IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

The Acceleration of multi-factor Merton model on FPGA

Author:
Pengyu Guo

Supervisor:
Luk Wayne

Submitted in partial fulfillment of the requirements for the MSc degree in MSc of Imperial College London

September 2020

Abstract

Credit risk stands for the risk of losses caused by unwanted events, such as the default of an obligor. The managing of portfolio credit risks is crucial for financial institutions. The multi-factor Merton model is one of the most widely used tools that modelling the credit risks for financial institutions. Typically, the implementation of the multi-factor Merton model involves Monte Carlo simulations which are time-consuming. This would significantly restrict its usability in daily credit risk measurement. In this report, we propose an FPGA architecture for credit-risk measurements in the multi-factor Merton models. The presented architecture uses a variety of optimisation techniques such as kernel vectorisation and loop unrolling, to optimise the performance of the FPGA implementation. The evaluation results show that compare to a basic C++ implementation running on a single core Intel i5-4210 CPU, our proposed FPGA implementation can achieve an acceleration of up to 22 times, with a precision loss of less than 10^{-8}.

Acknowledgments

I am extremely thankful for Prof Wayne Luk, for his academic guidance throughout the project. I am also extremely grateful to Dr Ce Guo, for his technical assistance, and also for the huge amount of confidence he placed in me to help me finish this project.

I would be also grateful to Prof Stephen Weston, for his devotion to his knowledge of the multi-factor Merton models.

I would also like to express my sincere thanks to my supportive parents, who give me this great opportunity to study at Imperial college London and for always being there for me.

Contents

Chapter 1

Introduction

Credit risk analysis and management have become one of the most important topics within a plethora of financial and lending institutions in recent years. Credit risk represents the risk of losses arising from some unexpected credit incidents such as the default of a counterparty, which often leads to incidents which have unwanted negative effects. When modelling credit risks, the key difficulties arise from the fact that default incidents are very unusual and that they always happen suddenly. However, when such incident occurs, it often leads to huge losses and disrupts the normal operations of financial institutions.

Concentration risks are referred to as a special type of credit risks, defined as the potential loss occurs from disproportionate loan distribution to single borrowers or regional sectors

Historical experience has already shown that the concentration risk in asset portfolios is one of the main reasons of bank distress. Furthermore, the huge amount of regulations imposed by supervisory authorities to regulate the concentration risks also illustrates the importance of diversifying loan portfolios for both nations and industries. Therefore, the measurement of potential concentration risks within a portfolio is very important.

In general, existing methods for measuring the concentration risks can be roughly divided into two groups [1]. The first group consists of asset-value models, in which the default of a corporation is modelled by the relationship between its assets and its liabilities at the end of a given period. Default risk in models of this type relies heavily on asset value's stochastic evolution, and default happens when the value of the variable that describes the asset value decreases below a specific value representing the liabilities threshold.

The second category is reduced-form models, which do not model the relationship between firm liabilities and firm asset value as what asset-value models did. In contrast, it models the process of defaults directly, instead of conducting a random evolution asset value process and model the default of a firm indirectly. In this type of models, a firm's probability of default is typically modeled as a distribution that depends on some economic covariables. However, compared to asset-value models, the reduced-form models are proved [2][3] to be a poor match for the observed real-world levels. Therefore, we select the asset-value models as the prototype of the concentration risk measurement method used in this work. Specifically, our study focuses mainly on one of the most famous and widely-used asset-value model - the multi-factor Merton model.

There are varieties of asset-Value models have already been developed by some prior researches, such as the KMV model[4] and the CreditMetrics model[5], both software-based and running on general CPUs. Therefore, these tools are typically serially-executed and lack parallel computing. This is especially troublesome in situations where the asset value process needs to be simulated for many scenarios. Because by doing so, it typically produces a high computational complexity due to the large dimensionality of counterparty information and therefore is time-consuming. Therefore, a high-performance implementation that focus on optimizing the execution speed of these models is needed to be developed to leverage the capabilities of asset-value models completely. Field Programmable Gate Array (FPGA)-based hardware methods has shown promising performance in terms of energy saving, and increasing the computing speed of the asset-value models [6].

To conclude, the aim of this project is to address the major challenge in concentration risk measurement: the time-consuming problem. The description of how we address such challenge is presented in Section 3. The evaluation of how well this challenge is met by our method is presented in Section 4.

Overall, in order to address the above challenge, this work proposes an FPGA-based acceleration approach to speed up the execution of the multi-factor Merton model so that the potential concentration risks of a portfolio can be measured more quickly.

To the best of the authors' knowledge, this is the first work that proposes FPGA implementation for the multi-factor Merton model. The purpose of this study is to make up the gap in this area and to develop an FPGA-based accelerator with high accuracy and fast processing speed.

The main contributions of this work are summarised as follows:

- We present a comprehensive set of basic manual optimisation methods for Intel FPGA SDK for OpenCL and discuss how each of them affects the performance of our FPGA implementation. By doing so, we hope to provide some insights to researchers who have similar studies as ours (Section 3.1 - 3.3);
- We quantitatively analysed the impact of using fixed-point and floating-point representation on the speed and accuracy of our FPGA implementation. In addition, we have also pointed out how to further improve the running speed of an FPGA implementation when a fixed-point representation is used (Section 3.4);
- We proposed an FPGA-based accelerator for the multi-factor Merton model to speed up the model execution. We show that compared to a basic C++ implementation running on a single core Intel i5-4210 CPU, the proposed FPGA implementation can achieve an acceleration of up to 22 times, with a precision loss of less than 10^{-8} (Section 3.5);
- We conducted a series of comprehensive evaluations on the accuracy and speed of our FPGA implementation. Besides, we have also compared our implementation with some previous similar designs. By doing so, we have identified some of the shortcomings of our implementation, and presented how can we improve our implementation in future work (Section 4);

Chapter 2

Background and related work

2.1 Background

Maintaining the stability of central counterparties(CCP) has become increasingly important in recent years due to the significant increase in central clearing. Among them, the management and modelling of the concentration risks are a core concern within the risk management processes at central counterparties as these risks can become systemic if a CCP is large enough. The concentration risk can be defined as the potential loss during the running of the CCP, because of an insufficient diversification of the CCP's collateral pool.

Recently, asset-value models obtained positive success in measuring the concentration risks. This type of models uses a geometric Brownian motion to simulate an obligor's asset value process. The default of the obligor in the model happens when its asset value at maturity time is lower than its liabilities.

The multi-factor Merton model [7] is one of the most popular asset-value models for concentration risk measurement. The multi-factor Merton model tries to interpret the asset of an obligor in terms of some potential economic variables so that these economic factors can explain some large portfolio losses. Besides, factor models' computational effort can be easily regulated by the number of economic variables included in the model. Being one of the asset-value models, in multi-factor Merton model, an obligor's default or survival is also made by comparing the obligor's asset value at maturity time to a certain default threshold.

Being one of the asset-value models, the multi-factor Merton model leads to a reliable risk analysis result. However, as mentioned in Section one, this kind of models are typically computationally intensive, which usually requires massive quantities of computational resources. Fortunately, these types of models usually have parallelism in many computational blocks and therefore it has a basis for parallelization by some hardware frameworks like FPGAs.

2.2 Related work

There is plenty of literature accelerating the multi-factor Merton model. Béresand and Bris [8] proposed a novel approach to speed up the execution of the multi-factor Merton model using importance sampling and GPUs. The accelerator uses a Gaussian mixture model to conduct importance sampling, and therefore avoid conducting time-consuming Monte Carlo simulations. They have found that the model run on an NVIDIA Kepler K20 accelerator is faster than run on the Intel Sandy Bridge E5-2470 CPU for a factor of 19 to 287, depending on the structure of the portfolio.

In a similar study, Zhang and Oosterlee [9] has proved that when the CPUs and GPUs are used concurrently to accelerate the model operation, different workloads partitions between CPU and GPU will affect the final acceleration effect. They have shown for European options that the highest speedup can be achieved if most of the basic mathematical operations, such as big metric multiplications, are performed directly on the GPU. By doing so, the bottleneck of limited bandwidth between CPU and GPU can be greatly avoided.

Besides, FPGAs based hardware acceleration is also often used to improve the running speed of asset-value models. Being one of the research group acting on FPGA accelerators, Tian and Benkrid [10] .implemented an FPGA-based option pricing accelerator on an asset-value model and achieves a speedup of 340 times over a software implementation running on an Intel Pentium IV CPU. Their later work that take use of Quasi Monte Carlo pricing strategy achieves 10 times speedup over an NVIDIA 8800GTX GPU and 500 times speedup over an Intel Xeon CPU[11].

A study given by Delivorias[12] also gives a similar result. They compared the acceleration effect of the Heston model simulation on both GPU and FPGA clusters. The

Heston model is also one of the typical asset-value models. The experiment result shows that: the FPGA version of the model run on Maxeler is 1.75 times faster than the GPU accelerator with 2x Tesla M2090.

In view of all that has been discussed above, it was finally decided to build an FPGA hardware design of the multi-factor Merton model in this project. The reasons why FPGAs is preferred over CPU and GPU are as follows. First of all, FPGAs presents a hardware implementation of the original algorithm whereas CPUs and GPUs accelerators are generally software-based. Due to its natural, the hardware is always faster than software. Furthermore, it has already been proved that using FPGA consumes less energy. A study given by Schryver et al. [13] shows that when doing a single-level Monte-Carlo simulation, a Tesla C2050 GPU achieves a 5.5x speedup by consuming 30% of the total energy. Whereas a Virtex-5 FPGA achieved the same throughput by only consuming 4% of the total energy.

2.3 Multi-factor Merton model

2.3.1 Model structure

As mentioned before, the multi-factor Merton model is used in this project to measure the concentration risk in credit portfolios. This section mainly introduces the structure of this model.

Consider a portfolio of N risky borrowers. Let $V_t^{(n)}$ to be the asset value of borrower n at time t (before the maturity time). Each borrower has a default threshold such that borrower n defaults if $V_t^{(n)}$ lowers than this threshold at maturity time. Therefore, in this model, $V_t^{(n)}$ can be regarded as a latent variable that driving the event of default.

Let r_n to be the n_{th} borrower's asset-value log returns, where T stands for the maturity time.

$$r_n = log(V_T^n / V_0^n) \tag{2.1}$$

r_n is assumed to depend linearly on K systematic risk factors $(X_1, X_2, ..., X_K)$ as well as on an idiosyncratic term ϵ_n. Both the systematic risk factors and the idiosyncratic term are standard normally distributed. In addition, ϵ_n is independent from the systematic risk factors X_K. Then, the asset value log-returns r_n can be rewrote in the following form:

$$r_n = \beta_n * Y_n + \sqrt{1-\beta_n^2} * \epsilon_n \tag{2.2}$$

Y_n denotes the borrower's composite factor, β_n measures the correlation between r_n and Y_n, ϵ_n denotes the idiosyncratic term.

Therefore, Y_n can be disintegrated into the K independent systematic factors we mentioned above:

$$Y_n = \sum_{k=1}^{K} \alpha_{n,k} * X_k \tag{2.3}$$

Where X_k represents the systematic risk factors $(X_1, X_2, ..., X_K)$ and $\alpha_{n,k}$ explains borrower n's dependence on a systematic factor X_k.

Since the composite risk factor Y_n and the idiosyncratic term ϵ_n are assumed to be independent, we can derive the following equation from (2.2):

$$V(r_n) = \beta_n^2 * V(Y_n) + (1-\beta_n^2) * V(\epsilon_n) \tag{2.4}$$

Then $\beta_n^2 * V(Y_n)$ captures borrower n's systematic risk, and $(1-\beta_n^2) * V(\epsilon_n)$ stands for the idiosyncratic term, which measures the other risk factors that cannot be explained by the systematic factors.

It is worth noting that, since X_k, r_n and ϵ_n are assumed to be standard normally distributed, then in order to make $V(Y_n) = 1$, the coefficient $\alpha_{n,k}$ in equation (2.3) must verify: $\sum_{k=1}^{K} \alpha_{n,k}^2 = 1$

With previous knowledge we can now measure the concentration risks of a portfolio. Let PD_n to be the one year default probability of borrower n: $PD_n = P(r_n < c_n)$. Since r_n are assumed to be standard normally distributed, c_n can be rewrite as:

$$c_n = \phi^{-1}(PD_n) \tag{2.5}$$

where ϕ is the cumulative distribution function.

If we plug equation (2.2) and (2.5) to the default condition $r_n < c_n$, we can obtain:

$$\phi_n < \frac{\phi^{-1}(PD_n) - \beta_n Y_n}{\sqrt{1-\beta_n^2}} \tag{2.6}$$

Thus, to conclude, the default probability of borrower n that conditional on the composite factor Y_n, is given by:

$$PD_n(Y_n) = \phi(\frac{\phi^{-1}(PD_n) - \beta_n Y_n}{\sqrt{1-\beta_n^2}}) \tag{2.7}$$

Then, the portfolio loss variable L, which measures the quality of the current credit portfolio, can be defined as follow.

$$L = \sum_{n=1}^{N} s_n * LGD_n * 1\{r_n < \phi^{-1}(PD_n)\} \tag{2.8}$$

Where s_n is the exposure share of borrower n: $s_n = \frac{EAD_n}{\sum_{n=1}^{N} EAD_n}$. EAD_n is the exposure of each borrower's loan. LGD_n determines the amount of loss when borrower n default. It is assumed that LGD are independent for different borrowers and for all other variables in the model. And $1\{.\}$ here denotes the indicator function.

2.3.2 Monte Carlo Simulation

Using equation (2.8), we can calculate the loss rate of a certain credit portfolio according to the given systematic risk factors. However, in order to more comprehensively assess the quality of this credit portfolio and detect the potential risks within it, we need to obtain the loss distribution of this credit portfolio. This can be done by conducting a Monte Carlo simulation based on different systematic risk factors.

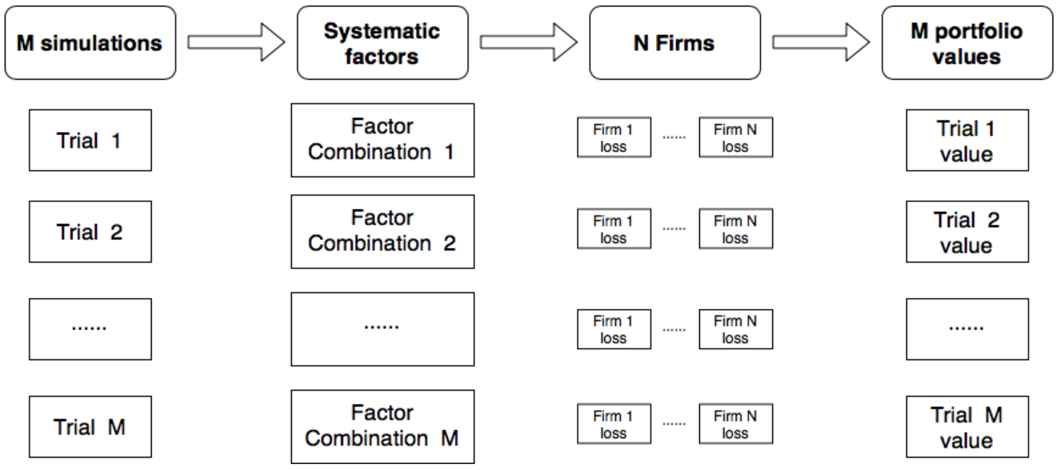


Figure 2.1: figure shows the procedure of the Monte Carlo Simulation

As shown in Figure 2.1, the Monte Carlo simulation involves M trials. A random set of systematic factors is generated in each trial, which is then used by firms to conduct the asset value process and decide whether the firm is default or not. Then, the portfolio loss value of each trial is calculated as:

$$L^i = \sum_{n=1}^{N} s_n * LGD_n^i * 1\{r_n < \phi^{-1}(PD_n)\}^i \tag{2.9}$$

The upper index i indicates the index of a particular Monte Carlo sample. It is worth noting that, we assume that portfolios are infinitely fine-grained so that the idiosyncratic risk term can be fully removed from the calculation.

Having the distribution of portfolio losses, the quality of a certain portfolio can then be measured using a metric called ”value at risk (VaR)”, which is often used by firms of the financial institutions to measure the risk of loss for investments. In our work, VaR is defined as the p quantile of $L = (L^1, L^2, ..., L^M)$.

Therefore, the M-trial Monte Carlo simulation approximates the value at risk of a certain portfolio as:

$$\begin{aligned} VaR(L) &= min\{L^i : \psi(L^i) \leq (1-p) * M\} \\ &= L^{[\lceil Mp \rceil]} \end{aligned} \tag{2.10}$$

Where $\psi(L^i) = \sum_{j=1}^{M}(L^j > L^i)$ and $L^{[\lceil x \rceil]}$ is the $x-th$ loss in the ascendant sorted loss sequence L.

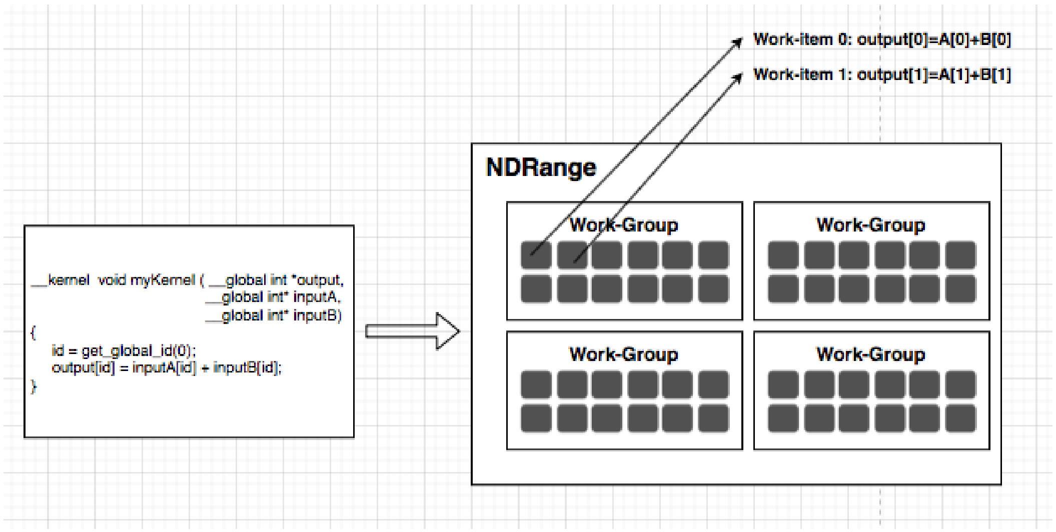
2.4 Intel FPGA SDK for OpenCL

This work uses Intel FPGA SDK for OpenCL to develop the FPGA implementation of the multi-factor Merton model. To make things easier for the reader to understand, before discussing the structure of our FPGA Accelerator, the internal structure of the Intel FPGA SDK for OpenCL, as well as its memory structure, is first introduced in this section.

2.4.1 OpenCL SDK programming model

In this section, we discuss how OpenCL programs are executed. OpenCL programs consist of one or more kernels, and each kernel represents a function of the OpenCL program code. The OpenCL SDK provides two different types of kernel: the NDRange kernel and the Single work-item kernel.

Consider a kernel that calculates the addition of two N-element Vectors. If the kernel is executed in NDRange mode, an index space is generated accordingly as shown in Figure 2.2.



```
__kernel  void myKernel ( __global int *output,
                          __global int* inputA,
                          __global int* inputB)
{
    id = get_global_id(0);
    output[id] = inputA[id] + inputB[id];
}
```

Work-item 0: output[0]=A[0]+B[0]
Work-item 1: output[1]=A[1]+B[1]
NDRange
Work-Group
Work-Group
Work-Group
Work-Group

Figure 2.2: NDRange kernel index space

Each point of this space represents a single work-item, and each work-item is responsible for executing the same piece of kernel code. In our example, each work-item is in charge of handling the addition of two elements. Besides, an unique ID called global ID is assigned to each work-item, such that different sets of data can be mapped to different work-items according to this global ID. Each work-group consists of a collection of work-items, and a collection of work-group forms an NDRange kernel. One of the biggest benefits of this design is that the work-items within the same work-group can share data through a fast on-chip local memory. Thus, there is no need to extract the same piece of data repeatedly while running different work items.

Imagine N = 128, that is, we are trying to add two vectors with length 128. Then there will be 128 work-items in total, each responsible for adding two single elements. Assume that the 128 work-items are divided into 8 work-groups, each consists of 16 work-items, and there are three available compute units. Then an NDRange kernel will execute as follow:

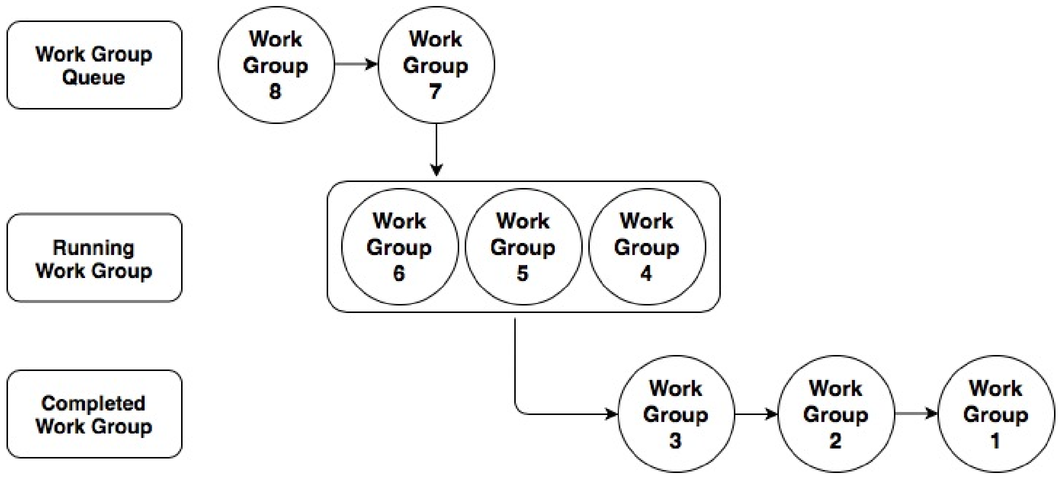


Figure 2.3: The execution of NDRange kernels

Since there are three compute units available, each time three work-groups are allowed to be processed in parallel. Moreover, the compiler also performs work-group level pipelining automatically. By doing so, it enables several work-groups to be executed concurrently in the same compute unit and the efficiency of all pipelines in the compute unit can hence be optimised.

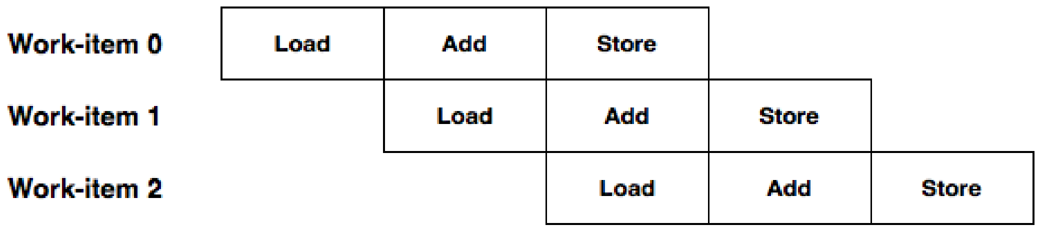


Figure 2.4: The execution of Single work-item kernels

The single work-item kernel works in a slightly different way - it has a single work-group that contains only one work-item. As shown in Figure 2.4, the offline compiler of OpenCL take use of a pipeline structure to speed up the computation for each kernel. A new work-item is inserted into the pipeline at each clock cycle, so that several work-items can be executed in parallel. By doing so, compare with replicating hardware for each work-item, the pipeline structure is more effective in terms of hardware resources as all work-items are executed in the same pipeline. Besides, the pipeline can make the memory bandwidth more efficient, as the input data of each work-item is loaded sequentially at each clock cycle. Since FPGAs' maximum external memory bandwidths are lower than that of GPUs, the pipeline technique is very important for FPGAs.

In addition to the different run time logic, another major difference between the two different kinds of kernel is the way they share data. The sharing of data between work-items in NDRange kernels is done by using a shared local memory as seen in Figure 2.5. Whereas single work-item kernels share data via a specific feedbacks channel as shown in Figure 2.6. This single work-item kernel's special technique of sharing data is a key factor in its ability to handle data dependencies between work items.

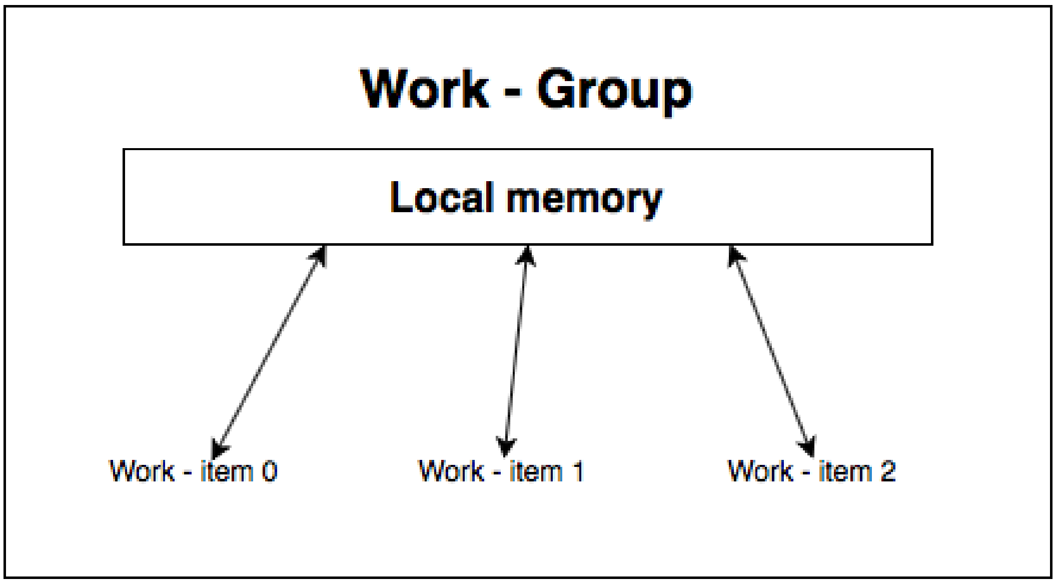
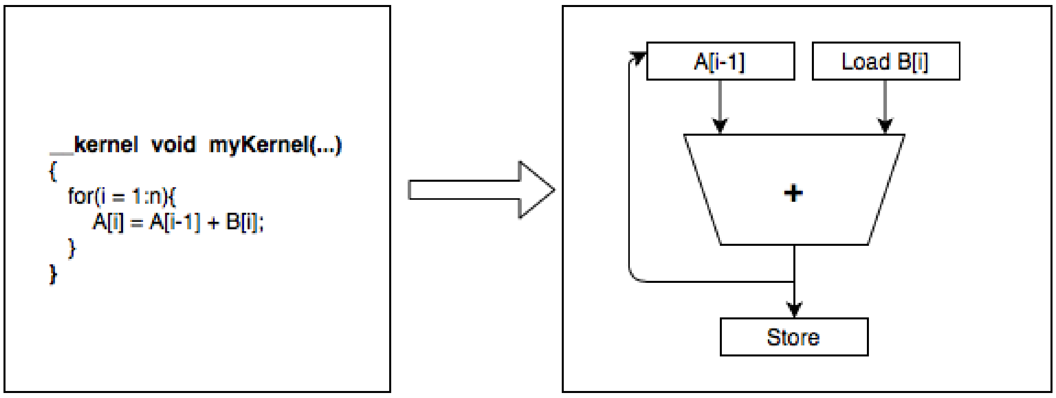


Figure 2.5: NDRange kernel's memory sharing strategy



```
__kernel void myKernel(...)
{
  for(i = 1:n){
    A[i] = A[i-1] + B[i];
  }
}
```

Figure 2.6: Single work-item kernel's memory sharing strategy

2.4.2 Memory structures

The OpenCL SDK supports four different memory types: private memory, local memory, global memory and constant memory, as shown in Figure 2.7.

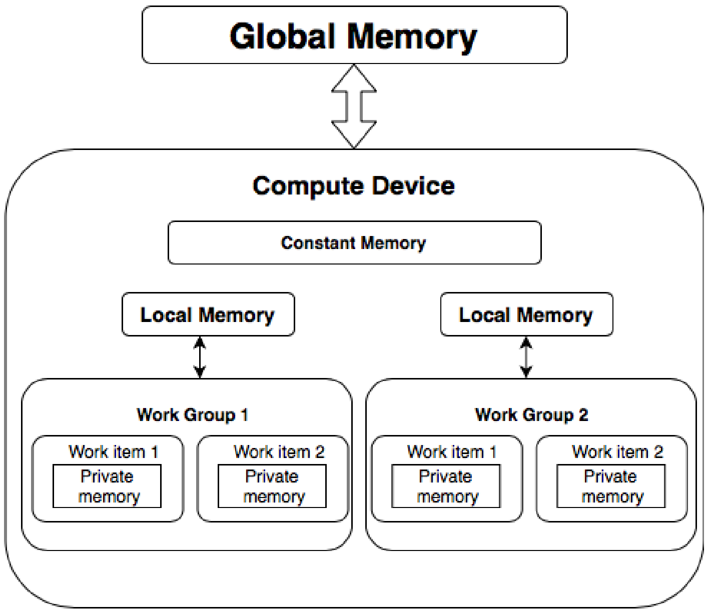


Figure 2.7: Memory structure of the Intel FPGA SDK for OpenCL

Global memory: The global memory is off-chip memory and has a large capacity. Although global memory storage has the highest access latency amongst the four memory types, it can still be very efficient. This is because the global memory bandwidth can be optimised by utilising LSU embedded caches which can provide coalescent memory access. Therefore, if there are repetitive and low-frequency global cache accesses, data is best to be stored in LSU caches. Compared with using global memory directly, a shorter access latency and a higher memory bandwidth can be achieved.

Constant memory: Constant memory locates in global memory, and therefore it is accessible to all work-groups. However, the compiler always loads it into a fast on-chip read-only cache at runtime. Therefore, by its nature, the access latency of the constant memory is much lower than that of the global memory. Thus, The constant memory is often used for data storage that require high-bandwidth memory access and is constant across several invocations of a kernel.

Local and private memory: Typically, due to the use of a small-size but high-performance local cache, local and private memory's access latency and memory bandwidth is far more better than that of the global memory. The major difference between private memory and local memory is their accessibility for different work-items. The local memory is visible to all work-items within the same work-group, whereas private variables are stored in private registers, and are accessible only to the work-item that own this private register.

Chapter 3

FPGA design and implementation

This chapter describes the implementation of our multi-factor Merton model in FPGA. In our work, we tried to use both NDRange mode and single work-item mode to implement our model and apply various methods to both modes to improve their performance. It is worth indicating that there is only one kernel function for each of the two implementations. Although distributing the model operation process to multiple kernels can significantly reduce the programming difficulty, the communication between different kernels has transmission consumption, which will affect the running speed. Therefore, we only keep one kernel function and let this kernel reproduce the asset value process of the multi-factor Merton model as we discussed in Section 2.3.1.

3.1 Basic C++ implementation

This section describes how ordinary C++ code reproduced the Monte Carlo Simulation of the multi-factor Merton model. First of all, suppose we have 1024 obligors and we need 32000 Monte Carlo simulation to measure the quality of a certain credit portfolio, and there are 10 systematic factors affecting each firm's asset value process. Then the input and output of the program will be as shown in Table 3.1:

Table 3.1: Data specification

Type	Variable name	Dimension	Description	Variables in the model
Input	S	1 * 1024	The exposure share of each obligor	s_n in equation 2.8
	LGD	1 * 1024	The loss given default of each obligor	LGD_n in equation 2.8
	R	32000 * 1024	The asset-value log returns of each obligor in each scenario	r_n in equation 2.8
	C	1 * 1024	The default threshold of each obligor	c_n in equation 2.5
	Beta	1 * 1024	The correlation between the asset-value of each obligor and its corresponding composite factors	β_n in equation 2.2
	X	32000 * 10	Systematic factors for each scenario	$\alpha_{n,k}$ in equation 2.3
	Alpha	10 * 1024	Each obligor's dependence on each systematic factors	X_K in equation 2.3
Output	Result	32000 * 1	The portfolio loss of each Monte Carlo simulation scenario	L in equation 2.8

In short, the basic C++ implementation performs the following:

Algorithm 1 Multi Factor Merton Model Monte Carlo Simulation

```
 1: for $i = 1$ to $numOfScenario$ do
 2:     portfolioLoss = 0.0
 3:     for $j = 1$ to $numOfOblg$ do
 4:         Extract the $S_j, LGD_j, C_j, \beta_j, \alpha_j$ from the input data.
 5:         Compute the composite risk factor: $Y_j = \sum_{k=1}^{K} \alpha_{n,k} X_k$
 6:         $ncdf = cdf((C_j - Y_j * \beta_j)/(sqrt(1 - \beta_j^2)))$
 7:         $portfolioLoss+ = S_j * LGD_j * (U_{i,j} < ncdf)$
 8:     end for
 9:     $Result[i] = portfolioLoss$
10: end for
```

The outer loop is responsible for conducting Monte Carlo simulation and recording the portfolio loss of each simulation. The inner loop iterate through each obligor and calculates their composite risk factors, then use this composite risk factor to conduct the evolution of their asset-value, and finally decide whether this obligor default or nor. This process can be described intuitively using the flow chart below.

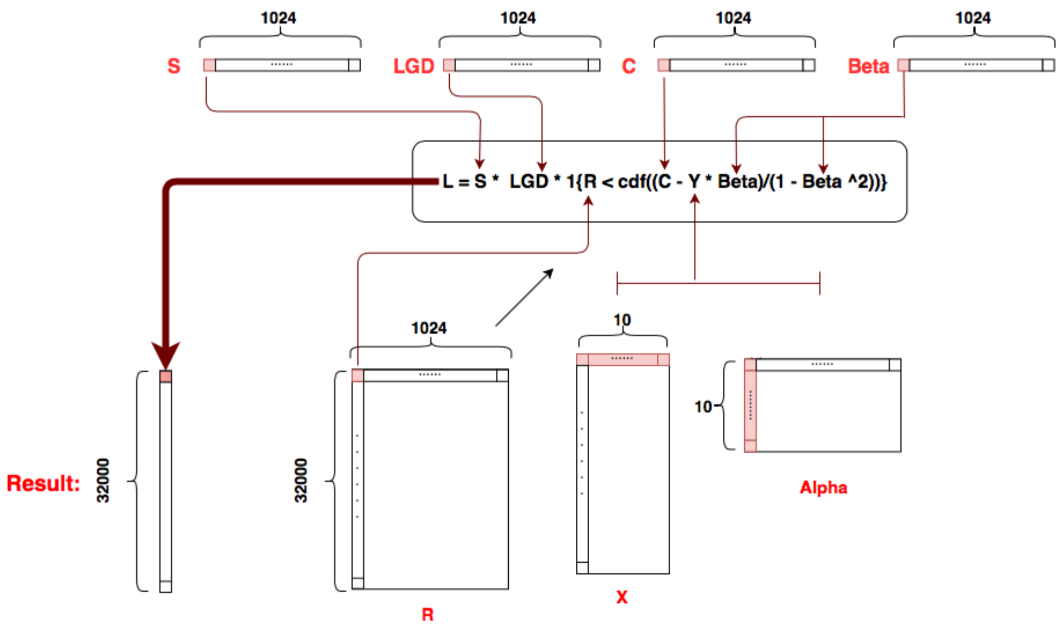


Figure 3.1: The computational spatial layout of the model

Moreover, as mentioned before, two different types of kernel are used to implement the FPGA version of the multi-factor Merton model. The main difference between them is the different data sharing strategy and the different division of work-groups/work-items. Since they have the same internal kernel logic , they can share the same set of hardware structure as presented in Figure 3.2.

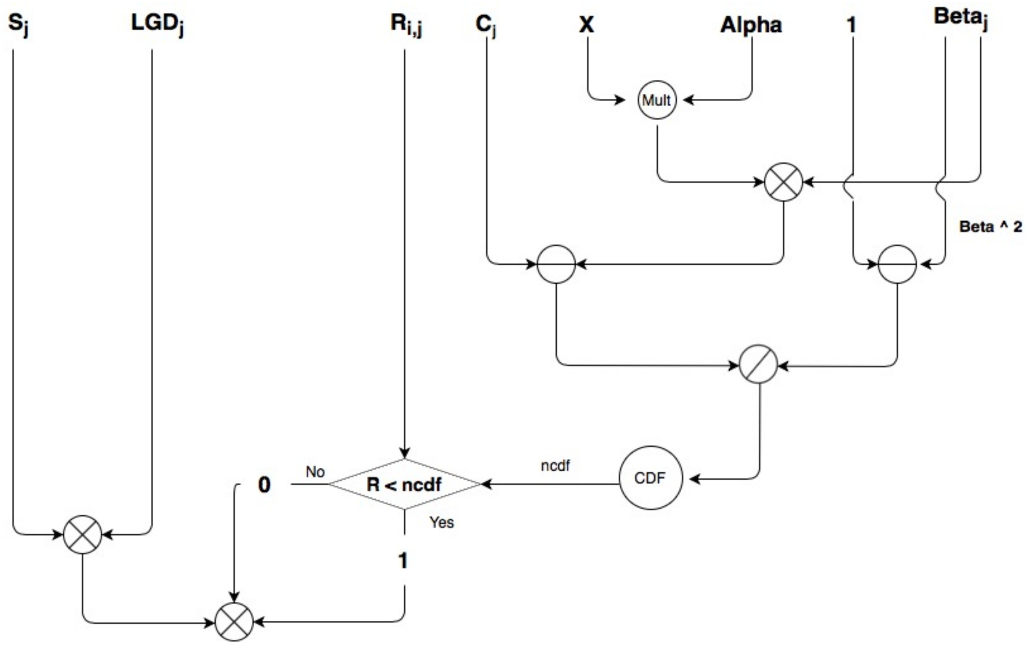


Figure 3.2: The hardware logic of both types of kernel

3.2 NDRange Mode

This section discusses our NDRange implementation of the multi-factor Merton model. The NDRange mode mainly benefits from the parallelization of individual work items. Therefore, programmers usually need to specify the thread-level parallelism mechanism explicitly. The following subsections present several different optimisation techniques under NDRange mode. These optimisation methods are evaluated separately in each subsection to analyse how each of them affects the performance of our implementation. The optimal combination of these optimisation techniques under NDRange mode is provided in Section 3.5. Besides, at the end of each subsection, an Arria 10 FPGA board was used to evaluate the performance of each optimisation technique, with a resource report attached. The execution time of the corresponding FPGA implementation was also compared to that of running on a single-core Intel Core i5-4210 CPU. A more comprehensive comparison is provided in Chapter 4 (i.e. compare the speedup effect of our FPGA implementation to the software implementation running on a 1/2/4/8 cores CPU).

3.2.1 Basic Implementation

This implementation is the normal NDRange mode kernel function of the multi-factor Merton model Monte Carlo Simulation with the references from the C++ implementation given in Algorithm 1. The NDRange kernel generates a deeply pipelined version of the kernel each time so that it can take advantage of the pipelining parallelism mechanism. Therefore, the outer *for* loop in Algorithm 1 is removed so that the C++ implementation can be parallelized. As shown in Algorithm 2, to index the specific work-item executed by the host, the OpenCL function $get\_global\_id()$ is used to determine the index of the Monte Carlo simulation for the kernel function. The rest of the code is identical to the C++ implementation.

Algorithm 2 multi-factor Merton model - NDRange Mode basic implementation

```
1: i = $get\_global\_id(0)$;
2: portfolio loss = 0.0
3: for $j = 1$ to $numOfOblg$ do
4:     Extract the $S_j, LGD_j, C_j, \beta_j, \alpha_j$ from the input data.
5:     Compute the composite risk factor: $Y_j = \sum_{k=1}^{K} \alpha_{n,k} X_k$
6:     $ncdf = cdf((C_j - Y_j * \beta_j)/(sqrt(1 - \beta_j^2)))$
7:     $portfolioLoss+ = S_j * LGD_j * (U_{i,j} < ncdf)$
8: end for
9: $Result[i] = portfolioLoss$
```

In the host side, after being initialised, the input data is then transferred to the FPGA side. Then the NDRange kernel is inserted into the command queue and starts to execute. Once all the threads running on the kernel finished their tasks, the kernel terminates and transfers the results back to the host side. Finally, the host side is responsible for the evaluation of the value at risk of this credit portfolio. An intuitive explanation of how the host and the FPGA collaborate is presented in Figure 3.3.

In the FPGA side, data transferred from the host side is first preloaded from global memory to constant memory. Then the kernel extracts data from the constant memory, uses the local memory to store and transform the runtime data, and write the results back to global memory in the end.

In our project, kernel computations are performed on constant memory mainly because it minimises the number of global memory access and reduces the memory latency through all work-groups and therefore improves the memory access efficiency. According to Zohouri et al.' s[14] study, due to its nature, constant caches are most appropriate for high-frequency data lookups that is constant across multiple invocations of a kernel. Besides, as suggested by the Intel Best Practice guide 2020[15], if there are any read-only data that is shared by all work-groups, then the data is better to be allocated in the constant memory. Our kernel precisely conforms to this case as according to the running process of the model we proposed in Figure 3.1, most of the input data, including S, LGD, C, etc., are all read-only data that shared by all work-groups.

To conclude, the basic implementation of the multi-factor Merton model in NDRange mode follows the procedure below, and its performance is as shown in Table 3.2.

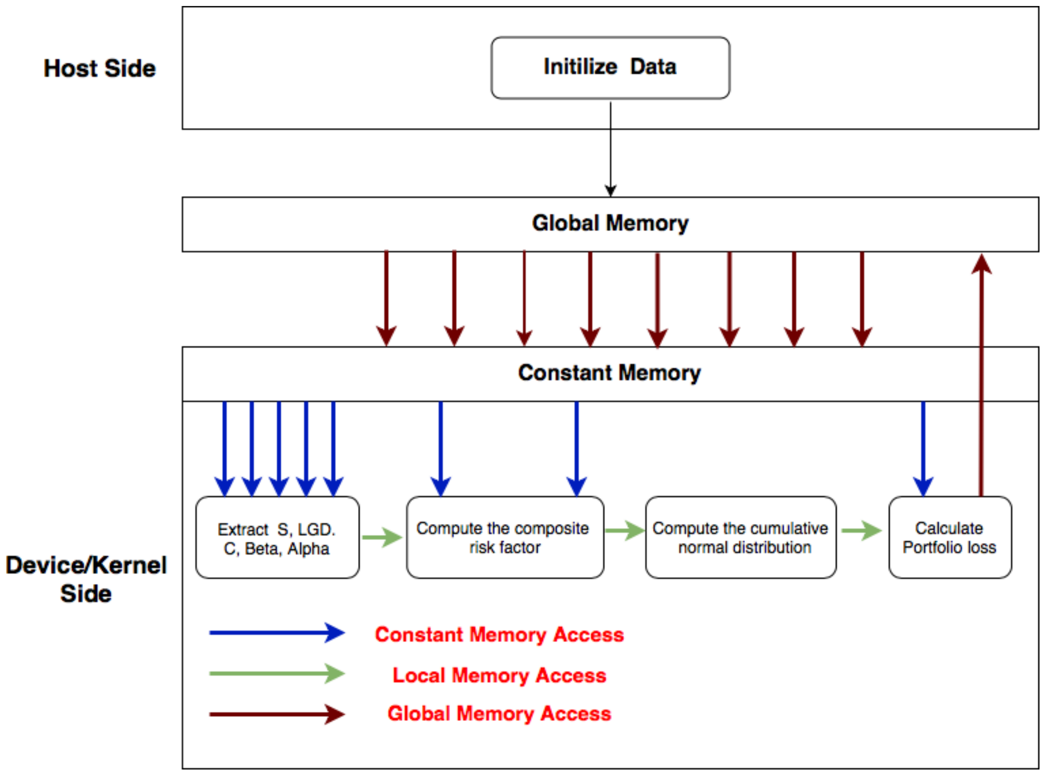


Figure 3.3: NDRange Basic implementation

Table 3.2: Resource utilisation - NDRange Basic Implementation

Parameters/Devices	FPGA	CPU
Time	12651 ms	16936 ms
Resource utilisation	Logic utilization = 20%	N/A
	ALUTs = 11% (91420/854400)	
	Dedicated logic registers = 10% (170876/1708800)	
	Memory blocks = 12% (322/2713)	
	DSP blocks = 4% (60/1518)	

3.2.2 Kernel vectorization implementation

The following figure shows the results of an Intel FPGA SDK for OpenCL benchmark test conducted by Jia et al. in [16]. This diagram describes the relationship between cache frequency/latency and the number of access ports while using the same constant cache.

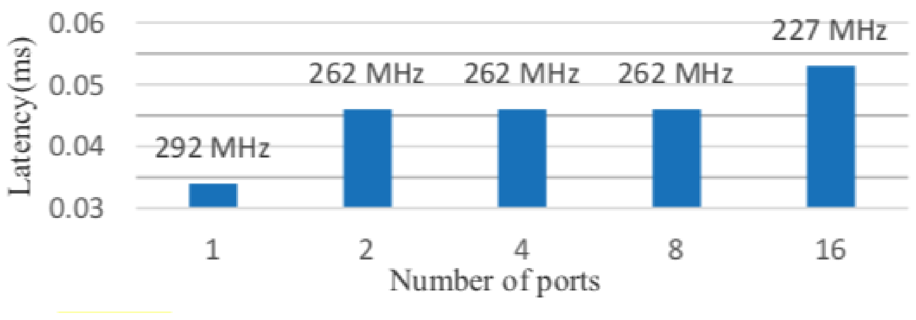


Figure 3.4: Constant cache latency and frequency with different number of ports (Obtained from "Tuning Stencil codes in OpenCL for FPGAs",2016)[16]

The latency of the constant cache accessed by a certain number of ports is described using the bars, and the numbers above each bar presents the corresponding frequency. It can be observed that the performance of the constant memory drops dramatically while the number of memory accessing requests is increased. Considering that in our implementation, the average number of ports access is 12 on average, the memory access speed of our NDRange implementation will be significantly affected. Therefore, we decided to use the technique of kernel vectorisation to alleviate this problem.

Kernel vectorisation allows work items to be executed in a single instruction multiple data (SIMD) fashion, and therefore it can be used to reduce the number of ports accessing constant memory, and consequently, the access latency can be reduced. The principle of kernel vectorization is as shown in Figure 3.5. Intuitively, this technique splits the work among all the work-items within the work-group. For instance, if a kernel is executing a work-group that contains 16 work-items, then a kernel vectorization with a factor of 2 will reduce the total number of work-items to 8. As a result, the first work item will take over the work of the first and the second work-item, and so on.

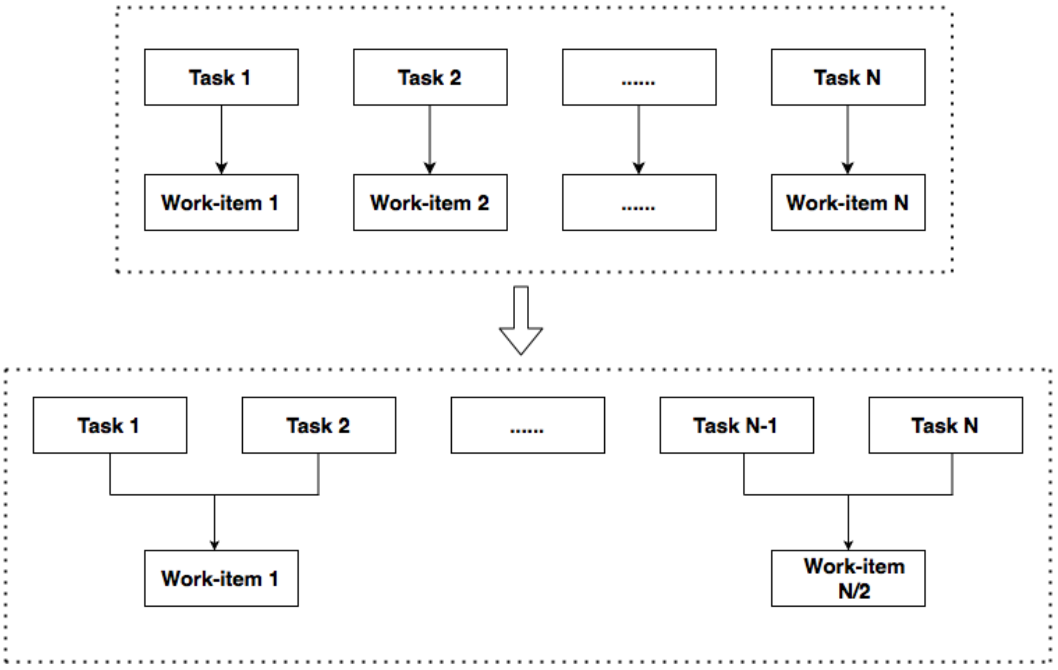
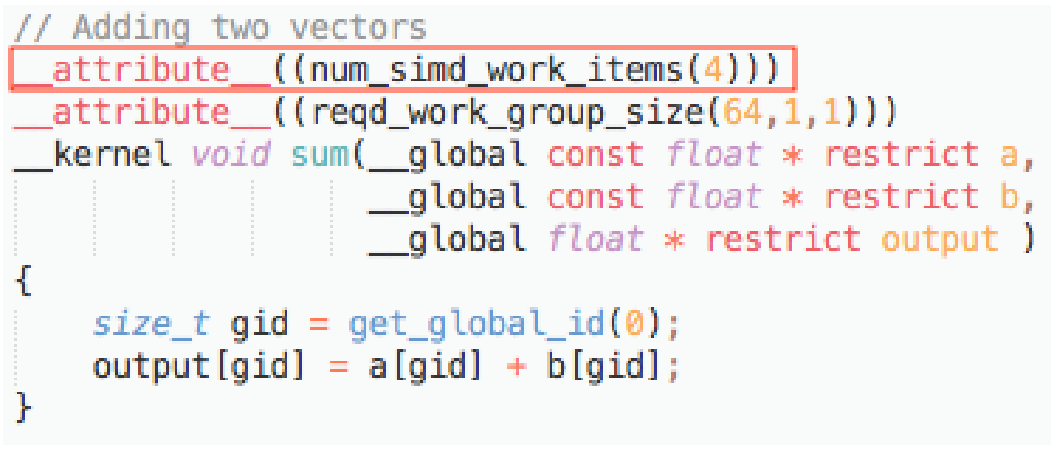


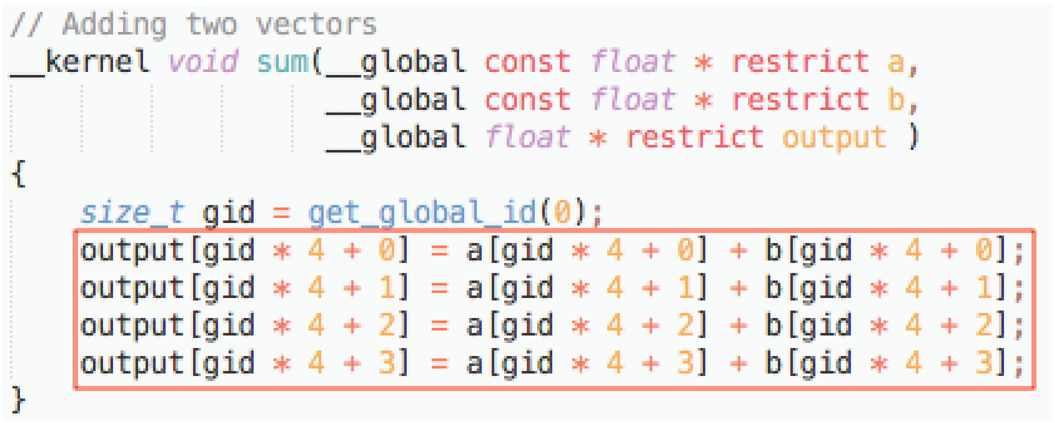
Figure 3.5: The key idea of kernel vectorization

The Intel FPGA SDK for OpenCL Offline Compiler contains a built-in library that translates mathematical operations in the kernel, such as addition and subtraction, to its corresponding SIMD operation. Therefore in our work, kernel vectorization is easily done by including the $num\_simd\_work\_items$ attribute in the kernel code. For example, in Figure 3.6 we apply a kernel vectorisation with a factor of four to the original kernel code that add two vectors. By setting the vectorization factor to 4, the compiler will be informed to vectorise the kernel four times to allow four work-items to be executed concurrently. That is, in each iteration, the kernel loads four elements from both arrays, instead of one element as before. The equivalent manually vectorisation implementation is shown in Figure 3.7. However, by doing so, we need to adjust the NDRange size to a quarter of what it used to be, because each work-item takes over four times as much work after the manual optimisation is implemented. Therefore, in order to decrease the programming difficulty, we use OpenCL's build in kernel vectorization command directly (as shown in Figure 3.6).



```
// Adding two vectors
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum(__global const float * restrict a,
                  __global const float * restrict b,
                  __global float * restrict output )
{
    size_t gid = get_global_id(0);
    output[gid] = a[gid] + b[gid];
}
```

Figure 3.6: Kernel Vectorization using OpenCL SDK



```
// Adding two vectors
__kernel void sum(__global const float * restrict a,
                  __global const float * restrict b,
                  __global float * restrict output )
{
    size_t gid = get_global_id(0);
    output[gid * 4 + 0] = a[gid * 4 + 0] + b[gid * 4 + 0];
    output[gid * 4 + 1] = a[gid * 4 + 1] + b[gid * 4 + 1];
    output[gid * 4 + 2] = a[gid * 4 + 2] + b[gid * 4 + 2];
    output[gid * 4 + 3] = a[gid * 4 + 3] + b[gid * 4 + 3];
}
```

Figure 3.7: Kernel Vectorization manually

To conclude, by using the Kernel Vectorization technique, the kernel can access multiple times as much data as before. This is indicated by the broad arrows in Figure 3.8.

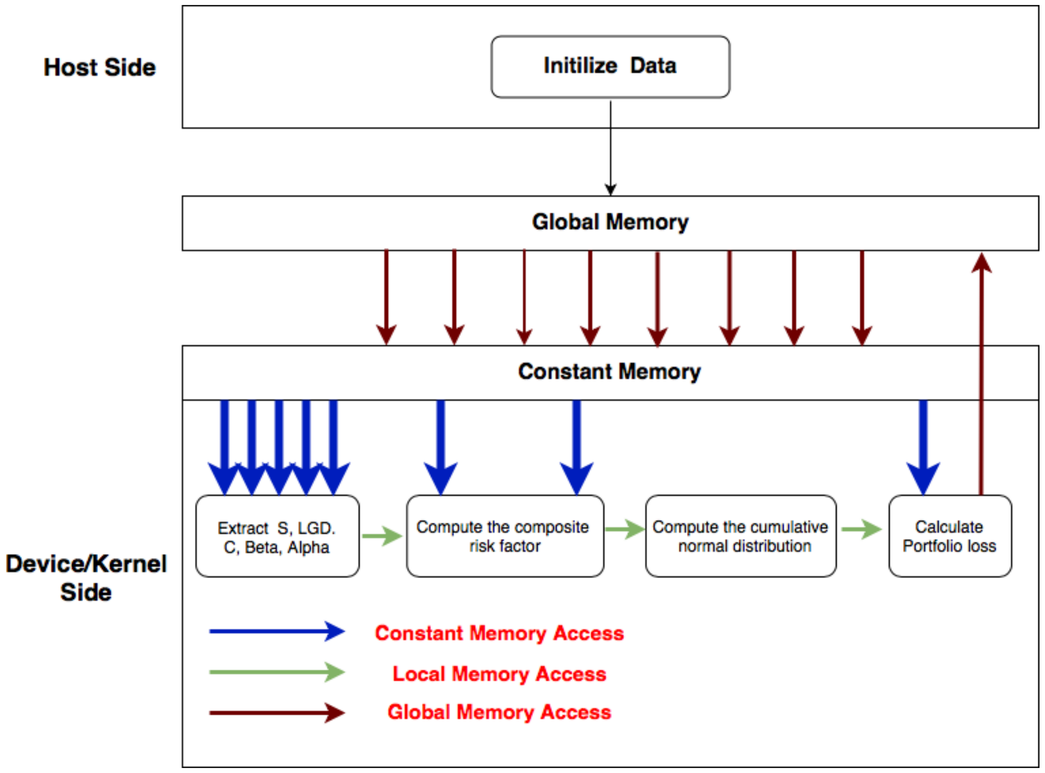


Figure 3.8: NDRange mode - Kernel Vectorization implementation

To execute multiple work-items concurrently, the compiler increase the hardware utilization to do so. In our Kernel Vectorization implementation, we have utilised a vectorisation factor of 16, and it can be seen that the resource utilisation has increased as shown in Table 3.3 when compared to the basic implementation's resource utilisation presented in Table 3.2. However, we could also observe that compare to the basic implementation of NDRange modes, vectorise the kernel helps to improve the performance by up to 200.9%. This indicates that our kernel vectorisation strategy is effective.

Table 3.3: Resource utilization - Kernel Vectorization Implementation

Parameters/Devices	FPGA	CPU
Time	6295 ms	16936 ms
Resource utilization	Logic utilization = 62%	N/A
	ALUTs = 39% (331507/854400)	
	Dedicated logic registers = 26% (444288/1708800)	
	Memory blocks = 44% (1191/2713)	
	DSP blocks = 45% (680/1518)	

3.2.3 Multiple compute units implementation

This section discusses the second optimisation technique under the NDRange mode - increase the number of compute units. In the NDRange mode, work-groups run on multiple compute units. Each compute unit is implemented as a unique pipeline, and can execute multiple work-groups at the same time. By default, work-groups are dispatched automatically by the FPGA hardware scheduler to available compute units. A compute unit is able to be assigned work-groups as long as it does not exceeds its maximum capacity. If there are two compute units available, then each compute unit is responsible for executing half of the work-groups. In case that each work-group takes the same amount of time to complete its tasks, then in theory, the total running time can be cut in half if there are enough hardware resources.

For example, if we decide to increase the number of compute units to 4, then the compiler will create four unique compute units as showed in Figure 3.9;

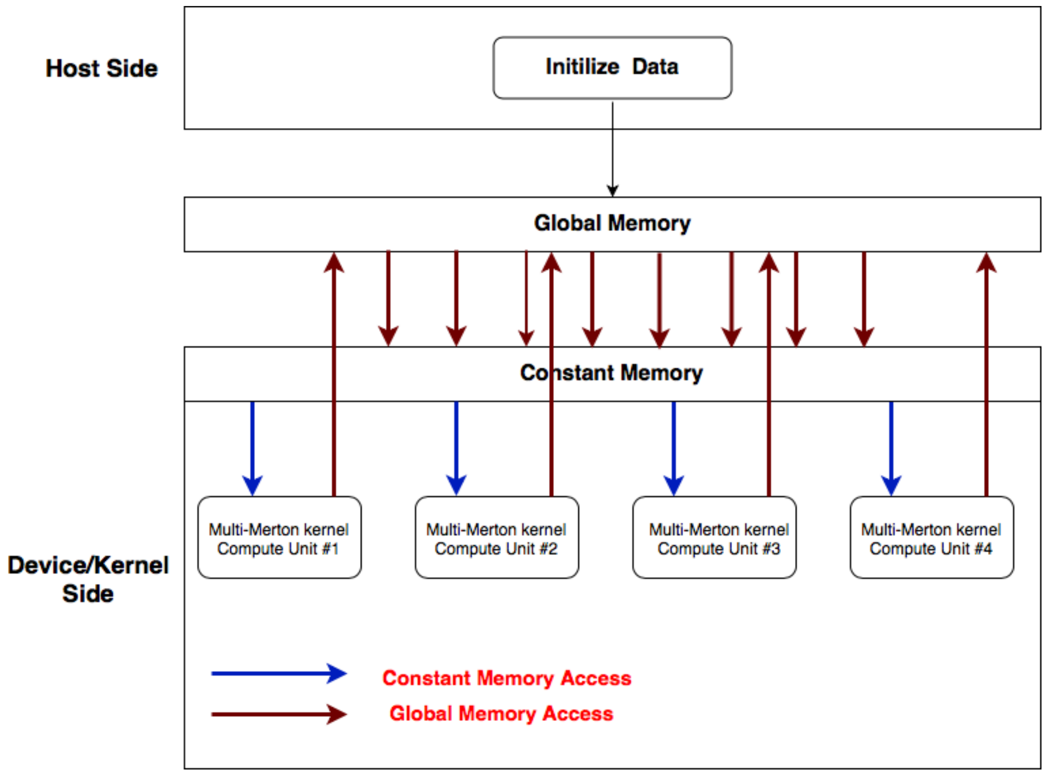
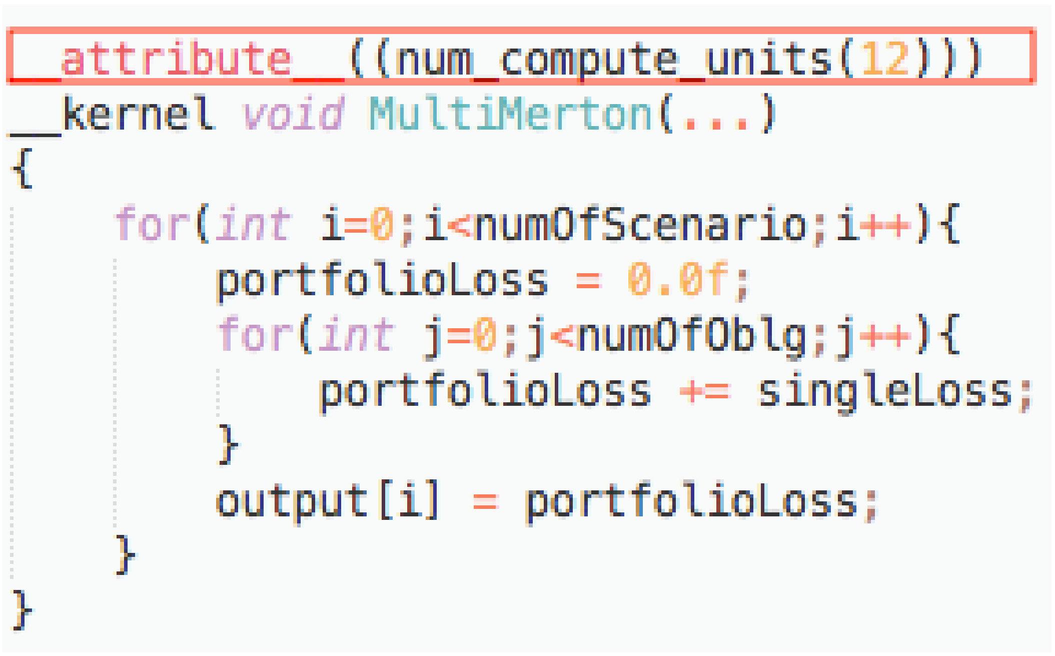

Host Side
Initilize Data
Global Memory
Constant Memory
Device/Kernel Side
Multi-Merton kernel Compute Unit #1
Multi-Merton kernel Compute Unit #2
Multi-Merton kernel Compute Unit #3
Multi-Merton kernel Compute Unit #4
Constant Memory Access
Global Memory Access

Figure 3.9: NDRange - Multiple Compute Units implementation

Therefore, we also tried to optimize our FPGA implementation by increasing the number of compute units per kernel to 12. In our work, this is done by specifying the number of compute units that the OpenCL FPGA compiler is expected to create using the *num_compute_units* attribute, as shown below. The corresponding performance is presented in Table 3.4.



```
__attribute__((num_compute_units(12)))
__kernel void MultiMerton(...)
{
    for(int i=0;i<numOfScenario;i++){
        portfolioLoss = 0.0f;
        for(int j=0;j<numOfOblg;j++){
            portfolioLoss += singleLoss;
        }
        output[i] = portfolioLoss;
    }
}
```

Figure 3.10: Pseudo code for multiple compute units implementation

However, through experiment we found that increase the number of compute units to improve the performance of our FPGA implementation leads to the following two problems.

- To increase the number of compute units, the compiler adds the necessary recourses to do so. It can be seen that compared with the basic implementation, the improvement of resource usage in Table 3.4 is similar to that of the kernel vectorization implementation in Table 3.3.

- A higher throughput can be achieved by increasing the number of compute units. However, as shown in figure 3.9, it do so at the expense of increasing memory bandwidth(4 loads and 4 store, instead of 1 load and 1 store of the kernel vectorization implementation).

Because of the above two reasons, from Table 3.4 we can find that when use 12 compute units to optimize the implementation, the kernel running speed was only improved a little bit compared to the basic NDRange implementation, and its acceleration effect is far from that of kernel vecterization.

Table 3.4: Resource utilization - 12 Compute Units Implementation

Parameters/Devices	FPGA	CPU
Time	11980 ms	16900 ms
Resource utilization	Logic utilization = 73%	N/A
	ALUTs = 43% (364828/854400)	
	Dedicated logic registers = 33% (562196/1708800)	
	Memory blocks = 63% (1710/2713)	
	DSP blocks = 47% (713/1518)	

3.2.4 Loop unrolling implementation

Since there are a large number of loops in our kernel code, we also tried the loop unrolling strategy to improve the performance of our loop Iterations. Loop unrolling is a loop transformation technique that helps to improve the running speed of a program. It works by replicating the body of a loop multiple times so that the number of iterations of a loop can be reduced. Intuitively, the aim is to flatten the loop structure and execute all iterations of the loop in one feed-forward path.

Loop unrolling optimizes the loop's performance in two aspects. First of all, it remove or reduce iterations. This increases the program's speed by eliminating the consumption of loop control instructions. Secondly, there exists a lot of loops that are not pipelined in the NDRange kernel. If the loops are not pipelined, then the optimal case will be that it can be completely unrolled so that the loop is removed. By doing so, for the offline compiler, the loop iterations is equivalent to be fully pipelined as there is no loop anymore. This can be explained more intuitively using the figure below, where the above one describes the running process without loop unrolling, while the bottom one shows the running process with fully loop unrolled.

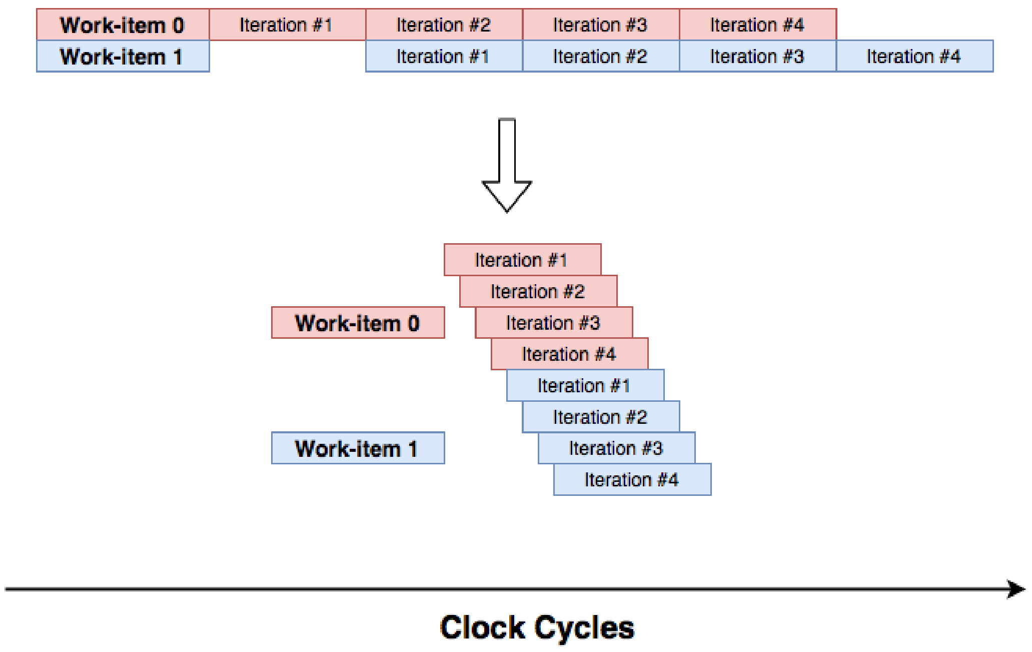
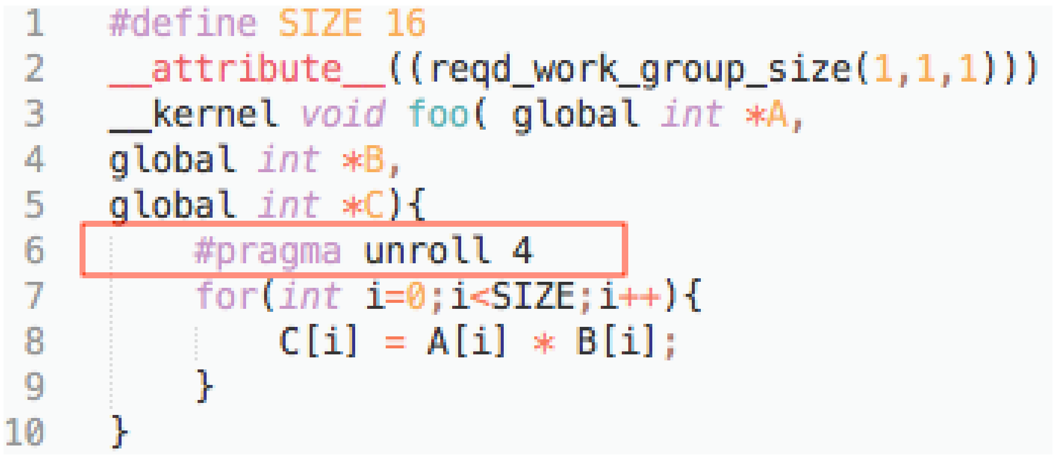


Figure 3.11: loop rolling vs loop unrolling

From the figure we can observe that, without loop unrolling, loops in a FPGA kernel is not well pipelined. Whereas unrolling the loop enables the operations inside a work item to be pipelined, this make the entire pipeline wider so that more tasks can be executed in parallel.

In our work, loop unrolling is done by adding a *#pragma unroll* attribute to the main loop, as shown in the code example below.



```
#define SIZE 16
__attribute__((reqd_work_group_size(1,1,1)))
__kernel void foo( global int *A,
global int *B,
global int *C){
    #pragma unroll 4
    for(int i=0;i<SIZE;i++){
        C[i] = A[i] * B[i];
    }
}
```

Figure 3.12: OpenCL SDK loop unroll 4 times

We implemented two different loop unrolling implementations that used two different combinations of the unrolling factor. Algorithm 3 shows our first loop unrolling implementation. The outer loop was unrolled with 4 times and the inner loop was fully unrolled. By doing so, from Table 3.5, it can be found that the algorithm has got a speed up of factor 8 at the expense of increased hardware utilization.

Algorithm 4 shows our second loop unrolling implementation, where the loop unroll factor of the outer loop was changed to 8. From Table 3.6, it can be observed that it consumes about 1.5 times the hardware utilization of the basic implementation to give us about the same improvement in performance which is very ineffective. Therefore, in our optimised implementation, the first loop unrolling factor combination is used.

Algorithm 3 multi-factor Merton model - Loop unrolling implementation 1

```
1:  i = $get\_global\_id(0)$;
2:  portfolioLoss = 0.0
3:  # pragama unroll 4
4:  **for** $j = 1$ to $numOfOblg$ **do**
5:      Extract the $S_j, LGD_j, C_j, \beta_j, \alpha_j$ from the input data.
6:      # pragama unroll
7:      Compute the composite risk factor: $Y_j = \sum_{k=1}^{K} \alpha_{n,k} X_k$
8:      $ncdf = cdf((C_j - Y_j * \beta_j)/(sqrt(1 - \beta_j^2)))$
9:      $portfolioLoss+ = S_j * LGD_j * (U_{i,j} < ncdf)$
10: **end for**
11: $Result[i] = portfolioLoss$
```

Table 3.5: Resource utilization - Loop Unrolling Implementation 1

Parameters/Devices	FPGA	CPU
Time	2445 ms	16936 ms
Resource utilization	Logic utilization = 35%	N/A
	ALUTs = 19% (162324/854400)	
	Dedicated logic registers = 18% (307583/1708800)	
	Memory blocks = 30% (811/2713)	
	DSP blocks = 14% (212/1518)	

Algorithm 4 Multi Factor Merton Model - Loop unrolling implementation 2

```
1: i = $get\_global\_id(0)$;
2: portfolioLoss = 0.0
3: # pragama unroll 8
4: for $j = 1$ to $numOfOblg$ do
5:     Extract the $S_j, LGD_j, C_j, \beta_j, \alpha_j$ from the input data.
6:     # pragama unroll
7:     Compute the composite risk factor: $Y_j = \sum_{k=1}^{K} \alpha_{n,k} X_k$
8:     $ncdf = cdf((C_j - Y_j * \beta_j)/(sqrt(1 - \beta_j^2)))$
9:     $portfolioLoss+ = S_j * LGD_j * (U_{i,j} < ncdf)$
10: end for
11: $Result[i] = portfolioLoss$
```

Table 3.6: Resource utilization - Loop Unrolling implementation 2

Parameters/Devices	FPGA	CPU
Time	2510 ms	16936 ms
Resource utilization	Logic utilization = 51%	N/A
	ALUTs = 28% (237523/854400)	
	Dedicated logic registers = 25% (427200/1708800)	
	Memory blocks = 46% (1245/2713)	
	DSP blocks = 27% (406/1518)	

3.3 Single work-item

The single work-item kernels is equivalent to NDRange kernels with only one work-group and each work-group contains only one work-item. Ordinary C/C++ programs is based on sequential models, which execute each element sequentialy, with no overlap between the execution of the elements, as described in Figure 3.13(left). Whereas in single work-item kernels, the compiler will infer pipelined execution across loop iterations and builds all the loops to iterate sequentially through the pipeline. That is, a work-item is inserted into the pipeline in each clock cycle so that multiple work-items can be executed in parallel to maximize the utilization of hardware resources.

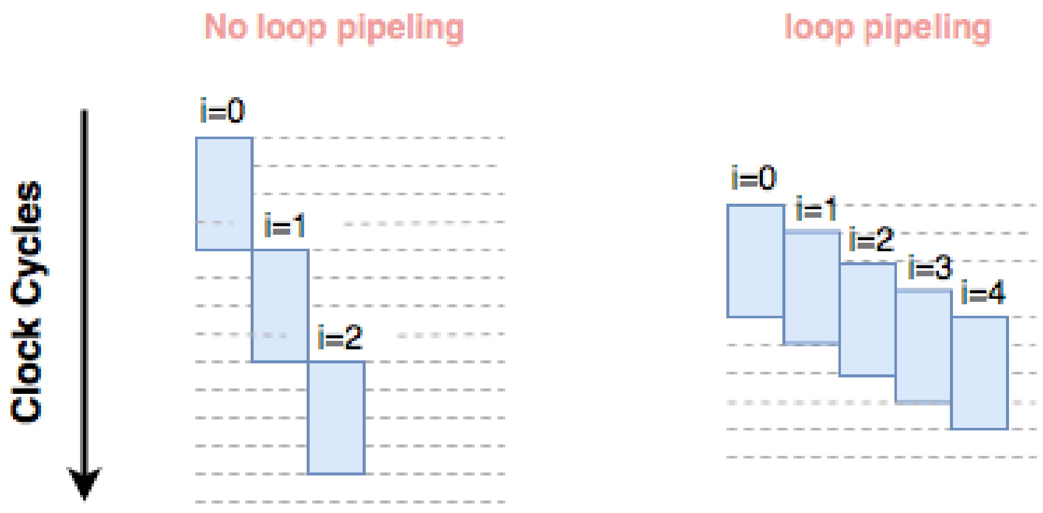


Figure 3.13: sequential mode vs loop pipeling

3.3.1 Single work-item basic implementation

The basic implementation of single work-item kernel is very similar to that of the original C++ implementation, as shown in the pseudo code below.

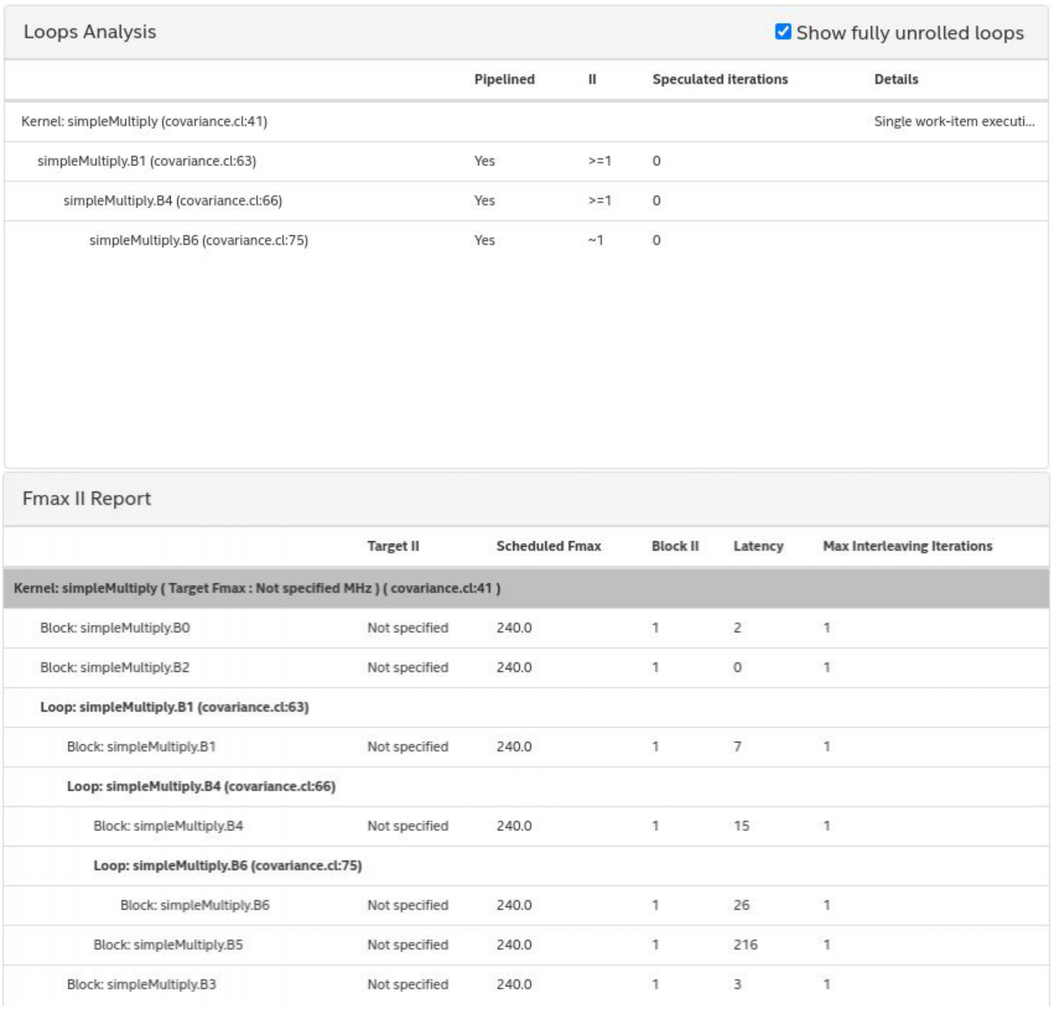
Algorithm 5 Single work-item mode - Basic implementation

```
1:  for $i = 1$ to $numOfScenario$ do
2:      portfolioLoss = 0.0
3:      for $j = 1$ to $numOfOblg$ do
4:          Extract the $S_j, LGD_j, C_j, \beta_j, \alpha_j$ from the input data.
5:          for $k = 1$ to $numOfFactor$ do
6:              $Y_j += X(i, j) * Alpha(j, k)$
7:          end for
8:          $ncdf = cdf((C_j - Y_j * \beta_j)/(sqrt(1 - \beta_j^2)))$
9:          $portfolioLoss += S_j * LGD_j * (U_{i,j} < ncdf)$
10:     end for
11:     $Result[i] = portfolioLoss$
12: end for
```

The basic implementation above has three sets of nested loops. The outer loop is responsible for doing Monte-Carlo simulations, the middle loop calculates each obigor's loss value, and the inner loop is responsible for calculating the composite risk factor of each obligor based on the systematic factors x and the dependence variable α.

Similar to our NDRange mode implementation, the basic implementation of single work-item kernel will also first transferred the data from the global memory to the constant memory. This is because the main difference between the single work-item kernel and the NDRange kernel is the different division of work-groups and work-items. As the data used by different work-groups in our model is identical, theoretically, single work-item kernels can adopt the same memory accessing method as NDRange Kernel models. By doing so, the memory access efficiency can be improved without affecting the accuracy of the running results.

As we said before, the single work-item mode mainly speed up the kernel through pipeline parallelization. Therefore, the OpenCL FPGA compiler always generates an optimization report on how well the single work-item kernel is pipelined, as shown in Figure 3.14.


Loops Analysis

☑ Show fully unrolled loops

	Pipelined	II	Speculated iterations	Details
Kernel: simpleMultiply (covariance.cl:41)				Single work-item executi...
simpleMultiply.B1 (covariance.cl:63)	Yes	>=1	0	
simpleMultiply.B4 (covariance.cl:66)	Yes	>=1	0	
simpleMultiply.B6 (covariance.cl:75)	Yes	~1	0	

Fmax II Report

	Target II	Scheduled Fmax	Block II	Latency	Max Interleaving Iterations
Kernel: simpleMultiply (Target Fmax : Not specified MHz) (covariance.cl:41)					
Block: simpleMultiply.B0	Not specified	240.0	1	2	1
Block: simpleMultiply.B2	Not specified	240.0	1	0	1
Loop: simpleMultiply.B1 (covariance.cl:63)					
Block: simpleMultiply.B1	Not specified	240.0	1	7	1
Loop: simpleMultiply.B4 (covariance.cl:66)					
Block: simpleMultiply.B4	Not specified	240.0	1	15	1
Loop: simpleMultiply.B6 (covariance.cl:75)					
Block: simpleMultiply.B6	Not specified	240.0	1	26	1
Block: simpleMultiply.B5	Not specified	240.0	1	216	1
Block: simpleMultiply.B3	Not specified	240.0	1	3	1

Figure 3.14: Single work-item kernel basic implementation performance report

The initiation interval (II) in the report refers to the launch frequency of a loop's new iteration. It describes the number of hardware clock cycles the pipeline must wait for before continuing with the next loop iteration. By referring the report, we can observe that the compiler successfully infers pipelined execution for the inner most loop and it shows an optimal loop performance as it has an II value of 1, which indicates that iterations were launched every cycle. The outer two loops were also pipelined successfully, but their iterations were launched every two cycles because they have a inner loop inside them. Overall, the optimization report shows that the kernel was well pipelined. From Table 3.7, we see that without any optimization methods, the basic implementation produced a 2x improvement in running speed compared to that of the C++ basic implementation on the CPU.

Table 3.7: Resource utilization - Single work-item basic implementation

Parameters/Devices	FPGA	CPU
Time	7392 ms	16936 ms
Resource utilization	Logic utilization = 21%	N/A
	ALUTs = 11% (93976/854400)	
	Dedicated logic registers = 10% (170879/1708800)	
	Memory blocks = 11% (279/2713)	
	DSP blocks = 4% (60/1518)	

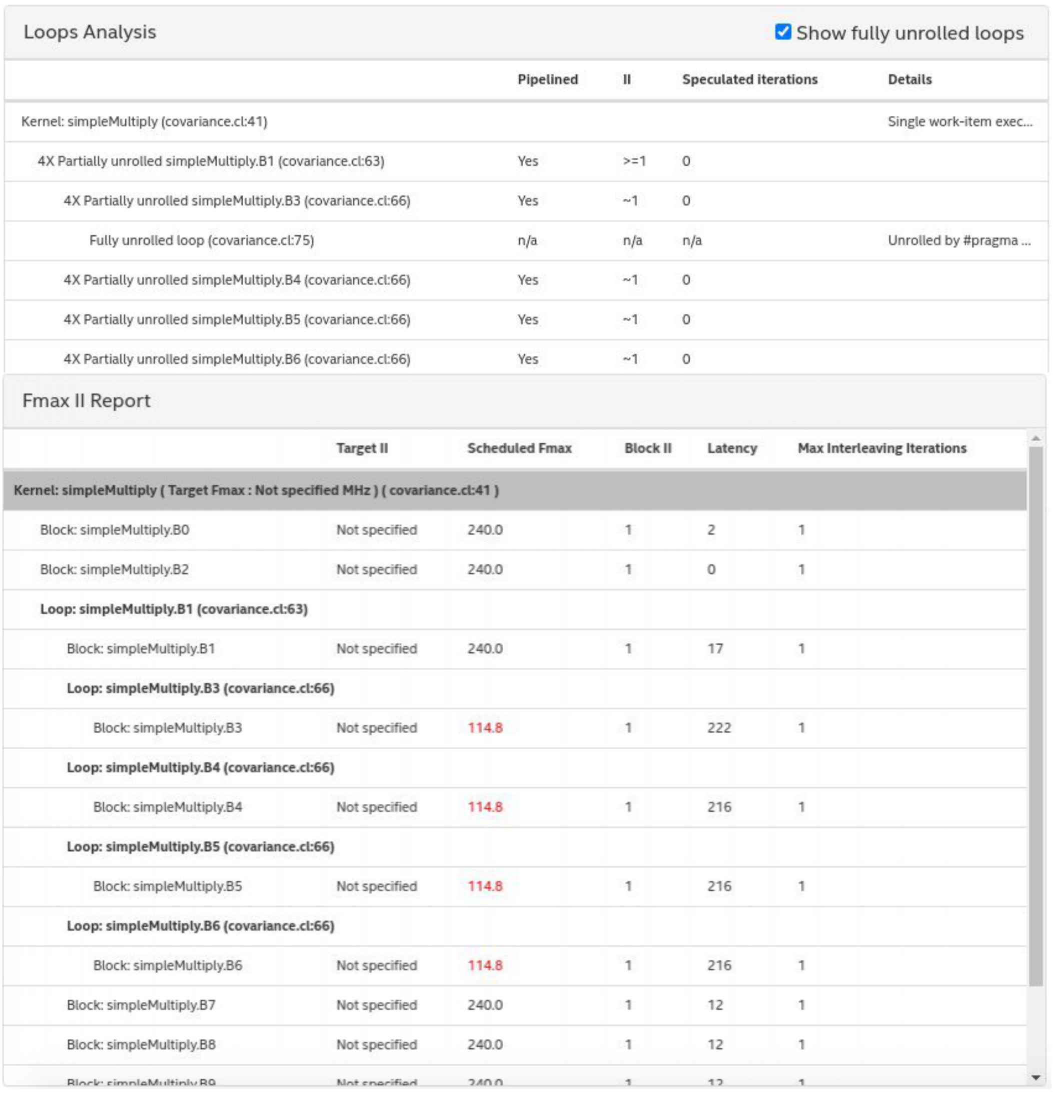
3.3.2 Loop unrolling implementation

Loop unrolling reduces the number of iterations of a loop at the expense of increasing hardware utilization. Since it is able to increase the throughput of the kernel and widen the pipeline, loop unrolling usually brings a speed up effect. Therefore, we have also tried to use loop unrolling to improve the performance of the single work-item kernel. In this implementation, as we can see from the code in Algorithm 6, we have unrolled the outer two loops with a factor of 4 and fully unrolled the inner most loop, like in our optimal loop unrolling implementation under NDRange mode.

Algorithm 6 Single work-item kernel loop unrolling implementation

```
1: #pragma unroll 4
2: for $i = 1$ to $numOfScenario$ do
3:     portfolioLoss = 0.0
4:     #pragma unroll 4
5:     for $j = 1$ to $numOfOblg$ do
6:         Extract the $S_j, LGD_j, C_j, \beta_j, \alpha_j$ from the input data.
7:         #pragma unroll
8:         for $k = 1$ to $numOfFactor$ do
9:             $Y_j + = X(i, j) * Alpha(j, k)$
10:        end for
11:        $ncdf = cdf((C_j - Y_j * \beta_j)/(sqrt(1 - \beta_j^2)))$
12:        $portfolioLoss + = S_j * LGD_j * (U_{i,j} < ncdf)$
13:    end for
14:    $Result[i] = portfolioLoss$
15: end for
```

From the optimization report in Figure 3.15, we can see that after using loop unrolling, both three loops are well pipelined, the initiation intervals are also within a reasonable range. This indicates that the use of the loop unrolling technique does not affect single work-item kernel's parallel pipelining structure. From Table 3.8, we can see that the performance has increased significantly benefits from the use of loop unrolling.


Loops Analysis

☑ Show fully unrolled loops

	Pipelined	II	Speculated iterations	Details
Kernel: simpleMultiply (covariance.cl:41)				Single work-item exec...
4X Partially unrolled simpleMultiply.B1 (covariance.cl:63)	Yes	>=1	0	
4X Partially unrolled simpleMultiply.B3 (covariance.cl:66)	Yes	~1	0	
Fully unrolled loop (covariance.cl:75)	n/a	n/a	n/a	Unrolled by #pragma ...
4X Partially unrolled simpleMultiply.B4 (covariance.cl:66)	Yes	~1	0	
4X Partially unrolled simpleMultiply.B5 (covariance.cl:66)	Yes	~1	0	
4X Partially unrolled simpleMultiply.B6 (covariance.cl:66)	Yes	~1	0	

Fmax II Report

	Target II	Scheduled Fmax	Block II	Latency	Max Interleaving Iterations
Kernel: simpleMultiply (Target Fmax : Not specified MHz) (covariance.cl:41)					
Block: simpleMultiply.B0	Not specified	240.0	1	2	1
Block: simpleMultiply.B2	Not specified	240.0	1	0	1
Loop: simpleMultiply.B1 (covariance.cl:63)					
Block: simpleMultiply.B1	Not specified	240.0	1	17	1
Loop: simpleMultiply.B3 (covariance.cl:66)					
Block: simpleMultiply.B3	Not specified	114.8	1	222	1
Loop: simpleMultiply.B4 (covariance.cl:66)					
Block: simpleMultiply.B4	Not specified	114.8	1	216	1
Loop: simpleMultiply.B5 (covariance.cl:66)					
Block: simpleMultiply.B5	Not specified	114.8	1	216	1
Loop: simpleMultiply.B6 (covariance.cl:66)					
Block: simpleMultiply.B6	Not specified	114.8	1	216	1
Block: simpleMultiply.B7	Not specified	240.0	1	12	1
Block: simpleMultiply.B8	Not specified	240.0	1	12	1
Block: simpleMultiply.B9	Not specified	240.0	1	12	1

Figure 3.15: Single work-item loop unrolling performance report

Table 3.8: Resource utilization - Single work-item loop unrolling implementation

Parameters/Devices	FPGA	CPU
Time	1592 ms	16936 ms
Resource utilization	Logic utilization = 85%	N/A
	ALUTs = 41% (347740/854400)	
	Dedicated logic registers = 31% (529714/1708800)	
	Memory blocks = 71% (1902/2713)	
	DSP blocks = 42% (637/1518)	

3.4 Fixed point vs Floating point

The input of the kernel that running the multi-factor Merton model consists of floating point numbers, which results in a huge consumption of hardware resources. However, in Intel FPGA SDK for OpenCL, fixed-point operations typically require less hardware resources than the equivalent floating-point operation. This indicates that we could save more hardware resources by using fixed-point data representations. Therefore, we have also tried to use fixed-point representation to improve the performance of our implementation

The fixed-point representation's bit length of the fractional parts and integer parts can be determined according to the sample data intensity range and a minimum fractional accuracy. The intensity range of the multi-factor Merton model's input data is quite straightforward to be determined. According to the requirements of the model, the input data of the model can generally be stored in 8-bit variables. However, the simulation procedure requires the input data to be summed, multiplied and scaled. Therefore, in order to avoid arithmetic overflow, it will be better to use a longer bit lengths. In order to figure out the most appropriate bit length of the fixed-point representation, an analysis of the input and intermediate data value intensities was performed. As a result, it was finally decided to adopt 17-bits fixed-point data representation: 4 bits for integer part and 13 bits for fractional part.

Since the Intel FPGA SDK for OpenCL does not support fixed-point representation, the conversion is done manually on the host side using the following equations:

$$fixed\_point\_input = floating\_point\_input * (1 << fractional\_bits) \tag{3.1}$$

$$floating\_point\_output = fixed\_point\_output/(1 << fractional\_bits) \tag{3.2}$$

The two tables below shows the influence of using fixed point on our optimised FPGA implementation (which is our best-performing implementations, this will be discussed in Section3.5). It can be observed that, as we expected, using fixed point number saves a lot of hardware resources, especially the resources of digital signal processing (DSP) blocks. Besides, it can be seen that compared to the floating point implementation, using fixed-point further increase the speed up effect for 15%.

This indicates that using fixed-point data is a efficient way to improve the running speed of our FPGA implementation.

Table 3.9: Resource utilization - NDRange kernel with fixed-point

Parameters/Devices	FPGA	CPU
Time	769 ms	16936 ms
Resource utilization	Logic utilization = 65%	N/A
	ALUTs = 40% (341760/854400)	
	Dedicated logic registers = 26% (444285/1708800)	
	Memory blocks = 46% (1245/2713)	
	DSP blocks = 44% (665/1518)	

Table 3.10: Resource utilization - NDRange kernel with floating-point

Parameters/Devices	FPGA	CPU
Time	884 ms	16936 ms
Resource utilization	Logic utilization = 67%	N/A
	ALUTs = 42% (358843/854400)	
	Dedicated logic registers = 29% (495352/1708800)	
	Memory blocks = 48% (1296/2713)	
	DSP blocks = 56% (850/1518)	

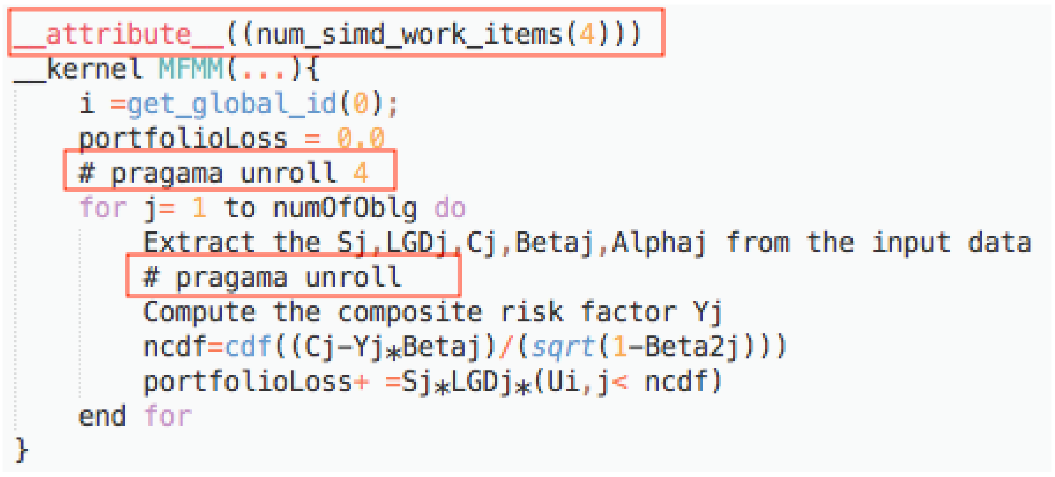
3.5 Optimised implementation

The performance of different optimisation techniques we discussed in previous sections is summarised in the table below.

Table 3.11: Performance comparison between different optimization techniques

Kernel	Implementation	Execution time	Speed up
NDRange	Basic implementation (fixed point)	12651ms	1.34x
	Kernel vectorization (fixed point)	6295ms	2.69x
	Multiple compute units (fixed point)	11980ms	1.41x
	Loop unrolling 1 (fixed point)	2445ms	6.93x
	Loop unrolling 2 (fixed point)	2510ms	6.75x
	Optimised implementation (fixed point)	769ms	22.02x
	Optimised implementation (floating point)	884ms	19.16x
Single work-item	Basic implementation (fixed point)	7392ms	2.29x
	Loop unrolling (fixed point)	1592ms	10.64x

Of all the optimization methods we have tried in the previous sections, including combinations of various optimization methods, experimentally we found that the method with the greatest improvement in speed was the NDRange kernel with loop unrolling and kernel vecterization, which is called optimised implementation in the table above. The pseudo code of of this implementation is given in the figure below.



```
__attribute__((num_simd_work_items(4)))
__kernel MFMM(...){
    i =get_global_id(0);
    portfolioLoss = 0.0
    # pragama unroll 4
    for j= 1 to numOfOblg do
        Extract the Sj,LGDj,Cj,Betaj,Alphaj from the input data
        # pragama unroll
        Compute the composite risk factor Yj
        ncdf=cdf((Cj-Yj*Betaj)/(sqrt(1-Beta2j)))
        portfolioLoss+ =Sj*LGDj*(Ui,j< ncdf)
    end for
}
```

Figure 3.16: Optimised FPGA implementation

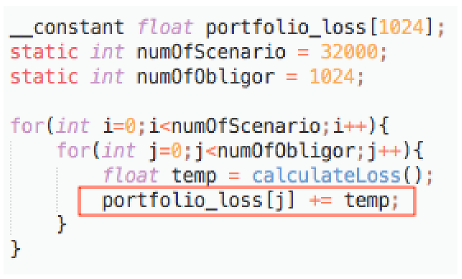
Within a single work-item kernel, there is a minimum distance of one clock cycle between two contiguous loop iterations. We can take advantage of this distance to transfer the data directly from the first loop iteration to the second iteration. Therefore, in single work-item kernels, it is possible to fully resolve iteration dependencies so that the initiation intervals can all be reduced to 1. However, as can be seen from Figure 3.15, the outermost loop in the code of single work-item kernel has an initiation interval that is greater than 1 because the loop dependencies inside it are not fully resolved. Therefore in our project , the single work-item kernel performed worse than the NDRange kernel because it did not fully exploit the power of pipeline execution.

Theoretically, If the single work-item kernel design is optimised with sufficient effort, it must overperform, or at least equal to the equivalent NDRange kernel design. However, within a limited time, we wish to cover more types of optimisations, such as algorithmic optimisations and precision optimisations. Although single work-item designs can lead to higher performance, they take a huge effort to optimise manually. Therefore in this project, we ended up with a good NDrange design and leaving the tedious single work-item kernel optimisation for future work.

Moreover, based on the optimised implementation of the NDRange kernel, we also used the following programming techniques to optimise the speed.

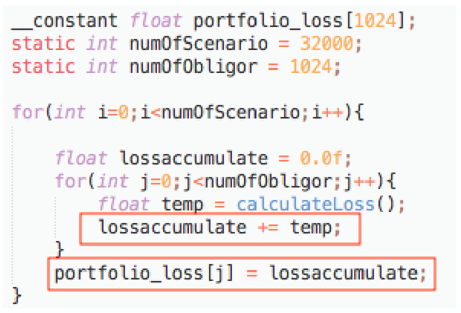
First of all, we avoid branches on global memory addresses. As suggested by Intel best practice guide 2020[15], for cases where in a kernel, the external memory address that is accessed from branches, instead of extracting data directly from the external memory as a criterion for branching, it is best if we pre-stored the results in temporary variables and use the temporary variable instead. By doing so, we can prevent the compiler from dynamic addressing at runtime and potentially allow coalesce memory accesses when loop unrolling is applied. Therefore, in the code of the FPGA side, we have moved all constant memory accesses out of branches and storing their value in some temporary variables. Then these temporary variables are used in the branch so to improve the performance of the model.

Secondly, we minimised the number of constant memory accesses. In our project, the performance of the implementation is improved by reducing accesses to the slower global memory and instead, storing all input data in the constant memory which has a faster accessing speed. As mentioned earlier, in NDRange Kernel, we can reduce the number of access ports of constant memory by using kernel vectorisation. In addition, we could also manually reduced the number of accesses to constant caches by using temporary registers.



```
__constant float portfolio_loss[1024];
static int numOfScenario = 32000;
static int numOfObligor = 1024;

for(int i=0;i<numOfScenario;i++){
    for(int j=0;j<numOfObligor;j++){
        float temp = calculateLoss();
        portfolio_loss[j] += temp;
    }
}
```

Figure 3.17: Unoptimised code

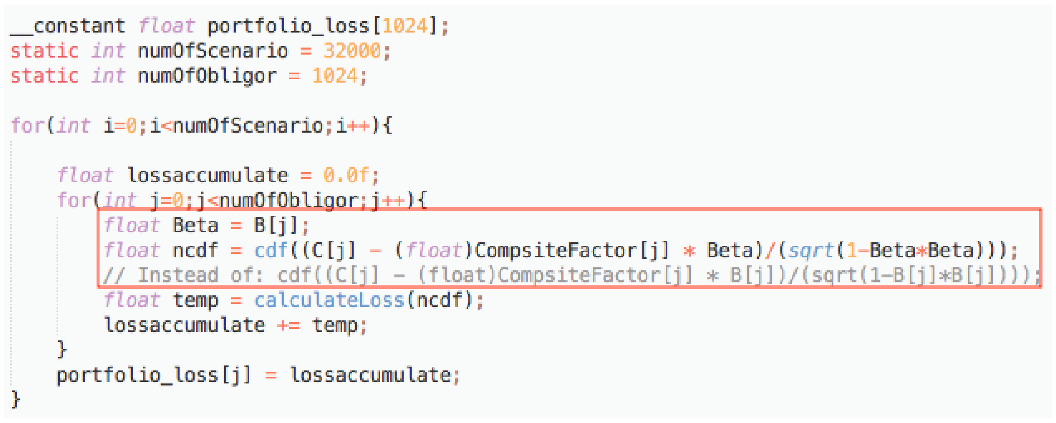


```
__constant float portfolio_loss[1024];
static int numOfScenario = 32000;
static int numOfObligor = 1024;

for(int i=0;i<numOfScenario;i++){

    float lossaccumulate = 0.0f;
    for(int j=0;j<numOfObligor;j++){
        float temp = calculateLoss();
        lossaccumulate += temp;
    }
    portfolio_loss[j] = lossaccumulate;
}
```

Figure 3.18: Optimised code

Figure 3.17 shows a code snippet in which a constant buffer is used to accumulate the loss of each obligor. As can be seen from the code, a write port is required to the constant buffer for each iteration. However, by moving the constant cache access outside of the loop and replacing it with a temporary variable as can be seen in Figure 3.18, the read port is removed, and therefore the overall performance can thus be improved.

Another case of reducing the number of constant buffers accessing is depicted in Figure 3.19. Since the program needs to access the same element in the same constant buffer repeatedly, in each iteration, the element is first extracted into a temporary register $Beta$, and then let the temporary register participate in the subsequent operations.



```
__constant float portfolio_loss[1024];
static int numOfScenario = 32000;
static int numOfObligor = 1024;

for(int i=0;i<numOfScenario;i++){

    float lossaccumulate = 0.0f;
    for(int j=0;j<numOfObligor;j++){
        float Beta = B[j];
        float ncdf = cdf((C[j] - (float)CompsiteFactor[j] * Beta)/(sqrt(1-Beta*Beta)));
        // Instead of: cdf((C[j] - (float)CompsiteFactor[j] * B[j])/(sqrt(1-B[j]*B[j])));
        float temp = calculateLoss(ncdf);
        lossaccumulate += temp;
    }
    portfolio_loss[j] = lossaccumulate;
}
```

Figure 3.19: Reducing the number of constant buffer accessing

Chapter 4

Evaluation

This chapter evaluates the optimised FPGA implementation for the multi-factor Merton model introduced in section 3.5 from three aspects: the accuracy of the result, the speed improvement and the comparison with other literature, which are carried out in Section 4.1, 4.2 and 4.3 respectively.

4.1 Accuracy evaluation

The accuracy of our FPGA implementation is evaluated from two aspects, carried out in section 4.1.1 and 4.1.2 respectively. For the FPGA implementation, we use the Intel(R) FPGA SDK for OpenCL(TM), version 19.4.0 build 64 Pro Edition. For the test data, 100 groups of data(1024 obligor, 32000 scenario, 10 systematic factors for each group) are generated randomly according to the requirement of the model data presented by Almqvist[17].

It is worth indicating that, the accuracy evaluation in section 4.1 is done by using the FPGA Emulator provided by Intel FPGA SDK for OpenCL, instead of a real FPGA Board. The FPGA emulator enables programmers to emulate the functionality of the kernel and figure out the problems of their design without executing it on an actual FPGA board. According to Intel best practice guide 2020[15], the running results generated by FPGA emulator are the same as those generated using an actual FPGA Board, so people usually use an emulator to verify the correctness of the program before trying to run on an FPGA Board. Therefore, considering that FPGA boards are expensive resources and the evaluation in this section only includes the measurement of the results correctness without any evaluation in terms of speed/energy consumption, we use the FPGA Emulator to evaluate our implementation.

It is also worth noting that, the data we generate already contains some extreme cases, that is, the data generated with some extreme distributions. By doing so, we want to demonstrate that our implementation would still work in extreme situations.

4.1.1 Result distribution analysis

The test set consists of 100 sets of randomly generated data that met the requirement of the model. As discussed in Section 2.3.1, each set of data is entered into the model, and a vector with a length of 32000 is generated accordingly. Each element of this vector represents the result of a Monte Carlo simulation run with this its corresponding input data, and therefore there are 32,000 scenarios in total .

In section 2.3.2, we have discussed that the quality of a certain portfolio is typically measured by the "value at risk" metric. Intuitively, in our model, the value at risk is measured by sorting the credit portfolio loss of 32,000 scenarios in ascending order and extracting the loss value at the corresponding confidence level. For example, if we want to measure the value at risk of our 32,000 simulations with confidence level P = 0.1, then we need to sort the results in ascending order and take the 3200_{th} data as a result.

In this section, we measured the value at risk of all results at the confidence level of P = 0.2, 0.4, 0.6 and 0.8, respectively. The corresponding box plot distribution is as shown in the figure below. The y-axis represents the value at risk of the result at a certain confidence level.

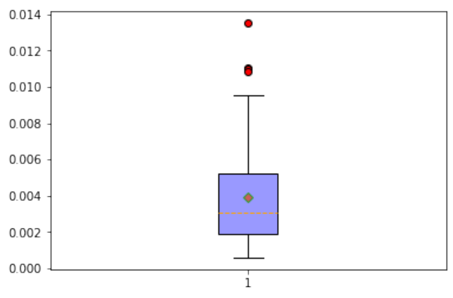


Figure 4.1: Loss distribution when p = 0.2

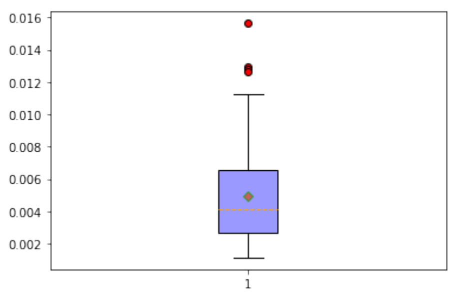


Figure 4.2: Loss distribution when p = 0.4

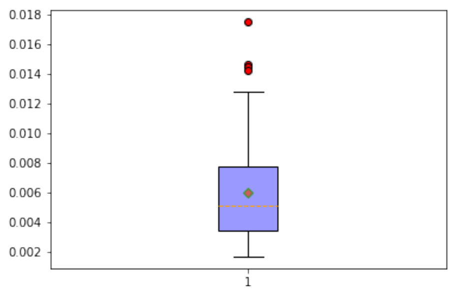


Figure 4.3: Loss distribution when p = 0.6

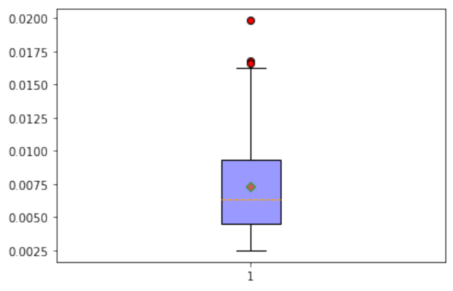


Figure 4.4: Loss distribution when p = 0.8

As can be seen from the figure, at the same confidence level, although there are a few outliers, most of the data is concentrated, and the fluctuation is generally less than 0.01. This indicates that the simulation converges reasonably well. This is promising because this is consistent with the real world: the distribution of the loss rate should not change too much, even if the overall environment change slightly. Because the quality of a credit portfolio should not be affected too much by slight changes of systematic factor values.

4.1.2 Result accuracy analysis

In this subsection, the results produced by our model are compared with the standard results to evaluate the accuracy of our model. As discussed in Section 3.4, our model uses two different types of data: single precision floating-point representation and fixed-point representation. Compared with the double-precision data representation used in the golden reference, single precision floating points and fixed points representation tend to be faster because they take up less space and provide wider memory bandwidth. However, this comes at the expense of precision losses.

Therefore, the test environment is set up to compare the results of our single precision floating-point and fixed-point implementation with the double-precision result generated by the standard Python code, which is considered to be the golden reference and available publicly to download. In this evaluation, We manually import the 100 sets of input data generated previously into the standard code and produce the corresponding 100 sets of standard output. The standard output is then compared with the results produced by our implementations. The following diagram quantitatively analyses the precision loss of the running results of all test data using different data representation. The x-axis represents different intervals precision loss, and the y-axis represents the number of test results falling within the corresponding precision loss interval. The precision loss of each test set was measured by the average loss of 32000 Monte Carlo simulations.

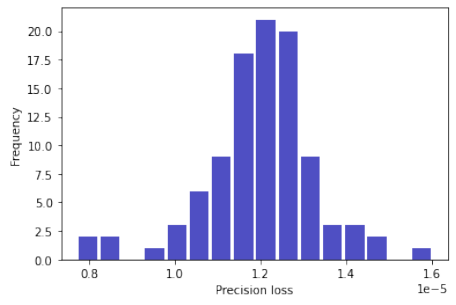


Figure 4.5: Fixed-point precision loss

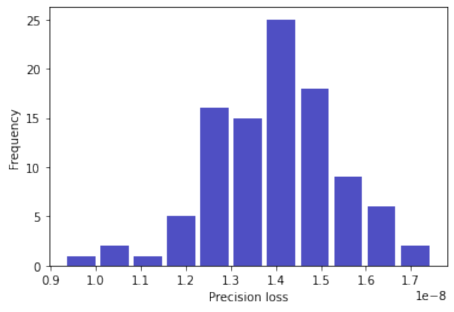


Figure 4.6: Floating-point precision loss

From the figure we can observe that, on average, the precision loss for the single precision floating-point result is on the order of 10^{-8}, while the error for the fixed-point result is on the order of 10^{-5}. The precision loss of both types of data representations did not fluctuate significantly between different test sets. Two conclusions can be drawn from the figure:

- The experimental results are in line with our expectations. Compared with single precision floating point, fixed-point representations have a more serious precision loss.

- According to Morgan's study [5], when measuring the credit risk of a credit portfolio, the precision loss should not exceed 10^{-8}. The result shows that the precision loss of our single precision floating-point result is within this error tolerance.

To conclude, based on the experimental results in sections 4.1.1 and 4.1.2, it can be proved our FPGA implementation can produce reasonable result. This is because the running results produce a reasonable distribution, and the precision loss of our implementation is within an acceptable range when using the single precision floating-point representation. Even though the precision loss of fixed-point representation exceeds the safe threshold, we can easily reduce the precision loss by increasing the bit length of the fixed-point representation, but this will increase hardware utilisation as well.

4.2 Performance Comparison Between CPU and FPGA Implementations

In this section, we compare the performance between CPU and FPGA implementations. The FPGA implementation is the optimised implementation we proposed in Section 3.5. The CPU implementation includes the C++implementation that we implemented ourselves and the Python implementation that is regarded as the golden reference. Same as before, for test data, the number of scenarios, the number of obligors and the number of systematic factors are determined as 32000, 1024, 10, respectively. For CPU based processing, we used Intel i5 4210 single/double/4/8 core CPUs to execute our C++ implementation. It is worth noting that the host side of our FPGA implementation uses the same CPU with a single core. For FPGA-based processing, we use the Arria 10 devices that built on TSMC's 20 nm process technology. For the FPGA implementation, we use the Intel FPGA SDK for OpenCL, Version 19.4.0.64.

The evaluation metric used in this section is the processing time, which is the time for executing the multi-factor Merton model at one time step. In the FPGA implementation, the processing time includes the time of data transfer between the host side and FPGA side. The figure below presents the processing time of 10 different executions of our optimised FPGA implementation.

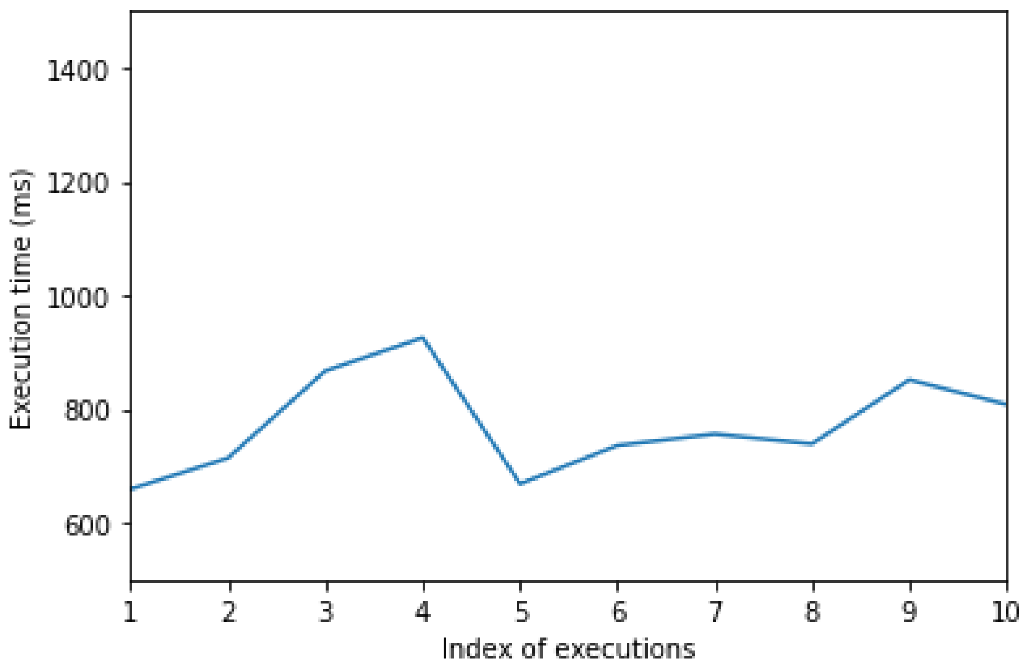


Figure 4.7: Processing time of our optimal FPGA implementation

As can be seen from the figure, the processing time of different test cases is roughly the same, and there are no extreme cases. This indicates that the performance of our implementation is stable. The table below gives a comparison of the processing time of different implementations. The result is measured as the average execution time of 10 executions for FPGA implementation, and 100 executions for Python and C++ implementation. The table also provides the speedup effect of all implementations relative to the one core C++ implementation.

Table 4.1: The comparison of the performance between software implementation and FPGA implementation

Device	Implementation	Processing time	Speed Up
FPGA	17 bits fixed-point	769 ms	22
	Single precision floating-point	884 ms	19.2
CPU	Python 1 core	43069 ms	-
	C++ 1 core	16936 ms	1.0
	C++ 2 cores	8913 ms	1.9
	C++ 4 cores	4838 ms	3.5
	C++ 8 cores	2454 ms	6.8

From the table we can observe that, the golden reference that programmed in Python is significantly slower than our equivalent C++ implementation. This is due to the nature of the programming language, which is exactly what we would expect. By observing the processing time of our C++ implementation, we can find that with the increasing number of CPU cores, the running speed has been significantly improved. However, there was a counter-intuitive phenomenon: the speed up effect and the number of CPU cores are not increased by the same proportion. For example, the execution time is not eight-fold for an 8-cores CPU. The reason is that when there are multiple processors running at the same time, the consumption of managing parallel tasks slows down the model execution.

On the other hand, the proposed FPGA hardware implementation performs well. Its fixed-point and floating-point versions are 22 and 19 times faster than the basic one core C++ implementation, respectively. Compare with the 8 cores C++ implementation, which is the fastest software implementation we have, the two data types of the proposed FPGA implementation still achieves an acceleration of 3.2 and 2.8 times, respectively.

Moreover, from the table we can also find that compared with using single precision floating-point representations, using fixed-point representations can improve the running speed of the model. In our experiment, when using the same hardware implementation, using fix-point brings us a speedup effect of 1.15 times. Of course, as mentioned in
the previous section, this leads to an increase of the precision loss, from 10^{-8} to 10^{-5}.

4.3 Result analyze

In this part, our work is compared to some prior designs. To the best of our knowledge, no FPGA implementation based on the multi-factor Merton model has been presented before, so we are not able to find literature that matches with our research. Instead, three representative, similar studies were selected to compare with our implementation.

4.3.1 Compare with Intel FPGA implementation of single factor Merton model

Although no one has researched on the FPGA implementation of the multi-factor Merton model before, a few companies such as Intel [18] are offering FPGA-based accelerators for computing the Black-Scholes Merton model and Monte-Carlo simulation for pricing options. The Black-Scholes Merton model is a simplified version of the multi-factor Merton model, and the two models are identical in many aspects.

Similar to our implementation, Intel also uses Monte Carlo simulations to simulate many possible paths of asset value process, and then derive an expected value for the payoff. In addition to the different structure of the model, the major difference between our implementation and Intel's is the way the data is transmitted. In Intel's FPGA Implementation of the Merton model, the data required for model calculation is generated on the FPGA side by Mersenne Twister Random Number Generator, and these data are then transmitted to the kernel to conduct model calculation. In our implementation, however, data is generated on the host side and then passed to the FPGA side. This puts a lot of pressure on memory bandwidth compared to Intel's approach and increases the program running time due to the data transmitting between CPUs and FPGAs. As a result, Intel's implementation achieves the following performance.

Table 4.2: The performance of Intel OpenCL single factor Merton model FPGA implementation

Device	Implementation	Speed Up
FPGA	18 bits fixed-point	146
	Single precision floating-point	41
CPU	C++ 1 core	1

Compared to the performance of the optimised implementation presented in Table 4.1, we can conclude the following two points: first of all, Intel's FPGA implementation can achieve a higher speedup effect. This is mainly because it generates input data directly on the FPGA side, which greatly reduces the transmission overhead between the device side and host side, consequently, the running speed is increased.

In addition, we can also find that Intel's fixed-point implementation is much faster than its floating-point implementations: compared with the corresponding floating-point implementation, Intel's fixed-point implementation achieved a speed increase of 300% while our fixed-point implementation only achieved a speed increase of about 15%. A possible reason for this could be that Intel incorporates a masking operation in their source code so that the offline compiler can reduce the hardware resources needed for fixed-point calculations. That is, Intel's fixed-point implementation retain a minimum data resolution that is necessary for model operation, whereas our implementation retains a full resolution.

For instance, if a 17-bit fixed-point representation is used in our model, then a 32-bit data type is required to store the data as OpenCL SDK only supports 8, 16, 32 and 64 bits data type. In this case, if we need to calculate the addition of two 17-bit fixed-point variables, then the hardware responsible for the addition of the extra 15 bits is wasted. To avoid wasting this unnecessary hardware resources, Intel's implementation uses bitmasks to inform the compiler to disregard the unnecessary bits during compilation time. Whereas our fixed-point implementation did not use this technology so we waste some hardware resources. Therefore in our project, using fixed-point does not improve our implementation as much as Intel does.

Overall, it provides us with a good idea about the future work by comparing our implementation to Intel's Merton model FPGA implementation.

4.3.2 Compare with CUDA GPU implementation of multi-factor Merton model

Li et al. .[19] is a research group that conducts similar research with us and is in a semi-cooperative nature. The major difference between our two research groups is that our main objective is to optimise the running speed of the multi-factor Merton model using FPGAs, while Li et al. mainly make use of GPUs to speed up the execution of the same model. Since the programming language, CPU and input data size used by our two research groups are different, for the fair comparison, we directly compare the speedup effect achieved by our two groups' best implementation.

Running on a GeForce GTX 1650 Ti GPU, Li et al. 's GPU-based implementation can improve the running speed of the C++ one core implementation by 22-23 times while ensuring accurate results, whereas we can speed up the same C++ implementation by 22 times while maintaining the same precision. Our implementation presents a comparable performance as Li's, but this is not the ideal situation. In Intel's research we discussed in the previous section, their FPGA implementation of the single factor Merton model is 5 times faster than the equivalent GPU implementation. This indicates that we still have room for improvement in our FPGA implementation.

4.3.3 Compare with an outstanding GPU implementation of the multi-factor Merton model

Prior to the work of Li et al., there were some mature studies that used GPU to accelerate the execution of the multi-factor Merton Model. One of the most representative one is the study given by Béresand and Bris [8]. They used two main methods to speed up the execution of the model: one is to use GPU to speed up the model computing, and the other is to use importance sampling to reduce the complexity of the model operation. By using importance sampling, the number of samples in important regions that may facilitate the calculation of portfolio loss could be increased, while the number of samples in unimportant regions can be greatly reduced, thus speeding up the calculation of the model.

As a result, running on a NVIDIA Kepler K20 GPU, their implementation speed up the multi-factor Merton model against the basic implementation by 19 x to 287 x times depending on the portfolio structure. In terms of the speeding up effect of the model, the GPU implementation proposed by Beresand and Bris significantly outperforms than that of our implementation. The main difference between the two implementations is that we do not use importance sampling to reduce the computational complexity of the model. However, experiments have shown that using importance sampling is a good way to accelerate the model execution. Therefore in the future work, it is also a good direction to try importance sampling in our implementation.

Chapter 5

Conclusion and future work

Due to the potentially devastating power of concentration risks, its management has become increasingly important in recent years in financial institutions. The multi-factor Merton model is a powerful tool for measuring and managing concentration risks. However, due to the requirement of a huge amount of Monte Carlo simulations, the execution of this model is usually very time-consuming. In this research, we propose an FPGA-based approach for the multi-factor Merton model, which is able to speed up the execution of the model and produce accurate results. Main contributions of this work involves:

- A comprehensive list of manual optimisation techniques for Intel OpenCL FPGA has been presented. Through experiments, we found that among all these optimization techniques, the NDRange kernel with loop unrolling and kernel vectorisation technique has the greatest acceleration effect on our model. However, theoretically, the single work-item kernel should improve the running speed of the model by more than or at least equal to the NDRange kernel. Therefore, in the future work, we will keep optimizing the design of the single work-item kernel to obtain a better performance.

- We found that compared with floating-point representations, using fixed-point numbers leads to a faster model execution speed and a higher precision loss. In our project, fixed-point representation increased the execution speed of the model by an additional 15% but also lead to a less accurate result compared to the floating point representation. However, we have also pointed out that the execution speed of fixed-point representations can be further improved by using the bitmask technique.

- We proposed an high-performance FPGA implementation for the multi-factor Merton model. Compared with the original C++ software implementation of the multi-factor Merton model on a one core Intel i5-4210 CPU, the proposed FPGA implementation achieves a speedup of 22 times. Compared with the same C++ implementation running on an 8 cores Intel i5-4210 CPU, our FPGA implementation achieves a speedup effect of 3.2 times. In addition, the precision loss of our proposed FPGA implementation fluctuates in a range of 10^{-8} to 10^{-5} depending on different types of data representations, which satisfies the requirement of concentration risk measurement.

Future work includes trying out several possible ways to optimise our implementation further. First of all, the data could be generated on the FPGA side so to reduce the data transmission consumption between the FPGA side and the host side. Secondly, some model-level optimisation methods, such as importance sampling, can be used to reduce the complexity of the model operation and thus improve its running speed. In addition, it is also important to use some hardware optimisation techniques such as bitmasks to optimise the speed of fixed-point operation.

Bibliography

[1] Timmins C, Schlenker W. Reduced-Form Versus Structural Modeling in Environmental and Resource Economics. Annual Review of Resource Economics. 2009 10;1:351–380. pages 1

[2] yu F. Default Correlation in Reduced-Form Models. Journal of Investment Management. 2005 04;3. pages 2

[3] Merton R. In: On the Pricing of Corporate Debt: The Risk Structure of Interest Rates; 2019. p. 79–102. pages 2

[4] Anderson-Cook C. Quantitative Risk Management: Concepts, Techniques, and Tools. Journal of the American Statistical Association. 2006 12;101:1731–1732. pages 2

[5] Morgan J, Grenfell D. The Benchmark for Understanding Credit Risk. 2020 08;. pages 2, 45

[6] Gokhale M, Cohen J, Yoo A, Miller W, Jacob A, Ulmer C, et al. Hardware Technologies for High-Performance Data-Intensive Computing. Computer. 2008 05;41:60 – 68. pages 2

[7] Lütkebohmert-Holtz E. Concentration Risk in Credit Portfolios; 2009. pages 4

[8] Béreš M, Briš R. In: Acceleration of multi-factor Merton model Monte Carlo simulation via Importance Sampling and GPU parallelization: Proceedings of the 1st International Conference on Applied Mathematics in Engineering and Reliability (Ho Chi Minh City, Vietnam, 4-6 May 2016); 2016. p. 107–118. pages 5, 51

[9] Zhang B, Oosterlee C. Acceleration of Option Pricing Technique on Graphics Processing Units. Concurrency and Computation Practice and Experience. 2014 06;26. pages 5

[10] Tian X, Benkrid K. Design and implementation of a high performance financial Monte-Carlo simulation engine on an FPGA supercomputer; 2009. p. 81 – 88. pages 5

[11] Tian X, Benkrid K. High-performance quasi-Monte Carlo financial simulation: FPGA vs. GPP vs. GPU. TRETS. 2010 11;3:26. pages 5

[12] Delivorias C. Case Studies in Acceleration of Heston's Stochastic Volatility Financial Engineering Model: GPU, Cloud and FPGA Implementations; 2012. . pages 5

[13] De Schryver C, Shcherbakov I, Kienle F, Wehn N, Marxen H, Kostiuk A, et al. An Energy Efficient FPGA Accelerator for Monte Carlo Option Pricing with the Heston Model; 2011. p. 468–474. pages 6

[14] Zohouri HR, Maruyamay N, Smith A, Matsuda M, Matsuoka S. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs; 2016. p. 409–420. pages 20

[15] Intel. Intel® FPGA SDK for OpenCLTM Pro Edition Best Practices Guide;. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf` Accessed Aug 8, 2020. [EB/OL]. pages 20, 40, 42

[16] Jia Q, Zhou H. Tuning Stencil codes in OpenCL for FPGAs; 2016. p. 249–256. pages 21

[17] Almqvist M. Master Thesis - Active Management of Non-Granular Loan Portfolios. 2015 4;. pages 42

[18] Intel. Monte Carlo Black-Scholes Asian Options Pricing Design Example;. `https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/black-scholes.html` Accessed Aug 8, 2020. [EB/OL]. pages 49

[19] Li L. Acceleration of the Multi-Factor Merton Model through GPU parallelisation. 2020 09;p. 53. pages 50

Appendix A

Legal and Ethical Considerations

The goal of this project is to develop an FPGA implementation of the multi-factor Merton model, which is able to measure the potential concentration risk in credit portfolios. we are not aware of any possible negative impacts this project would have. The table on the next page shows a completed ethics checklist. To conclude, there are no legal or professional issues involved in this project.

Our project is software-based and does not involve any experiments on humans or animals. Besides, all of the data used in this project is generated randomly according to some predefined rules, and therefore we does not violate any GDPR or other data privacy regulations.

We use Intel OpenCL SDK in our project, which has a free academic license. The other libraries used in our project are all open sources and therefore this project does not contains license issues.

Table A.1: Ethics checklist

	Yes	No
Section 1: HUMAN EMBRYOS/FOETUSES		
Does your project involve Human Embryonic Stem Cells?		✓
Does your project involve the use of human embryos?		✓
Does your project involve the use of human foetal tissues / cells?		✓
Section 2: HUMANS		
Does your project involve human participants?		✓
Section 3: HUMAN CELLS / TISSUES		
Does your project involve human cells or tissues? (Other than from “Human Embryos/Foetuses” i.e. Section 1)?		✓
Section 4: PROTECTION OF PERSONAL DATA		
Does your project involve personal data collection and/or processing?		✓
Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		✓
Does it involve processing of genetic information?		✓
Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.		✓
Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?		✓
Section 5: ANIMALS		
Does your project involve animals?		✓
Section 6: DEVELOPING COUNTRIES		
Does your project involve developing countries?		✓
If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?		✓
Could the situation in the country put the individuals taking part in the project at risk?		✓
Section 7: ENVIRONMENTAL PROTECTION AND SAFETY		

Does your project involve the use of elements that may cause harm to the environment, animals or plants?		✓
Does your project deal with endangered fauna and/or flora /protected areas?		✓
Does your project involve the use of elements that may cause harm to humans, including project staff?		✓
Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?		✓
Section 8: DUAL USE		
Does your project have the potential for military applications?		✓
Does your project have an exclusive civilian application focus?		✓
Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?		✓
Does your project affect current standards in military ethics – e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?		✓
Section 9: MISUSE		
Does your project have the potential for malevolent/criminal/terrorist abuse?		✓
Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?		✓
Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?		✓
Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?		✓
textbfSECTION 10: LEGAL ISSUES		
Will your project use or produce software for which there are copyright licensing implications?		✓
Will your project use or produce goods or information for which there are data protection, or other legal implications?		✓

SECTION 11: OTHER ETHICS ISSUES		
Are there any other ethics issues that should be taken into consideration?		✓