discretized integral can be solved via *Fast Fourier Transform* (FFT) algorithm, with $O(J\log(J))$ complexity where J is the total number of quadrature points. As mentioned earlier, here we focus on how this error is propagated during the course of COS approximation. Let us denote the error term arising from numerical integration as $\mathcal{O}(Q, TOL)$ - as the notation indicates, it depends on the number of integration points Q and the tolerance of truncating integration range TOL. This error term is propagated via the Fourier series as follows:

Lemma 3.4.1.2.

$$\tilde{F}_K^{\Sigma}(x) = F_K^{\sigma}(x) + \mathcal{O}(\sqrt{K}) \cdot \epsilon(J, TOL)$$

, where J is the number of points adopted in the Clenshaw-Curtis cubature, TOL is the integration truncation error.

We omit the proof here for brevity, and refer to [34] for a detailed discussion.

3.4.5. Computation of Risk Measures

In Section 2.4, we have already seen the definitions of various credit risk measures. Here, we see how they are calculated under the COS method. The computation of VaR and ES are trivial. Since, once we have recovered the CDF, i.e $\tilde{F}_K(x)$, the q-th quantile can be solved numerically, $\tilde{F}_K(x) = q$. ES can be evaluated by for e.g. a numerical integration scheme.

As we have seen in Section 2.4.1, there exist standard decompositions of ES and VaR of the portfolio loss that follow the properties of Euler risk allocation. From equations (2.6) and (2.8) the decomposition of VaR and ES essentially is a problem of solving a conditional loss distribution. This is again solvable by applying the COS to conditional characteristic function. It follows from eqn (2.6),

$$\begin{aligned}
CES_n &= l_n P(x_n \le \xi_n | L \ge VaR_\alpha) \\
&= l_n \frac{P(x_n \le \xi_n, L \ge VaR_\alpha)}{P(L \ge VaR_\alpha)} \\
&= \frac{l_n PD_n}{\alpha} P(L \ge VaR_\alpha | x_n \le \xi_n)
\end{aligned} \tag{3.22}$$

To solve the conditional probability $P(L \ge VaR_\alpha | x_n \le \xi_n)$, we can start from its characteristic function $\phi_{n,L}$,

$$\begin{aligned}
\phi_{n,L}(\omega) &= \mathbb{E}\left[e^{i\omega L} | x_n \le \xi_n \right] \\
&= \frac{1}{PD_n} \mathbb{E}\left[\left(\prod_{j \ne n} \mathbb{E}\left[e^{i\omega l_n \cdot \mathbb{1}_{n \le \alpha_j(z_j)}} | Z = z \right] \right) \cdot \mathbb{E}\left[e^{i\omega l_n \cdot \mathbb{1}_{\epsilon_n \le \alpha_n(z_n)}} | Z = z \right] \right]
\end{aligned} \tag{3.23}$$

where $\mathbb{E}\left[e^{i\omega l_n \cdot \mathbb{1}_{n \le \alpha_j(z_j)}} | Z = z \right] = 1 + P(\alpha_j(z_j))(e^{i\omega l_j} - 1)$ and $\mathbb{E}\left[e^{i\omega l_n \cdot \mathbb{1}_{\epsilon_n \le \alpha_n(z_n)}} | Z = z \right] = P(\alpha_n(z_n)) \cdot e^{i\omega l_n}$

From eqn (2.8) it follows that,

$$\begin{aligned}
CVaR_\alpha &= l_n \cdot P(x_n \le \xi_n | L = VaR_\alpha) \\
&\approx l_n \cdot PD_n \cdot \frac{P(VaR_\alpha - \varepsilon \le L \le VaR_\alpha + \varepsilon | x_n \le \xi_n)}{P(VaR_\alpha - \varepsilon \le L \le VaR_\alpha + \varepsilon)}
\end{aligned} \tag{3.24}$$

Above we replace the $L = VaR_\alpha$ by approximating within ε tolerance of VaR_α. COS method can be used to evaluate the probabilities in eqn. (3.24).

3.5. Test examples

We use the methodology described in the previous sections, to determine the loss distribution and the corresponding risk measures. More specifically, we see the performance (i.e accuracy and computational speed) of the COS method on the CPU and benchmark it against conventional approaches like the Vasicek one-factor model and Monte Carlo simulations. The reason is two fold. First, our intention is to gain fidelity on our implementation of the COS method. Furthermore, it allows us to compare the performance of the present serial algorithm of the COS against its GPU-parallel versions in the later sections.
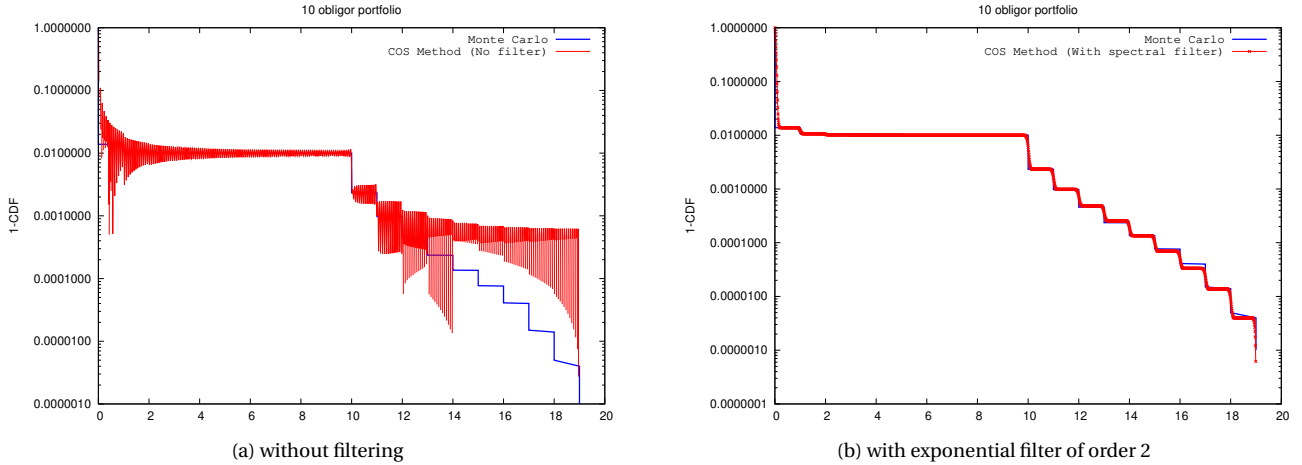
Figure 3.2: Recovered CDF from Monte Carlo and COS methods for a test portfolio with 10 obligors with 2 Factor Gaussian copula.

3.5.1. Artificial portfolio

We first test the COS method with a small portfolio with 10 obligors with sample characteristics. This is achieved by setting probability of default and loss-at-default of the first obligor to be 10 times as large as of the other obligors. We use the two-factor Gaussian copula correlation between all obligors. Specifically, we set the default probabilities of the obligors to be $PD_1 = 0.01$, $PD_n = 0.001$, $\forall n = 2, \ldots, 10$. The loss at default of obligors was taken to be $l_1 = 10$, $l_n = 1$, $n = 2, \ldots, 10$, respectively. Without loss of generality the factor loadings are taken to be $\beta_{n,1} = 0.2$ and $\beta_{n,2} = 0.8$ and are set constant for all obligors. The piecewise constant CDF of the portfolio loss with name concentration is more likely to have a wide step at the location of VaR, particularly if the concentrated obligor defaults with a large probability at the event of VaR. In this case the loss CDF is a step function of few steps, and therefore we can notice the Gibbs phenomenon at the steps very clearly. Thus, the objective of this test case is to show that the filtered Fourier cosine series can recover the piecewise constant CDF. Figure 3.2 plots the recovered CDF of the portfolio loss from Monte carlo method with 100000 simulations compared with the result obtained from COS in both cases. As the plot indicates, the filtered COS approximation is accurate. in Figure 3.2.

3.5.2. Verification with the One Factor Vasicek model

In 3.1.3 we described the model of Vasicek [40]. Here, we verify our recovered CDF from COS with the Large Homogeneous Pool (LHP) approximation from [40]. As described in earlier sections, the LHP approximation uses a one-factor Gaussian copula to represent the default correlation structure. The portfolio contains an infinite number of entities, which all have the same characteristics (e.g. PD_n, l_n, β_n). So, we check whether our piecewise constant CDF converges $a.s$ in the limiting case, to the computed Vasicek CDF with the same input data. Figure 3.3 shows the convergence.

3.5.3. S&P rated portfolio

We further give a numerical example with a large portfolio with 1000 obligors to mimic real-world situations. We first sample ratings of the obligors uniformly from the levels AAA, AA, A, BBB, BB, B and CCC, and the assign to each obligor the S&P[1] default probability by rating. Then the loss-at-default per obligor is uniformly sampled from [10,1000]. Finally, a few name concentrations are created by means of multiplying the losses of some CCC obligors by a factor of 50 or 10. The reference values for calculating the relative errors are generated using the COS method, where we set the number of COS terms to be 800, the tolerance level of the integration range truncation error equal to be $1e - 9$, the number of integration points to be 200. The benchmark MC results are obtained with 1 million simulations.

As witnessed in the previous numerical example, the Figure 3.5 (c), confirms a good match of the loss distributions from the MC method and the COS method. As demonstrated in the Figure 3.5 (d), we see the errors of

---

[1] Standard and Poor's credit rating system url: https://www.spglobal.com/ratings/en/research/articles/200429-default-transition-and-recovery-2019-annual-global-corporate-default-and-rating-transition-study-11444862, Accessed: February 2, 2021
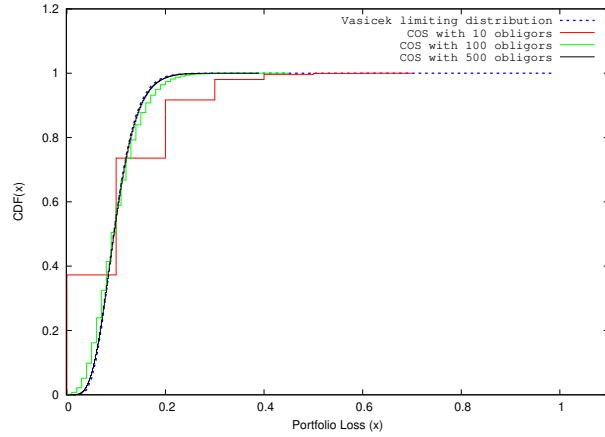
Figure 3.3: Comparison of Loss CDF for $PD = 0.1$, $\beta = 0.2$
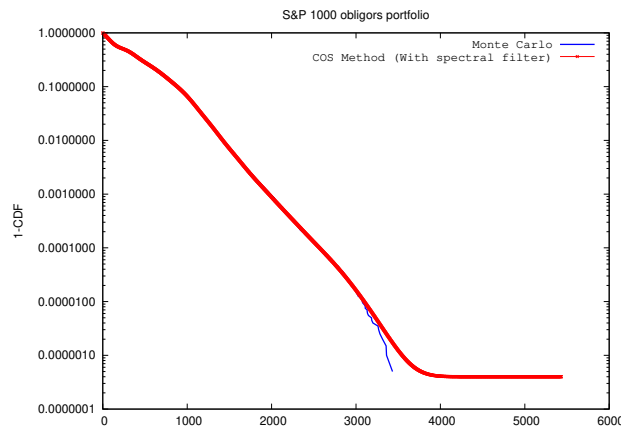
Figure 3.4: recovered CDF vs portfolio loss

the COS method converge in a regular pattern for computed risk metric VaR with the increase in COS terms. The errors are computed with respect to the benchmark MC results.

3.5.4. Remarks

The error convergence plots of the recovered CDF and computed VaR (in Figures 3.5 a)- d)) show a unifying trend of rapid convergence. The behavior is dependent on the number of COS terms (K) and the number of quadrature points Q, as described in Lemma 3.4.1.2 respectively. Thus, the COS method showcases an exponential rate of convergence, and a large number of COS terms for a fixed numerical integration error, is therefore not necessary. Tables 3.1 and 3.2 show the effect of the spectral filter on the convergence rate. Two types of spectral filters are tested: the exponential filters (with orders p = 2 and 4), with the number of the cosine terms from 40 to 110. We define the observed convergence rate as the ratio of the logarithm of

COS Terms (K)	CDF L_2 Error	Conv. Rate	Rel. VaR Error	Conv Rate	Rel. ES Error	Conv Rate
40	1.158679	1.18	8.28729282e-02	2.04	7.01107709e-02	2.03
50	0.890122	1.17	5.24861879e-02	2.08	4.45518110e-02	2.05
60	0.717936	1.19	3.59116022e-02	2.38	3.06258678e-02	2.36
70	0.5967301	1.23	2.48618785e-02	1.88	2.12674604e-02	1.83
80	0.5060751	1.28	1.93370166e-02	2.82	1.66508867e-02	2.83
90	0.4352105	1.33	1.38121547e-02	2.11	1.19180550e-02	2.07
100	0.3783001	-	1.10497238e-02	-	9.58074650e-03	

Table 3.1: Observed convergence rates for second order (p=2) exponential filter (S & P 1000 obligors)

(a) L_2 norm of error of recovered CDF (10 obligors)

(b) L_1 norm of error of risk measure VaR (10 obligors)

(c) L_2 norm of error of recovered CDF (1000 obligors)

(d) L_1 norm of error of risk measure VaR (1000 obligors)
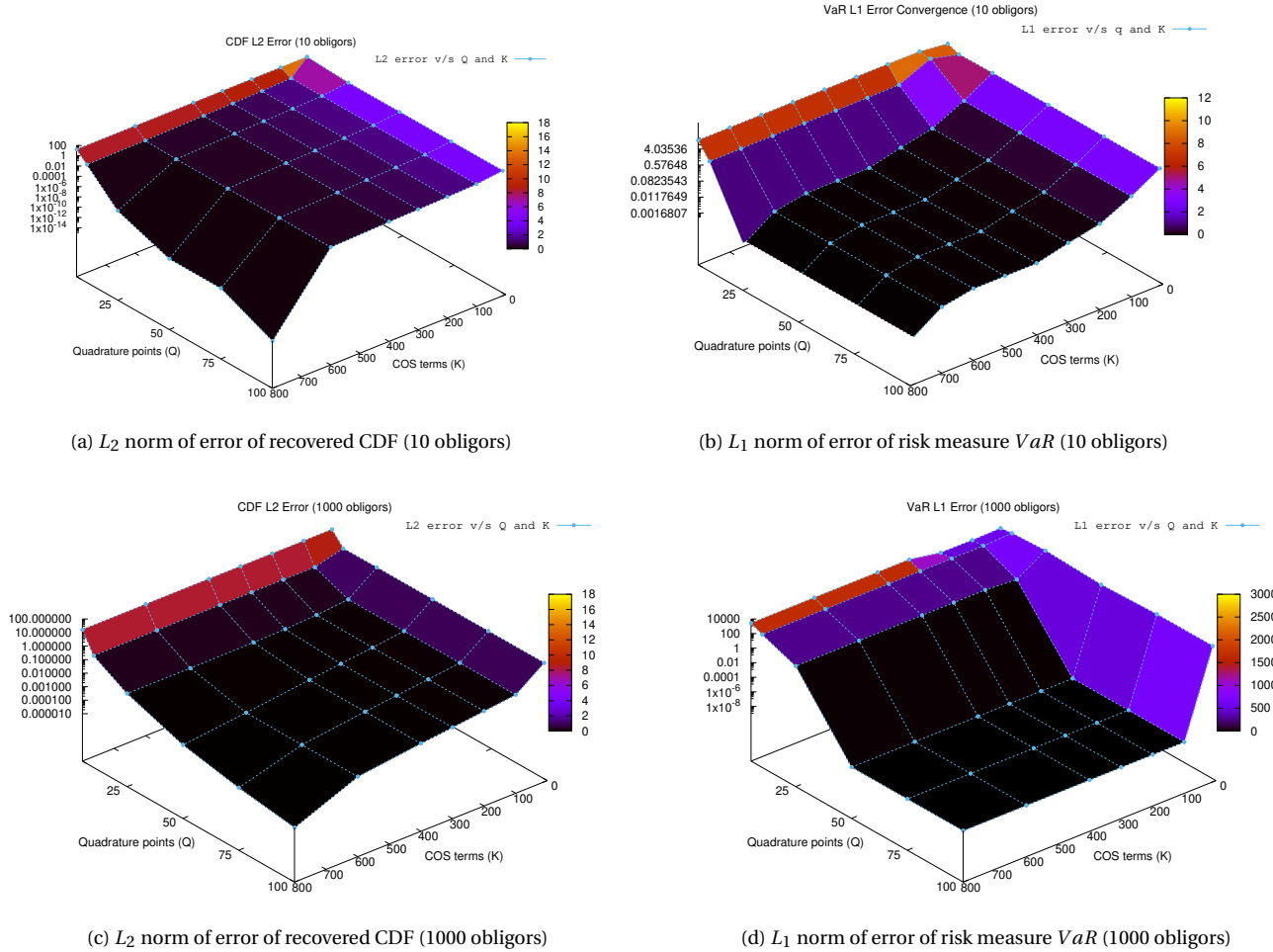
Figure 3.5: Error convergence behaviour of the 2-factor Gaussian copula model on an example with 10 and 1000 obligors. The dependence follows a similar trend dependent on the bounds described in Lemma 3.4.1.2

the decrease in the relative error to the logarithm of the increase in the number of the cosine terms. In the column "Conv.Rate", we calculate the convergence rate of the adjacent characteristic. The calculated rates of the CDF, as observed, are close to the theoretical lower bound prescribed in Lemma 3.4.1.1. The VaR and ES convergence rates for $p = 2$ show a convergence rate much faster than the theoretical rate, close to the order of the filter. Similar observations have been reported in [34]. In the case of $p = 4$, the computed values show a much accelerated rate of convergence for the VaR and the ES, in contrast to the CDF. The convergence of ES is less regular than VaR, as suggested by the observed convergence rate. It can be due to that we compute ES as an integration of VaR numerically, whereby extra numerical errors are introduced. Overall, this shows that increasing the order of the filter, we can obtain better convergence properties of the computed values. Other computed risk measures are listed in Appendix .

3.6. Summary

This chapter has described various approaches available in the literature to model portfolio loss distributions. We defined linear factor copulas that are fundamental tools in modeling the correlations between various external factors, while computing the overall creditworthiness of the obligor. We progressed from the analytical Vasicek model, that is used for simulating loss distribution of homogenous portfolios to giving a brief account of the Monte Carlo methods, the canonical choice for such problems. Finally, we detailed the COS method and how it can be used to compute various risk measures. With the help of two test portfolios, our results show that the COS method accurately recovers the loss distribution and its related risk measures. Furthermore, the error convergence of the approximated distribution as well as the VaR is exponential in nature dependent on

COS Terms	CDF L_2 Error	Conv. Rate	Rel. VaR Error	Conv Rate	Rel. ES Error	Conv Rate
40	3.77415353e-01	1.59	5.52486186e-03	3.1	3.72938106e-03	3.5
50	2.64167142e-01	2.05	2.76243092e-03	20.64	1.70187843e-03	0.74
60	1.81761446e-01	2.38	3.14953092e-03	122.06	1.94994318e-03	14.32
70	1.25838835e-01	2.4	1.85988793e-11	-	2.14507615e-04	6.93
80	9.12513065e-02	2.08	1.85988793e-11	-	8.49160363e-05	8.31
90	7.13840234e-02	1.57	1.85988793e-11	-	3.22751804e-05	10.75
100	0.3783001	-	1.85988793e-11	-	1.03888681e-05	-

Table 3.2: Observed convergence rates for second order (p=4) exponential filter (S & P 1000 obligors)

the number of COS terms and the number of quadrature points used in the numerical integration scheme. The order of the spectral filter used while computing the cosine series expansion, also determines the speed of convergence. We see an accelerated convergence rate in various derived risk measures, when using a filter of higher order. Going forward, the above listed results help us fix our input parameters based on the desired level of accuracy. Additionally, these results also serve as a benchmark for our GPU implementations.

4

Implementation & Results

4.1. Overview

Our aim in this study is to accelerate the computation of the portfolio credit and its risk sensitivities. In the previous chapter, we have seen several models proposed in the literature for simulating these loss distributions. Section 3.5 illustrated the application of the COS method to calculate the loss distribution and the corresponding risk sensitivities within a Gaussian two-factor copula model, with the help of 2 test portfolios. Even though we already showcased the final computed results from the algorithm. These have already been extensively documented in the literature. Our contribution in this work, is to find an GPU parallelized performance-optimal version of the COS method and showcase the improvement over the current state of the art methods like the Monte Carlo models. We begin by explaining the sequential implementation of the COS method. All our test cases calculate the portfolio loss distribution, more specifically the CDF in addition to risk measures like value at risk, expected shortfall and other conditional measures like CVaR and CES. We consider portfolios of sizes varying between one and ten thousand obligors, which are typical sizes of assets in a banking book. For all portfolios, a more realistic case would be if we consider S&P rated default probabilities and loss-at-default per obligor values sampled from $[10, 1000]$, similar to the example given in Section 3.5.3. The following main steps are taken to calculate the loss distribution and risk measures:

1. collect portfolio data

2. compute Fourier coefficients based on the expressions given in Section 3.4

3. Evaluate the loss distribution using the Fourier cosine approximation

4. Compute the risk measures such as VaR from the calculated loss distribution

We first explain the case of a purely sequential implementation of the method developed in Python and C. Next, we describe our different GPU based implementations using C and CUDA. In this way, we can be able to compare our performance in each case systematically, highlighting the scope of each along the way.Then, perhaps we can come up with an optimal GPU-parallel algorithm. These approaches should also provide a taste of the "embarrassingly parallel" nature of the COS method, and thereby hinting a potential to obtain favorable speedups. Finally, we compare our *best possible* with the CPU and GPU versions of a simple Monte Carlo method, that is currently used in practice. In undertaking this type of comparative analysis one is always faced with the question; what should one compare against? In parallel computing two primary schools of thought exist. Best sequential vs. best parallel implementations and best equivalent vs. best parallel implementation. The second argues that for the fair comparison both programs should perform exactly the same work whereas the former focuses only on performance. In this work both approaches are used. In the first instance comparisons between purely sequential and CUDA enabled implementations are best equivalent. In the second case where further specific host or device side optimisations yield performance improvements these are also reported. A second consideration concerns the terminology used. A primary motivation for this work is to exploit the GPU for performance improvements in terms of wallclock execution time. It is tempting to think in terms of speed-up and efficiency. However, in the context of parallel computing speed-up and efficiency are measurements used to quantify performance improvements in homogeneous systems. Detailed performance metrics that we shall make use of, are described in Appendix B.

Portfolio Size	Function	Exec. Time (s)	% of Total time
1000	compute Fourier coeffs	13.29	89.8
	Data Initialization	0.43	2.9
	Compute Risk Measures	0.005	0.034
	All other activities	1.08	7.3
10000	compute Fourier coeffs	67.65	92.3
	Data Initialization	0.0007	0.001
	Compute Risk Measures	1.465	0.2
	All other activities	5.497	7.5

Table 4.1: Profiling results for different portfolio sizes for the sequential version on C

4.2. COS Method: Sequential Implementation
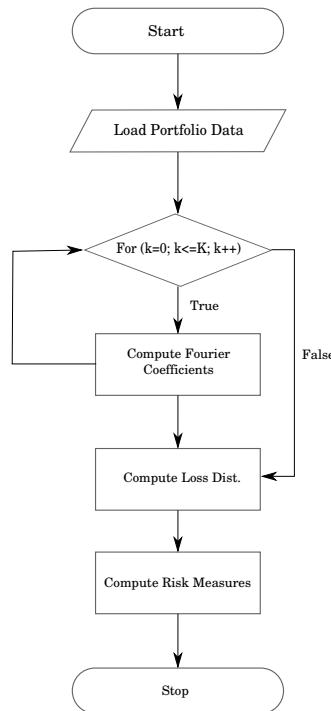
Figure 4.1: Flowchart of the ported C implementation

For the needs of this project, we first analyzed the existing Python implementation to compute the loss distributions and the risk measures, as depicted in the flowchart in Fig.4.1. It can be seen that the algorithm can be divided into three main tasks: data initialization, computation of the Fourier coefficients, and finally the recovery of the loss distribution. From further statistical interpretations on the computed loss distribution, we can obtain the various risk measures and sensitivities. The extracted Python model was then ported to C, since the eventual goal was to implement the algorithm in CUDA. The implementation follows the computation detailed in Algorithm 2 in Section 3.4 step to step.

4.2.1. Profiling
The CPU version of the algorithm was studied with the aim of identifying potential performance bottlenecks as well as devising a parallelization strategy. It also helps us profile the code and provide an estimate of the computational complexity of our algorithm. We use gprof [15], a UNIX based performance analysis tool, for profiling our implementation on the CPU. Clearly, there is a bottleneck in computing the Fourier coefficients. As we have seen in Section 3.4.1, it involves the evaluation and the subsequent integration of the characteristic function.

4.2.2. Computing the Fourier Coefficients

Profiling results indicate that the calculations of Fourier coefficients/characteristic function over different values of ω (Refer Eqn. (3.13)), takes the most amount of computes. It is imperative that the final performance optimal version of the COS includes optimizations of these calculations first.In that way, we can expect to see performance gains. In this section, we try one such approach albeit over the sequential code. The aim is to port an efficient but also a suitable version of the sequential code to the GPU. The Fourier coefficients A_k defined in Eqn. (3.15), depends on the computation of a multidimensional integration over the hyper-rectangle Ω^d, where d is the number of factors in the model. We follow the Clenshaw-Curtis cubature scheme for our implementation (See Appendix A for more details). Implementing a general multidimensional integration is an intricate problem, one that requires careful consideration. The dimensions of the integration are unknown at compile time. Furthermore, as the number of sampling points required increases to the order of the dimension, the parameters in the program demand a larger memory space. We followed two popular techniques used in the literature to perform multidimensional integration in our serial C implementation and mark their performance in our application.

Tail Recursion
Essentially, the problem of numerical integration over multiple dimensions can be seen as a case of nested summation. Naive loop iteration is the classical way to implement to such an operation. As an alternative, a recursive tail call otherwise known as tail recursion can be utilized. The advantage of this method, is that we do not need the dimension at compile time and solves the space inefficiency problem. The schematic showing the tail recursion procedure specific to our implementation, details the exact process followed while computing the integral sum.

Listing 4.1: Numerical Integration using Tail Recursion

```c
double complex integrate(size_t dim, integrand func
    , const double a, const double b)
{

  //return func value if 1st dimension
  if(dim==0)
    return func(); //integrand

  double complex J=0;

  //loop over quadrature points
  for(size_t q=0; q<Q; q++)
    J+= (w[q]*integrate(dim-1,func,a,b));
  J*=(b-a)/2;
  //final value of the integral
  return J;
}

int main(void)
{
  const int d = ... //dimension
  ...
  //integrate over d dimensions under limits [a,b]
  double complex integral = 0.;
  integral = integrate(dim, phi, a, b);
  return 0;
}
```
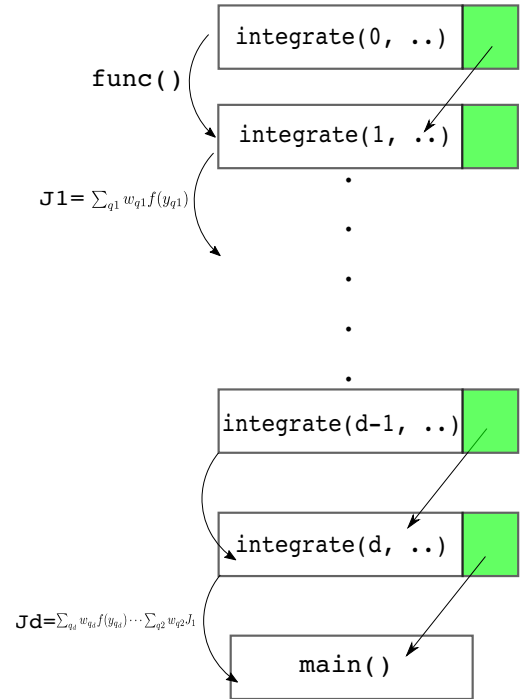

Figure 4.2: Scheme depicting the stack frame along with function calls and returns during recursion

Here, the routine `integrate` contains the call to itself as the last operation as long as the control is within the loop. `func()` is only called when the check for the inner most dimension is evaluated to be true. Since all the previous calls to the function have been stored by the compiler on the stack, this results in a cascading effect of return values from `func()` till the outermost dimension is reached and eventually the sum is stored and the final result is returned.

Matricization
Another way to look at multidimensional integration is with the help of tensors. By using tensor notation, we can represent a general nested summation if we consider $\bar{W}$ is rank-d tensor computed using,

$$\tilde{W}_Q^d = \bigotimes_{j=1}^{d} \tilde{w}_Q^j, \qquad Y_Q^d = \bigotimes_{j=1}^{d} y_Q^j$$

where, $\tilde{w}_Q^j = [\tilde{w}_0^j, \tilde{w}_2^j, \dots, \tilde{w}_p^j, \dots, \tilde{w}_Q^j] \in \mathbb{R}^{Q+1}$ $y_Q^j = [y_0^j, y_2^j, \dots, y_p^j, \dots, y_Q^j] \in \mathbb{R}^{Q+1}$ are the precomputed weights and the sampling points from the Clenshaw-Curtis rule (See Appendix A) and $\otimes$ is the Kronecker product, and similarly, $\Phi(Y_Q^d) \in \mathbb{R}^{(Q+1)^d}$ is the rank-d tensor representing the characteristic function evaluated at the sampling points. Numerical integration then just becomes the tensor contraction operation over,

$$I \approx \tilde{W}_Q^d * \Phi(Y_Q^d) \tag{4.1}$$

where, $*$ represents the contraction operation.

From a computational perspective, it is important to be able to transform the shape of a tensor so that it can be represented as a matrix. We need three pieces of input in addition to the data itself to effectively reshape the data: the size of the tensor, the modes that are mapped to the rows of the matrix, and the modes that are mapped to the columns of the matrix. Hence, the matricization of any general tensor $\mathscr{X} \in \mathbb{R}^{q_1 \times q_2 \times \dots q_Q}$ can be defined as,

$$\mathbf{X}_{\mathscr{R} \times \mathscr{C} : q_Q} \in \mathbb{R}^{M \times N}$$

with $M = \prod_{i \in \mathscr{R}} q_i$ and $N = \prod_{i \in \mathscr{C}} q_i$ The indices in $\mathscr{R}$ are mapped to the rows and the indices in $\mathscr{C}$ mapped to the columns. We then have,

$$(\mathbf{X}_{\mathscr{R} \times \mathscr{C} : q_Q})_{m,n} = x_{q_1 q_2 \dots q_Q}$$

We do not indulge in more details here. A nice review of matricization of tensors and along with its applications has been studied in [22]. For our current work, we limit our dimensions to two factors. This choice is important for our GPU implementations as we are limited by the device's global memory capacity. We shall see this in detail in the upcoming sections. Finally, implementing the summation of over the indices of the resulting 2-D matrices can be achieved using simple loops. Hence, we do not detail this approach here.

Timing Runs & Remarks
Here we see the CPU run times of both the approaches for numerical integration described above for different number of COS terms (K). We see that matricization has a consistent speedup of almost 50% over

COS terms (K)	Exec. Time (s) Tail Recursion	Exec. Time (s) Matricization
40	10.865	5.627
50	13.030	6.523
60	14.875	7.291
70	20.402	9.136
80	21.233	9.626

Table 4.2: Execution times for the serial COS implementation with different integration strategies (1000 obligors Gaussian 2 factor copula)

the tail recursion approach as we increase the number of integrations, related to the additional COS terms that are computed. This confirms the existing knowledge that recursions are time intensive operations over naive iterations. Recursion is usually slower and uses more memory because of the overhead of creating and maintaining stack frames. Going forward, we stick to matricization strategy as our ultimate aim is to find a performance optimal version of the algorithm. The matricization approach also has the added advantage that it involves linear algebra operations over large dense matrices, perfectly suited for utilizing efficient GPU algorithms.

4.3. COS Method: Parallel Implementation

Based on the profiling results of the sequential algorithm, we find that the time intensive part is computing the Fourier coefficients taking almost 90% of total runtime. Therefore, it makes sense that we target our efforts to optimize the performance of this part first. Each Fourier coefficient A_k inside the loop over the COS terms

(K) requires three main steps. First, we need to compute the characteristic function integrand grid defined in Eqn. (3.13) over the complex space, $\Phi(\mathbf{z}; \omega_k) \in \mathbb{C}^{(Q+1)^d}$. This term is reduced to $\phi(\omega_k) \in \mathbb{C}$ through numerical integration and then to finally obtain the scalar, $A(\phi) \in \mathbb{R}$ (See Eqn. (3.15)). These steps are repeated over K terms to get our whole Fourier cosine basis $\{A_k : 0 \le k \le K\}$, which can then be used to compute our approximation to the loss distributions as explained in Eqn. (3.16). To be efficient in our efforts, it helps to study the sub-tasks individually. Flowchart in Figure 4.3 depicts the tasks to compute the coefficients. As we observe, there arise multiple approaches to parallelize these steps. We now explain these choices individually in detail.
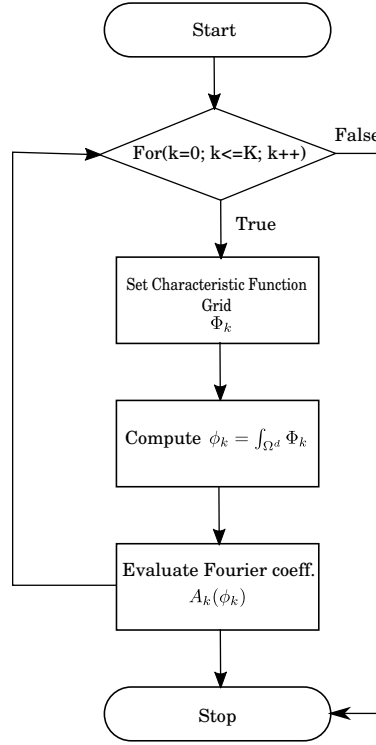

Figure 4.3: Flowchart for the subtasks during the computation of Fourier cosine coefficients $\{A_k : 0 \le k \le K\}$ where K is the number of COS terms

4.3.1. GPU0: Subdividing the Omega dimension
As established in earlier sections, to create our Fourier basis vector, we need to compute multiple Fourier cosine coefficients, denoted by A_k. As a first step, a simple loop unrolling can performed to distribute the evaluation of each A_k over available processing elements. We see that A_k depends on the computing of $\omega_k = \frac{k\pi}{l_b - l_a}$. Thus, this can be subdivided over K threads. For establishing a naming convention, we term this approach GPU0. The COS method has a exponential rate of convergence (see Section 3.5.4), hence it is needless to use more number of COS terms and since we know, that launching a few threads in CUDA would lead to under utilization of the available hardware, we do not expect this version of parallelization to be an optimal one in terms of performance. In fact, in some cases, the performance is observed to be lower than the corresponding sequential version as the time taken by initializing the CUDA API environment and data transfers between device and CPU memories over weigh any time reduction achieved through parallelization. Nonetheless, this approach is our starting point and will help us understand more specialized kernels, as we shall explain them in the next sections.

Results
To find the optimal thread division, we follow 2 types of GPU thread divisions- one where all threads allocated over a single block (pt:1xn) and another where 32 threads are taken per block (pt:32xn). As observed in the plot, the latter performs slightly better with regards to computation time. Therefore, the division of work over constant 32 threads per block is a better alternative than using just a single block with all threads. In the

COS terms (K)	GPU (C/CUDA)	CPU (C)	Speedup	CPU (Python+Numpy)	Speedup
32	12.311	4.997	0.41	4.601	0.37
64	12.439	9.398	0.76	8.154	0.66
128	12.444	18.396	1.48	14.77	1.19
256	13.307	37.061	2.79	28.89	2.17
512	13.375	72.122	5.39	57.735	4.32
1024	13.675	143.24	10.47	117.69	8.61

Table 4.3: Execution times and corresponding Speedup ratios for two sequential implementations (C and python) with respect to CUDA GPU0 implementation for various COS terms (K) for computing the CDF of 1000 S&P obligors 2 factor Gaussian copula model. The definition followed for Speedup is the total CPU wallclock time divided by the total GPU wallclock time

particular example, we can see that the GPU version is faster than the CPU version only when the K taken is large. However we note that it is not necessary to take such a large values of K in the COS method in practice, as with $K = 256$, the distribution already in the order of 10^{-6}. Therefore, the COS method with smaller values of K is not really advantageous on the GPU. Profiling results on GPU0 show that the maximum amount of time

Figure 4.4: Execution times and corresponding Speedup ratios for two sequential implementations (C and python) with respect to CUDA GPU0 implementation

is still the computation of the characteristic function in every ω. Therefore, there is a need to optimize this portion further.

4.3.2. GPU1: Subdividing the Integral

As mentioned in 4.3.1, the preliminary GPU implementation of the COS method with smaller values of K is not really advantageous. The loss values already are within the required tolerance. However, there are subtasks within the algorithm which can be optimized. In equation (3.13), for each ω, we need to compute a multidimensional integral for the characteristic function over the factors of the copula. Numerically, an integration is just a nested summation over the sampling points and the cubature weights. We have that is,

$$\phi_L(\omega) = \int_{\Omega^d} \Phi(\mathbf{z};\omega)d\mathbf{z} \approx \sum_{q_d} w_{q_d} \cdots \sum_{q_1} w_{q_1} \Phi(\mathbf{z};\omega)$$

We have already seen this in our sequential implementation (See Section 4.2.2 and Appendix A for Clenshaw-Curtis explainations). In order to effectively port this function to a parallel implementation, we can use the

Portfolio size (N)	GPU (C/CUDA)	CPU (Python+Numpy)	Speedup
1000	0.875	28.512	32.59
2000	1.066	54.351	50.99
3000	1.174	81.294	69.25
4000	1.369	119.16	87.04
5000	1.54	163.08	105.90
6000	1.773	215.853	121.74
7000	1.8831	265.87	141.19
8000	2.088	328.65	157.40
9000	2.305	380.87	165.24
10000	2.476	424.63	171.50

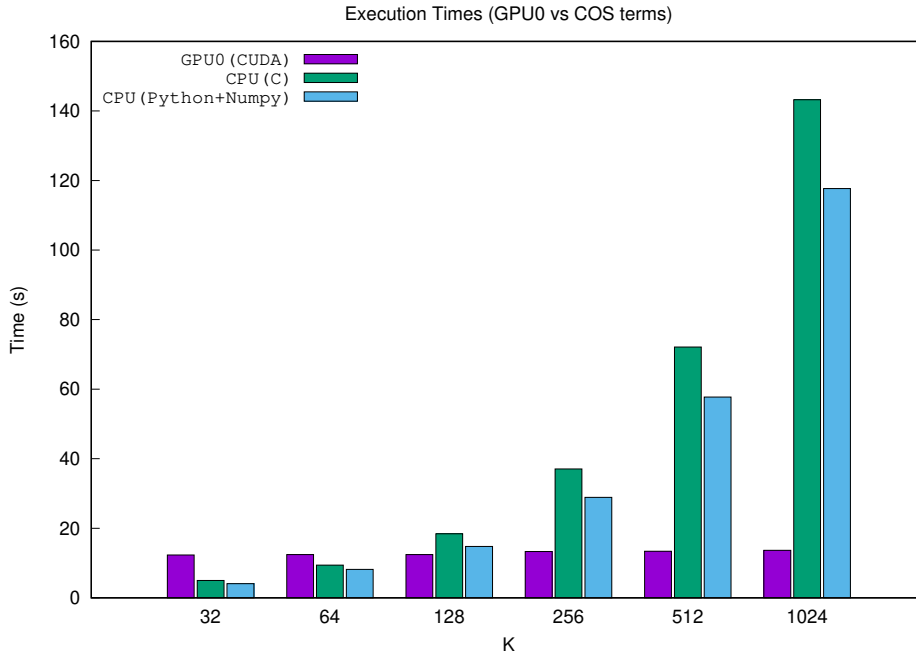Table 4.4: Execution times and corresponding Speedup ratios for two sequential implementation in python with respect to CUDA GPU1 implementation for various portfolio sizes. Here we omit our C implementation results since the CPU timings on both Python and C were almost similar.

"Reduce" operation. As we have already explained in the introductory sections in 2.4.5, numerical integration or a weighted summation is well suited for parallel processing. Due to the commutative and associative properties of the addition operator, there is no dependence of the computation if we split our task into multiple sub-integrals. This is the essence of the `GPU1` approach. Here, we subdivide the integration domain into multiple sub-domains for efficient parallel computation. Finally, the multiple results from each sub-domain are further reduced into a final sum. With the help of CUDA's thread hierarchy, we can subdivide both the computation of each A_k (essentially the `GPU0` approach) and the integration over multiple blocks and threads respectively. An introductory note listing the important concepts of CUDA programming methodology is given in Section 2.1.3. Figure 4.5 graphically illustrates the entire process in the case for 2-factors.

(a) Subdivision of work over CUDA threads and blocks. Each $\phi(\omega)$ is separated over blocks, meanwhile for each block the characteristic function grid is calculated.

(b) Summing of all $\Phi(\omega; \mathbf{z})$ across threads using `atomicAdd()` directive inside a block
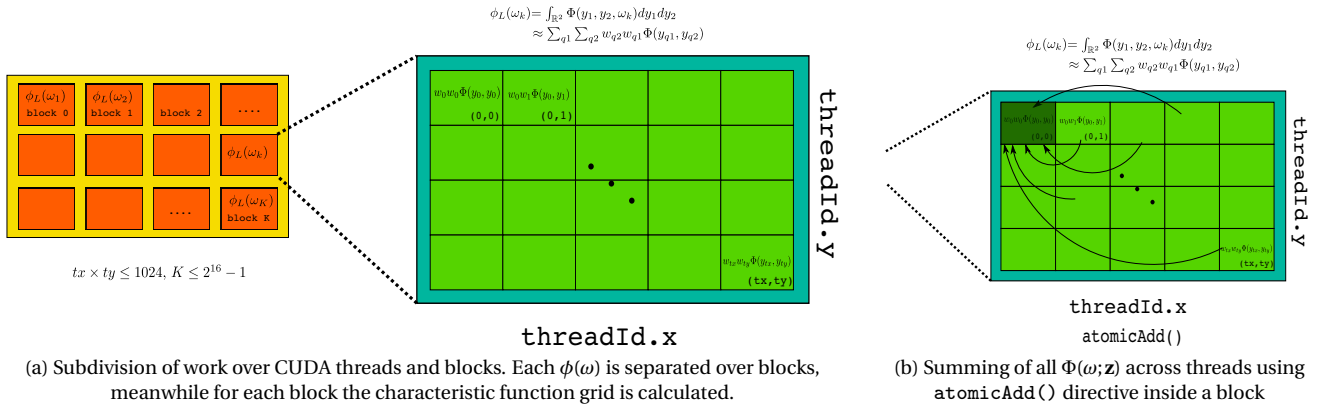
Figure 4.5

Results

The well known bound for time complexity of a parallel sum reduction [21] is known to be $\mathcal{O}(Q/P + \log_2 P)$, where Q in our situation becomes the number of quadrature points and P is the number of threads. Since the number of quadrature points are fixed for various portfolio sizes, we see a scaling of $\mathcal{O}(\log_2 P)$. In our examples, the total number of threads is product of COS terms and the quadrature points.

$$P \approx 1024 \times K$$

where, 1024 are the maximum number of threads in a block. We track the results of `GPU1` method with different portfolio sizes and compute the speedups. As we observe in Table 4.4, the speedup with respect to the sequential COS method is significant as we increase the portfolio size by a factor of 100x or more. This is a good advantage to have considering in realistic industry cases, we generally have portfolio sizes of more than 1000 obligors.

Figure 4.6: Execution times for GPU1 and GPU2 in comparison with sequential implementation. Both GPU implementations show a order of magnitude performance gain for a 2-factor Gaussian copula model.

4.3.3. GPU2: **Reducing the Characteristic Function**

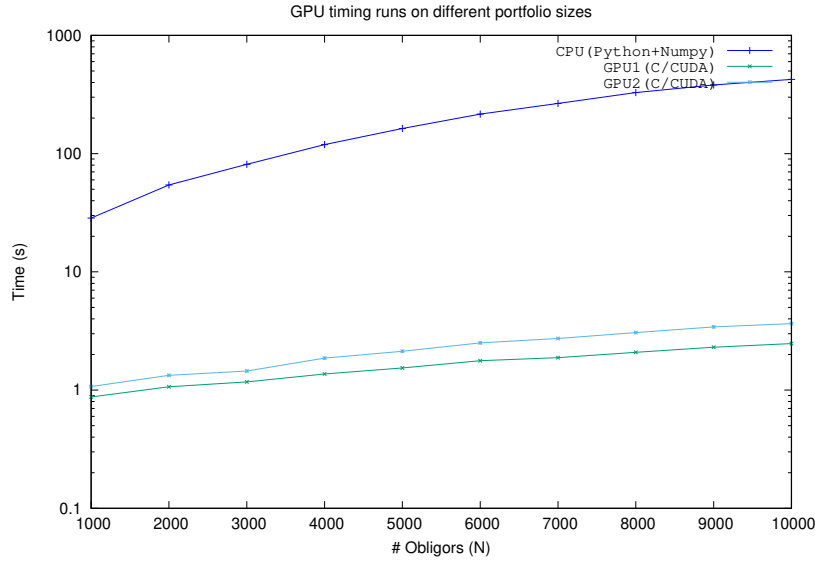To further identify areas of improvement, we concentrate on our computation of the characteristic function $\Phi(\omega; \mathbf{z})$ instead of $\phi(\omega)$, which is essentially the sampled values for the integrand. Recall from eqn. (3.13) the computation as follow,

$$\phi_L(\mathbf{z}; \omega) = \int_{\Omega^d} f_{\mathbf{Z}}(\mathbf{z}) \prod_{n=1}^{N} \underbrace{\left[1 + P(\epsilon_n \leq \alpha_n(\mathbf{z}_n))(e^{i\omega l_n} - 1) \right]}_{X \in \mathbb{C}^{N \times (Q+1)^d}} d\mathbf{z}. \tag{4.2}$$

The product term, as demarcated above is the expectation of the characteristic function of the n-th obligor. Hence, the computation of this value can be considered independent, before a product operator reduces it to single value. Analogous to a parallel summation, the multiplication operator is also a commutative and an associative operator, hence each process can compute a single multiplicand independently. This approach has the advantage of scaling over the number of obligors N, as well the number of quadrature points Q. A step forward in comparision with GPU1, and hence the naming convention, GPU2. A representative matrix is shown in Figure to visualize how we implement the reduction. The size of the matrix is typically in the range of what we should expect while computing the loss distribution in a large portfolio.
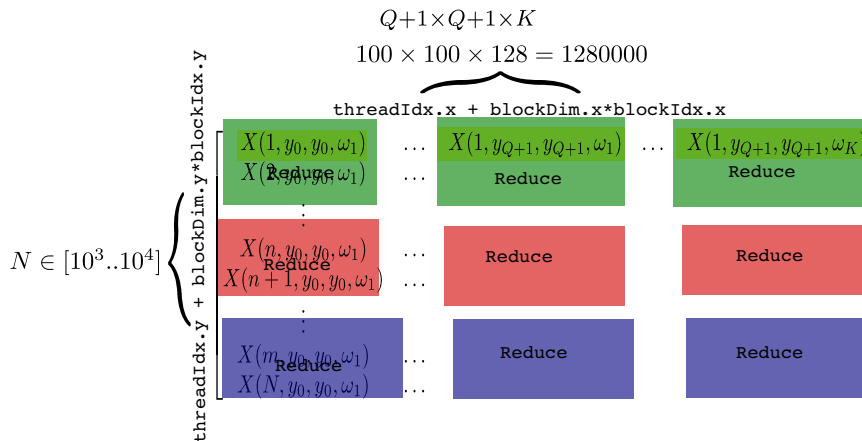

Figure 4.7: Here X is the tensor represented in 2-D matrix from Eqn. (4.2) in a two-factor model. Each different "Reduce" performs the multiplicative reduction in a single thread.

Portfolio size (N)	GPU2 (C/CUDA)	CPU (Python+Numpy)	Speedup
1000	1.072	28.512	26.60
2000	1.333	54.351	40.77
3000	1.449	81.294	56.10
4000	1.863	119.16	63.96
5000	2.127	163.08	76.67
6000	2.505	215.853	86.17
7000	2.731	265.87	97.35
8000	3.068	328.65	107.12
9000	3.424	380.87	111.24
10000	3.648	424.63	116.40

Table 4.5: Execution times and corresponding Speedup ratios for two sequential implementation in python with respect to CUDA GPU2 implementation for various portfolio sizes

Results & Remarks

GPU2 speedup ratios show impressive speedups with respect to their sequential counterparts, almost 70-100x when computing larger portfolios. However, GPU1 is slightly faster when compared to the walltime elapsed during execution. This maybe due to the sequential execution of the K dimension. The tensor X in Eqn. (4.2) is of the size $X \in \mathbb{C}^{N \times (Q+1)^d}$. Typically, a complex single precision float has the size of 8 bytes. Hence the total size required to store the entire tensor X for $N = 10000$ becomes almost 102 GigaBytes! Much larger than the device memory which is typically around 12GB-32GB even in the latest GPU cards. Thus, it is impractical to store such a large matrix at once in the memory for computation. We reduce the size of the matrix in our implementation, therefore we compute the reduced matrix $\tilde{X} \in \mathbb{C}^{N \times (Q+1)^d}$, computing the K dimension sequentially. This is a limitation in the GPU2 approach, in contrast to the GPU1 where there is no concern of memory storage. This sequential computation of K different kernels in a loop, affects the performance gain in GPU2 kernel.

However, there is an additional benefit to this approach compared to GPU1. We have so far been testing with portfolios to compute their loss distribution approximation in accordance to Eqn. (3.18). From the computed approximation, we can find risk measures such as VaR and ES of the portfolio at any confidence interval. There are however, other risk measures that our GPU approach must efficiently compute. Some of them we have already seen in Section 3.4.5, where we introduced the conditional risk measures such as CES and $CVaR$. Recall from Eqn. (3.22)- (3.24), the CES and the CVaR both require the computation of $P(L \geq VaR_\alpha | x_n \leq \xi_n)$. This probability is evaluated again through the COS method. We know that the characteristic function of this conditional loss distribution is just a slight modification of the original characteristic function (See Eqn. (3.23)). Therefore, in GPU2 approach, we can directly parallelize the modified characteristic function along with the original one. Therefore, the computation now includes two different multidimensional integrals. In addition to Eqn. (4.2), the kernel must also compute,

$$\phi'_n(\mathbf{z};\omega) = \int_{\Omega^d} f_{\mathbf{Z}}(\mathbf{z}) \underbrace{\prod_{j=1, j \neq n}^{N} \left[1 + P(\alpha_j(z_j))(e^{i\omega l_j} - 1) \right] \cdot P(\alpha_n(z_n)) \cdot e^{i\omega l_n}}_{X'_n} d\mathbf{z}. \tag{4.3}$$

A faster trick is to compute, $X'_n = \frac{X}{P(\alpha_n(z_n)) \cdot e^{i\omega l_n}}$. This way we can reuse X that is already available through computation of GPU2. Computing X'_n is just one floating point operation through this optimization, which has negligible time addition in contrast to GPU1, where the computation of the kernel with the modified characteristic function needs to be redone. We see this effect in Figure 4.8 for multiple portfolio sizes conditional to a defined defaulted obligor ($n = 12$).
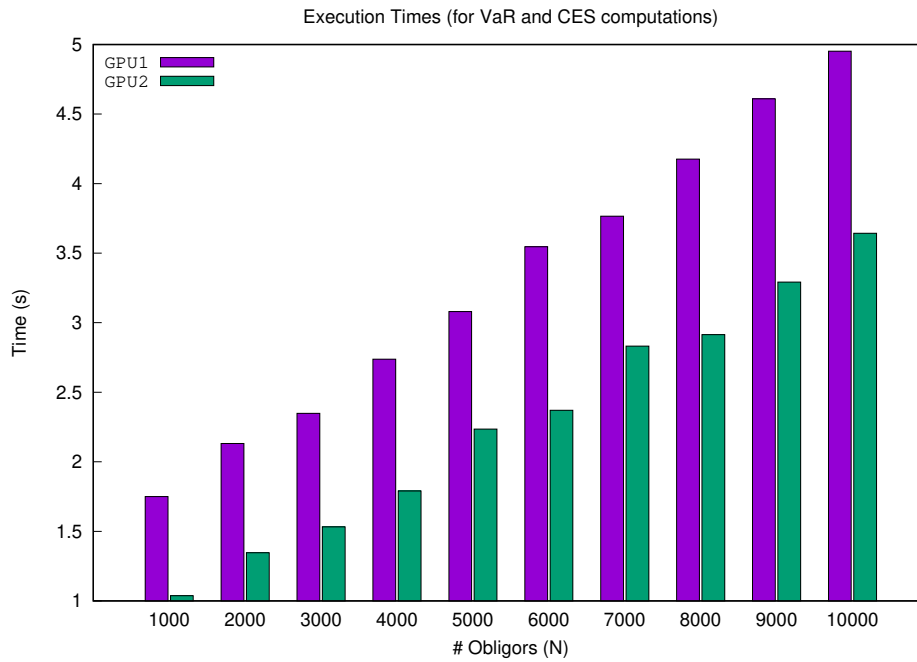
Figure 4.8: Execution times for CES_{12} and VaR computations

Portfolio size (N)	GPU1	GPU2	Speedup
1000	1.75	1.037	1.69
2000	2.132	1.346	1.58
3000	2.348	1.532	1.53
4000	2.738	1.79	1.53
5000	3.08	2.235	1.38
6000	3.546	2.37	1.50
7000	3.766	2.832	1.33
8000	4.176	2.914	1.43
9000	4.61	3.292	1.40
10000	4.952	3.643	1.36

Table 4.6: Execution times and corresponding Speedup ratios for GPU1 and GPU2 for various portfolio sizes computing CES_{12} and VaR. W.l.o.g, we consider the 12th obligor to be the defaulted obligor.

5

Concluding Remarks

This study was conducted to explore the potential of the COS method on the GPU hardware for applications in estimating the portfolio loss distributions and its sensitivities in factor copula models. The initial literature study provided an opportunity to gather the theoretical background necessary for understanding the problem of computing portfolio credit risk measures using factor copula based COS method and its potential for data parallelism. We briefly outlined the various modelling techniques to compute the portfolio loss distribution and the corresponding risk sensitivities. Traditional methods still used in this field use Monte Carlo approaches eventhough a reliable method, but unfortunately are computationally slow. Our focus was on the relatively new COS method is based on the Fourier cosine expansion of the characteristic function of the portfolio loss random variable. The semi-analytical COS method is an attractive alternative that demonstrates superior computational speed as well as accuracy. However, a drawback of the method is the impact on computational speed when extending the higher dimensional spaces, such as with the multifactor copula models. However, due to the "embarrassingly parallel" nature of the algorithm, the COS method can be optimized further with the help of data parallelism concepts such as the GPU device. Hence in Chapter 4, we described our different approaches to implement the GPU parallel version of the COS method and profiled our results. First our sequential implementation involved two approaches to compute the multidimensional integration that arose in the computation of the characteristic function. The difference in performance between the matricized and the recursive approach led us to choose the matricized version and set our direction towards further optimizations. Profiling of the sequential code showed a clear hotspot in terms of the latency of the algorithm. This was the computation of the Fourier coefficients. These observations led us to our first implementation with respect to computing the K Fourier coefficients over separate threads. The performance improvement was not significant in the case of lower number of COS terms. And since, the COS method has an exponential convergence rate, it is unnecessary to compute with larger K. After profiling the first approach, it was found that a considerable amount of time is still spent in the computation of the characteristic function integral for each of the coefficients. Thus, our next two approaches tackled this aspect. GPU1 divided the domain of integration over several threads within a block whereas, we divided the computation of the each Fourier coefficient A_k over different blocks. This resulted in significant speedups with respect to the sequential implementation. In pursuit of wringing out further performance optimization, we tried to divide the calculation of the characteristic function within the integral, which we termed as GPU2. Even though in the computation of the loss distribution, VaR and ES, the GPU1 is faster than GPU2. Reusing the tensor X in Eqn. (4.2) for computing the CES values, leads GPU2 to be doubly faster than GPU1. Hence, our preferred candidate for an performance optimal GPU implementation of the COS method, with minimal difference in accuracy with respect to the sequential version.

5.1. Discussions
This section reflects on some points of the thesis. We combine this with some directions for further research.

5.2. Scope of this work
This work was performed with the cooperation of ING's Quantitative Risk Analysis team. Our main aim was to address the following questions:

- We have noted the computational complexities of the "workhorse" Monte Carlo methods and the COS method in computing large portfolios with high dimensional correlated structures. Is it possible to find an optimal parallel implementation ? Our achieved results show that both our GPU implementations is almost 20-100x faster than the sequential versions, indicating the suitability of the COS method on the GPU framework. Therefore, the achieved results can be beneficial to compute loss distributions and risk measures of large realistic portfolios, that are commonly used during risk calculations for a financial institution in faster but also with accuracy.

- As outlined in the previous subsections, the parallel optimization of the COS method can be achieved by different approaches. Which was the most appropriate approach to reduce the execution time or does the most efficient approach include a careful combination of them all? What could be understood about of performance as well as in the accuracy of the parallel version over the existing serial routine ? We find that the optimal version GPU2 answers the above questions. This was an approach that was a culmination of all other approaches within itself. As we follow top-down approach, we go from a simple loop unrolling to the optimal calculation of the most innermost iteration, that was the calculation of the characteristic function. Finally, this version resulted in significant performance upgrades over not only the sequential versions but also the other GPU implementations, including the ones available externally from CUDA toolkit like `thrust` and `cuBLAS`.

- Our final aim was to compete with the canonical Monte Carlo method, that is currently widely used in the industry. We show that with the performance comparison between the GPU implementation of a naive Monte Carlo method, and the COS method. The COS method is shown have a speedups of almost 5, with minimal effect on the accuracy of the computed distribution and the risk measures. Hence, the performance metrics indeed improves upon the current industry standard.

5.2.1. Factor Copula models

Our current numerical experiments utilized a straightforward implementation of the two factor Gaussian copula correlation structure. However, many different model configurations are possible. First it would be interesting to compare the results from the factor model in our situation, for example with the Gaussian-t model (Ref. 3.1.2) applied by [34]. This will be straightforward addition to the method described here, since the only affected expression will be evaluating the inverse cummulative distribution function of the threshold defined in (3.8). Next, while computing the loss distribution, there are different systematic factors we could have applied for modelling the capital charge. The 'sector' and 'country' factor are the two systematic factors the BCBS suggests. Hence, our approach accelerates Default Risk Charge calculations, since we have considered two systematic factors in this study. An interesting extension to this work will be to find out the performance difference with multifactor copulas, allow for a more detailed specification of correlation structure between obligors.

5.2.2. Numerical Cubature

Finally, another direction for further research is utilizing sophisticated numerical cubature techniques present in literature. Our approach relies heavily on the tensor product grid approach to cubature. Alternatives include dimensionality reduction techniques such as the Principal component analysis (PCA). It aims to reduce the dimensionality of highly correlated data by finding a small number of independent linear combinations that approximately captures all the characteristic behaviours of the original data. In the same vein as factor copula models although PCA is not a model in itself, it can be used as a way of constructing appropriate factors for a factor model. The key mathematical result behind the technique is the theorem of *spectral decomposition* of normal matrices. Although this is a popular approach largely discussed in the literature, further exploration can be made for the use the principal components computed, within an data-parallel version of the factor copula based COS method.

<div align="right">

A

</div>

Numerical Integration

Integration is one of the fundamental concepts of Calculus alongside differentiation. Even though, the Lebesgue integral of many functions is proved to exist mathematically, practical computations demand its evaluation. In many cases, we make do with its numerical approximate. In numerical integration, the integrand is evaluated at section of finitely many sample points. Thus, the finite sum approximation of the sum of these sample points is popularly known as quadrature, in the univariate case and cubature in higher dimensions. For instance, while computing the characteristic function at a certain ω, we have the following expression,

$$\mathscr{I} = \phi_L = \int_{\mathbb{R}^d} \Phi(\vec{y}; \omega) d\vec{y}$$

$$\approx \sum_{q_1=0}^{Q} w_{q_1} \cdots \sum_{q_d}^{Q} w_{q_d} \Phi(y_{q_1}, \ldots, y_{q_d}; \omega)$$

To accurately approximate the integral over higher dimensional spaces, several sampling techniques have been developed. The type of sampling of Q such points in the integration domain leads to different schemes and represents a significant challenge in mathematics. In this appendix,, we motivate our choice of the Clenshaw-Curtis cubature scheme. We briefly detail the procedure used to calculate the weights and the sampling points for the Clenshaw-Curtis rule. We then compare our chosen scheme with the regular Trapezoidal Rule and observe their differences in convergence behaviors.

A.1. Clenshaw-Curtis Quadrature

Without indulging in intricacies, we show the computation of an one-dimensional integration using the Clenshaw-Curtis scheme. This procedure is analogous in higher dimensions, which has been extensively been covered in the literature. A good reference is We have,

$$\int_{-1}^{1} \Phi_L(y) dy = \int_{0}^{\pi} \Phi(\cos\theta) \sin\theta d\theta$$

$$\approx \frac{a_0}{2} \underbrace{\int_{0}^{\pi} \sin\theta d\theta}_{2} + \sum_{k=1}^{\infty} a_k \underbrace{\int_{0}^{\pi} \cos(k\theta) \sin\theta}_{\frac{1+(-)^k}{1-k^2}}$$

$$= a_0 + \Sigma_{k=1}^{\infty} \frac{2a_{2k}}{1-(2k)^2}$$

$$= W^T \underbrace{\vec{a}}_{DCT}$$

where we take, $\Phi(\cos\theta) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(k\theta)$ and and

$$\vec{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_Q \end{bmatrix} \qquad W = \begin{bmatrix} W_0 \\ W_1 \\ \vdots \\ W_Q \end{bmatrix}; \qquad W = \begin{cases} 1 & k=0 \\ \frac{2}{1-k^2} & k \neq 0 \,\&\,\text{even} \\ 0 & k\,\text{odd} \end{cases}$$

Here, DCT refers to Discrete Cosine Transform method. We know, $\vec{a} = [a_0, a_1, \ldots, a_Q]^T$ are the $(Q+1)$ Fourier coefficients of the periodic function $\Phi(cos\theta)$. Then, by DCT,

$$\vec{a} = \Lambda\Phi(y;\omega)$$

where, $\Phi(y;\omega) = [\Phi(y_{q_0}), \ldots, \Phi(y_{q_d})]^T$ and $\Lambda \in \mathbb{R}^{Q+1,Q+1}$ is the Fourier matrix,

$$\Lambda = \frac{2}{Q} \begin{bmatrix} \frac{1}{2} & 1 & \cdots & \frac{1}{2} \\ \frac{1}{2} & \cos\frac{\pi}{Q} & \cdots & \frac{1}{2}\cos\pi \\ \frac{1}{2} & \cos\frac{2\pi}{Q} & \cdots & \frac{1}{2}\cos 2\pi \\ \cdots & & \ddots & \\ \frac{1}{2} & \cos\pi & \cdots & \frac{1}{2}\cos(Q\pi) \end{bmatrix}$$

A.1.1. Precomputing the weights

We know that the integral can be represented as the weighted summation of the Fourier coefficients $\vec{a}$. This summation can be rearranged as follows,

$$\int_{-1}^{1} \Phi_L(y)dy = W^T\vec{a}$$
$$= (W^T\Lambda)\Phi(y;\omega)$$
$$= (\Lambda^T W)^T\Phi(y;\omega)$$
$$= \underbrace{(\Lambda^T W)^T}_{\text{DCT Type 1}} \Phi(y;\omega)$$
$$= \tilde{w}^T\Phi(y;\omega)$$

Since, the Fourier matrix Λ is rearranged to operate on W now, instead of Φ, the term $\Lambda^T W$ is another DCT of type I. This allows the weight vector to be independent of the function vector at sampling points. The modified quadrature weights $\tilde{w}$ can now be computed with the help of Fast Fourier Discrete Cosine Algorithms in $\mathcal{O}(Q\log Q)$ time. This when extended to higher dimensions is particularly beneficial as the weights $\tilde{w}$, needed to be computed only once and can be reused for every other dimension, as follows,

$$\mathcal{I} = \phi_L \approx \sum_{q=0}^{Q} \tilde{w}_{q1} \cdots \sum_{q_d}^{Q} \tilde{w}_{qd}\Phi(y_{q_1}, \ldots, y_{q_d};\omega)$$
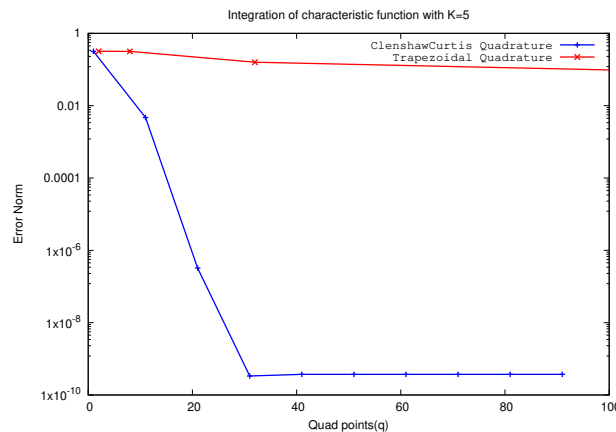
A.2. Effect on Convergence

Figure A.1: Computation of one-factor Gaussian characteristic function at $\omega = \frac{5\pi}{l_b - l_a}$

Although the choice of the numerical quadrature scheme to be used while computing the characteristic function depends on the individual, the superior convergence properties of the Clenshaw-Curtis rule makes

COS Terms	Execution Time (s) (Clenshaw-Curtis)	Execution Time (s) (Trapezoidal)
70	0.13975	0.651096
90	0.28075	1.308046
110	0.63329	2.957913
130	1.5189	7.094365
150	3.1304	14.621308
200	14.683	68.579506

Table A.1: Observed Execution times (s) with different integration schemes for COS method for SP 1000 obligors in one-factor Gaussian copula model (for integration truncation error = 1.e-7)

it a ubiquitous choice when using the COS method. As a side exercise, we compared the computation of the characteristic function through the Trapezoidal integration rule and compared it with the Clenshaw-Curtis quadrature scheme. For a particular ω, the characteristic function ϕ was computed and the convergence was observed, as shown in Fig. A.1. The result clearly shows the superior convergence properties of the Clenshaw-Curtis rule over the Trapezoidal integration rule. The convergence to the desired accuracy takes order of magnitude fewer sampling points than the Trapezoidal Rule. This also directly translated in the execution times of the COS method. (see Table A.1)

B

Test Setup and Performance Metrics

We detail our test setup used for our experimental runs. Specifically, we list out the hardware and software details of all the platforms we are using, and also discuss some common performance metrics that we profile for our different implementations.

B.1. Hardware/Software Platform

We make use of the DAS-5 [3](The Distributed ASCI Supercomputer 5) cluster designed by the Advanced School for Computing and Imaging for running and profiling our CPU and GPU implementations. DAS-5 runs on CentOS Linux release 7.4.1708, with the standard compute node equipped with Intel Xeon E5-2630-V3 CPUs. The GPU nodes that we utilize are equipped with NVIDIA TitanX-Pascal generation cards with 12 GB onboard memory. For developing and running our code, we use the NVIDIA CUDA 10.0 toolkit software platform for our GPU implementations. Specific hardware/platform specifications are listed in Table B.1

Specification	CPU: Intel Haswell E5-2630-v3	GPU: NVIDIA TitanX-Pascal
Cores/Threads	8/16	28/3584
Processor Frequency (GHz)	2.4	1.4
On Chip Memory (GB)	-	12
Cache Size (KB)	256 (L2)	3072(L2)
Memory Bandwidth (GB/s)	59	480
Throughput	130 GFLOPS	11 TFLOPS
Software Platform	gcc 6.3.0 /Python 3.4.5	NVIDIA CUDA 10.0 SDK

Table B.1: Hardware/Software Configuration on DAS-5

B.2. Performance Metrics

To do comparative analysis between our sequential and GPU accelerated implementations, we mark our performance against these common metrics listed below.

- **Latency** (T): The execution time or latency is the total time elapsed from giving input to the program to the outputting of results. Since our study involves both sequential and parallel computing algorithms, we use the wall-time measurement on the CPU and CUDA specific event APIs to record elapsed time taken by kernel executions.

- **Memory Bandwidth** is the rate at which data is transferred to and fro from the global memory. Theoretical bandwidth is the peak data read/write processing capability of a GPU, and can be computed using the given hardware specifications. Our aim is to compare this metric with our achieved bandwidth or effective bandwidth. This can be calculated by the sum of the total bytes written(b_w) and the total

bytes read (b_r) divided by the elapsed kernel time (T).

$$MB = \frac{b_r + b_w}{T}$$

- **Speedup** (S) Plainly speaking, Speedup is the ratio of execution times T_1/T_2 between two different implementations. Within Parallel computing this metric can have two different definitions. When the computational load is shared between p processors, "*Best effective*" Speedup can be given by,

$$S = \frac{\text{Execution time on 1 CPU}}{\text{Execution time on p CPUs}}$$

Whereas, the "*Best possible*" Speedup is the straightforward metric that is ratio the execution times between the 2 implementations, for instance,

$$S = \frac{\text{Execution time-CPU Sequential Version}}{\text{Execution time- GPU Implementation}}$$

. In our study, the aim is to compare the performance between a purely sequential algorithm with that of a GPU enabled accelerations. Hence, we use the latter definition in our results.

- **Compute Throughput** In contrast to the memory bandwidth which measures the memory throughput, compute throughput is a metric used to quantify the computational efficiency. It is thus calculated as the total number of floating point operations within a kernel divided by the corresponding elapsed execution time. It is measured in a unit called Floating Point Operations Per Second (FLOPS).

$$FLOPS_{effective} = \frac{\text{\# of floating point operations}}{T}$$

C

Further Results

C.1. Monte Carlo Method: Parallel Implementation

We consider the results with respect to Monte Carlo method implemented on CUDA with the help of external libraries such as `cuRand`. Each Monte Carlo instance is given its separate thread. The Monte Carlo method, even though remains at a constant time complexity, is not faster than the COS for our test cases until 10000 obligors. Our achieved results show that the `GPU2` approach for the COS method is faster than the Monte Carlo on the GPU by almost a factor of five.
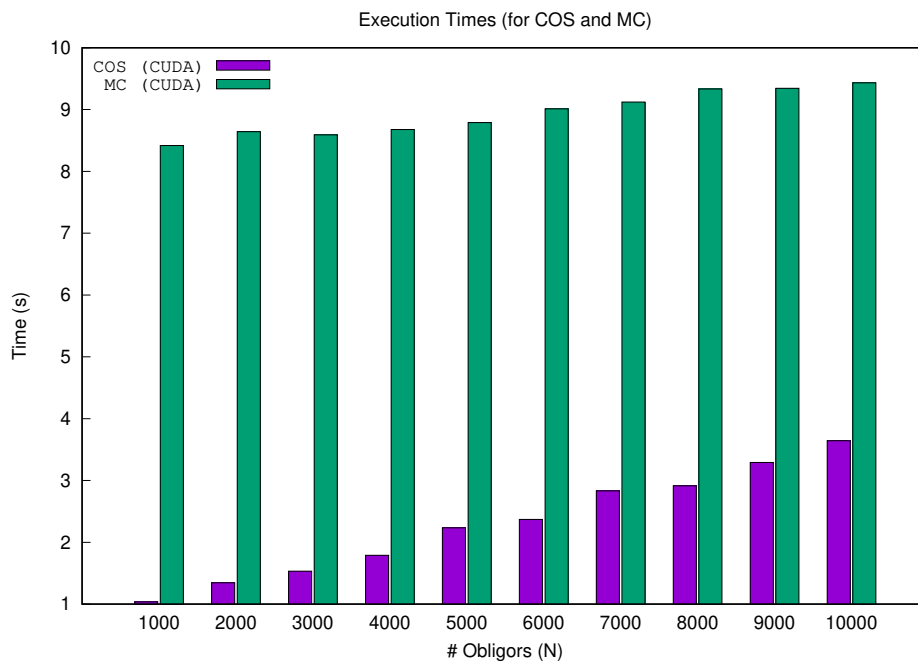

Figure C.1: Execution times for COS and MC on GPU. MC is run with 1e+6 simulations.

C.2. Comparision with External Libraries

Here we detail our achieved results with external CUDA libraries for implementing the parallel reduction for summation and multiplication. For our tests, we chose the prevalent method used in `thrust`, `thrust_reduce_by_key()` and for summation, `cuBlas` routine, `cublasCgemv` that performs complex matrix-vector multiplication. For summing a matrix along its rows, it is equivalent to compute the matrix-vector multiplication of the desired matrix with a vector of ones. We show that for equivalent sizes of different matrices that we see in our test

45

Portfolio Size	GPU1	GPU2	thrust_reduce	cublas Cgemv (with plan)
1000	0.875	1.037	1.0865	1.0965
5000	1.54	2.235	0.2679	0.2563
10000	2.476	3.643	4.122	3.892

Table C.1: Execution times for implementations with different reduce approaches

cases, both GPU1 and GPU2 outperform the implemented solutions in external CUDA libraries. Here, the difference in timings only stem from reducing the $\Phi(\omega; \mathbf{z})$ for various portfolio sizes, since all other parts remain common amongst all implementations.

Bibliography

[1] Carlo Acerbi and Balazs Szekely. Back-testing expected shortfall. *Risk*, 27(11):76–81, 2014.

[2] Leif Andersen, Jakob Sidenius, and Susanta Basu. All your hedges in one basket. *RISK-LONDON-RISK MAGAZINE LIMITED-*, 16(11):67–72, 2003. URL `http://www.ressources-actuarielles.net/EXT/ISFA/1226.nsf/d512ad5b22d73cc1c1257052003f1aed/bf571acf7dbca8cbc12577b4001e3664`.

[3] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(05):54–63, may 2016. ISSN 1558-0814. doi: 10.1109/MC.2016.127.

[4] January BCBS. Minimum capital requirements for market risk. 2016.

[5] Michele Bonollo, Luca Di Persio, and Luca Prezioso. The default risk charge approach to regulatory risk measurement processes. *Dependence Modeling*, 6(1):309–330, 2018.

[6] P Bratley and BL Fox. Implementing sobols quasirandom sequence generator (algorithm 659). *ACM Transactions on Mathematical Software*, 29(1):49–57, 2003.

[7] Xavier Burtschell, Jonathan Gregory, and Jean-Paul Laurent. A comparative analysis of cdo pricing models under the factor copula framework. *The Journal of Derivatives*, 16(4):9–37, 2009.

[8] Gemma Colldeforns-Papiol, Luis Ortiz-Gracia, and Cornelis W Oosterlee. Quantifying credit portfolio losses under multi-factor models. *International Journal of Computer Mathematics*, 96(11):2135–2156, 2019.

[9] Michel Crouhy, Dan Galai, and Robert Mark. A comparative analysis of current credit risk models. *Journal of Banking and Finance*, 24(1):59–117, 2000. ISSN 0378-4266. doi: https://doi.org/10.1016/S0378-4266(99)00053-9. URL `https://www.sciencedirect.com/science/article/pii/S0378426699000539`.

[10] Stewart DENHOLM, Hiroaki INOUE, Takashi TAKENAKA, Tobias BECKER, and Wayne LUK. Network-level fpga acceleration of low latency market data feed arbitration. *IEICE Transactions on Information and Systems*, E98.D(2):288–297, 2015. doi: 10.1587/transinf.2014RCP0011.

[11] Matthew F. Dixon, Thomas Bradley, Jike Chong, and Kurt Keutzer. Chapter 25 - monte carlo–based financial market value-at-risk estimation on gpus. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 337 – 353. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-385963-1. doi: https://doi.org/10.1016/B978-0-12-385963-1.00025-3. URL `http://www.sciencedirect.com/science/article/pii/B9780123859631000253`.

[12] FANG Fang and Kees Osterlee. The cos method: An efficient fourier method for pricing financial derivatives. *SIAM Journal of Scientific Computing*, 2008. doi: 10.1137/080718061. URL `https://ir.cwi.nl/pub/13283/13283A.pdf`.

[13] Anne Gelb and Sigal Gottlieb. The resolution of the gibbs phenomenon for fourier spectral methods. *Advances in The Gibbs Phenomenon. Sampling Publishing, Potsdam, New York*, 2007.

[14] Michael B Gordy. A risk-factor model foundation for ratings-based bank capital rules. *Journal of financial intermediation*, 12(3):199–232, 2003.

[15] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, April 2004. ISSN 0362-1340. doi: 10.1145/989393.989401. URL `https://doi.org/10.1145/989393.989401`.

[16] Peter Grundke. Computational aspects of integrated market and credit portfolio models. *OR Spectrum*, 29(2):259–294, 2007.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012. ISBN 978-0-12-383872-8.

[18] Xinzheng Huang and Cornelis Oosterlee. Saddlepoint approximations for expectations and an application to cdo pricing. *SIAM J. Financial Math.*, 2:692–714, 01 2011. doi: 10.1137/100784084.

[19] Ken Jackson, Alex Kreinin, and Xiaofang Ma. Loss distribution evaluation for synthetic cdos. Working Paper, 2007.

[20] Mark Suresh Joshi. *More mathematical finance*. Pilot Whale Press, 2011.

[21] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[22] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[23] Jean-Paul Laurent and Jon Gregory. Basket default swaps, cdos and factor copulas. 2005.

[24] Jean-Paul Laurent, Michael Sestier, and Stéphane Thomas. Trading book and credit risk: How fundamental is the basel review? *Journal of Banking & Finance*, 73:211–223, 2016.

[25] Pierre L'ecuyer, Richard Simard, E Jack Chen, and W David Kelton. An object-oriented random-number package with many long streams and substreams. *Operations research*, 50(6):1073–1075, 2002.

[26] David X Li. On default correlation: A copula function approach. *The Journal of Fixed Income*, 9(4):43–54, 2000.

[27] Alexander McNeil, Rüdiger Frey, and P Embrechts. *Quantitative Risk Management: Concepts, Techniques, and Tools*, volume 101. 10 2005.

[28] Robert C Merton. On the pricing of corporate debt: The risk structure of interest rates. *The Journal of finance*, 29(2):449–470, 1974.

[29] NVIDIA and J.P Morgan. Tesla case study: Gpus increase speed of risk computations while reducing costnvidia and j.p. morgan. 2011.

[30] Basle Committee on Banking Supervision. *Revisions to the Basel II Market Risk Framework: Updated as of 31 December 2010*. Bank for International Settlements, 2011.

[31] Michail Papadimitriou, Joris Cramwinckel, and Ana Lucia Varbanescu. Speed-up computational finance simulations with opencl on intel xeon phi. In Frédéric Desprez, Pierre-François Dutot, Christos Kaklamanis, Loris Marchal, Korbinian Molitorisz, Laura Ricci, Vittorio Scarano, Miguel A. Vega-Rodríguez, Ana Lucia Varbanescu, Sascha Hunold, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer, editors, *Euro-Par 2016: Parallel Processing Workshops*, pages 199–208, Cham, 2017. Springer International Publishing. ISBN 978-3-319-58943-5.

[32] Simon J. Rees and Joseph Walkenhorst. Chapter 24 - large-scale credit risk loss simulation. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 323 – 335. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-385963-1. doi: https://doi.org/10.1016/B978-0-12-385963-1.00024-1. URL http://www.sciencedirect.com/science/article/pii/B9780123859631000241.

[33] John Shalf. The future of computing beyond moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, 2020.

[34] Xiaoyu Shen, Fang Fang, and Chujun Qiu. Using the cos method to calculate risk measures and risk contributions in multifactor copula models. 2020.

[35] S. Solomon, R.K. Thulasirman, and P. Thulasiraman. Option pricing on the gpu. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 289–296, 2010. doi: 10.1109/HPCC.2010.54.

[36] BASLE COMMITEE ON BANKING SUPERVISION. Amendment to the capital accord to incorporate market risks. *Basle, Switzerland, jan*, 1996.

[37] Top500. Top500 list. Technical report.

[38] Anson H.T. Tse, David B. Thomas, K. H. Tsoi, and Wayne Luk. Efficient reconfigurable design for pricing asian options. *SIGARCH Comput. Archit. News*, 38(4):14–20, January 2011. ISSN 0163-5964. doi: 10.1145/1926367.1926371. URL https://doi-org.tudelft.idm.oclc.org/10.1145/1926367.1926371.

[39] Hervé Vandeven. Family of spectral filters for discontinuous problems. *Journal of Scientific Computing*, 6(2):159–192, 1991.

[40] Oldrich Alfons Vasicek. The distribution of loan portfolio value. *Citeseer*, 2002.

[41] Bowen Zhang and Cornelis W. Oosterlee. Acceleration of option pricing technique on graphics processing units. *Concurrency and Computation: Practice and Experience*, 26(9):1626–1639, 2014. doi: https://doi.org/10.1002/cpe.2825. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.2825.