Imperial College London
Department of Computing

Accelerating Reconfigurable Financial Computing

Hong Tak Tse (Anson)

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the Imperial College
January 2012

Declaration

This thesis is a presentation of my original research work. The contributions of others are involved, every effort is made to indicate this clearly in the references to the literature and acknowledgment of collaborative research.

Signature: .....

Date: .....

Abstract

This thesis proposes novel approaches to the design, optimisation, and management of reconfigurable compute accelerators for financial computing. There are three contributions. First, we propose novel reconfigurable designs for derivative pricing using both Monte-Carlo and quadrature methods. Such designs involve exploring techniques such as control variate optimisation for Monte-Carlo, and multi-dimensional analysis for quadrature methods. Significant speedups and energy savings are achieved using our Field-Programmable Gate Array (FPGA) designs over both Central Processing Unit (CPU) and Graphical Processing Unit (GPU) designs. Second, we propose a framework for distributing computing tasks on multi-accelerator heterogeneous clusters. In this framework, different computational devices including FPGAs, GPUs and CPUs work collaboratively on the same financial problem based on a dynamic scheduling policy. The trade-off in speed and in energy consumption of different accelerator allocations is investigated. Third, we propose a mixed precision methodology for optimising Monte-Carlo designs, and a reduced precision methodology for optimising quadrature designs. These methodologies enable us to optimise throughput of reconfigurable designs by using datapaths with minimised precision, while maintaining the same accuracy of the results as in the original designs.

Acknowledgements

I would like to express my greatest gratitude to my supervisors Professor Wayne Luk and Dr. David Thomas. It would not have been possible to write this doctoral thesis without their support and patience. Their ideas, advice and knowledge guide me a right direction of research to complete the thesis.

Special thanks to Mr. Gary Chow Chun Tak, for his contributions to Chapter 6. We collaborated, shared our ideas and finally came up with the idea on mixed precision methodology (Section 6.3). He also defined the partitioning schemes (Section 6.4), proposed the optimisation algorithm (Section 6.5) and helped implementing some of the hardware designs (Section 6.6). I would also like to acknowledge Dr. Kuen Hung Tsoi and Mr. Qiwei Jin for their help in solving technical issues for the experiments and proof-reading this thesis. I express my gratitude to fellow colleagues in Custom Computing Group of Imperial College London: Dr. Chi Wai Yu, Dr. Chun Hok Ho, Mr. Adrien Le Masle, Mr. Brahim Benkaoui, Prof. Yuet Ming Lam, Dr. Van Fu, Prof. Qiang Liu, Dr. Timothy Todman, Dr. Tobias Becker, Mr. Philip Potter and Prof. Peter Jamieson for their time of discussions and experience sharings.

I would like to thank Prof. John Lui, Prof. Philip Leong and Mr. Ricky Tsui for their reference letters when I was applying the Croucher Foundation scholarship. And I sincerely thank the Croucher Foundation for the financial support in this whole research period.

In addition, I thank my friends in London and Hong Kong for encouraging and supporting me.

The support of UK EPSRC, FP7 EPiCS and REFLECT projects, the HiPEAC NoE, MAXELER Technologies, Celoxica and Xilinx is gratefully acknowledged.

Dedication

To my parents: *Chun Sang Tse* and *So Fan Cheng* -

who bring me to this world and take care of me in my childhood.

To my brother and sister: *Sze Tak Tse* and *Wai Tak Tse* -

who help solving my problem in my life.

To my friends: -

who grow up with me and accompany me when I am alone.

Publications

Journal Papers

1. Anson H.T. Tse, David B. Thomas and Wayne Luk, "Design Exploration of Quadrature Methods in Option Pricing", *IEEE Transactions on Very-Large Scale Integration (VLSI) Systems* (Accepted for publication), 2012.

2. Anson H.T. Tse, David B. Thomas, K.H. Tsoi and Wayne Luk, "Efficient Reconfigurable Design for Pricing Asian Options", *ACM SIGARCH Computer Architecture News*, vol.38, no.4, pp.14-20, Sept. 2010.

3. K.H. Tsoi, Anson H.T. Tse, Peter Pietzuch and Wayne Luk, "Programming Framework for Clusters with Heterogeneous Accelerators", *ACM SIGARCH Computer Architecture News*, vol 38, no 4, pp.53-59, Sept. 2010.

Conference Papers

1. Anson H.T. Tse, Gary C.T. Chow, Qiwei Jin, David B. Thomas and Wayne Luk, "Optimising Performance of Quadrature Methods with Reduced Precision", *International Symposium on Applied Reconfigurable Computing* (Accepted for publication), 2012.

2. Gary C.T. Chow, Anson H.T. Tse, Qiwei Jin, David B. Thomas, Philip Leong, and Wayne Luk, "A mixed precision Monte Carlo methodology for reconfigurable accelerator systems", *In Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* (Accepted for publication), 2012.

3. Anson H.T. Tse, David B. Thomas, K.H. Tsoi and Wayne Luk, "Dynamic Scheduling Monte-Carlo Framework for Multi-Accelerator Heterogeneous Clusters", *In Proc. International Conference on Field-Programmable Technology (FPT)*, pp.233-240, 2010.

4. Anson H.T. Tse, David B. Thomas and Wayne Luk, "Accelerating Quadrature Methods for Option Valuation", *In Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp.29-36, 2009.

Short Papers

1. Anson H.T. Tse, David B. Thomas, K.H. Tsoi and Wayne Luk, "Reconfigurable Control Variate Monte-Carlo Designs for Pricing Exotic Options", *In Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pp.364-347, 2010.

2. Anson H.T. Tse, David B. Thomas and Wayne Luk, "Option Pricing with Multi-Dimensional Quadrature Architectures", *In Proc. International Conference on Field-Programmable Technology (FPT)*, pp.427-430, 2009.

Abbreviations

ASIC	-	Application Specific Integrated Circuit
CAD	-	Computer-Aided Design
CPU	-	Central Processing Unit
DP	-	Double Precision
DSP	-	Digital Signal Processor
FP	-	Floating Point
FPGA	-	Field Programmable Gate Array
FPU	-	Floating Point Unit
GPU	-	Graphics Processing Unit
HDL	-	Hardware Description Language
I/O	-	Input / Output
LSB	-	Least-Significant Bit
LUT	-	Look Up Table
MC	-	Monte-Carlo
MSB	-	Most-Significant Bit
RFC	-	Reconfigurable Financial Computing
SRAM	-	Static Random Access Memory
VHDL	-	Very High Speed Integrated Circuit Hardware Description Language

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation: Demand from the financial industry

The financial derivatives trade sees constant innovation and development, with new types of options introduced in each year, offering increasingly sophisticated features and complex settlement terms. Although the basic European option can be priced with a closed-form solution, many other derivatives with knock-out/knock-in features (e.g. Accumulator, Decumulator, and Barrier Options), changing strike prices, or discrete settlement days, have no simple solution and so their price must be approximated using numerical techniques. Many financial derivatives involve multiple underlying assets, which increases the dimensionality of the problem, so computational complexity often scales exponentially with the number of underlying assets.

The derivative pricing time is critical for trader for hedging and market making purpose. Also, huge amounts of computational resources are needed when many complex options are being revalued overnight under many different scenarios for risk management purpose. Energy consumption of computation is also a major concern when the computation is performed 24 hours a day, 7 days a week. As a result, the financial industry has seen a sharp increase in computational demand [1] [2]. There has been much interest in reducing the pricing latency and increasing the pricing throughput in order to gain competitive advantage. It is also important to increase the energy efficiency and reduce the cost of computing hardware in order to reduce the overall business cost.

In 2010, it is estimated that 41 million servers on the planet consumed around 18,118 billion kWh electricity each year when the energy for associated cooling and power distribution is included [3]. Increasing the number of traditional servers is not a viable solution, so there is a need for research into new types of solution, such as FPGA acceleration technology.

1.2 Motivation: Change of the computing technology

Moore's Law [4] (the doubling of transistors on chip every 18 months) has been a fundamental driver of computing technology for the previous 50 years. Moore's Law and Dennard scaling [5] has resulted in exponential performance increase of single-core processor. Since 2005, processor designers have increased the core counts to continuously exploit Moore's Law scaling, due to the end of Dennard scaling [6] [7]. The focus of computing technology has switched from performance-centric serial computation to energy-efficient parallel computation. However, the increasing number of components on a chip, combined with decreasing energy scaling, is leading to the phenomenon of "Dark Silicon" [7]. The power density of a chip is too high to use all components at once. It has been predicted that the performance of processors in 2024 will have only 7.9 times average speedup over the processors in 2008, leaving a near 24 times gap from a predicted of 32 times speedup according to Moore's law. These challenges are changing the computer technology to emphasize on efficiency, and driving chips to use multiple different components, each carefully optimised to efficiently execute a particular type of task [8].

One solution to improved energy efficiency is to use application-optimised processors and accelerators. By optimising these components for specific application, their energy efficiency can be increased by orders of magnitude. However, specialisation comes with a loss of generality. Therefore, there is a significant burden on system designers and application developers to choose the right combination of processors and accelerators, and how to apply optimisation in the applications. How to determine the right mix or choice of processor and accelerators for a specific application domain (e.g. financial computing), and how to optimise the design of accelerators in a specific application domain is of great research interest.

1.3 Objectives

Reconfigurable Financial Computing (RFC) is a technique to use reconfigurable hardware as an accelerator for financial computing. Reconfigurable hardware such as field-programmable gate array (FPGA) has been commonly used in communication and networking applications [9]. It is also widely applied for application acceleration in a wide variety of areas, such as video-processing, bioinformatics and cryptography [10, 11, 12, 13], where a large proportion of program time is spent on numerical computation. The benefits of incorporating FPGAs in a system design have been demonstrated in numerous research papers [14, 15] especially in the area of computational finance [16, 17]. Some financial institutions have been actively researching and seeking the opportunities to accelerate financial computing with FPGA. For example, FPGA accelerated credit derivatives pricing is adapted in the investment bank J.P. Morgan [18, 19].

FPGAs provide customisable floating-point number operation which could be exploited for additional speedup. Reduced-precision data-paths usually have higher clock frequencies, consume fewer resources and offer a higher degree of parallelism for a given amount of resources compared with full precision data-paths. Although the use of reduced precision can lead to higher performance, it also affects the accuracy of the results.

Graphical Processing Unit (GPU) is another popular choice in high performance computing recently. GPUs use the same types of floating-point number representation and operation as CPUs, namely IEEE-754 double precision and IEEE-754 single precision. GPUs are shown to provide significant speedup for many applications including in financial computing, especially when single precision is used [20].

Numerical methods for derivative pricing can be roughly divided into two groups: Monte-Carlo methods, which work forwards from the current asset price to expiry time using multiple randomly chosen paths; and lattice methods, which work backwards from exercise time to the current price, using a pre-determined lattice of asset prices and times. Quadrature methods are subsets of the lattice methods that are very powerful of pricing path-dependent options where the path is monitored in discrete time points [21].

With regards to the above circumstances, we defined our objectives in this thesis as:

- *Generic architecture for derivatives pricing*: it must be possible to support multiple derivative types with minimal manual effort. Numerical methods including both Monte-Carlo and lattice methods must be supported in order to cover as much derivatives as possible. Optimisation techniques based on generic derivatives pricing methods must be provided. Comparison between different accelerators in terms of performance and energy efficiency must be explored in detail.

- *Automated management*: the system must automatically adjust the workload balance between different accelerators including both FPGAs and GPUs. The workload adjustment policy must be configurable for a pre-defined objective. The system must be scalable to support a large computation problem.

- *Precision optimisation*: the custom numerical representation should be optimised automatically. The error incurred by custom numerical representation must be analysed. Models and algorithms for performance and accuracy optimisation must be problem independent. The performance and energy efficiency gains by precision optimisation must be explored in order to determine the right choice or right mix of processors and accelerators.

We firstly present our novel accelerated reconfigurable hardware architectures and optimisation techniques for option pricing based on Monte-Carlo methods [22, 23] (Chapter 3) and quadrature methods [24, 25, 26] (Chapter 4). The performance of the FPGA designs are compared with those of both GPU and CPU designs. The performance and energy efficiency of different designs are studied and discussed in depth. Then we present a scalable distributed framework for collaborative financial computing for multi-accelerator heterogeneous clusters including both FPGAs, GPUs and CPUs [27] (Chapter 5). Lastly, we present performance optimisation methodologies and techniques for both Monte-Carlo methods and quadrature methods by exploiting the customisable precision property of FPGAs. A mixed precision methodology and a reduced precision methodology are proposed to maximize the performance of Monte-Carlo methods [28] (Chapter 6) and quadrature methods respectively [29](Chapter 7).

1.4 Research Approach and Contributions

The research approach aims at accelerating and optimising reconfigurable financial computing in a generic way. Therefore, the hardware architectures, frameworks, methodologies, and optimisation techniques in each chapter can be used for the pricing of a wide range of different options. In Chapter 6 and Chapter 7, the mixed precision and reduced precision methodologies can also be applied in other problem domains apart from financial option pricing.

In each chapter, case studies and detailed experiments are carried to demonstrate the effectiveness of our proposed methodologies. The computing performances and energy consumptions are measured for different computational devices including FPGA, GPU and CPU. The comparison results of these computational devices in financial computing are one of the key aspects that we are interested in this thesis as they provide references for financial institutions when designing high performance derivatives pricing infrastructures.

Our research focuses on two popular and equally important option pricing methods: quadrature methods and Monte-Carlo methods. Quadrature methods are fast and accurate for many derivatives, and Monte-Carlo methods are the only computational feasible methods when the derivatives involve many underlying assets.

The main contributions of this thesis corresponding to our objectives are:

- (*Generic architecture for derivatives pricing*) A novel parallel hardware architecture using Monte-Carlo methods for the pricing of a wide range of exotic options. This includes the detailed parametric design of arithmetic Asian options pricer and the control variate optimisation technique. The performance comparison results show a speedup of 24 times for the FPGA over CPU. (Chapter 3)

- (*Generic architecture for derivatives pricing*) A novel parallel hardware architecture for option pricing based on quadrature methods. This includes techniques for pricing options with multiple dimensions and an approach of automatically generating multi-dimensional hardware cores.

The experimental results show that FPGA design is 4.6 times faster and 25 times more energy efficient than a software design running on a comparable CPU. (Chapter 4)

- (*Automated management*) A scalable distributed financial computing framework which enables all accelerators including FPGAs and GPUs in a multi-accelerator heterogeneous cluster to work collaboratively on the same problem. This includes a dynamic runtime scheduling system which enables the designer to improve the utilisation efficiency. Two practical examples are developed using the proposed framework and their performances under different scheduling policies are evaluated. (Chapter 5)

- (*Precision optimisation*) A mixed precision methodology for Monte-Carlo methods which constructs the FPGA data-path with an aggressively reduced precision and corrects the finite precision error by auxiliary sampling. This work presents the error analysis, techniques for partitioning workloads, and optimisation algorithms of the proposed methodology. Three case studies show a performance gain of 2.9 to 7.1 times is achieved with the mixed precision FPGA design over the original double precision FPGA design. (Chapter 6)

- (*Precision optimisation*) A reduced precision methodology for quadrature methods which determines the optimal precision and integration grid density by constructing a set of Pareto frontier points satisfying the error tolerance level. The work includes the optimisation modeling, an accuracy analysis and the optimisation algorithms. Case studies demonstrate that a performance gain of 4 times speedup is achieved using the reduced precision FPGA design over the original double precision FPGA design. (Chapter 7)

1.5 Thesis Organisation

This thesis is organised as Figure 1.1. The grey boxes indicate the objectives that are related to our chapters. Chapter 2 describes the background and related work in reconfigurable financial computing. Chapter 3 and 4 present our novel reconfigurable hardware design and techniques for option pricing based on Monte-Carlo methods and quadrature methods. Chapter 5 presents a scalable distributed

framework for collaborative financial computing on multi-accelerator heterogeneous clusters including both FPGAs, GPUs and CPUs. Chapter 6 and 7 describe performance optimisation techniques for both Monte-Carlo methods and quadrature methods. A mixed precision methodology and a reduced precision methodology are proposed to maximize the performance of Monte-Carlo methods and quadrature methods correspondingly. Finally, Chapter 8 summarises the thesis and suggests for future work.

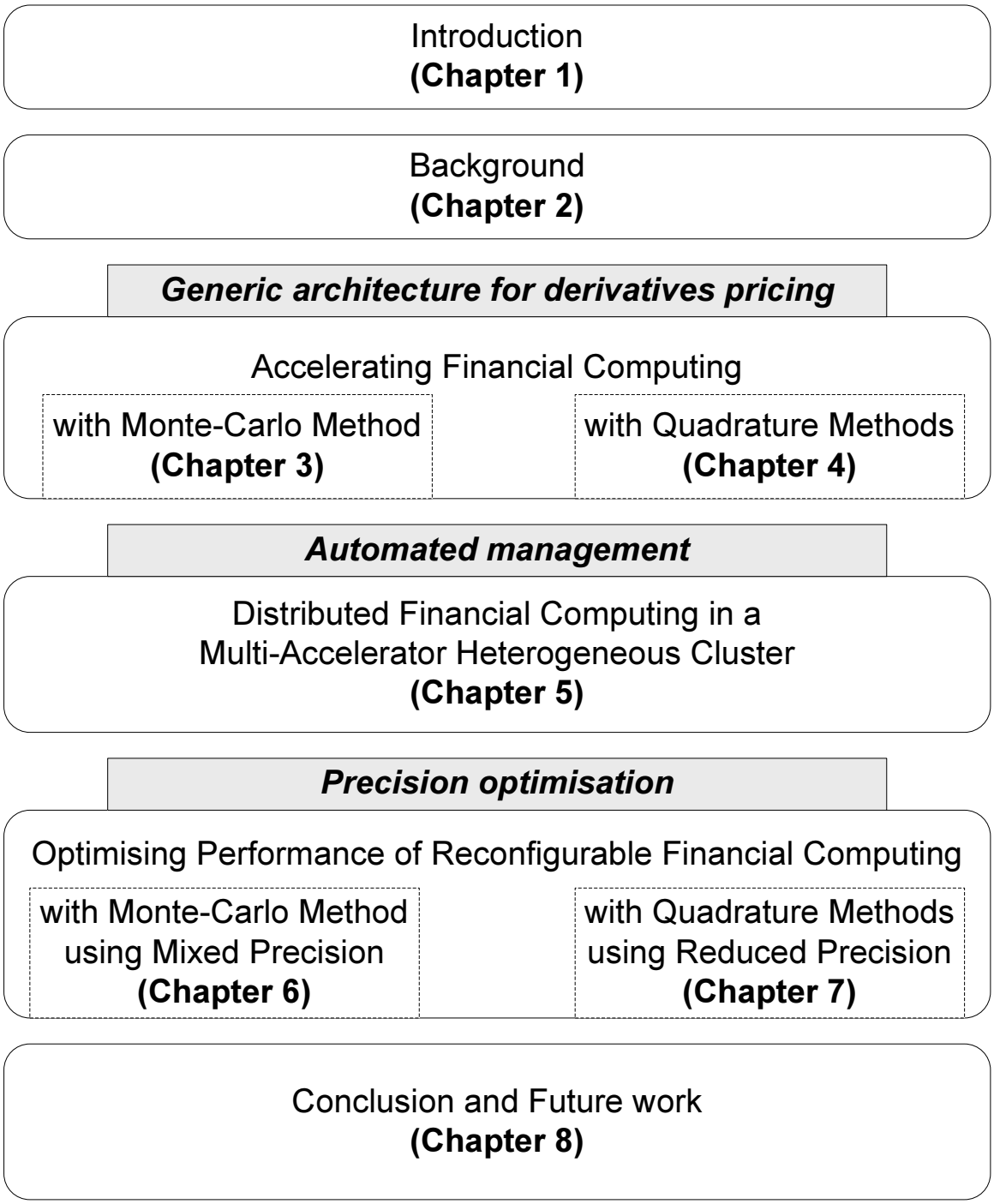

Figure 1.1: The organisation of this thesis.

Chapter 2

Background

This chapter presents the background knowledge and related works in financial computing and reconfigurable computing. Section 2.1 provides background knowledge of option pricing including the option pricing model and the examples of exotic options. Section 2.2 introduces the numerical methods used in option pricing and the corresponding related works. Section 2.3 presents the background knowledge and related works of algorithmic trading using reconfigurable devices. Section 2.4 introduces different computational devices and analyses their difference and strengths. Section 2.5 presents the previous works on bit-width optimisation using FPGA. Section 2.6 presents the previous works on cluster computing involving accelerators. Section 2.7 provides the background knowledge of hardware description languages that are used in this thesis.

2.1 Option Pricing

An option is a type of financial instrument which provides the owner of the option with the right, but not the obligation, to buy or sell an underlying asset such as a stock or bond at some point in the future. A *call option* allows the option owner to buy the underlying asset for some pre-agreed strike price K, while a *put option* gives them the right to sell at price K. The decision to exercise the option (i.e. buy or sell the asset) is always made by the option owner, and the option issuer has to abide by that decision, so the option owner must pay the issuer to create the option. Hence putting an accurate

value on an option is critical for both parties.

For simple European call options (also known as vanilla call options), the owner can exercise only at the expiry date. If the underlying asset price S at expiry date is higher than the strike price K, the owner can profit by buying the stock at lower price K from the option issuer and then immediately selling it at the higher price S in the market, providing a gain of $(S - K)$. If the underlying asset price is lower than the strike price, $S < K$, then the gain is zero because the option will not be exercised.

The payoff of European call option on expiry is

$$P_{call} = \max(S - K, 0) \tag{2.1}$$

and the payoff of European put option at expiry is

$$P_{put} = \max(K - S, 0). \tag{2.2}$$

2.1.1 Option Pricing Model

A common assumption is that stock price follows a geometric Brownian motion. That is,

$$\frac{dS}{S} = \mu dt + \sigma dW_t \tag{2.3}$$

where W_t is a Brownion motion (random walk), S is the underlying stock price, μ is the drift of the stock price, t is time and σ is the volatility. Using risk-neutral measure [30], we have the following equation:

$$\frac{dS}{S} = r dt + \sigma dW_t^Q \tag{2.4}$$

where r is the risk-free interest rate.

By solving the above stochastic differential equation (SDE) using Ito's lemma, we have the following

Black-Scholes partial differential equation and stock price dynamic equation [31, 32]:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{2.5}$$

$$S_{i+1} = S_i e^{((r-\frac{\sigma^2}{2})\delta t + \sigma\sqrt{\delta t}W)} \tag{2.6}$$

where V is the price of the option, σ is volatility of the underlying asset, δt is the time period between two time steps, W is a Gaussian random number $\sim \mathcal{N}(0,1)$, S_i is the underlying stock price at step i and S_{i+1} is the underlying stock price at step $i+1$.

Under this model, the price of an European option at present time can be calculated with a closed-form solution called the Black-Scholes formula. The price of more complex options (exotic options) are usually calculated by numerical methods base on Equation 2.5 or Equation 2.6. Figure 2.1 shows an example of random walk asset paths based on the stock price dynamic equation (Equation 2.6).

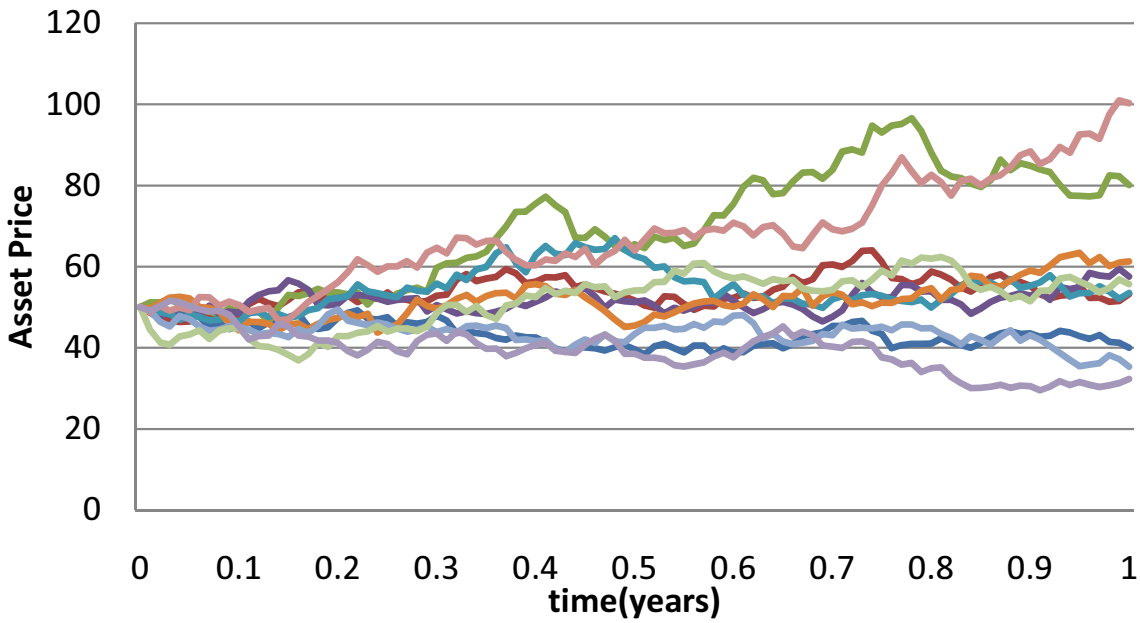

Figure 2.1: Example of random walk asset paths.

2.1.2 Exotic Option

Exotic option is a derivative which has features making it more complex and usually has no closed-form solution. For path-dependent exotic options, the payoff on expiry depends on the entire underlying asset movement path. Examples of these exotic options include lookback options, arithmetic Asian options and barrier options.

Lookback Options

The payoff of lookback options depends on the maximum value of the stock price in the whole period. It is defined as:

$$P_{call} = \max(\max(S_0, S_1, ..., S_n) - K, 0) \tag{2.7}$$

where $S_0, ..S_n$ are the asset price at time step $0...n$.

Arithmetic Asian Options

For an arithmetic Asian option [33], the payoff is calculated using the arithmetic average of the prices over the life time of the option. One advantage of this option type is that it is more difficult for the option issuer to manipulate market prices to reduce the option payoff, as the payoff depends on the path followed by the asset price, not just the price at expiry.

The payoff of an arithmetic Asian call option is:

$$P_{call} = \max\left(\frac{1}{n+1}\sum_{i=0}^{n} S_i - K, 0\right) \tag{2.8}$$

where $S_0, ..S_n$ are the asset price at time step $0...n$.

Barrier Options

Barrier options are path-dependent options where the payoff also depends on a predetermined barrier level B. *In* options start their lives worthless and only become active when the underlying asset moves across the barrier B level as known as "knock-in" barrier price. *Out* option starts their lives as active and become worthless when the underlying asset moves across the "knock-out" barrier price. There are four main types of barrier options: up-and-out, down-and-out, up-and-in and down-and-in. Figure 2.2 shows the payoff function of an up-and-out barrier option at maturity. However, the payoff of an up-and-out barrier option is also time-dependent and will be zero if the price of underlying asset moves up across the barrier level before maturity.

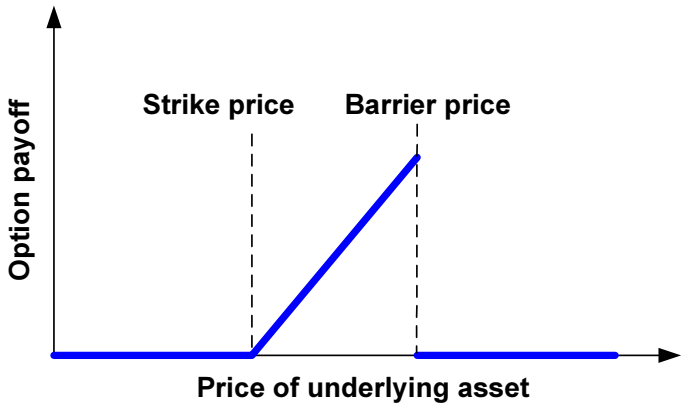

Figure 2.2: The payoff function of an up-and-out barrier option at maturity.

- Up-and-out: The spot price is below the barrier level at the beginning. The option is worthless and is knocked out once the asset moves up across the barrier level.
- Down-and-out: The spot price is above the barrier level at the beginning. The option is worthless and is knocked out once the asset moves down across the barrier level.
- Up-and-in: The spot price is below the barrier level at the beginning. The option becomes active is knocked in once the asset moves up across the barrier level.
- Down-and-in: The spot price is above the barrier level at the beginning. The option becomes active and is knocked in once the asset moves down across the barrier level.

Barrier options can also be divided into two categories: discrete and continuous. For a continuous barrier option, the knock-in or knock-out barrier event is considered immediately if the asset moved across the barrier line. For a discrete barrier option, the knock-in or knock-out barrier event is checked at a discrete time (e.g. end of the day or end of the month), hence less sensitive to market manipulation.

In addition, **moving** barrier options are particularly difficult to price. They have multiple different barrier prices B_m at different time period m. There is no closed-form solution for discrete moving barrier options.

2.1.3 Stochastic Volatility

Financial equations are often based on many assumptions. The most famous Black-Scholes equation relies on a constant volatility assumption [31]. In fact, it is well-known that the volatility is not constant in reality. A solution is to employ a stochastic volatility model. One of the most commonly used stochastic volatility models is Generalized Autoregressive Conditional Heteroskedasticity (GARCH) models [34]. A commonly used GARCH (1,1) model defines volatility (σ) by the following equations:

$$\begin{aligned} \sigma_i^2 &= \sigma_0 + \alpha\sigma_{i-1}^2 + \beta\sigma_{i-1}^2 W^2 \\ &= \sigma_0 + \sigma_{i-1}^2(\alpha + \beta W^2) \end{aligned} \tag{2.9}$$

where σ_i is the volatility of the asset at time step i, α and β are pre-calibrated model constants, σ_0 is the volatility at the start time and W is a Gaussian random number following a $\mathcal{N}(0, 1)$ distribution.

2.2 Numerical Methods for Option Pricing

Numerical techniques have been developed to value complex derivative products. They can be roughly divided into two groups: lattice methods, which work backwards from exercise time to the current price, using a pre-determined lattice of asset prices and times; and Monte-Carlo methods, which work forwards from the current asset price to expiry time using multiple randomly chosen

paths. Lattice methods include tree-based (binomial and trinomial trees), finite-difference and quadrature methods. We briefly introduce the related works with hardware acceleration using these methods, and then describe the details of Monte-Carlo and quadrature methods.

- *(Lattice) Tree-based methods:* Tree-based methods use a "discrete-time" model of the varying price over time of the underlying financial instrument. Valuation is performed iteratively, starting at each of the final nodes (those that may be reached at the time of expiration), and then working backwards through the tree towards the first node (valuation date) [35].

  A pipelined hardware architecture has been developed for the binomial and trinomial option models [36]. However, tree-based methods contain two main types of error: "distribution error" and "non-linearity error". Distribution error occurs because a continuous log-normal distribution is approximated by a discrete distribution. Non-linearity error occurs because the tree grid cannot cater for non-linearity in option price for certain values of the underlying asset. Non-linearity in option pricing is frequent for exotic options: for example in a discrete barrier option, at every barrier there is a non-linearity in the option price.
- *(Lattice) Finite-difference methods:* Finite-difference methods solve the Black-Scholes partial differential equation by discretising both time and the price of the underlying asset, and mapping both onto a two-dimensional grid [37]. Valuation is performed iteratively similar to tree-based method. There are three kinds of finite-difference methods: implicit, explicit and Crank-Nicolson.

  A parallel hardware architecture has been developed to support concurrent valuation of independent options with 12 times speedup [38]. Similar to tree-based methods, finite-difference methods suffer from "non-linearity error" as the grid cannot cater the non-linearity points.
- *(Lattice) Quadrature methods:* Quadrature methods have been applied in different areas including pricing options [39], modeling credit risk [40], solving electromagnetic problems [41] and calculating photon distribution [42]. A numerical approach for option pricing based on quadrature methods has been proposed, which overcomes the "distribution error" and the "non-linearity error" [21], and demonstrates accurate and fast calculation. Hardware acceleration of quadrature methods are not reported before this thesis and there is no previous performance results. Therefore, a generic hardware architecture of quadrature methods in option pricing is

presented in this thesis in Chapter 4 and the reduced precision optimisation methodology is proposed in Chapter 7.

- *Monte-Carlo methods:* Monte-Carlo methods are particularly suitable for implementation in FPGAs, as they contain abundant parallelism. An early FPGA-accelerated Monte-Carlo application for the BGM interest rate model [16] using customised data widths achieved a 25 times speedup over software. An automated methodology has been developed which produces optimized pipelined designs with thread-level parallelism based on high-level mathematical descriptions of financial simulation [43]. A stream-oriented FPGA-based accelerator with higher performance than GPUs and Cell processors has been proposed for evaluating European options [44]. More recent work has focused on considering more complex types of Monte-Carlo simulation, such as American exercise features [45]. However, no precision optimisation methodology has reported from the above works. In addition, optimising generic option pricing by its statistical property and by collaborative computing with other accelerators are not yet reported. In this thesis, the use of control variate technique for generic option pricing is proposed in Chapter 3, the design framework and techniques of automated collaborative computing with other accelerators is proposed in Chapter 5 and the mixed precision optimisation methodology is proposed in Chapter 6.

2.2.1 Monte-Carlo Methods

Monte-Carlo methods are a class of algorithms based on randomisation which are extensively used in many applications in science and engineering. The idea is to generate a huge number of random paths for each probabilistic variable, then take the average of the results. Consider a sequence of mutually independent, identically distributed random variables, X_i from a Monte-Carlo simulation. If, $\text{Sum}_N = \sum_{i=1}^{N} X_i$, and the expected value, I, exists, the Weak Law of Large Numbers states that if $p(x)$ is the probability of x, for $\epsilon > 0$, the approximation approaches the mean for large N [46],

$$\lim_{N\to\infty} p\left(|\frac{\text{Sum}_N}{N} - I| > \epsilon\right) = 0 \tag{2.10}$$

Table 2.1: An example of stock price paths ($S_0 = 1.00, K = 1.03, T = 2, r = 0.1$)

Path:	$t = 0$	$t = 1$	$t = 2$	Avg Price	Payoff
Path 1	1.00	1.22	1.25	1.16	0.13
Path 2	1.00	1.18	1.41	1.20	0.17
Path 3	1.00	0.92	0.88	0.93	0.00
Path 4	1.00	1.11	1.32	1.14	0.11
Path 5	1.00	0.99	1.09	1.03	0.00
Path 6	1.00	1.16	1.09	1.08	0.05
Path 7	1.00	1.19	1.39	1.19	0.16
Path 8	1.00	0.91	0.86	0.92	0.00
Path 9	1.00	1.22	1.21	1.14	0.11
Path 10	1.00	0.94	0.84	0.93	0.00
Avg Payoff					0.07

Moreover, if the variance σ^2 exists, the Central Limit Theorem states that for every fixed a,

$$\lim_{N\to\infty} p\left(\frac{\text{Sum}_N - NI}{\sigma\sqrt{N}} < a\right) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{a} e^{-z^2/2}dz \tag{2.11}$$

that is, the distribution of the standard error is normal [47].

We illustrate the idea of Monte-Carlo methods in option pricing with an arithmetic Asian option as an example. The arithmetic Asian option has the following parameters S_0 (spot price) $= 1.00$, K (strike price) $= 1.03$, r (risk-free interest rate) $= 0.1$, T (time to maturity) $= 2$ and $steps = 2$. Table 2.1 shows an example of simulated stock price paths. Firstly, 10 stock price paths from $t = 0$ to $t = 2$ are simulated. Then the average stock price for each path is calculated as in 'Avg' column. The payoff of each path is then calculated according to Equation 2.8 as in 'Payoff' column. Finally, the average payoff across all these paths is calculated. The final result is the expected value of the arithmetic Asian call option at $t = 2$. The arithmetic call option value at present time can be obtained by discounting this final answer backward by multiplying e^{-rT}. The option price in the above example is 0.057.

2.2.2 Control Variate Monte-Carlo Method

When Monte-Carlo method is used for option pricing, the payoff of the option is the variable that is simulated. We can construct a confidence interval of our estimated expected payoff based on the

number of simulations.

The 99% confidence interval of the payoff is given by:

$$\textit{Payoff}_{99\%} = \left[\bar{x} - 2.58\frac{\sigma_x}{\sqrt{N_{mc}}}, \bar{x} + 2.58\frac{\sigma_x}{\sqrt{N_{mc}}}\right] \tag{2.12}$$

where $\bar{x}$ is the estimated expected payoff , N_{mc} is the number of simulations and σ_x is the standard deviation of payoff x. The 99% confidence interval means the actual value has a chance of 99% to be inside the range of the interval [48].

Therefore, to improve accuracy (reduce confidence interval length) by a factor of n, the number of Monte-Carlo simulations has to be increased by a factor of n^2, which is the reason for the high computational complexity of Monte-Carlo methods.

Variance reduction techniques aim at shortening the interval by reducing the variance instead of increasing the number of simulations. The control variate method is a variance reduction technique which estimates the target value using a control variable y [49]. The variable $\bar{y}$ is computed using the same set of random data of the computation of $\bar{x}$. The true expected value of $E(y)$ should be calculable using a closed-form solution. The control variate estimator of x_c is given by:

$$x_c = x + c(y - E(y)) \tag{2.13}$$

Therefore, $E(x_c)$ = $E(x)$, and $Var(x_c)$ is minimized by choosing $c = -Cov(x, y)/Var(y)$, such that

$$Var(x_c) = Var(x) - \frac{Cov(x, y)^2}{Var(y)} \tag{2.14}$$

As a result, the variance of the estimated value is reduced and thus the length of confidence interval is shortened. The higher the correlation between the control variable and the target estimating variable, the higher effectiveness of control variate method. The required number of simulations could be significantly reduced for a given confidence interval.

This control variate Monte-Carlo (CVMC) method can be applied to exotic option pricing. Apart

from simulating the payoff of the target exotic option, the payoff of a correlated control option is also simulated at the same time. The only condition is, the closed-form solution of the control option must be known.

2.2.3 Quadrature Methods

Quadrature methods are numerical methods for approximating an integral by evaluating at a finite set of integration points and using a weighted sum of these values. To apply quadrature methods for option pricing, the Black-Scholes partial differential equation is transformed to an integral form. The details of the transformation and the mapping to the hardware will be described in Chapter 4. After determining the boundary conditions according to the number of dimensions and the option type, the integral is evaluated by one of the quadrature rules. There are many different rules of numerical integral evaluation. Two of the most common rules include the trapezoidal rule and Simpson's rule [50]:

Trapezoidal rule: The trapezoidal rule is the simplest quadrature method but is the slowest to converge. It converges at a rate of $(\delta y)^2$. The approximation equation is:

$$\int_a^b f(y)dy \approx \frac{\delta y}{2}\{f(a) + 2f(a+\delta y) + 2f(a+2\delta y)\cdots + 2f(b-\delta y) + f(b)\} \tag{2.15}$$

Simpson's rule: This is the most popular method for approximating integrals. It converges at a rate of $(\delta y)^4$. The approximation equation is:

$$\int_a^b f(y)dy \approx \frac{\delta y}{6}\{f(a) + 4f(a+\frac{1}{2}\delta y) + 2f(a+\delta y) + \cdots + 2f(b-\delta y) + 4f(b-\frac{1}{2}\delta y) + f(b)\} \tag{2.16}$$

Quadrature methods are powerful ways of pricing path-dependent options where the path is monitored in discrete time. A lookback discrete barrier option priced using quadrature methods is more than 1000 times faster than using the trinomial method, while achieving a more accurate result [21].

2.3 Algorithmic Trading

Algorithmic trading is a computer-based approach to execute buy and sell orders on financial instrument such as securities (e.g. stocks, bonds, and options). Financial traders exercise investment strategies using autonomous high-frequency algorithmic trading by real-time market events. As a result, algorithmic trading is dominating financial markets now and accounts for over 70% of all trading in equities [51]. To take advantage of the timely market information, the algorithmic trading engine must be able to respond quickly. Existing pure software solutions are no longer able to provide low latency solutions. There is a need for hardware acceleration for the algorithmic trading engine.

Reconfigurable hardware is a highly desirable platform for an algorithmic trading engine. An FPGA accelerated low-latency market data feed processing engine is presented which is able to process up to 3.5M messages per second [52]. An implementation of "Participate" algorithms for trading equity orders in reconfigurable hardware is presented and shows a 133 times speedup over a software implementation [53]. An analysis of using run-time reconfiguration of reconfigurable hardware to modify trading algorithms is also presented [54]. An event processing hardware is described in [55]. This work described a soft-processor-based architecture, a hardware architecture and a hybrid architecture. An end-to-end latency comparison shows that the hybrid architecture is 10 times faster than the software based solution. An FPGA implementation of a low-latency financial feed handler is presented and has a deterministic latency of $2.7\mu s$ while the CPU-based design has a non-deterministic latency (due to the operating system layer) of $38 \pm 22\mu s$ [56].

2.4 Computational Devices

This section presents the basic information for different computational devices including CPU, GPU and FPGA. A general comparison between these devices is shown in Table 2.2. The throughput and energy consumption between these devices are application dependent. Therefore, we made every effort to obtain a fair comparison between these devices in different aspects of financial computing in this thesis. Based on case studies and experiment results, the performance and energy consumption

comparisons are shown in each chapter.

	CPU	**GPU**	**FPGA**
Clock rate	high	high	low
Power consumption	high	high	low
Parallelism	low	high	high (depends on the size of FPGA)
Pipelining	low	medium	high
Reconfigurability	low	low	high
Instruction set	fixed	fixed	flexible
Floating-point precision	double / single	double / single	flexible

Table 2.2: A general comparison between different computational devices.

2.4.1 CPU

Central Processing Units (CPUs) were the most common processing devices. It was invented as a single core at the beginning. The processing power was increased by increasing the maximum frequency for each new generation. Since the heat generated by the high frequency have reached to a maximum threshold and the transistor sizes have reduced greatly in recent years, multi-core CPU architectures such as Intel Core2 have been developed. The multi-core CPUs use the shared memory for communication, and are synchronised through shared cache. Each thread is processed in one core at a time (or two threads are processed in one core simultaneously in hyper-threading design).

The computational algorithms are stored as a program and executed by CPUs in four main steps: fetch, decode, execute, and writeback. The instruction is fetched from the program memory to determine what the CPU should do. The instruction is than decoded to the opcode (operation type) and operands (memory location, value or other additional information). Then the instruction is executed and the result is stored in registers or memory. A management and scheduling unit in CPU is used for branch prediction, instruction ordering and execution. Although the clock rate of CPU is high, memory access and the execution cycles are often the bottlenecks. The power consumption of CPU is also high due to the high clock rate.

2.4.2 GPU

Graphics Processing Units (GPUs) are special processors that accelerates graphic processing with high memory bandwidth. They traditionally reside on a graphics card such as NVIDIA GeForce or ATI Radeon series and are dedicated to floating point operations. The GPUs dedicate most of the silicon area for floating point unit which include texture, scalar and vector processors for graphics computations. As a result, massive instruction-level parallelism can be achieved. Also, thread-level parallelism is used to hide latency. Threads are grouped as warps and executed in batches. Because of the large number of floating point processing units, GPUs are used to accelerate floating point applications [57] [58]. The clock rate and power consumption of GPUs are relatively high.

General-purpose computing on graphics processing units (GPGPU) is the technique of using a GPU, which typically handles computation only for computer graphics, to perform computation in other general applications. It is becoming more popular because of the application programming interface (API) and the programming language is becoming less complex for general application development.

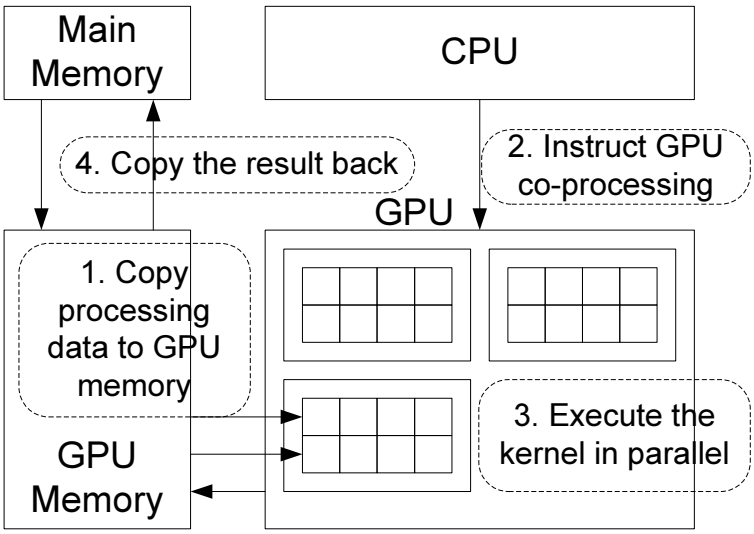

Figure 2.3: A typical CUDA co-processing flow.

Compute Unified Device Architecture (CUDA) is developed by NVIDIA to enable developer to use "C" like programming language to write a computing kernel and gain access to the memory and computational elements of the GPUs. A typical CUDA co-processing flow involve 4 steps and is shown in Fig. 2.3:

- Copy processing data to GPU memory from main memory of the host.
- Instruct GPU to start processing.
- Wait till the threads inside GPU finished executing the kernel in parallel.
- Copy the result back to main memory.

Under CUDA, a function can be compiled into a "kernel". Each computation grid consists of a grid of thread blocks. The "kernel" is executed by all threads in parallel. Each block has a unique ID; so has each thread. Fig. 2.4 shows the organization of the CUDA computation grid.

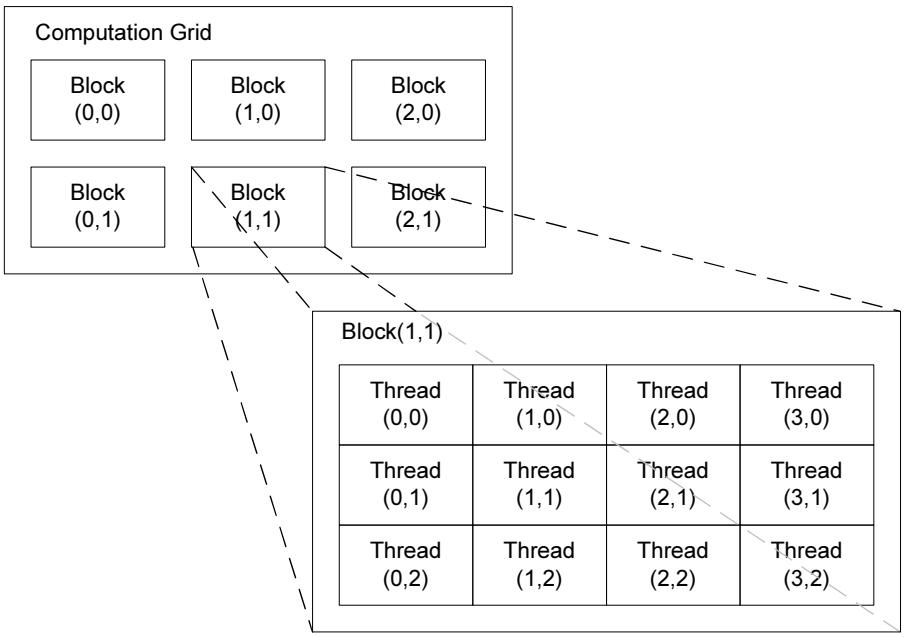

Figure 2.4: Diagram for the CUDA computation grid.

OpenCL (Open Computing Language) is another popular choice for GPU programming when the target GPU is not from NVIDIA. OpenCL provides a common language, programming interfaces, and hardware abstractions enabling developers to accelerate applications with task-parallel or data-parallel computations in a heterogeneous computing environment consisting of the host CPU and any attached OpenCL devices [59]. It is an open standard that can be used to program CPUs, GPUs, and other devices from different vendors, while CUDA is specific to NVIDIA GPUs. It has been adopted by Intel, AMD, NVIDIA, and ARM.

There have been much research on the comparison between CUDA and OpenCL. Since OpenCL is a portable language for GPU programming, its generality may result to a performance penalty. It has

been shown that the performance of data transferring and kernel execution is faster using CUDA than OpenCL when two implementations are running in nearly identical code [60]. It has been shown that CUDA performs at most 30% better than OpenCL in most benchmark applications. However, OpenCL can achieve similar performance to CUDA after some manual tuning to the OpenCL code [61], .

2.4.3 FPGA

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing. The configuration is usually specified using a hardware description language (HDL). The HDL code is then synthesized, placed and routed according to the FPGA vendor tools to generate a bit-stream file in order to configure the FPGA device. The ability to update the functionality or partial re-configuration of the portion of the design makes FPGA an attractive alternatives to application-specific integrated circuit (ASIC) as FPGA has a lower non-recurring engineering costs.

FPGAs contain programmable logic components called “logic blocks”, and routing components for the interconnection of the logic blocks. Figure 2.5 shows a general architecture of an FPGA.

Modern FPGAs contain fixed high-level functionality blocks such as multipliers, generic DSP blocks, embedded processors, high speed I/O logic and embedded memories. The inherent parallelism of the logic resources on FPGAs allows a high computational throughput even at a low clock rates. The computation on FPGA can be completely pipelined, which enhances the throughput significantly. FPGA has been commonly used in communication, networking and video encoding applications [9, 62, 10].

Many publications have reported that fined-grained parallelism based on FPGAs can result in outstanding performance over traditional general-purpose processors. Example of applications include cryptography [63, 64, 65], the computation problem SAT [66, 67], medical [12, 11] and physics [68, 69]. Hence, high performance computing with FPGAs is becoming a popular research topic.

The flexibility of bit-width for fix-point and floating-point operations offers an additional performance

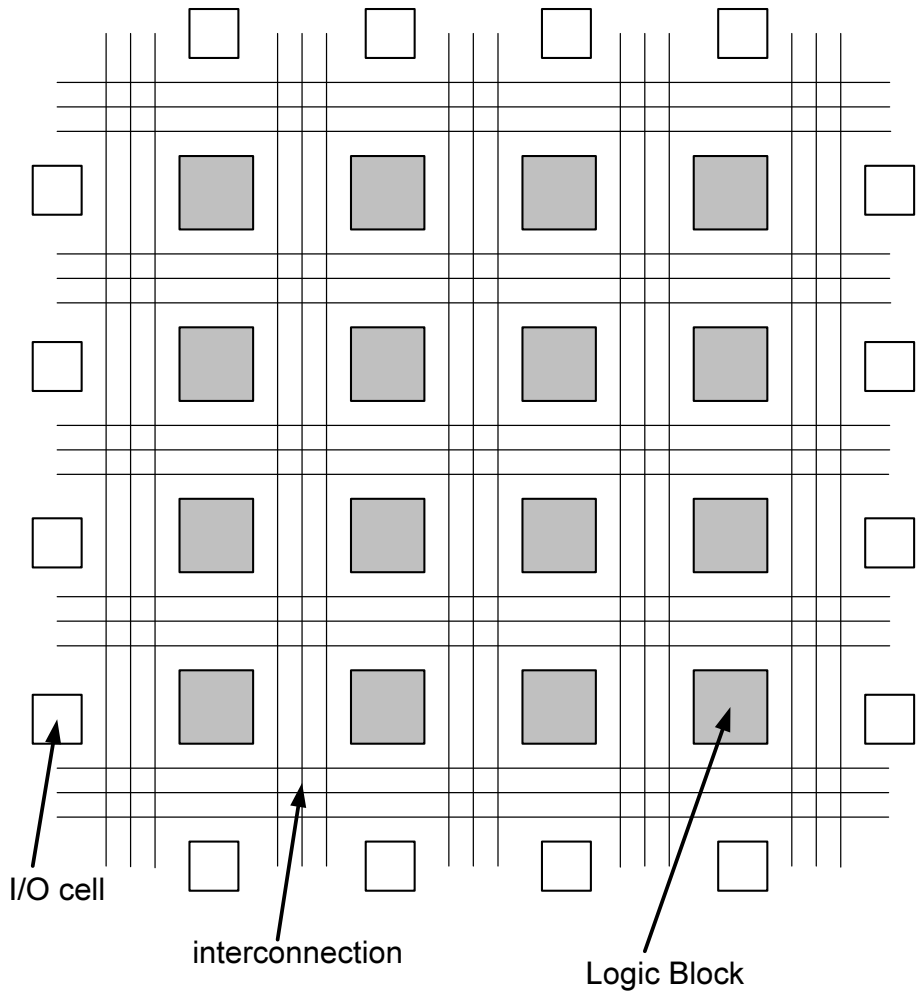

Figure 2.5: A general architecture of an FPGA.

gain opportunity. The related works will be presented in Section 2.5.

2.5 Bit-width Optimisation with Reconfigurable Hardware

FPGAs provide customisable floating-point number operation which could be exploited to provide additional speedup. Reduced precision floating-point operators usually have higher clock frequencies, consume fewer resources and offer a higher degree of parallelism for a given amount of resources compared with double precision operators. However, the use of reduced precision affects the accuracy of the numerical results.

Finite precision error ϵ_{fin} is the error due to non-exact floating-point arithmetic. Floating-point num-

ber representation in computer has a finite significant bit-width. The rounding of the intermediate or final result leads to precision. Decreasing the bit-width of the floating-point number representation generally leads to a larger finite precision error ϵ_{fin} and decreases the accuracy of the result.

The benefits for reduced precision designs are well-known. For instance, it has been shown [70] that appropriate word-length optimisation can improve the area of adaptive filters and polynomial evaluation circuits by up to 80%, power reduction of up to 98%, and speed of up to 36% over common alternative design strategies. Therefore, how to perform bit-width optimisation has been an important research issue.

One common approach is to develop an accuracy model which relates output accuracy with the precisions of the data formats being used in the data-path. The area and delay of data-paths with different precisions both are modeled and combined with the accuracy model. The design with a minimum area-delay product can be obtained from the models. Common accuracy modeling approaches include simulation approach [71], interval arithmetic [72], backward propagation analysis [73], affine arithmetic [74] [75] [76] [77], SAT-Modulo theory [78] and the polynomial algebraic approach [79]. More recently, a mixed-precision methodology is presented and shows an additional performance gain of 7.3 times over the original FPGA-accelerated collision detection algorithm [80].

2.5.1 Bit-width Optimisation of Monte-Carlo Method

Methods for dealing with finite precision error in FPGA-based Monte-Carlo simulations can be classified into two categories. In the first category, only standard precisions such as the IEEE single/double precision are used in sampling data-paths [81, 82]. Users are responsible for determining whether the finite precision error is acceptable, because the FPGA Monte-Carlo engines will follow the result of software exactly.

In the second category, error bounds of the finite precision error are constructed and the precision of the sampling data-path is adjusted such that the error bounds are smaller than the error tolerance. In [83], the maximum relative error of the sampling data-path is used to construct the error bound. The maximum relative error can be characterised using analytical methods such as interval [84] or

affine arithmetic [85]. However, these approaches do not take into account that finite precision errors from different sample points might have different signs and would cancel out each other. Hence there is usually an over-estimation of finite precision error in Monte-Carlo simulation.

In [86], test runs with a pre-defined number of sample points are used to figure out the maximum percentage error due to finite precision effect empirically. The finite precision error of MC simulations using the same data-path and the same number of sample point are then assumed to share the same error bound. Such assumption may not be valid and thus the empirical error bound can only be used as a reference rather than a rigorous bound.

In [87], a design is proposed with both high precision and reduced precision data-paths used in computing cumulative distribution functions (CDFs). The two CDFs are compared using a Kolmogorov-Smirnov test, the distance score of which is then used to control the precision of the reduced precision data-path adaptively such that finite precision error is within the range of error tolerance.

In Chapter 6, we proposed a mixed precision methodology in Monte-Carlo option pricing to correct the finite precision error instead of passively estimating the error bound as other research. Also, in Chapter 7, we proposed a reduced precision optimisation methodology by trading off both the precision and the integration grid density to obtain the optimal throughput.

2.6 Multi-Accelerator Heterogeneous Cluster

Domain specific processors with specialized instructions or logic blocks usually outperform traditional CPUs due to their more efficient use of silicon area and higher hardware parallelism. So it is common to see Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) used as accelerating co-processors in high performance computing (HPC) systems.

Techniques from distributed computing have been a solution for HPC for many years. The computation task in an application is decomposed into smaller tasks which are performed by computing nodes which communicate through a network.

A multi-accelerator heterogeneous cluster is a cluster consists of multiple different types of accel-

erators or computational devices (e.g. FPGAs and GPUs). It is very different from a homogeneous cluster which consists of the same type of computational resources only (e.g. CPUs only). Apart from using accelerators for application acceleration, one can combine accelerators to perform distributed computing in a heterogeneous cluster to further improve the performance of the application. However, there are still some key challenges when building practical applications for a multi-accelerator heterogeneous cluster.

The first challenge is the difference in programming models and difference in tools between conventional software programming and these hardware accelerators. Having different types of accelerators within the system makes the situation even more complex as they communicate with the CPU in different ways. This complicated application structure and the high non-recurring engineering (NRE) cost per application become the major barriers when utilizing heterogeneous clusters.

The second challenge is the different types of hardware accelerators are usually customized for specific computation and communication patterns. Thus the performance of them will vary from application to application. Some accelerators may outperform others in computational speed while some accelerators may consume less energy. How to efficiently schedule the tasks for different accelerators is one of the challenging problems. On top of this is the synchronization and data transfer overhead, which increases the uncertainty of the overall achievable performance.

The third challenge for a distributed HPC system is to distribute tasks efficiently. The overhead will become dominant as the number of tasks increases, according to the law of diminishing returns. The communication between distributed tasks will contribute to the overhead. This suggests that applications with a large number of divisible tasks, and a small number of inter-task communications will benefit the most.

Clusters with FPGAs as accelerators have been studied and developed in both academic and industrial fields. In 2004, the Cray XD1 computer [88] achieved 58 GFlops with 12 Opteron CPUs and 6 Xilinx Virtex-II FPGA devices on a single motherboard. In 2007, a cluster with 64 Virtex-4 FPGA devices was built in the Maxwell project [89]. Each FPGA in the Maxwell cluster can achieve up to 2.5 times speedup in a face recognition application when compared with the software implementation.

Clusters with GPUs often have a larger number of floating point units and higher operating frequency than FPGA devices can support. The programming interface of GPU devices, such as CUDA [90], also helps to promote their popularity in HPC systems. In 2008, an updated version of the TSUBAME (Tokyo-tech Supercomputer and Ubiquitously Accessible Mass-storage Environment) system [91], achieved 56.43 TFlops for solving dense linear equations. In addition to custom vector processors, this supercomputer is also equipped with 170 nVidia Tesla C1070 cards. In [92] the authors studied the performance of a GPU cluster, which is 2.8 times faster and consumes 28.3 times less energy than a CPU cluster.

In 2009, the Quadro Plex (QP) Cluster [93] was built by NCSA in UIUC. For each of the 16 nodes in the QP prototype, there are two AMD Opteron CPUs, four nVidia G80GL GPUs and one Xilinx Virtex-4 LX100 FPGA. This system may achieve 23 TFlops (single precision) theoretically. In 2010, the Axel cluster [94] from Imperial College London demonstrated the collaboration between heterogeneous accelerators. With Xilinx Virtex-5 FPGAs and nVidia C1060 GPUs working together, this 16-node cluster achieved over 22 times speedup in a N-body simulation application over a 16-node CPUs only cluster.

However, no publication has addressed the issue of how to automatically allocate and adjust the workload balance between different accelerators and how to optimise the workload allocation for a pre-defined objective. Seeing the potential speedup with multiple accelerators work collaborative together, we proposed a scalable framework with dynamic scheduling to provide automated management for collaborative financial computing in Chapter 5.

2.7 Hardware Description Language

Hardware Description Languages (HDL) are programming languages that are designed to program FPGAs. The most common and primitive HDLs are VHDL and Verilog. They are full-featured languages that support synthesis and simulation. However, they are considered to be relatively hard to learn. For application development, algorithms have to be broken down into smaller hardware component. The input, output and intermediate states of the component in each clock cycle are

described explicitly. The connections between components are also described explicitly. Therefore, they are very different from the typical sequential software languages like “C” or “C++”.

Handel-C is a behavioral language for FPGA design and is based on the ANSI-C programming language. Handel-C is a superset of the ANSI-C language and contains additional constructs in exploiting and abstracting complexities present when programming to hardware. A Handel-C program requires that a clock construct be used. Often, this is set to the clock rate of the target device. Groups of statements may be encased within PAR and SEQ code blocks, indicating that the statements should execute in parallel (in the same clock cycle) or in a sequence (one after the other), respectively. The Handel-C specification also introduces the idea of channels: a link between parallel branches of code to allow intercommunication [95].

HyperStreams is a high-level abstraction language and library in Handel-C. It can produce a fully-pipelined hardware implementation with automatic optimization of operator latency at compile time. This feature is useful when implementing a complex algorithm core [96].

The Data Stream Manager (DSM) was designed by Celoxica to enable OS-independent hardware/software co-design between applications written in C/C++ on a microprocessor host and Handel-C on a reconfigurable hardware target [97]. Simply put, it is an API that makes it easy for programmers to have their C or C++ applications communicate with Handel-C code running on an FPGA coprocessor, thereby allowing data movement between C/C++ applications and reconfigurable hardwares. However, we must declare and initialize DSM interfaces in both the hardware and software sides and specify when to move data in and out during the design phrase in a low-level perspective.

MaxCompiler is a high-level tool developed by Maxeler Technologies for application acceleration on a Maxeler FPGA system [98]. The FPGA is configured with one or more hardware kernels and a manager. The computation intensive part of the application is written in Java language following Maxeler API and compiled as a hardware kernel. The kernel adopts a streaming programming model and supports customisable data formats.

There are also much research on the integration frameworks between reconfigurable hardware and software design. An IGOL (Imaging and Graphics Operator Libraries) framework is proposed for

developing reconfigurable data processing application [99]. A middleware platform is built using reflective component model [100]. A design methodology is presented which enables designers to combine cycle-accurate descriptions with behavioral descriptions [101]. A framework for developing FPGA-based configurable computing machines application is discussed [102]. A high-level component-based methodology and design environment for application-specific multicore SoC architectures is presented [103]. *Gezel* language is introduced for an electronic system level design flow which supports abstraction and reuse [104]. A parallel programming library is described to transform C# parallel programs into circuits for realization on FPGAs [105].

2.8 Summary

This chapter provides the background knowledge and related works in financial computing and reconfigurable computing for this thesis. The background knowledge of option pricing including the option pricing model and the examples of exotic options is presented in Section 2.1. Numerical methods used in option pricing including tree-based methods, finite-difference methods, Monte-Carlo methods and quadrature methods, and the corresponding related works are presented in Section 2.2. The background knowledge and related works of algorithm trading using reconfigurable devices are presented in Section 2.3. The differences and strengths of different computational devices are discussed in Section 2.4. The previous works on bit-width optimisation using FPGA and on cluster computing involving accelerators are presented in Section 2.5 and Section 2.6 respectively. At last, the background knowledge of hardware description languages is presented in Section 2.7.

Chapter 3

Accelerating Monte-Carlo Methods for Option Valuation

3.1 Motivation

Financial analysis and pricing applications are often computationally intensive, so there has been much interest in FPGA-accelerated option pricing. Numerical techniques (lattice and Monte-Carlo methods) are used for option valuation when there is no closed-form solution. Lattice methods implemented in FPGAs include binomial trees [36]. Such algorithms are generally more efficient than Monte-Carlo methods, but they cannot easily handle more complex features, such as the path-dependence found in some exotic options (e.g. Asian options).

Monte-Carlo methods are particularly suitable for implementation in FPGAs, as they contain abundant parallelism. Early FPGA-accelerated Monte-Carlo application includes the simulation of BGM interest rate model [16]. More recent work has focused on considering more complex types of Monte-Carlo simulation, such as American exercise features [45]. However, none of the previous works explored the use of control variate technique. Control variate is one of the variance reduction techniques which aims at reducing variance as well as the computation time for a given accuracy for Monte-Carlo simulation [49, 106]. This chapter explores the control variate Monte-Carlo method in

FPGAs for generic exotic option pricing.

The contributions in this chapter are:

- A parallel hardware framework using the control variate Monte-Carlo method for pricing exotic options.
- A detailed hardware design of arithmetic Asian option pricing using both control variate Monte-Carlo method and pure Monte-Carlo method under this framework.
- Evaluation of the FPGA and GPU implementations versus a multi-threaded software implementation on Intel Xeon 2.5GHz CPU, showing 24 times speedup for the FPGA, and 10 times for the GPU. We also explored the trade-off of the accuracy gain versus the parallelism reduction when using control variate Monte-Carlo method instead of pure Monte-Carlo method. The chapter shows that using control variate Monte-Carlo method is 2 times faster than using pure Monte-Carlo method in FPGA for a given confidence interval (accuracy).

3.2 Parallel Hardware Architecture for Exotic Options Pricing

As discussed in Section 2.1, under Black-Scholes model, the stock price movement is governed by a geometric Brownian motion process, and the stock price is given by Equation 2.6:

$$S_{i+1} = S_i e^{((r-\frac{\sigma^2}{2})\delta t+\sigma\sqrt{\delta t}W)}$$

where r is the interest rate, σ is the volatility of the underlying stock price, δt is the time period between two time steps, W is a Gaussian random number $\sim \mathcal{N}(0,1)$, S_i is the underlying stock price at step i and S_{i+1} is the underlying stock price at step $i+1$. We could define the following equations:

$$drift = \left(r - \frac{\sigma^2}{2}\right)\delta t \tag{3.1}$$

$$vsqrdt = \sigma\sqrt{\delta t} \tag{3.2}$$

such that

$$S_{i+1} = S_i e^{drift + vsqrdt \times W} \tag{3.3}$$

The values of $drift$ and $vsqrdt$ can be precomputed in advance, so that the stock price can be simulated with these two static values.

To simulate the target exotic option price using control variates, a control option price is also computed with the same set of stock price path. The statistical result (the variance of the control option payoff and the covariance between target and control option payoff) is required for the final adjustment. A one-pass variance and covariance computation method is used with the following equations:

$$Var(x) = E(x^2) - E^2(x) \tag{3.4}$$

$$Cov(x, y) = E(xy) - E(x)E(y) \tag{3.5}$$

Therefore, the control variate Monte-Carlo option pricing algorithm is designed as in Algorithm 1. We define *t_temp* and *c_temp* to be the temporary updating variables for the target and control option payoffs calculation. They are initialized by option specific initialization functions *init()* at the beginning of each path simulation (line 8-9). They are then updated with the corresponding updating functions *update()* after each step of stock price movement (line 13-14). The corresponding option payoffs of the target and control options are calculated by their option specific functions *calculate()* after a completed path is simulated (line 16-17). The sum of the target option payoff (*t_sum*), the sum of control option payoff (*c_sum*), the sum of the square of control option payoff (*c2_sum*) and the sum of the target option payoff times control option payoff (*tc_sum*) are accumulated correspondingly (line 18-21).

In the final stage of the algorithm, the simulated target option payoff, control option payoff, variance of the control option payoff, covariance of target and control option payoff are computed from *t_sum*, *c_sum*,*c2_sum* and *tc_sum* (line 23-28). The true value of the control option payoff is calculated with the closed-form equation (line 29). This closed-form equation depends on which type of control

Algorithm 1 Control variate Monte-Carlo pricing algorithm

1: (Let o be all the option parameters)
2: *t_sum* = 0 //target option payoff sum
3: *c_sum* = 0 //control option payoff sum
4: *c2_sum* = 0 //square of control option payoff sum
5: *tc_sum* = 0 //target times control option payoff sum
6: **for** $i = 1$ to N_{mc} **do**
7: $S = S_0$
8: *t_temp* ← $init_t$(*t_temp,o*)
9: *c_temp* ← $init_c$(*c_temp,o*)
10: **for** $i = 1$ to *Steps* **do**
11: W ← *NextRandomNumber()*
12: $S \leftarrow Se^{drift+vsqrdt\times W}$
13: *t_temp* ← $update_t$(*t_temp,S,o*)
14: *c_temp* ← $update_c$(*c_temp,S,o*)
15: **end for**
16: *t* ← $calculate_t$(*t, t_temp,o*)
17: *c* ← $calculate_c$(*c, c_temp,o*)
18: *t_sum* ← *t_sum* + *t*
19: *c_sum* ← *c_sum* + *c*
20: *c2_sum* ← *c2_sum* + c^2
21: *tc_sum* ← *tc_sum* + *c* × *t*
22: **end for**
23: $E(t)$ ←(*t_sum* / N_{mc})
24: $E(c)$ ←(*c_sum* / N_{mc})
25: $E(c^2)$ ←(*c2_sum* / N_{mc})
26: $E(tc)$ ←(*tc_sum* / N_{mc})
27: *Var(c)* ← $E(c^2) - (E(c))^2$
28: *Cov(t,c)* ← $E(tc) - E(t)E(c)$
29: *True(c)* ← $control\_option\_true\_equation(o)$
30: *adjustment* ← $-\frac{Cov(t,c)}{Var(c)}(E(c) - True(c))$ (Using Equation 2.13)
31: E_{cv}*(t)* ← $E(t) + adjustment$ (Using Equation 2.13)
32: *TargetOptionPrice* ← $e^{-rT}E_{cv}$*(t)*

option is used. If an European option is used as an control option, the closed-form equation will be the Black-Scholes formula [31]. The final target option price is then obtained with the control variate adjustment (using the Equation 2.13 as presented in Section 2.2.2) and discounted backward to present time (line 30-32).

Different types of exotic options have different *init()*, *update()* and *calculate()* function. Table 3.1 shows these functions content for some exotic and European options.

We therefore present our overall hardware design architecture as Fig. 3.1. There are two main types

Table 3.1: The *init()*, *update()* and *calculate()* function for some example options

	init(x,o)	*update(x,S,o)*	*calculate(y,x,o)*
Arithmetic Asian call options	$x \leftarrow S_0$	$x \leftarrow x + S$	$y \leftarrow Max(0, x/(steps+1) - K)$
Geometric Asian put options	$x \leftarrow S_0$	$x \leftarrow x \times S$	$y \leftarrow Max(0, K - x^{1/(steps+1)})$
Fixed strike lookback call options	$x \leftarrow S_0$	$x \leftarrow Max(x, S)$	$y \leftarrow Max(0, x - K)$
Up and out barrier call options	$x_1 \leftarrow 0$; $x_2 \leftarrow S_0$	*if* $S > B, x_1 \leftarrow 1$; $x_2 \leftarrow S$	*if* $x_1 = 1, y \leftarrow 0$, *else* $y \leftarrow Max(0, x_2 - K)$
European options	$x \leftarrow S_0$	$x \leftarrow S$	$y \leftarrow Max(0, x - K)$

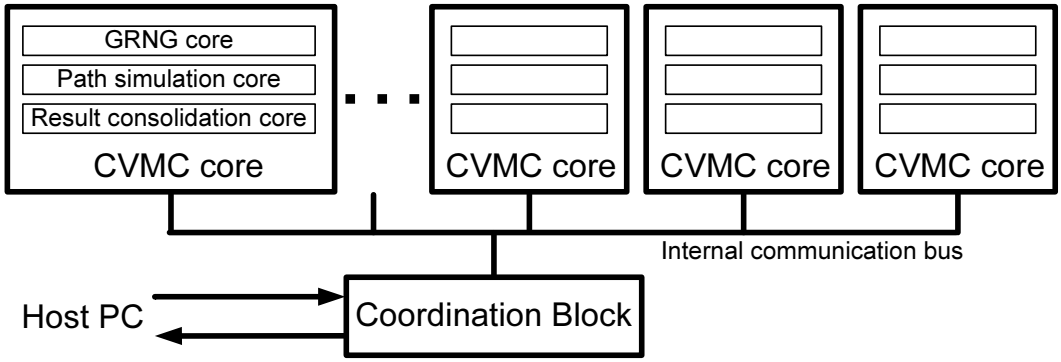

Figure 3.1: Overall hardware architecture.

of components in the design: one or more identical control variate Monte-Carlo (CVMC) cores; and a single shared Coordination Block (CB). The CVMC cores contain a Gaussian random number generator (GRNG) core, a path simulation core and a result consolidation core; each CVMC core is capable of generating random asset price paths, calculating payoffs of the target option and control option, and accumulating the payoffs and payoffs related statistical result. In other words, each CVMC core is capable of executing the main for-loop in Algorithm 1. Multiple identical CVMC cores are instantiated to make maximum use of the device. The total number of simulations required is distributed equally to each CVMC core.

The block diagrams of the path simulation core and result consolidation core inside the CVMC core are shown in Fig. 3.2. The logic for *update()* and *calculate()* function of *t_temp*, *c_temp*, *t* and *c* are located inside their corresponding block of path simulation core. As pipelined operators are used and the output of all "update blocks" and "sum blocks" will be fed back to the input of themselves, there is a pipelined loop for each of the "update block" and "sum blocks". The number of the pipelined stages must be identical for all the pipelined loops in order to guarantee a consistent computation schedule. Let p be the maximum number of pipeline stages for these pipelined loops. Pipelined registers are added to ensure the number of pipelined stages of all loops equal to p. As the feedback result will

reappear only after p stages, we simulate a batch of p paths at the same time in this pipelined fashion.

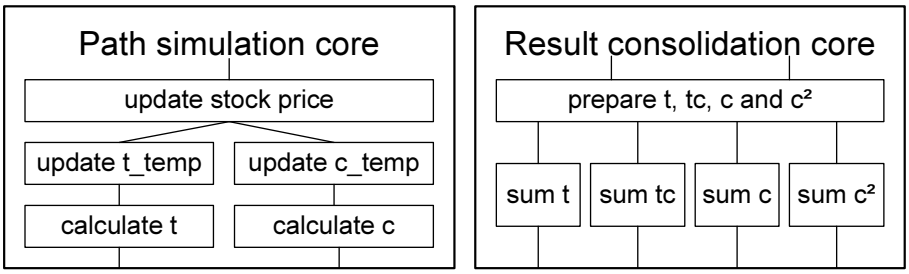

Figure 3.2: Block diagram of path simulation core and result consolidation core.

The number of pipeline stages for all "calculate blocks" must be the same as well to guarantee the valid results of t and c arrive at the result consolidation core at the same cycle. The "calculate blocks" output only when that p simulations reaches the end of the path (i.e. S reached S_n and n is the total number of steps). Therefore, p path simulations are completed every $p \times steps$ cycles. These t and c results are used in result consolidation core until the completed number of batches reaches the required number of batches. Let N_{batch} be the required number of batches, N_{mc} be the required number of Monte-Carlo simulations and C be the number of CVMC cores in the hardware. N_{batch} is defined as:

$$N_{batch} = \left\lceil \frac{N_{mc}}{p \cdot C} \right\rceil \tag{3.6}$$

The Coordination Block (CB) manages the CVMC cores, allowing them to work in parallel to price the same option. The CB is also responsible for communicating with the external controller, for example a PC. With the precious timing control, all the *t_sum*, *c_sum*, *c2_sum* and *tc_sum* computed in CVMC cores will be sent back to CB, and then transfer to the external host for the final post-processing. The Gaussian random number generators in CVMC cores are also initialized by the CB. Different sequences of bits are connected to different Gaussian random number generators as the random seeds.

3.3 Case Study: Asian Options Pricing

In this section, we present the detailed FPGA design of CVMC arithmetic Asian option pricing using our designed hardware architecture. There is no closed-form solution for arithmetic Asian options and

pricing them fast and accurately is a challenging problem in finance. Therefore, arithmetic Asian options are perfect candidates to be priced using the hardware accelerated CVMC framework. European options are chosen as control options, as they have a closed-form solution.

For an arithmetic Asian option [33], the payoff is calculated using the arithmetic average of the prices over the life time of the option. One advantage of this option type is that it is more difficult for the option issuer to manipulate market prices to reduce the option payoff, as the payoff depends on the path followed by the asset price, not just the price at expiry.

The payoff of an arithmetic Asian call option is introduced in Chapter 2 Section 2.1 in Equation 2.8 as:

$$P_{call} = \max\left(\frac{1}{n+1}\sum_{i=0}^{n} S_i - K, 0\right)$$

where $S_0, ..S_n$ are the asset price at time step $0...n$.

A common assumption is that asset prices move according to a log-normal random walk. Under this model, price of an European option at present time can be calculated with a closed-form solution called the Black-Scholes Equation [31]. However, there is no such solution for arithmetic Asian options, due to their highly path-dependent properties. Monte-Carlo methods are commonly used to solve this problem.

3.3.1 FPGA design: CVMC core

Gaussian random number generator

Random Number Generators (RNGs) are a key component of any Monte-Carlo simulation, as they provide the underlying stochastic factor that allows the average behaviour to be explored. For this reason it is critical that the RNGs produce a high-quality stream of random numbers: the numbers must appear independent, with no correlations or patterns in the sequence; and the statistical distribution must be indistinguishable from the target distribution, in this case the Gaussian distribution.

These independence requirements are particularly important in accelerated Monte-Carlo, where 2^{30} random samples can be generated and consumed in one second. Any correlations or biases can easily distort the overall results of the simulation, so the period of the RNG must be 2^{128} or more.

This work uses the piecewise linear generation method [107], which provides high-quality fixed-point Gaussian samples, while using only a small amount of logic and block-RAMs. A particular advantage of the method is that it does not use any DSP blocks, freeing these up for the use in the path generation and payoff logic. To provide a good approximation to the Gaussian distribution, two independent piecewise linear RNGs are used, both of which provide a good approximation to the Gaussian distribution. The outputs of the generators are then added together, providing a better approximation to the Gaussian distribution, due to the Central Limit Theorem.

The resulting Gaussian RNG uses a single block-RAM and around 600 slices, and produces a stream of 24-bit fixed-point random numbers, with a period of 2^{128}. The quality of the stream has been checked with the Chi-squared test for sample sizes up to 2^{32}, and shows no significant deviation from the Gaussian distribution.

Path simulation

The *init()*, *update()* and *calculate()* functions of arithmetic Asian call options and the controlling European options are shown earlier in Table 3.1. Therefore, the architecture of the path simulation core is designed as in Fig 3.3.

The static input parameters include S_0, K, $vsqrdt$, $drift$ and $steps$ (number of simulation steps). The dynamic input parameter is the Gaussian random number W. The underlined parameter near each operator is the number of pipeline stages (latency) of that operator. Therefore, it takes $d_a + d_m + d_e$ clock cycles for W to reach the second multiplication operator.

There are 3 multiplexers namely MUXA, MUXB and MUXC controlling the computation flow. MUXA selects S_0 at the beginning in order to calculate the S_1 price. The signal *s_path* indicates the updated S in the path and is feed back to MUXA. Therefore, MUXA selects signal *s_path* afterward to provide a loop for iterating next S_i. The loop containing MUXA and the second multiplication

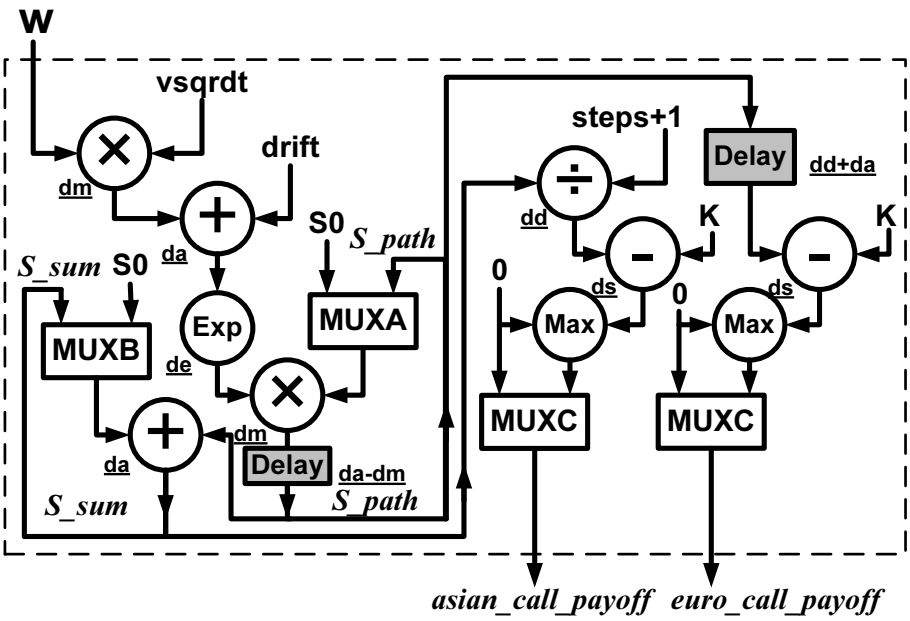

Figure 3.3: Architecture of the price movement path simulation core.

operator is the "stock price updating loop".

MUXB selects S_0 at the beginning in order to compute the sum of price S_0 and S_1 to the result signal *S_sum*. *S_sum* is then feed back to MUXB to form the "sum of price updating loop". MUXB selects *S_sum* afterward after the S_0 + S_1 computation.

The number of the pipelined stages must be identical for all the pipelined loops in order to guarantee a consistent computation schedule. Let p be the maximum number of pipeline stages for these pipelined loops. Pipelined registers are added to ensure the number of pipelined stages of all loops equal to p. In this architecture, $p = d_a$. Therefore, a $d_a - d_m$ cycles pipelined delay register is inserted after the second multiplication operator for balancing.

As the computation is pipelined, the feedback result will reappear at the MUXA and MUXB after p cycles. Therefore, we simulate p paths at the same time in this pipelined fashion. The computed 1 step of S for the first simulation will arrive MUXA and MUXB just after the other $p-1$ computations of the other simulations.

MUXC selects the output of max operator only when that p simulations reached the end of the path (i.e. *S_path* reached S_n). Therefore, MUXC only selects the max operator output for p cycles as the *asian_call_payoff* at that moment, and selects value 0 for the rest of time. In conclusion, p path simulations are completed after $p \times steps + 3d_a + d_m + d_e + d_d + d_s$ cycles for the pipeline stages.

The whole process repeats and we could expect another p completed path simulations after another $p \times steps$ cycles. Table 3.2 summarize the behavior of the MUXs in the path simulation core.

Table 3.2: MUXs' behavior in path simulation

MUX	Selecting behavior:
MUXA	Select S_0 for first $d_a + d_m + d_e$ cycles. Repeat: Select S_0 for p cycles and then select *S_path* for $p \times (steps - 1)$ cycles
MUXB	Select S_0 for first $2d_a + d_m + d_e$ cycles. Repeat: Select S_0 for p cycles and then select *S_path* for $p \times (steps - 1)$ cycles
MUXC	Select 0 for first $3d_a + d_m + d_e + d_d + d_s$ cycles. Repeat: Select 0 for $p \times (steps - 1)$ cycles and then select max() output for p cycles

Result consolidation

The architecture of the result consolidation core is shown in Fig. 3.4. As discussed in the previous subsection, a batch of p payoff results (Asian call payoff and European call payoff) are generated for every $p \times steps$ cycles and passed to the result consolidation core. The product of Asian and European call payoff, and the square of the European call payoff are computed with two multipliers. These results are accumulated until the completed number of batches reaches N_{batch}. Two 8 stage delay registers are inserted to balance the timing with these two multipliers.

All MUXD type multiplexers select 0 for initialization, then they select the accumulated result (e.g. signal *a_sum*) afterward to form "sum of result loop". When the number of batches reached N_{batch}, we have to aggregate the final p consecutive sum of result (e.g. *a_sum*) together. It is a bit complicated to aggregate these p consecutive values using p stages pipelined adders. The solution is to make use of multiplexers (MUXE) and registers with a special clock-enable timing. All MUXE type multiplexers select the output of the delays or multipliers at the beginning, then they select the output of the D-type register afterward. As the input of the D-type register is connected with the output of those adders, these D-type registers and MUXE form another feedback loops. Table 3.3 shows the behavior of MUXs in the result aggregation core with the actual number of cycles.

The clock-enable of the D-type registers are controlled by a special signal sequence "accr" in order to

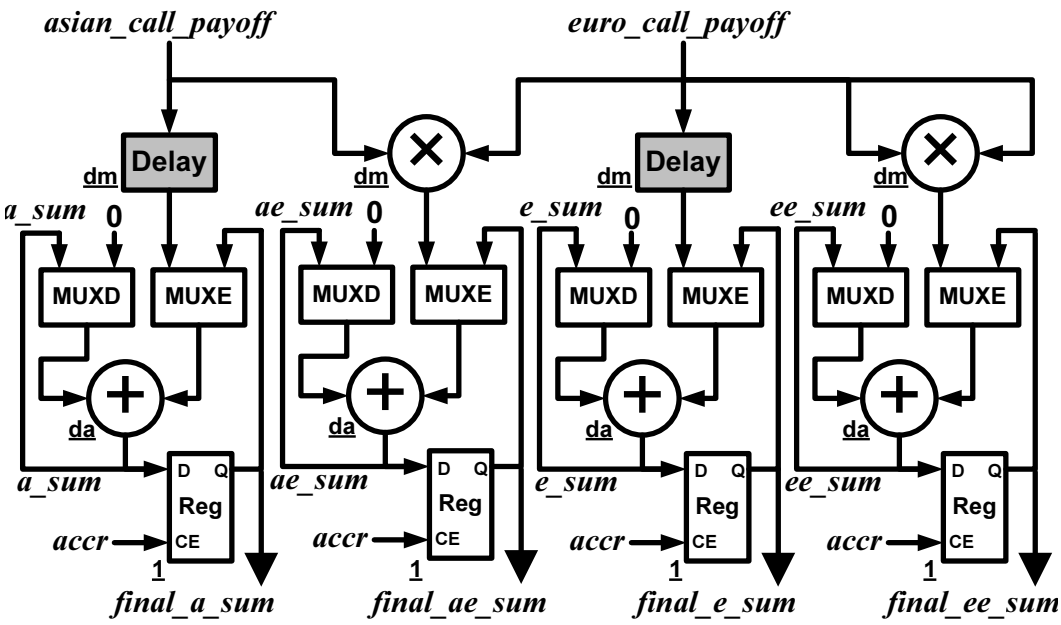

Figure 3.4: Architecture of the result consolidation core.

Table 3.3: MUXs' behavior in result consolidation

MUX	Selecting behavior:
MUXD	Select 0 for $p \times steps$ + $4d_a + 2d_m + d_e + d_d + d_s$ cycles and then select AdderOut afterward.
MUXE	Select result from multipliers or delays for $p \times steps \times N_{batch}$ + $5d_a + 2d_m + d_e + d_d + d_s$ cycles and then select D-type RegOut afterward.

achieve the final accumulation. The "accr" is set to 1 for a dedicated timing so as to buffer the desired intermediate output of the adder. The desired intermediate result stays at the output of the register and the output of the MUXE.

Let x be the index of clock cycle, the sequence of signal "accr" is defined as the following equation:

$$accr(x) = \begin{cases} 1 & \text{if } x \bmod 2^{k+1} = 2^k - 1, \\ & pk \leq x < p(k+1), \\ & \forall k \in N, 0 \leq k \leq \log_2(p) \\ 1 & \text{if } x \bmod 2^k = 2^{k-1} - 1, \\ & pk \leq x < p(k+1), \\ & \forall k \in N, \log_2(p) < k \leq \log_2(p) + 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

The number of cycles required to obtain the aggregated results at register out is $\lceil p(\log_2(p) + 1) \rceil$. As $p = 12$ in this case, the final sum will appear at the output of the D-type registers 56 cycles later.

This register, with a special clock-enable signal sequence, is of general use for a design requiring a "reduce" function in a "map-reduce" computation with commutative operator with any number of pipeline stages. If a multiplier is used as the commutative operator, the result of $\prod_{i=1}^{p} Y_i$ will be computed at the register output instead of $\sum_{i=1}^{p} Y_i$.

3.3.2 FPGA design: Coordination Block

The Coordination Block is the main control unit of the hardware architecture and provides the communication with the host PC. The Asian option parameters are first sent from host PC to the Coordination Block. The Coordination Block then distributes the parameters to all CVMC cores. The communication time between the FPGA and PC is negligible as there are only tens of bytes of input parameters and results transferred between them.

The Coordination Block also controls the overall timing of the computation. It generates 5 types of MUX selection signals and 1 type of "accr" signal sequence to all CVMC cores as discussed in previous subsection. The timing of generating these signals is followed strictly by the requirement as in Table 3.2 and Table 3.3. Instead of implementing counters and finite state machines in each CVMC core, we implement them in the Coordination Block only to reduce logic redundancy.

Path delay optimization

All counters, condition checking and controlling logic are implemented in the Coordination Block only instead of in the CVMC cores. In this way, the logic redundancy is significantly reduced. However, the use of global controlling signals may suffer from a decrease of clock rate due to a long critical path delay. Path delay consists of 2 parts: *logic delay* and *routing delay*. When there are many computational cores, the routing delay of the controlling signal from the Coordination Block to the farthest computational core will be significant, and the performance of a parallel architecture will be drastically reduced. Therefore, the hardware design of the Coordination Block is optimized carefully

Algorithm 2 Monte-Carlo pricing algorithm

```
payoffSum = 0
for i = 1 to NumberOfSimulation do
    SumOfPrice = S_0
    S = S_0
    for i = 1 to Steps do
        W ← NextRandomNumber
        S ← S e^(drift+vsqrdt×W)
        SumOfPrice ← SumOfPrice + S
    end for
    payoff ← SumOfPrice / (Steps+1) - K
    payoff ← max(0,payoff)
    payoffSum ← payoffSum + payoff
end for
Price ← e^(-rT)(payoffSum/NumberOfSimulation)
```

with path delay partitioning. Pipeline registers are inserted in the controlling signal paths to minimize the routing delay. Therefore, a high clock rate can be maintained while maximizing the degree of parallelism.

3.3.3 FPGA design: Pure MC core

A pure MC version of Asian option pricer is also designed for performance comparison. The hardware architecture of the price movement path simulation is based on the pure MC Asian option pricing algorithm (Algorithm 2), which is shown in Figure 3.5.

3.3.4 GPU design

Our implementation on GPU is based on Compute Unified Device Architecture (CUDA) API provided for nVidia GPUs. We design our CUDA implementation of Asian option pricing by 2 procedures, namely Gaussian random number generator procedure and path simulation procedure.

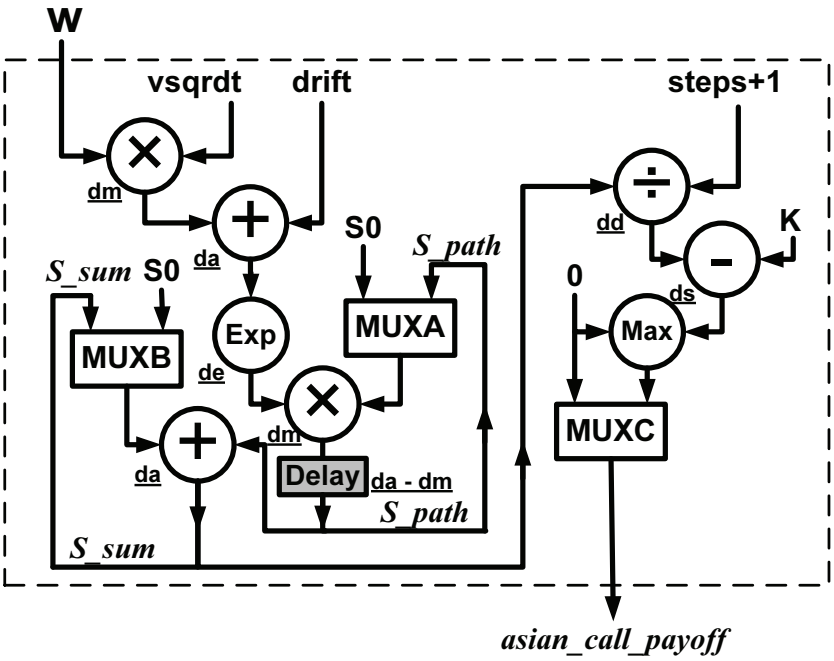

Figure 3.5: Architecture of pure MC path simulation core. The underlined parameter denotes operator latency.

Gaussian random number generator procedure

In this procedure, we first allocate the GPU's global memory space for the total amount of random numbers that we needed for the simulations. If the number of simulations is N, the number of steps is M and single precision is used, we allocate 4NM bytes of global memory in GPU. Then we execute the Mersenne Twister random number generator kernel using all the threads to generate random numbers at the memory space [108]. A Box-Muller transformation kernel is then executed on that memory space to form Gaussian random numbers. The random number generation procedure is optimised by ensuring all threads generate random numbers in the global memory in a fully parallel manner and the reading and writing to the global memory are coalesced.

Path simulation procedure

In the path simulation procedure, each thread simulates the price movement path and sums up the payoff in the shared memory. Therefore, these payoff sums can be accessed by other threads in the same block. The first thread in each block then sums up all the payoff sums within the same block and stored it in the global memory location. Finally, a final aggregation kernel is executed by 1 thread

Table 3.4: xc5vlx330t FPGA resource consumption

	10 CVMC Cores		**16 pure MC Cores**	
Resource	**Used**	**%**	**Used**	**%**
Slices	44,118	85%	41,968	80%
FFs	130,195	62%	110,789	53%
LUTs	79,587	38%	95,749	46%
RAM	10	3%	16	4%
DSP48Es	180	93%	192	100%

only. This thread sums up the results by all blocks from the global memory location and returns the total payoff sum to the main program. The main program then computes the option price from the returned result.

3.4 Performance Comparison

This section investigates the advantage of using the CVMC method over the pure Monte-Carlo method for arithmetic Asian option pricing on FPGA. We also compare the performance of the implementations against GPU and CPU.

Our FPGA implementations targeted a Xilinx xc5vlx330t FPGA chip on an Alpha Data ADM-XRC-5T2 card. We design our hardware architecture manually in VHDL to maximize performance. The design is synthesized, mapped, placed and routed using Xilinx ISE 10.1.03. Single precision floating point arithmetic is used. There are 10 CVMC cores in the design. Resource utilisation is summarised in Table 3.4. We have also implemented arithmetic Asian option pricing using the pure Monte-Carlo method on xc5vlx330t. As the number of floating point operators is decreased, 16 pure MC cores can be fitted in an xc5vlx330t device.

We choose an arithmetic Asian call option with parameters S_0*=100, K=105, v=0.15, r=0.1, T=1* and *steps=365*. The number of Monte-Carlo simulations is 1,000,000. The $Var(t)$, $Var(c)$ and $Cov(t, c)$ of the tested arithmetic Asian call option is 33.47, 152.36 and 59.54 respectively. The 99% confidence interval of the option price is [3.392, 3.408]. The 99% confidence interval length is 0.016.

Fig. 3.6 shows the number of simulations required versus the 99% confidence interval length for using

pure Monte-Carlo and control variate Monte-Carlo methods to price arithmetic Asian options. We can see that the number of simulations required for the pure Monte-Carlo method is 3.28 times more than the control variate Monte-Carlo method. One may consider that the pure Monte-Carlo core in FPGA consumes fewer resources, and therefore the degree of parallelism is higher. Our result also shows that 6 more cores can fit in the xc5vlx330t FPGA using pure Monte-Carlo. However, the reduced parallelism with fewer cores for the control variate method is more than compensated by the benefit of reduced variance. For a given confidence interval length (accuracy), the 10 CVMC FPGA cores is 2 times faster than the pure Monte-Carlo FPGA implementation with 16 cores as shown in Fig. 3.7.

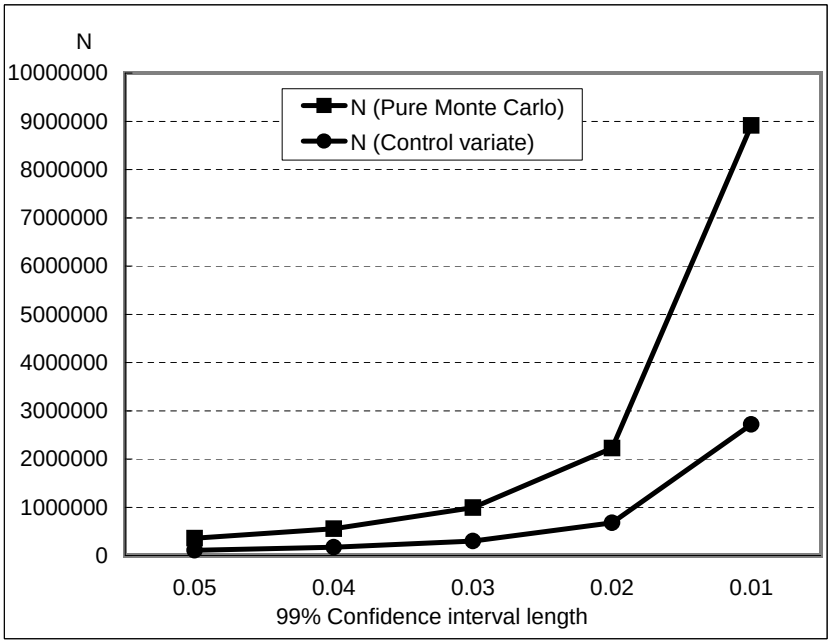

Figure 3.6: The required number of simulation versus the 99% confidence interval length.

The acceleration of the FPGA implementations and GPU implementations of Asian option pricing using CVMC method are in comparison to a reference software implementation. The reference PC used an Intel Xeon quad-core E5420 2.5GHz processor with 16GB RAM. The multi-threaded software implementation is written using C language with Intel Math Kernel Library (MKL) and compiled using Intel compiler *icc* with maximum speed optimization options which fully utilize *SSE2* parallelism. The targeted GPU is nVidia Tesla C1060 with 4GB of on board RAM. The GPU implementation is a translation from the software C code using CUDA API. The Mersenne Twister and Box-Muller transformation is used for Gaussian random number generation. The CUDA code is written in a way to ensure all GPU cores execute concurrently for random number generation and path simulation. The

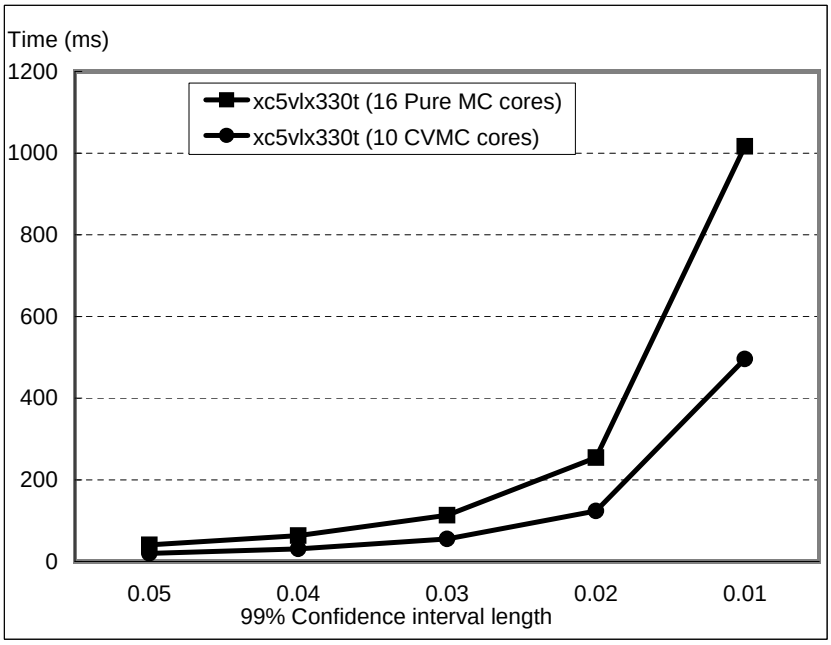

Figure 3.7: The required computation time versus the 99% confidence interval length.

summary of the performance comparison is shown in Table 3.5.

From the results, it can be seen that a speedup of 24 times over the CPU is achieved by xc5vlx330t FPGA. For the performance of the GPU, a speedup of 10 times is achieved by Tesla C1060. The xc5vlx330t FPGA is 2.4 times faster than the Tesla C1060 GPU. The maximum power usage of different devices is also estimated. The power usage of xc5vlx330t is estimated by Xilinx XPower Estimator 11.4.1 with toggle rate 100% and clock rate 200MHz. It is impossible for all the Flip-Flops to toggle at all times. Setting toggle rate to 100% is purely for estimating the maximum bound of power usage. The maximum energy consumption and the energy efficiency can also be estimated. From the Table 3.5, we can see that GPU is 4 times more energy efficient than CPU and FPGA is 66.6 times more energy efficient than CPU.

Table 3.5: Performance of the Asian option pricing using CVMC method

	FPGA	**GPU**	**CPU**
Type	xc5vlx330t	Tesla C1060	Xeon E5420
Frequency	200MHz	1.3GHz	2.5GHz
Time (ms)	184ms	443ms	4,446ms
Speedup	**24x**	10x	1x
Max power (W)	29W	200W	80W
Max energy consumption (J)	5.3J	88.6J	355.7J
Normalised energy efficiency	**66.6x**	4.0x	1.0x

3.5 Summary

This chapter presents a high performance hardware architecture for exotic option pricing using the control variate Monte-Carlo (CVMC) method. To our knowledge, this is the first reported hardware implementation of CVMC method in the literature. Hardware implementations of arithmetic Asian option pricing using both CVMC method and pure Monte-Carlo method are described. Our result shows the reduced number of cores for the control variate method is more than compensated by the benefit of reduced variance. For a given confidence interval length (accuracy), the 10 CVMC cores FPGA implementation is 2 times faster than the pure MC FPGA implementation with 16 cores.

The performance of CVMC FPGA design is compared with a GPU design using CUDA and a multi-threaded software design. By exploiting the efficient Gaussian random number generators, massive parallelism and highly pipelined datapath, our FPGA implementation outperforms a comparable software implementation running on a quad-core CPU by 24 times, and outperforms the GPU implementation by 2.4 times.

There has been no previous work on using control variate method to perform Monte-Carlo simulation in option pricing. There are some similar previous works which employ pure Monte-Carlo for financial computing. In [43], five different types of financial random walks were implemented in hardware and in average 80 times faster than a software implementation running on a single-core CPU. Those random walks are roughly 20 times faster when comparing with a quad-core CPU. Hardware accelerated American option pricer based on Monte-Carlo method is presented in [45] and shows a speedup of 20 times compared with a single-core CPU. The speedup figure of this American option is only around 5 times when comparing with a quad-core CPU. Therefore, our FPGA design of a more complex exotic option pricer presented in this chapter outperforms both the simple financial walk simulator and American option pricer in previous research.

In addition, the FPGA implementation consumes much less power than the GPU and software implementation. This improvement in speed and power consumption provides an attractive solution to financial institutions to shorten the pricing time and reduce costs by energy savings.

Chapter 4

Accelerating Quadrature Methods for Option Valuation

4.1 Motivation

Financial institutions continually invent new ways to repackage and modify financial products in order to satisfy the needs of different investors. While some basic financial options can be priced with a closed-form solution, many other derivatives with knock-out/knock-in features (e.g. Accumulator, Decumulator, and Barrier Options), changing strike prices, or discrete settlement days, have no known closed-form solution. Numerical techniques are used to value these complex derivative products. Numerical methods for derivative pricing can be roughly divided into two groups: Monte-Carlo methods, which work forwards from the current asset price to expiry time using multiple randomly chosen paths; and lattice methods, which work backwards from exercise time to the current price, using a pre-determined lattice of asset prices and times. In Chapter 3, we presented the acceleration methodology for Monte-Carlo methods. In this chapter, we explore the acceleration methodology of quadrature methods, which are subsets of the lattice methods and are very powerful for pricing options when their paths are monitored in discrete time points [21].

Quadrature methods have been applied in different areas including modeling credit risk [40], solving electromagnetic problems [41] and calculating photon distribution [42]. It is a powerful way of pricing

path-dependent options where the path is monitored in discrete time points. A lookback discrete barrier option priced using quadrature methods is more than 1000 times faster than using the trinomial method, while achieving a more accurate result [21].

Using quadrature methods to price a single simple option is fast and can typically be performed in milliseconds on desktop computers. However, quadrature methods can become a computational bottleneck when a huge number of complex options are being revalued in real-time using live data-feeds. Many financial derivatives now involve multiple underlying assets instead of just one. As the computation complexity increases exponentially with the number of underlying assets (i.e. the number of dimensions), how to accelerate the quadrature option pricing becomes a significant problem. Energy consumption of computation is also a major concern when the computation is performed 24 hours a day, 7 days a week.

This chapter explores the acceleration of quadrature computation using different computational devices including Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). The main contributions of this chapter are:

- A novel parallel hardware architecture for option pricing based on quadrature methods (Section 4.3).
- Techniques for multi-dimensional option pricing and a model of the computational complexity (Section 4.4).
- An approach for generating multi-dimensional quadrature evaluation cores for FPGA and GPU (Section 4.5).
- Comparison of performance and energy consumption of FPGAs, GPUs and CPUs for quadrature evaluation across different number of dimensions (Section 4.6).

4.2 Option pricing and quadrature methods

To understand option pricing with quadrature methods, we first consider the Black and Scholes partial differential equation [31] for an option with an underlying asset following geometric Brownian motion

with continuous dividend yield:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r - D_c)S\frac{\partial V}{\partial S} - rV = 0 \tag{4.1}$$

where $V(S,t)$ is the price of the option, S is the value of the underlying asset, t is time, r is risk-free interest rate, σ is volatility of the underlying asset, E is exercise price, and D_c is continuous dividend yield.

According to [21], the following standard transformations

$$x = \log(S_t/E), \qquad y = \log(S_{t+\Delta t}/E)$$

give us the solution of $V(x,t)$ as:

$$V(x,t) = A(x)\int_{-\infty}^{+\infty} B(x,y)V(y,t+\Delta t)dy \tag{4.2}$$

where

$$A(x) = \frac{1}{\sqrt{2\sigma^2\pi\Delta t}} e^{(-kx/2)-(\sigma^2 k^2 \Delta t/8)-r\Delta t} \tag{4.3}$$

$$B(x,y) = e^{-((x-y)^2/2\sigma^2\Delta t))+(ky/2)} \tag{4.4}$$

$$k = \frac{2(r - D_c)}{\sigma^2} - 1 \tag{4.5}$$

Equation (4.2) contains an integral which cannot be evaluated analytically. Although for European options they can be converted to the probability density function for the normal distribution, for more complicated options numerical techniques are required to evaluate the integrals. For the evaluation of other complicated options such as discrete barrier options and American options, the valuation problem can be arranged to exploit consecutive time intervals and apply Equation (4.2) iteratively from maturity time. The value of $V(y,T)$ at maturity time (T) is determined according to the payoff function of the option. For example, the values of $V(y,T)$ for an European option is given by:

$V(y,T) = E(e^y - 1)$ for $y > 0$ and $V(y,T) = 0$ for $y <= 0$.

There are many different methods of numerical integral evaluation. Two of the most common methods include the trapezoidal rule and Simpson's rule [50] and their equations are stated in Section 2.2.3 as Equation 2.15 and Equation 2.16.

4.3 Parallel Architecture

Using quadrature methods from Equation (2.15) or Equation (2.16), the option value $V(x,t)$ from Equation (4.2) can be computed as:

$$\begin{aligned} V(x,t) &= A(x)\int_{-\infty}^{+\infty} B(x,y)V(y,t+\Delta t)dy \\ &\approx A(x)I_{oc}\sum_{i=0}^{N} I_i \cdot B(x, y_{min}+i\delta y)V(y_{min}+i\delta y, t+\Delta t) \end{aligned} \tag{4.6}$$

The sequence of integration coefficients I_i and the value of the outer integration coefficient I_{oc} depend on the type of quadrature method used. For example, the sequence of I_i is $1, 2, 2, \ldots, 2, 1$ and the value of I_{oc} is $\frac{\delta y}{2}$ for trapezoidal rule.

The major calculation part of $V(x,t)$ is the summation of $B(x, y_{min}+i\delta y)V(y_{min}+i\delta y, t+\Delta t)$ times I_i for all i. Similarly, the values of $V(y_{min}+i\delta y, t+\Delta t)$ are computed by the summation of $B(y_{min}+i\delta y, y_{min}+j\delta y)V(y_{min}+j\delta y, t+2\Delta t)$ times I_j for all j. Therefore, the computation is a backward iterative process from the maturity time. A graphical representation of the process is illustrated in Fig. 4.1.

The value of δy determines the density of the integration. As the underlying asset follows a lognormal distribution and the change of price exhibits Brownian motion, the value of y fluctuates proportional to $\sqrt{\Delta t}$. As a result, we define the grid density factor $K1$ from:

$$\delta y = \sqrt{\Delta t}/K1 \tag{4.7}$$

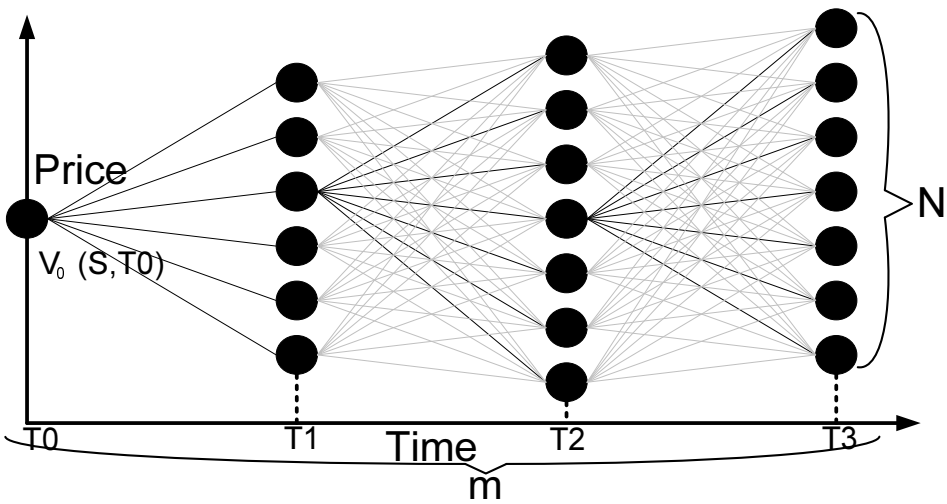

Figure 4.1: The backward iteration process.

Therefore, increasing the value of $K1$ leads to a smaller value of δy and a denser grid.

It is not possible to integrate a function from $-\infty$ to $+\infty$ numerically in practice. Therefore, the quadrature methods evaluate from a sufficiently small value y_{min} to a sufficiently large value y_{max}. We define the grid size factor $K2$ from:

$$y_{max} = x + K2 \cdot \sigma\sqrt{\Delta t} \tag{4.8}$$

$$y_{min} = x - K2 \cdot \sigma\sqrt{\Delta t} \tag{4.9}$$

As a result, a large value of $K2$ leads to a large value of y_{max} and a small value of y_{min}, resulting in a wide grid. $K2$ can also be viewed as the number of standard deviations from y to the original position of x after Δt.

Table 4.1 shows some of the pricing equations for different option types according to [21]. The pricing equations are slightly different in terms of the integration range and the evaluation flow. For discrete barrier options, the result of C_{m+1} is required for the evaluation of C_m. The final option value C_0 has to be evaluated iteratively. Table 4.2 shows the computational complexity for different types of options. The computation complexity depends on the evaluation flow, the number of integration grid points N and the number of time steps m.

A key result from Table 4.1 and Table 4.2 is that all option pricing equations require the evaluation of

Table 4.1: The pricing equations for various types of options.

Option type:	Pricing equation:
European	$V(x,t) \approx A(x) \int_0^{N+\delta y} B(x,y)V(y,t+\Delta t)dy$
Discrete barrier call	$C_m(x,T_{m-1}) \approx A(x) \int_{b_m}^{y_{max_m}} B(x,y)C_{m+1}(y,T_m)dy$
Bermudan put	$P_m(x,T_{m-1}) \approx A(x) \int_{b_m}^{y_{max_m}} B(x,y)P_m(y \geq b_m,T_m)dy + Ee^{-r\Delta t_m}N(-d_2) - Ee^{x-D_c\Delta t_m}N(-d_1)$
American call	$C_m(x,T_{m-1}) \approx A(x) \int_{y_{min_m}}^{b_m} B(x,y)C_m(y \leq b_m,T_m)dy + E_M e^{x-D_c\Delta t_m}N(d_1) - E_m e^{-r\Delta t_m}N(d_2)$

Table 4.2: The computational complexity for some example options. N denotes the number of integration grid points and m denotes the number of time steps.

Option type:	Number of integration	Number of evaluation of B(x,y)	Number of evaluation of A(x)
European	$O(1)$	$O(N)$	$O(1)$
Discrete barrier call	$O(Nm)$	$O(N^2m)$	$O(Nm)$
Bermudan put	$O(Nm)$	$O(N^2m)$	$O(Nm)$
American call	$O(Nm)$	$O(N^2m)$	$O(Nm)$

a similar integral on $B(x,y)V(y,t+\Delta t)$. Using quadrature methods requires the evaluation of the function $B(x,y)$ intensively, which is the computation bottleneck. Although function $A(x)$ is also required to be evaluated repeatedly, the computation complexity for $A(x)$ is lower than $B(x,y)$ from Table 4.2.

4.3.1 System architecture

Our system architecture is not designed for pricing a specific option, so the flexibility to support all kinds of options must be considered. It has been shown that most of the equity options can be expressed in integral forms and solved by quadrature methods including: European options, discrete barrier options, moving discrete barrier options, Bermudan put options, American call options, and lookback options [21, 109]. However, the quadrature evaluation procedures are slightly different for different types of options. Different types of options have different discontinuities, which lead to different integral boundaries. Some options contain option specific parameters: for example, the knock-out prices and number of periods are required for discrete barrier options. Although European options can be priced with a single quadrature step, most of the other options need to be evaluated iteratively from the price in period of m to $m-1$. Therefore using different number of quadrature steps is required. As a result, our system is designed to provide efficiency in hardware evaluation of the integral and flexibility for a general option pricing framework, as illustrated in Fig. 4.2.

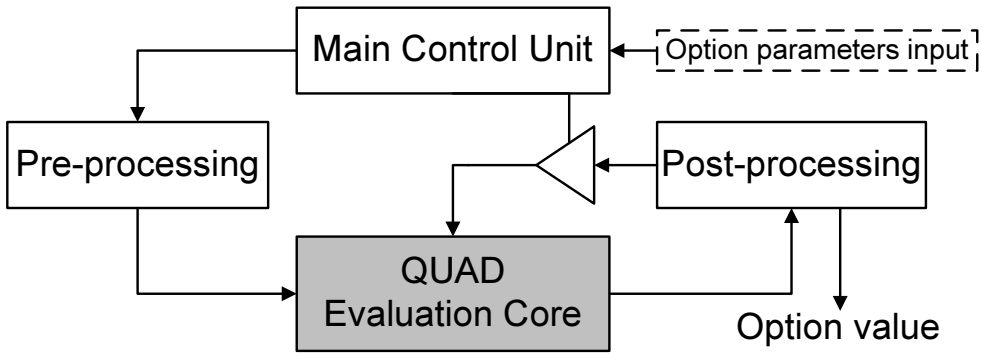

Figure 4.2: System architecture of a generic option valuation system based on quadrature methods.

The system architecture of the generic option valuation system using quadrature method is shown in Fig. 4.2. The architecture consists of the following components: (a) a pre-processing block, (b) one or more QUAD(quadrature) evaluation cores, (c) a post-processing block, and (d) a main control unit. Data input to the system are: $K1, K2, T, So, E, r, D_c, \sigma$, option-type and option-specific-parameters. The option-type and option-specific-parameters provide the flexibility to support the pricing of multiple types of options. For example, we could specify the number of periods (m) and the knock-out/knock-in prices (b) for barrier options.

A typical option evaluation flow is illustrated in Fig. 4.3. The main control unit accepts the basic option input, selects the corresponding option evaluation equation and coordinates with the pre-processing and post-processing blocks. The pre-processing block computes the non-repeated values such as δy, y_{max} and y_{min}. It then generates the set of y_i, V_i and x for the QUAD(quadrature) evaluation cores. The QUAD evaluation cores evaluate the integral value based on Equation (4.6). The post-processing block combines the integral value with the value of $A(x)$ and produces the value of $V(x,t)$. The main control unit then decides whether $V(x,t)$ is the final solution or a temporary result for the next iteration.

The QUAD evaluation core is implemented in hardware for three main reasons. First, more than one QUAD evaluation core fits on a single FPGA. Therefore, several quadratures can be evaluated simultaneously to exploit parallelism. Second, the evaluation of the function $B(x,y)$ could be implemented in pipelined hardware which is fast and efficient. The value of $B(x,y)$ can be obtained in every clock cycle. Third, as shown in Table 4.2, the evaluation of the quadrature is the computation bottleneck,

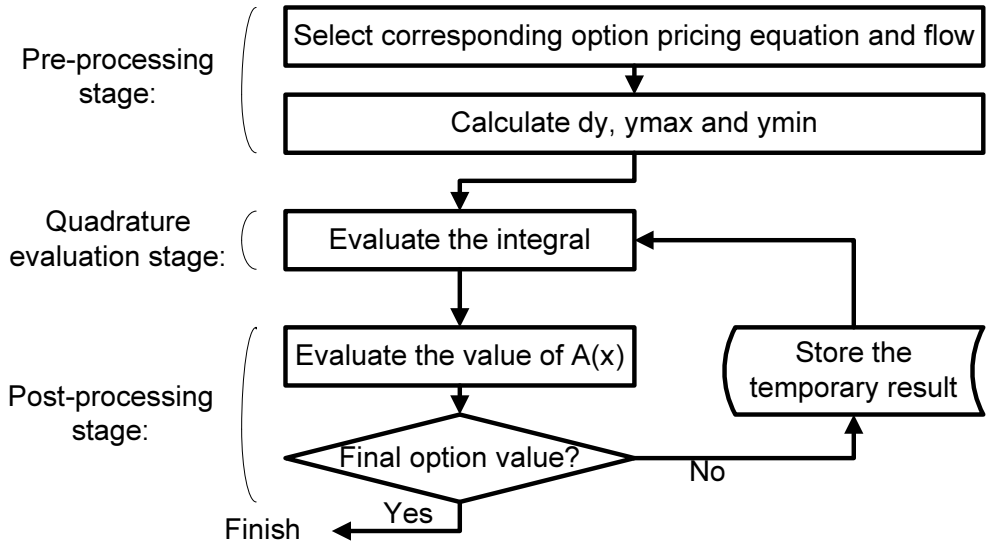

Figure 4.3: The option evaluation flow.

which would benefit from hardware acceleration.

The main control unit, pre-processing and post-processing blocks are implemented in software for the following reasons. (a) It increases the flexibility to support other options. (b) The evaluation in pre-processing and post-processing blocks is not the performance bottleneck; implementing them in hardware would not improve performance significantly.

The proposed architecture offers fast and parallel hardware cores for repeated numerical integrations, while supporting a versatile option evaluation platform.

A straight-forward way of optimising the QUAD evaluation core is to create a tree of pipelined operators from the equations directly. Fig. 4.4 shows an operator tree based on Equation 4.2 to Equation 4.5.

In Fig. 4.4, $\Delta t, x, y_i, \sigma, r, D_c, V(y_i, t{+}\Delta t)$ and I_i are fed to the evaluation tree continuously. However, the straight-forward implementation consumes a large amount of hardware resources as it requires many floating-point operators. The optimized design is shown in Fig. 4.5, and will be used to produce implementations on both FPGA and GPUs (Section 4.5).

The optimized quadrature operator tree takes the following data input: $x, y_i, C1, C2, I_i$ and $V(y_i, t +$

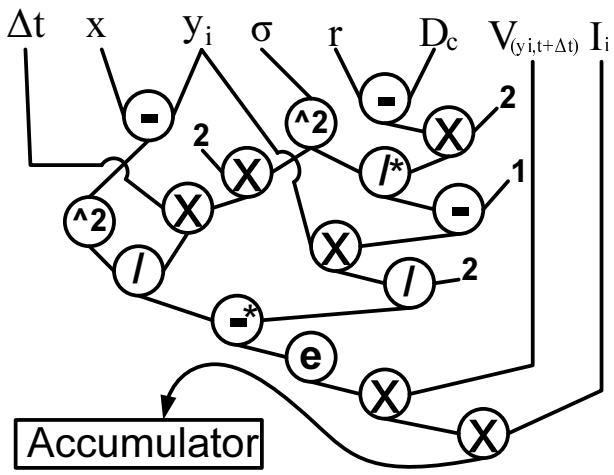

Figure 4.4: An operator tree diagram for a straight-forward design by creating the operators from Equation 4.2 to Equation 4.5 directly (the operator with '*' denotes the operation from right to left).

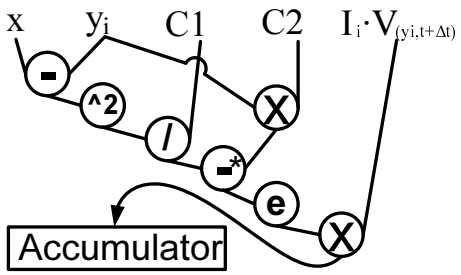

Figure 4.5: An operator tree diagram for optimized design.

Δt). We define:

$$C1 = 2\sigma^2 \Delta t \tag{4.10}$$

$$C2 = \frac{r - D_c}{\sigma^2} - \frac{1}{2} \tag{4.11}$$

The operator tree is optimized by identifying the non-changing nodes during the pipelined evaluation. The values of $C1$ and $C2$ are fixed for the values of $y_i, I_i, V(y_i, t + \Delta t), i \in [0, N]$. Therefore, $C1$ and $C2$ can be pre-computed in the pre-processing stage and passed to QUAD evaluation cores. The hardware size is therefore reduced significantly and the number of parameters is also reduced. The parameters of I_i and $V(y_i, t + \Delta t)$ are passed to the QUAD evaluation cores together. For an integration grid with N steps, the total number of parameters required is of the order $2N$, a 33% reduction from the original design which is of the order $3N$. Table 4.3 summarizes the differences between the original design and the optimized design.

Table 4.3: Comparing the original and optimized designs.

	Original	Optimized
Number of $exp(x)$ operators	1	1
Number of $\times$ operators	8	3
Number of $\div$ operators	3	1
Number of $-$ operators	4	2
Number of input parameters	$3N$ + 5	$2N$ + 3

4.4 Multi-dimensional Quadrature Analysis

To extend the design to support multiple underlying assets, we first consider the Black and Scholes partial differential equation [39] for an option with all underlying assets following geometric Brownian motion:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}\sigma_i\sigma_j\rho_{ij}S_iS_j\frac{\partial^2 V}{\partial S_i \partial S_j} + \sum_{i=1}^{d}(r - D_i)S_i\frac{\partial V}{\partial S_i} - rV = 0 \tag{4.12}$$

with the logarithmic transformations of $x_i = \log(S_i)$ to be the chosen nodes at t and $y_i = \log(S_i)$ to be the chosen nodes at $t + \Delta t$. Let R be the matrix such that element $R_{ij} = \rho_{ij}$. According to [39], the solution is:

$$V(x_1, \ldots, x_d, t) = A\int_{-\infty}^{+\infty}\ldots\int_{-\infty}^{+\infty} V(y_1, \ldots, y_d, t + \Delta t)B(x_1, \ldots, x_d, y_i, \ldots, y_d)dy_1 \ldots dy_d \tag{4.13}$$

where

$$B(x_1, \ldots, x_d, y_i, \ldots, y_d) = exp(-\frac{1}{2}\boldsymbol{\alpha}^T R^{-1}\boldsymbol{\alpha}), \tag{4.14}$$

$$\alpha_i = (x_i - y_i + C1_i)/C2_i \tag{4.15}$$

Equation (4.13) is the fundamental equation for multi-dimensional option pricing containing an integral which cannot be evaluated analytically. The number of dimensions for this integration is given by the total number of assets. $C1_i$ and $C2_i$ will be calculated in the pre-processing stage to improve

performance.

$$C1_i = (r - D_i - \frac{\sigma_i^2}{2})\Delta t$$

and

$$C2_i = \sigma_i(\Delta t)^{1/2}$$

All quadrature methods discretize the continuous integration range into a set of grid points. The function value $f(y)$ is evaluated at these grid points and multiplied with integration coefficients. As from Equation (2.16), the integration coefficients for Simpson's rule are $\{1, 4, 2, 4, 2, \ldots, 2, 4, 1\}$. Under multi-dimensional quadrature methods, the product rule is used to determine the coefficient. The effective integration coefficients are calculated by the product of all original integration coefficients in their corresponding dimensions. For example in a 2D case, the integration coefficients for the grid points using Simpson's rule are:

$$\begin{bmatrix} 1, & 4, & 2, & 4, & 2, & \cdots, & 2, & 4, & 1 \\ 4, & 16, & 8, & 16, & 8, & \cdots, & 8, & 16, & 4 \\ 2, & 8, & 4, & 8, & 4, & \cdots, & 4, & 8, & 2 \\ 4, & 16, & 8, & 16, & 8, & \cdots, & 8, & 16, & 4 \\ 2, & 8, & 4, & 8, & 4, & \cdots, & 4, & 8, & 2 \\ \vdots, & \vdots, & \vdots, & \vdots, & \vdots, & \ddots, & \vdots, & \vdots, & \vdots \\ 2, & 8, & 4, & 8, & 4, & \cdots, & 4, & 8, & 2 \\ 4, & 16, & 8, & 16, & 8, & \cdots, & 8, & 16, & 4 \\ 1, & 4, & 2, & 4, & 2, & \cdots, & 2, & 4, & 1 \end{bmatrix}$$

Fig. 4.6 shows the graphical representation of the iteration process of a 2-dimensional barrier option pricing. We define N as the number of possible values (grid points) for y_1 and assume the number of grid points for all y_i is the same. We define d as the number of dimensions and m as the number of time intervals. For each time step, the number of integrations required is equal to the number of grid points, which is N^d. As a result, the total number of integrations required for multiple time steps American options and Barrier options is $N^d m$. The total number of evaluations of B is $N^d m \times N^d = N^{2d} m$.

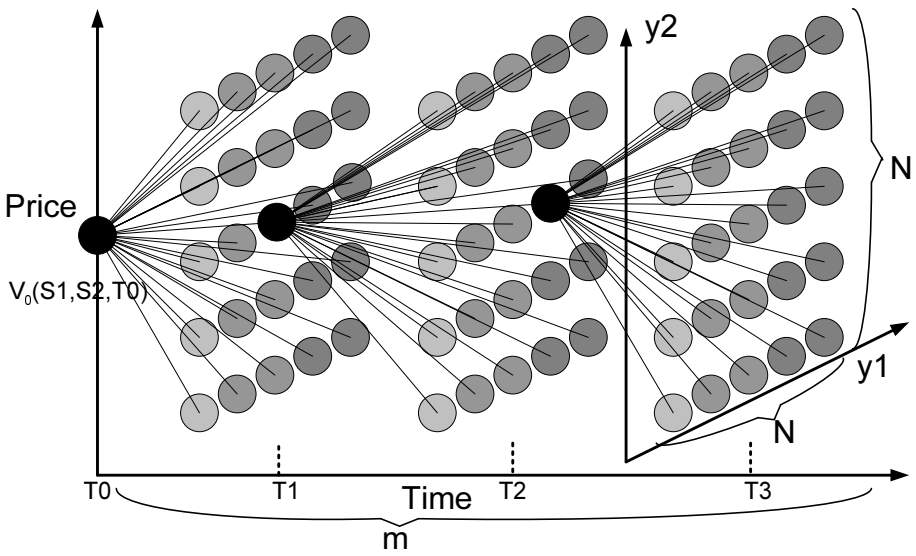

Figure 4.6: The iteration process of a 2D barrier option.

Next, consider complexity analysis of our designs. The optimized number of operators required for the calculation of column matrix α is d for $(-)$ operator, d for $(+)$ operator and d for $(\div)$ operator. For matrix multiplication of $\alpha^T R^{-1}\alpha$ in Equation (4.14), the number of $(\times)$ operators required is $d(d+1)$ and the number of $(+)$ operators required is $(d-1)(d+1)$. The rest of Equation (4.14) requires one more $(\times)$ operator and one more exponential operator. Table 4.4 shows the summary of operator requirement for the evaluation of $B(x_1, \ldots, x_d, y_i, \ldots, y_d)$ and Table 4.5 shows the summary of computation complexity for some example options.

Table 4.4: The operators count for the evaluation of B.

Operator type:	**Count**
$+$	$d^2 + d - 1$
$-$	d
$\times$	$d^2 + d + 1$
$\div$	d
exp	1
Total:	$2d^2 + 4d + 1$

Table 4.5: The computation complexity for some example multi-dimensional options.

Option type:	Number of integration	Number of evaluation of B	Total number of floating-point operations
European options	$O(1)$	$O(N^d)$	$O(N^d(2d^2 + 4d + 1))$
Barrier options	$O(N^d m)$	$O(N^{2d} m)$	$O(N^{2d}(2d^2 + 4d + 1)m)$
American options	$O(N^d m)$	$O(N^{2d} m)$	$O(N^{2d}(2d^2 + 4d + 1)m)$

The computation time can be estimated by assuming that a 10 GFLOPs processor is used (the peak performance of a Pentium 4 3.2GHz CPU is around 6.4 GFLOPs) and all floating point operators take

the same amount of time. Fig. 4.7 shows that the computation time required is drastically increased with the number of dimensions. It can be seen that pricing a European option with 7 underlying assets takes 14.7 days with this processor at peak performance; however, it takes over 5 years with 8 assets. Hence other methods, such as using a cluster of accelerators, are required for designs beyond 7 dimensions.

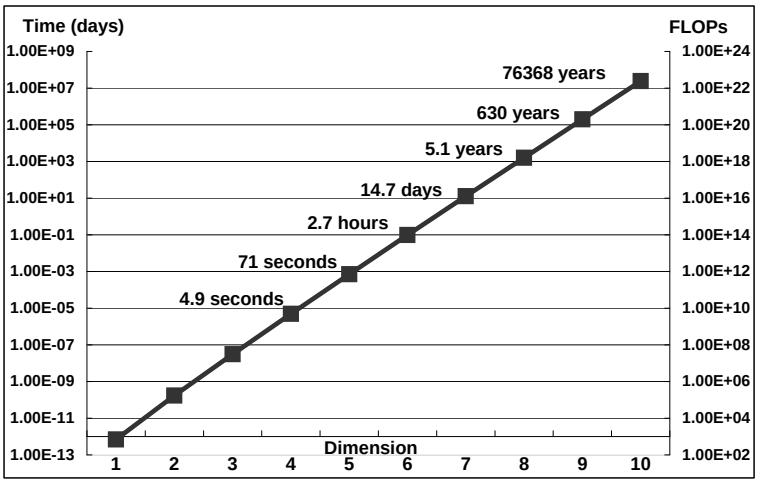

Figure 4.7: The time required for the pricing of European options. (n = 100)

4.5 FPGA and GPU designs

4.5.1 Single dimension QUAD evaluation core on FPGA

Our FPGA implementation of the QUAD evaluation cores is based on *HyperStreams* and the Handel-C programming language. *HyperStreams* is a high-level abstraction language and library [44]. It can produce a fully-pipelined hardware implementation with automatic optimization of operator latency at compile time. This feature is useful when implementing a complex algorithm core.

Fig. 4.8 shows a pipelined QUAD evaluation core based on the design in Fig. 4.5. The grey boxes denote the pipeline balancing registers that are allocated automatically by *HyperStreams*. The QUAD evaluation core produces the value of $B(x, y)$ in Equation (4.2) for every clock cycle. For an FPGA running at 100MHz, the QUAD evaluation core can produce 100M partial integral values per second.

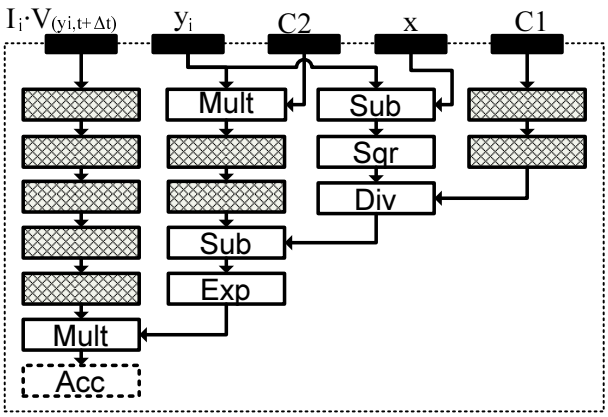

Figure 4.8: Pipelined QUAD evaluation core for FPGA.

4.5.2 Multiple dimensions QUAD evaluation core on FPGA

The most challenging part of the multi-dimensional design is to support an arbitrary number of dimensions. The hardware evaluation cores are completely different for different dimensions as the underlying logic and the number of pipeline stages are different. Our approach provides a generic architecture that produces hardware designs specialised for a given dimension.

The hardware multi-dimensional QUAD evaluation core involves three major parts. The first part is the evaluation of the vector α from Equation (4.15). The second part is the matrix multiplication of $\alpha^T R^{-1} \alpha$. The last part is the rest of the integration.

Fig. 4.9 shows a $\alpha^T R^{-1} \alpha$ design for 2D QUAD evaluation. The $\alpha^T R^{-1} \alpha$ design becomes more complex for higher dimensions, with an increasing number of operators and pipeline stages. An evaluation core generator is developed to produce designs for different dimensions automatically.

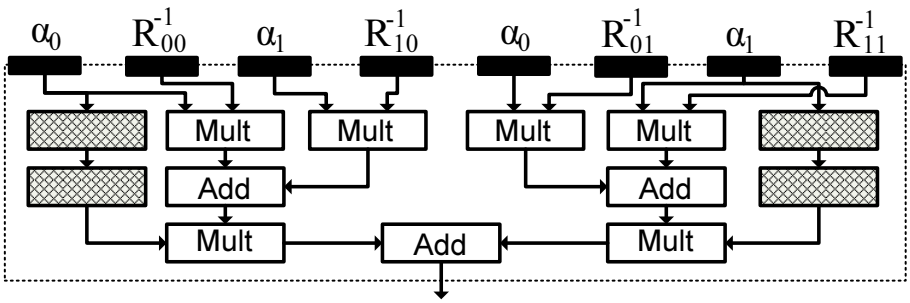

Figure 4.9: Pipelined $\alpha^T R^{-1} \alpha$ design for 2D QUAD evaluation.

The flow of the QUAD evaluation core generator is shown in Fig. 4.10. The generator accepts 2

input parameters: number of dimensions and the precision (single or double). An operator tree is generated and stored in a temporary file. Finally, the operator tree file is parsed and the corresponding *HyperStreams* and Handel-C codes are generated. The Handel-C code can then be compiled for simulation or bit-stream generation.

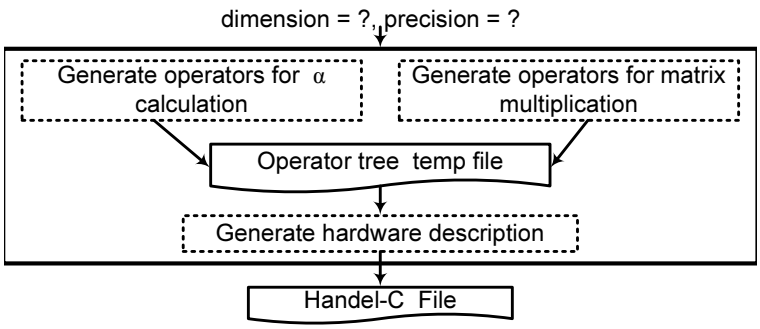

Figure 4.10: Generating multi-dimensional QUAD evaluation.

The operator tree generation consists of two main parts. The first part is the operator tree generation for the vector α calculation. It is generated according to Equation (4.15) and replicated d times with respect to dimension d. Therefore, the logic resources required grow proportionally to d for the α calculation part. The second part is the operator tree generation for the matrix multiplication $\alpha^T R^{-1} \alpha$ from Equation (4.14). The numbers of $(\times)$ and $(+)$ operators required for this matrix multiplication are $d(d+1)$ and $(d-1)(d+1)$ respectively. Therefore, logic resources required grow proportionally to d^2. Finally, the operator trees from the above two parts are combined with the rest of the quadrature operators.

Table 4.6 shows the FPGA device utilization figures for the QUAD evaluation core in different dimensions and precisions. The targeted FPGA is Xilinx Virtex-4 xc4vlx160 and the designs are compiled using DK5.1 and Xilinx ISE 10.1. The result indicates that the FPGA device is fully utilized for 1 dimension under double-precision and is fully utilized for 5 dimensions under single-precision. The result also shows that for 1 dimension, multiple QUAD evaluation cores could be fitted into a single FPGA in order to exploit parallelism.

Table 4.6: The logic utilization of QUAD evaluation core in different dimensions. Asterisk (*) indicates that the place and route procedure cannot be completed.

	FPGA - Virtex-4 xc4vlx160							
Precision	single						double	
Dimension	1	2	3	4	5	6	1	2
DSPs	34 (35%)	51 (53%)	75 (78%)	96 (100%)	96 (100%)	(*)	96 (100%)	(*)
LUTs	19,713 (14%)	32,053 (23%)	43,580 (32%)	60,058 (44%)	70,909 (52%)	(*)	53,792 (39%)	(*)
FFs	16,605 (12%)	22,418 (16%)	30,005 (22%)	41,466 (30%)	54,569 (40%)	(*)	39,281 (29%)	(*)
Slices	20,200 (29%)	27,970 (41%)	38,006 (56%)	51,862 (76%)	67,582 (99%)	(*)	51,396 (76%)	(*)
Clock Rate	100MHz	91.2MHz	89.7MHz	91.5MHz	88.0MHz	(*)	81.9MHz	(*)

4.5.3 QUAD evaluation core on GPU

Our implementation on GPUs is based on Compute Unified Device Architecture (CUDA) API for nVidia GPUs [90]. The QUAD evaluation core is implemented in CUDA as a kernel to exploit parallelism. Similar to the implementation on FPGA, we implement the evaluation core in CUDA based on the optimized operator tree. In addition, the whole integration is segmented to support different blocks and threads in the CUDA environment. Each thread would evaluate a set of partial integrals and accumulate the result. The first thread in each block then adds up the results from all the threads within the same block. The main thread then adds up all the results from all the blocks. The CUDA pseudo code for the QUAD evaluation kernel is shown in Fig. 4.11. The grid size and block size is set to 60 and 256 respectively. Registers per thread is 16 and the occupancy of each multiprocessor is 100%.

4.6 Evaluation and comparison

In this section, the performance and energy consumption of different implementations of QUAD evaluation core are studied. We choose the pricing of 1,000 European options with grid density factor $K1 = 400,000$ and grid size factor $K2 = 10$ as the benchmark. The typical $K1$ value of 400 produces highly accurate results, but the reason for choosing a much larger value is to facilitate performance analysis of the QUAD evaluation cores with a longer evaluation time. No matter what values of $K1$ or $K2$, the QUAD evaluation cores are still responsible for the computation bottleneck of option pricing of the order N^2m as shown in Table 4.2 or $N^{2d}m$ in multi-dimensional cases. Simpson's rule is preferable to the trapezoidal rule in our system as the error terms of Simpson's rule decrease at a rate

```
void GPU_EvaluationCore()
{
   unique_thread_ID = Num of block * block_ ID + thread_ID_per_block
   THREAD_COUNT  = Num of thread in a block * Num of block in a grid

   for (i = unique_thread_ID ; i <N; i += THREAD_COUNT)
         evaluate partial integral on yi and Vi, and accumulate on local register;

   copy local register value to shared memory
   Synchronize with all other threads.
   if  (thread_ID_per_block==0)              // the first thread in each block
         Sum up all partial integrals from all threads in the same block.

   Synchronize with all other threads.
   if (unique_thread_ID==0)                  // the main thread
         Sum up all partial integrals from all the first threads in all blocks.
}
```

Figure 4.11: CUDA pseudo code for QUAD evaluation kernel.

of $(\delta y)^4$ which produces more accurate results with the same hardware complexity. Therefore, Simpson's rule is adopted for performance analysis. The performance and energy consumption analysis for the pricing of 1-underlying, 2-underlying and 3-underlying assets European options are studied.

The FPGA and GPU implementations are compared to a reference software implementation. The reference CPU is Intel Xeon W3505 2.53GHz dual-core processor. The software implementation is written using C language. It is optimized with multi-threading using OpenMP API and compiled using Intel compiler (icc) 11.1 with -O3 maximum speed optimization option and SSE enabled. Intel Math Kernel Library is used. The targeted FPGA is Xilinx Virtex-4 xc4vlx160 in the RCHTX card. The designs are compiled using DK5.1 and Xilinx ISE 9.2. The targeted GPU is nVidia Geforce 8600GT with 256MB of on board RAM and nVidia Tesla C1060 with 4GB of on board RAM. The time measured for the GPU is the execution time of the evaluation kernel only. The time for copying the data from the main memory to the global memory of GPU is excluded. Similarly, the data transfer time for copying the data from main memory to the block RAM of FPGA is excluded. The performance figures obtained reflect the pure processing speed of the underlying devices only.

We measure the additional power consumption for computation (APCC) with a power measuring setup involving multiple equipments. A FLUKE i30 current clamp is used to measure the additional

AC current in the live wire of the power cord during the computation. This current clamp has an output sensitivity of $S = 100mV/A$ in $\pm 1mA$ resolution. The output of the clamp is measured in mV scale by a Maplin N56FU digital multi-meter (DMM), collected through a USB connection and logged with open source QtDMM software. APCC is defined as the power usage during the computation time (run-time power) minus the power usage at idle time (static power). In other words, APCC is the dynamic power consumption for that particular computation. Since the dynamic power consumption fluctuates a little, we take the average value of dynamic power to be the APCC.

The additional energy consumption for computation (AECC) is defined by the following equation:

$$AECC = APCC \times Total\ Computational\ Time. \tag{4.16}$$

Therefore, AECC measures the actual additional energy consumed for that particular computation.

The summary of the performance comparison of 1D, 2D and 3D QUAD evaluation core is shown in Table 4.7, Table 4.8 and Table 4.9.

Table 4.7: The performance and energy consumption comparison of different implementation of 1D QUAD evaluation core. The Geforce 8600GT has 32 processors, the Tesla C1060 has 240 processors and the Xeon W3505 has two processing cores.

	FPGA		GPU			CPU
	Virtex-4 xc4vlx160		Geforce 8600GT	Tesla C1060		Xeon W3505
Technology	90nm		80nm	65nm		45nm
Release date	Sep 2004		Apr 2007	Sep 2008		Mar 2009
Arithmetic	single	double	single	single	double	double
Clock Rate	100MHz	81.9MHz	1.35GHz	1.3GHz	1.3GHz	3.6GHz
Replicated cores	3	1	-	-	-	-
Processing Speed (M values/sec)	300.0	81.9	114.4	546.5	288.7	65.3
Time for 10^9 values (s)	3.3	12.21	8.74	1.83	3.46	15.31
Acceleration (replicated cores)	**4.59x**	**1.25x**	**1.75x**	**8.37x**	**4.42x**	**1x**
APCC for 10^9 values (W)	4.18	3.3	40.55	102.00	99.00	23.60
AECC for 10^9 values (J)	13.93	40.29	354.42	186.64	342.92	361.30
Normalized energy efficiency	**25.93x**	**8.97x**	**1.02x**	**1.94x**	**1.05x**	**1x**

4.6.1 Performance Analysis

From the results of Table 4.7 for 1D case, it can be seen that the FPGA implementation on the xc4vlx160 achieved 4.59 times acceleration using single-precision with 3 replicated QUAD cores,

Table 4.8: The comparison of different implementation of 2D QUAD evaluation core.

	FPGA	GPU			CPU
	Virtex-4 xc4vlx160	Geforce 8600GT	Tesla C1060		Xeon W3505
Arithmetic	single	single	single	double	double
Clock Rate	91.2MHz	1.35GHz	1.3GHz	1.3GHz	3.6GHz
Replicated cores	1	-	-	-	-
Processing Speed (M values/sec)	91.20	95.50	509.60	284.40	50.82
Time for 10^9 values (s)	10.96	10.47	1.96	3.52	19.68
Acceleration	**1.79x**	**1.88x**	**10.03x**	**5.60x**	**1x**
APCC for 10^9 values (W)	2.64	37.02	91.00	83.00	21.16
AECC for 10^9 values (J)	28.95	387.64	178.57	291.84	416.37
Normalized energy efficiency	**14.38x**	**1.07x**	**2.33x**	**1.43x**	**1x**

Table 4.9: The comparison of different implementation of 3D QUAD evaluation core.

	FPGA	GPU			CPU
	Virtex-4 xc4vlx160	Geforce 8600GT	Tesla C1060		Xeon W3505
Arithmetic	single	single	single	double	double
Clock Rate	89.7MHz	1.35GHz	1.3GHz	1.3GHz	3.6GHz
Replicated cores	1	-	-	-	-
Processing Speed (M values/sec)	89.70	81.70	489.20	272.80	33.69
Time for 10^9 values (s)	11.15	12.24	2.04	3.67	29.68
Acceleration	**2.66x**	**2.43x**	**14.52x**	**8.10x**	**1x**
APCC for 10^9 values (W)	3.08	38.74	91.00	87.00	19.8
AECC for 10^9 values (J)	34.34	474.17	186.02	318.91	587.71
Normalized energy efficiency	**17.12x**	**1.24x**	**3.16x**	**1.84x**	**1x**

and achieved 1.25 times acceleration using double-precision. For GPUs, a speedup of 1.75 times is achieved by Geforce 8600GT and a speedup of 8.37 times is achieved by Tesla C1060 in single-precision. In double-precision, the Tesla C1060 has shown a 4.42 times speedup over the reference CPU, while there is no double-precision support in the Geforce 8600GT.

It would be fair to compare Virtex-4 FPGA with Geforce 8600GT GPU because of similar fabrication technology. Xeon W3505 is selected to be a CPU reference because it represents the processing power of most workstations and it has a similar architecture to the latest CPU. We included a set of comparable devices - Virtex-4 FPGA, Geforce 8600GT GPU and Xeon W3505 CPU. We estimated that Virtex-5 FPGA performs at least 4 times faster than Virtex-4 as Virtex-5 has 4 times more slices than Virtex-4 and with higher clock frequency. We found that Tesla C1060 GPU is more than 4 times faster than Geforce 8600GT from Table 4.7. We also estimate that the performance of the latest Intel Core i7 CPU will be around 4 times faster than Xeon W3505 according to their number of cores and frequency ratios.

From Table 4.8 and Table 4.9, it can be seen that the performance of xc4vlx160 FPGA in 2D and 3D cases is not as good as in 1D case. The reason is that the xc4vlx160 FPGA is fully utilized in 1D case with 3 replicated QUAD evaluation cores. However, only one QUAD evaluation core can be fitted in the xc4vlx160 FPGA in 2D and 3D cases and there are many unused logic resources. From this point of view, we can conclude that an algorithm with a smaller computation core is more suitable to FPGA because it is easier to replicate multiple smaller computation cores to fully utilize the resources in the FPGA. The worst scenario, like our 2D case, involves a computation core that consumes just above 50% FPGA resources; it precludes replication so possibly wasting resources.

Although complex algorithms can be implemented easily in FPGAs with *HyperStreams*, maximum performance and utilization of FPGA resources is not guaranteed, as there is a tradeoff when using *HyperStreams* between development time and the amount of acceleration that can be achieved. However, our *HyperStreams* implementation still provides a satisfactory result with significant acceleration over the software implementations. Therefore, *HyperStreams* is useful for producing prototypes rapidly to explore the design space. Further optimization can be applied after a promising architecture is found.

Fixed-point implementations usually enable FPGA to achieve the best performance [70]. However, it is not applicable to quadrature methods as the range of the numerical values spreads widely from small size partial integral values to large size complete integral values.

4.6.2 Energy consumption analysis

Next, consider the energy efficiency of different devices. It is interesting to note that the xc4vlx160 FPGA demonstrates the greatest energy efficiency regardless of the technology differences. In single dimension case, xc4vlx160 is 25.9 times more energy efficient than Xeon W3505, 25.4 times more energy efficient than Geforce 8600GT, and 13.4 times more energy efficient than Tesla C1060.

Fig. 4.12 shows a scatter plot graph of the computation time versus the energy consumption (AECC) of different devices implementing the 1D QUAD evaluation core. From this graph, the highest computational performance is achieved using Tesla C1060 GPU and the lowest energy consumption is achieved using xc4vlx160 FPGA. Therefore, Geforce 8600GT and Xeon W3504 are considered to

be inefficient for this application. Tesla C1060 and xc4vlx160 are the fastest and the most energy efficient respectively for this application.

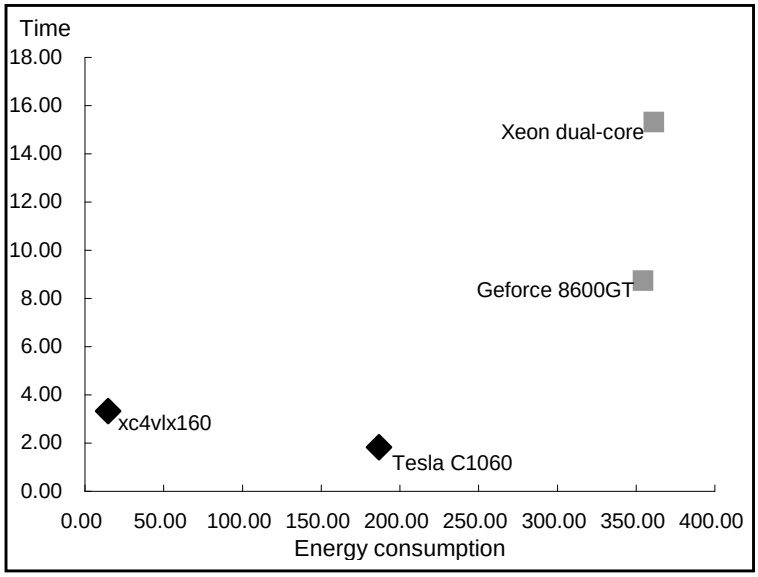

Figure 4.12: The computational time and energy consumption relationship of different devices.

4.7 Summary

This chapter proposes a novel parallel architecture for hardware accelerated option pricing based on quadrature methods. Our proposal includes a highly pipelined datapath capable of supporting quadrature evaluation in parallel. We explore implementations for quadrature evaluation in FPGA and GPU technologies. A tool is developed for automatic production of hardware designs with a given number of dimensions.

The performance and energy consumption of FPGA and GPU implementations are compared against each other and compared against a multi-threaded software implementation on a CPU. The results show that FPGA implementation is 4.6 times faster than the CPU, 1.75 times faster than a GPU in comparable technology and 1.8 times slower than the latest GPU. In addition, the FPGA is up to 25 times more energy efficient than a CPU and a GPU in comparable technology. The energy efficiency of FPGA against other devices in multi-dimensional cases is similar to the 1D case.

There is no previous work nor previous performance result for this problem. The closest research

work is presented in [38], where a parallel hardware architecture is proposed to accelerate option pricing based on explicit finite difference method. It uses a different CPU and a different compiler as the base reference, but it uses the same GPU (Geforce 8600GT) and FPGA (XC4VLX160) as in this chapter. It demonstrated a speedup of only 1.3 times using XC4VLX160 FPGA over Geforce 8600GT GPU. Therefore, our FPGA design using quadrature method (1.75 times faster than GPU) achieved a better speedup than previous FPGA design using finite-difference method (1.3 times faster than GPU) while both designs are based on lattice methods.

Chapter 5

Distributed Financial Computing in Heterogeneous Cluster

5.1 Motivation

A multi-accelerator heterogeneous cluster is a cluster consists of multiple different types of accelerators or computational devices (e.g. FPGAs and GPUs). It is very different from a homogeneous cluster which consists of the same type of computational resources only (e.g. CPUs only). In the previous two chapters (Chapter 3 and Chapter 4), we presented the design and optimisation techniques to use FPGA or GPU as accelerators for generic option pricing with both Monte-Carlo and lattice methods. To further improve the option pricing performance, one may consider using all FPGAs and GPUs as accelerators at the same time in a heterogeneous cluster to perform the computation collaboratively. However, there are still some key challenges when building practical applications to run collaboratively on a multi-accelerator heterogeneous cluster, such as the scalability of the system, the diversity of programming models, tool chains and interfaces; and the difficulty in scheduling the task according to the goal. In this chapter, we tried to address these challenges in Section 5.2 and Section 5.3 when designing the heterogeneous framework.

Focusing our research on the Monte-Carlo (MC) simulation problems enables a better system optimization in a domain specific way. A large Monte-Carlo simulation problem can also be sub-divided

into smaller problems due to the associativity and commutativity nature of the Monte-Carlo simulation. Therefore, we address the above challenges by designing a versatile distributed framework on the heterogeneous cluster architecture targeted in Monte-Carlo problem domain. The main contributions of this chapter include:

- A scalable distributed Monte-Carlo framework for multi-accelerator heterogeneous clusters is proposed. In this framework, various computational units including CPUs, GPUs and FPGAs work collaboratively to share the workload in the simulation process. Each device is controlled by a working process and communicate in a unified way.
- Various load balancing schemes are modeled and evaluated for the proposed framework. Dynamic runtime scheduling is enabled to improve the utilization efficiency of all available computing resources in the system and to minimize the communication overhead.
- Two applications are developed and mapped in the proposed framework. The performance of different dynamic scheduling policies in these practical examples is evaluated. The speed and energy consumption trade-off of different accelerator allocations is discussed and analyzed with the Efficient Allocation Line (EAL) approach.

In the chapter, Section 5.2 explains the details of our proposed distributed MC framework. Section 5.3 presents the models and implementations of different dynamic scheduling policies. Section 5.4 presents the implementation details of two applications (Asian-Option pricing and GARCH asset simulation) using the proposed framework. Section 5.6 evaluates the measured results of these two applications running on a cluster of accelerated computers. Different dynamic scheduling policies are compared and the speed and energy consumption trade-off between different accelerator allocation policies is discussed. Finally, Section 5.7 summarizes our achievements and future work.

5.2 Heterogeneous Framework

There are three major concerns when we design the computing framework in a multi-accelerator heterogeneous cluster, and they are:

- the scalability of the framework to handle more time consuming computation by adding additional hardware resources in a hierarchical way (Section 5.2.1),
- the flexibility of the framework for application programmers to use the original tool-chain of the accelerators (Section 5.2.2),
- and the efficiency of the framework to allocate resources according to the performance goal are the three major concerns of our distributed framework (Section 5.3).

Therefore, we designed our heterogeneous Monte-Carlo framework without creating another layer in programming language level and without altering the original tool-chain of each type of accelerators in order to provide the flexibility for the application programmer. The framework provides a unified hierarchical model such that a Monte-Carlo simulation is divided into sub-tasks and distributed to the lower layers recursively. It is highly scalable as a simulation task can be distributed across different accelerators in a single server node, across different server nodes in a cluster or even across several heterogeneous clusters. Extensible dynamic scheduling policies can be designed in the distributor processes such that the sub-tasks can be allocated to the worker process based on the computational performance or even energy consumption.

5.2.1 Overall hierarchy

The overall framework for distributed Monte-Carlo simulation on a multi-accelerator heterogeneous cluster is shown in Fig. 5.1. There are two major processes in this framework: MC distributors and MC workers. MC distributors wait for the Monte-Carlo parameters and task size as a form of MC request from their parent MC distributor or from the user. The MC distributor then partitions the task and distributes the sub-tasks to their child MC distributors or MC workers. No simulation is done on the MC distributor. Their functionality is implied by their name – to distribute the Monte-Carlo simulation tasks to their connecting child processes. Each MC worker is responsible for the execution of part of the simulation. They pass the simulation parameters to the underlying “kernel” and get the partial simulation result back from it. In a multi-accelerator environment, each “kernel” holds a specific computational hardware resource such as FPGA, GPU or CPUs.

Fig. 5.1 only shows a two layer MC distributor network. In fact, the framework is highly scalable since there could be more than two and no upper limit for the number of layers of MC distributors. Additional layers of MC distributors could be inserted between the user node and the cluster. For example, when there are 3 heterogeneous clusters (A,B,C) from different organizations, they could collaborate by inserting a layer of 3 MC distributors namely DA, DB and DC. The MC distributer at the user node distributes the sub-tasks to DA, DB and DC. DA then further partitions and distributes the sub-tasks to the MC distributors of the nodes in cluster A. Similarly for DB and DC.

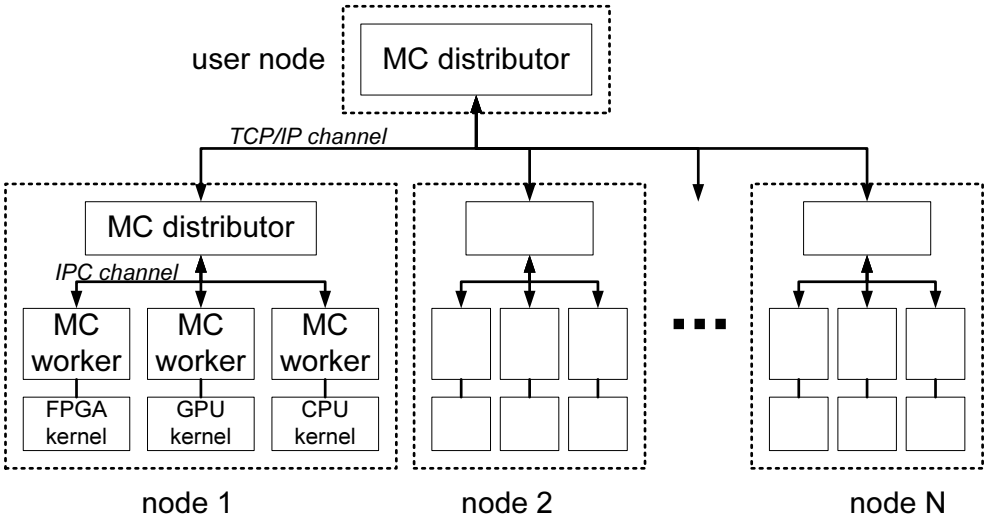

Figure 5.1: The overall framework.

5.2.2 MC processes

The MC workers are the main simulation units. Fig. 5.2 shows the work flow of the MC worker. The MC workers wait for the MC request (MC parameters and the task size), then forward the MC request to their computation hardware (FPGA, GPU or CPUs) and execute the kernel via the hardware driver. Therefore, the computation kernel can be optimised using the native tool-chain for each accelerator. When the computation results are returned, the MC workers report them to their parent MC distributor. The reported results include the aggregated simulation results and the actual completed task size by the kernel. The actual completed task size could differ from the MC request due to hardware specific constraints (e.g. number of cores and memory limit).

The MC distributors are key elements in the distributed Monte-Carlo framework. The work flow of the MC distributor is shown in Fig. 5.3. The MC distributors wait for the MC request from their parent

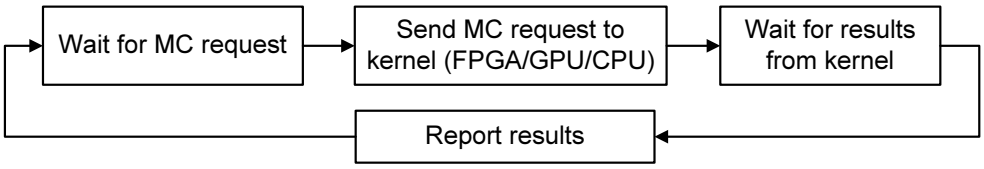

Figure 5.2: The work flow of MC workers.

process or user input, then they partition the MC request to several sub-tasks based on the scheduling policy. The partial MC requests for those sub-tasks are then sent to the child MC distributors or MC workers. When one of the child processes reports the partial results, the MC distributors aggregate the results until the task is completed (the sum of reported task completion size = the required task size in the MC request). When the task is not completed yet, the MC distributor adjusts the sub-task size for the reporting process according to the scheduling policy. Another partial MC request is sent to the reporting process with an updated sub-task size. When the task is completed, the MC distributors report the aggregated result to the parent process (or user). The "task size" discussed here can be the number of simulations, the number of particles or any form of computation tasks of that particular Monte-Carlo simulation.

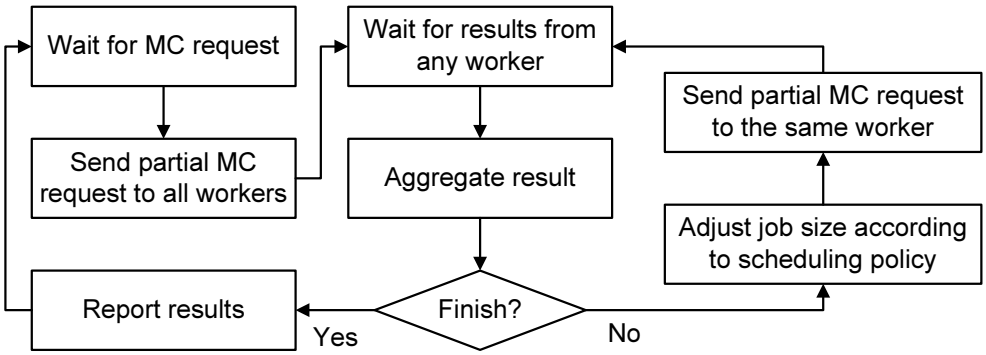

Figure 5.3: The work flow of MC distributors.

The intra-node communication between MC distributor and MC workers within the same node is realized by interprocess communication channel (IPC). The inter-node communication between MC distributors of different nodes is realized by the TCP/IP channel with MPI as the session layer.

5.3 Scheduling Policies

In a multi-accelerator heterogeneous cluster, the computational performance differs between different nodes and between different accelerators of the same node as well. Improper task distribution could lead to a drastic performance reduction.

For example, consider a node consisting of one FPGA and one CPU, and the processing speed of FPGA and CPU is 1000 simulations per second and one simulation per second respectively. If 2000 simulations are required and the MC distributor simply distributes 1000 simulations to the MC worker of FPGA and CPU equally at the beginning, the total execution time will be 1000 seconds and the FPGA will be idle for 999 seconds. Such inefficient task allocation leads to poor performance and imbalanced resource utilization. In contrast, if the MC distributor distributes one simulation to both MC workers and distributes another one simulation to them after they reported the result, the execution time is around 2 seconds computation time plus a large amount of message passing overhead and latency between hardware and software.

For the above simple example, one may be able to determine the "optimal task distribution" by pilot running the simulation in each of the devices and distribute the tasks according to their computational performance (1000:1 in this case) provided that the computational time is deterministic for each accelerator. However, such deterministic assumption is often invalid as many Monte-Carlo simulation problems involve non-deterministic run-time (such as solving PDEs). The computation performance for some devices (such as CPU) also depends heavily on the server status.

Therefore, the scheduling policy is a critical factor for the collaborative computing performance in a multi-accelerator heterogeneous cluster. Our solution involves introducing one static and two dynamic scheduling policies. The dynamic scheduling policies enable the task size allocated to the child processes to grow adaptively according to their performance. The performance evaluation of these policies will be discussed in Section 5.6. The initial task size for all child processes is defined as TS_{init}. The task size for child i at the jth time of simulation is defined as TS^i_j. Therefore, $TS^i_1 = TS_{init}$ for all i. The remaining uncompleted task size of the MC distributor is defined as R_d. The updating of R_d and aggregating of the returned results from MC workers are done by MC distributors before

and after each scheduling.

5.3.1 Constant-Size policy

The Constant-Size scheduling policy is the simplest form of static scheduling policy in which the task size stays constant for each child at all times. The Constant-Size scheduling policy is defined as:

$$TS^i_{j+1} = \min(TS^i_j, R_d) \tag{5.1}$$

The number of TS_{init} (TS^i_1) is critical for constant-size scheduling policy. A small value of TS_{init} might cause a large amount of message passing overhead. A large value of TS_{init} might cause the slowest MC worker to affect the overall computation performance.

5.3.2 Linear-Incremental policy

The Linear-Incremental scheduling policy is defined as:

$$TS^i_{j+1} = \min((TS^i_j + c), R_d), \tag{5.2}$$

where c is a constant. It is a dynamic scheduling policy which increases the task size of the MC worker linearly. Eventually, the task size allocated for the faster child TS^{i1}_{j1} is larger than the slower child TS^{i2}_{j2} as $j1 > j2$. The task size allocated to each child will grow proportionally to their corresponding processing rate slowly.

5.3.3 Exponential-Incremental policy

The Exponential-Incremental scheduling policy is defined as:

$$TS^i_{j+1} = \min((TS^i_j \times m), R_d), \tag{5.3}$$

where m is a constant. This dynamic scheduling policy increases the task size of the MC worker exponentially with a factor of m. Similar to Linear-Incremental policy, the task size allocated for the faster child will be larger than the slower child after a period of time. The task size allocated to the child processed will grow proportionally to their processing rate at a much faster rate.

5.3.4 Throughput-Proportional policy

The Throughput-Proportional scheduling policy is defined as:

$$TS^i_{j+1} = \min\left(\left(\frac{Throughput^i}{\sum Throughput^i} \times TS_{max}\right), R_d\right), \tag{5.4}$$

where TS_{max} is a constant representing the maximum total task size for all MC workers. The $Throughput^i$ is calculated by the computational throughput of that MC worker in its previous task. It is defined as $TS^i/UsedTime$. This dynamic scheduling policy aims at allocating the tasks proportionally according to the worker's computational throughput in the previous task.

5.3.5 Energy-Proportional policy

The Speed-Proportional scheduling policy is defined as:

$$TS^i_{j+1} = \min\left(\left(\frac{EnergyPerTask^i_j}{\sum EnergyPerTask^i_j} \times TS_{max}\right), R_d\right), \tag{5.5}$$

where TS_{max} is a constant representing the maximum total task size for all MC workers. The $EnergyPerTask^i$ is calculated by the with the power times the used time of that MC worker in its previous task. It is defined as $\frac{DynamicPower^i \times UsedTime}{TS^i_j}$. This dynamic scheduling policy aims at allocate the tasks proportionally according to the worker's computational energy such that each underlying worker

consumes the similar amount of energy. The dynamic power of MC worker i ($DynamicPower^i$) can be determined with a power meter.

5.3.6 Other possible policies

Apart from the basic scheduling policies stated above, we can also employ a mixed scheduling policy, such as using Linear-Incremental policy at the beginning and then change the policy to Constant-Size after certain iteration. The scheduling policy in this framework is highly flexible and can be optimized for any goal. It is up to the application engineer to design their own scheduling policies for their own target. For example, If the energy usage of the MC worker can be profiled and fed back to the MC distributor, an Energy-Equal scheduling policy can be defined such that each MC worker consumes the same amount of computational energy. An energy-efficient MC worker keeps computing most of the time, while a less energy-efficient MC worker will be idle occasionally to keep the same amount of energy usage. The idle accelerators can therefore be used in another application.

5.4 Applications

We have implemented two applications in our proposed framework, namely

- Asian option pricing using control variate method,
- GARCH asset simulation.

5.4.1 Asian option pricing using control variate method

Arithmetic Asian options provide a payoff depending on the arithmetic average price of the underlying during the option life-time. This averaging makes arithmetic Asian options cheaper and less sensitive to market manipulation, but also means there is no closed-form solution for the pricing. The payoff equation is shown in Section 2.1 as Equation 2.8.

Monte-Carlo methods provide an accurate way to price Asian options, but have slow convergence, so a huge number of simulations is needed. Therefore, arithmetic Asian options are perfect candidates to be priced using a multi-accelerator heterogeneous cluster. In Chapter 3, we present the FPGA and GPU accelerated designs of Asian option pricer based on control variate Monte-Carlo method. In this chapter, we use Asian option pricer as an application in our presented multi-accelerator heterogeneous cluster framework. The computing is performed by all FPGAs, GPUs and CPUs collaboratively in the cluster.

5.4.2 GARCH asset simulation

Our second application is the simulation of GARCH volatility model. The volatility of an underlying asset is not constant in reality. One solution is assuming a stochastic volatility such as GARCH (1,1) model as presented in Section 2.1.3. We simulate the volatility in each time step according to Equation 2.9 with an additional Gaussian random number generator and price an European option accordingly.

5.5 FPGA and GPU designs

5.5.1 FPGA kernels

For both applications, we design the FPGA kernels as shown in Fig. 5.4. There are two main types of components in the design: one or more identical Monte-Carlo cores, and a single shared Coordination Block (CB). The MC cores contain a Gaussian random number generator (GRNG) core and a simulation core. The GRNG uses the piecewise linear generation method [107] which produces a stream of 24-bit fixed-point random numbers, with a period of 2^{128}.

The MC core in our Asian option pricing application is capable of generating random asset price paths, calculating payoffs of the Asian option and European option, and accumulating the payoffs and payoffs related statistical result. In other words, each MC core is capable of executing the MC

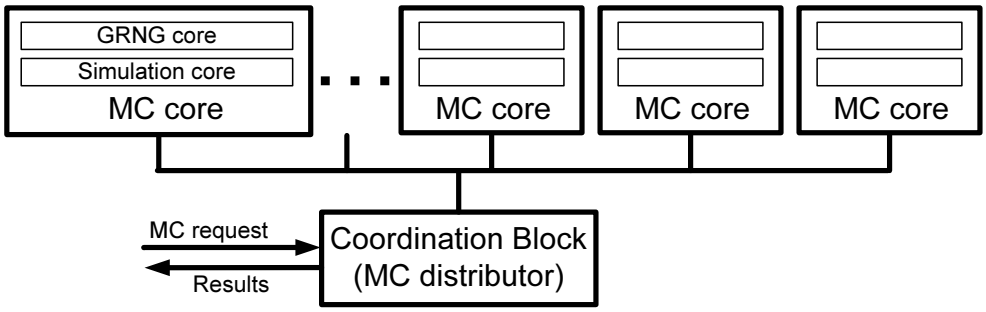

Figure 5.4: The hardware design of FPGA kernel.

part of CVMC Algorithm. Multiple identical MC cores are instantiated to make the maximum use of the device. The required number of simulations is distributed equally to each MC core.

The MC core in our GARCH asset simulation application is responsible for the generation of random numbers, simulation of the stochastic volatility movement, and simulation of the asset movement with respect to the volatility.

The Coordination Block (CB) manages the MC cores, allowing them to work in parallel to price the same option. The CB is also responsible for communicating with the host by accepting MC request and reporting MC results. The Gaussian random number generators in MC cores are also initialized by the CB. Different sequences of bits are connected to different Gaussian random number generators as the random seeds. The CB can also be viewed as a MC distributor employing Constant-Size scheduling policy. Constant-Size scheduling policy is the best choice as all MC cores finish the computation in the exact same cycle.

The hardware architecture of the simulation core for GARCH asset simulation is shown in Fig. 5.5. The grey boxes in Fig. 5.5 indicates the pipelined registers inserted to balance the number of pipeline stages for all feedback updating loops for the stochastic volatility and asset prices.

Our FPGA kernels target a Xilinx xc5vlx330t FPGA chip on an Alpha Data ADM-XRC-5T2 card, which contains 51,840 slices, 192 DSP48E and 324 BlockRAM units. We design our hardware architectures for both applications manually in VHDL to maximize performance. The design is synthesized, mapped, placed and routed using Xilinx ISE 10.1.03. Single precision floating point arithmetic is used. The number of MC cores for Asian option pricing is 10. The number of MC cores for GARCH asset simulation is 12. The summary of resource consumption for both applications is

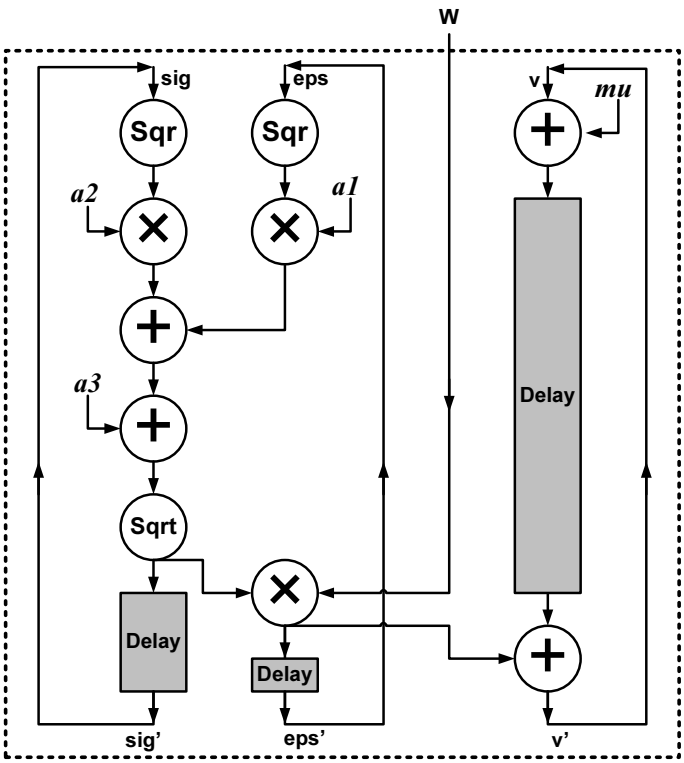

Figure 5.5: The hardware architecture of GARCH asset simulation core.

Table 5.1: xc5vlx330t FPGA resource consumption

	Asian option pricing		**GARCH simulation**	
MC Cores	10		12	
Resource	**Used**	**%**	**Used**	**%**
Slices	44,118	85%	37,205	71%
FFs	130,195	62%	118,261	57%
LUTs	79,587	38%	59,313	28%
RAM	10	3%	12	3%
DSP48Es	180	93%	192	100%

shown in Table 5.1.

5.5.2 GPU kernels

Graphics Processing Units (GPUs) have been used for acceleration in many application domains. They are Single Instruction Multiple Data (SIMD) computing devices. Parallelizable tasks are executed on the GPU as a "kernel" by a computation grid. The "kernel" is executed by all threads in parallel with the same code, but on different sets of data.

The co-processing flow of GPUs provides a good match to our design framework. The MC request

containing MC parameters and task size is firstly copied to the GPU data memory. The MC results are copied back to the memory of the MC worker after the execution of the GPU kernel.

We design our CUDA kernels for Asian options pricing and GARCH asset simulation using two procedures, namely Gaussian random number generator procedure, and a path simulation procedure.

In the Gaussian random number generator procedure, uniform random numbers are first generated and stored in the GPU's global memory space using the Mersenne Twister algorithm in parallel with all threads. Then the uniform random numbers are transformed into Gaussian random numbers using the Box-Muller method [110]. The memory space for storing Gaussian random numbers is allocated by the MC worker once at the beginning. In our target implementation on nVidia Tesla C1060, 2GBytes are allocated, which can accommodate 512MBytes of single-precision Gaussian random number. Such memory constraints may lead to the completed number of simulations to be less than the requested number of simulations, which will be notified by the MC worker to its parent.

In the path simulation procedure of Asian option pricing, each thread simulates the price movement path as in the CVMC Algorithm and computes the Asian and European option result in the shared memory. In the path simulation procedure of GARCH asset simulation, each thread simulates the volatility dynamics as in Equation 2.9 and updates the asset price accordingly.

5.5.3 CPU kernels

In both applications, we implement the CPU kernels in the C language and use the Intel Math Kernel Library (MKL) for the random number generation. The Mersenne Twister algorithm is used as the random number base and Box-Muller is used for Gaussian number transformation. The code is compiled with Intel compiler (icc) 11.1 with -O3 maximum speed optimization options and SSE enabled. OpenMP is used to parallelize the computation with the multi-core capabilities of CPUs. The parallel FOR `#pragma` directive is used to parallelize the main loop, so that loop iterations can be executed in parallel using multiple CPUs.

Table 5.2: Performance of Asian option pricing

	FPGA	**GPU**	**CPUs**	**Collaboration (Upper bound)**	**Collaboration (Actual)**
Brand	Xilinx	Nvidia	AMD	-	-
Type	Virtex-5	Tesla	Phenom	-	-
Model	xc5vlx330t	C1060	9650	-	-
Freq.	200MHz	1.3GHz	2.3GHz	-	-
Qty	1	1	2	1+1+2	1+1+2
Time	18.3s	25.5s	399.6s	10.4s	11.8s
Speedup	21.8x	15.7x	1x	38.4x	33.8x

5.6 Performance Evaluation

In this section, we evaluate the results of the two applications used in our framework. We investigate the effect of different dynamic scheduling policies on the computational performance using the Asian options pricing engine in Section 5.6.1. The performance, energy consumption and efficient accelerator allocation will be discussed using the GARCH asset simulation example in Section 5.6.2. We carry out our experiments on an accelerator cluster, which consists of 8 server nodes. Each server node consists of two AMD Phenom 9650 Quad-Core 2.3GHz CPUs, one nVidia Tesla C1060 GPU and one Xilinx Virtex-5 xc5vlx330t FPGA.

5.6.1 Dynamic scheduling analysis of a single node

The performance of different accelerator combinations for the pricing of an Asian call option is studied. The computational and load-balancing performance of different dynamic scheduling policies is also presented. We choose a 10-year arithmetic Asian call option with parameters $S_0 = 100$, $K = 105$, $v = 0.15$, $r = 0.1$, $T = 10$ and $steps = 365$. The number of Monte-Carlo simulations is 10,000,000.

The performance comparison for the pricing of Asian option with individual accelerators and multi-accelerator collaboration is shown in Table 5.2. The optimized multi-threaded CPU kernel executed by two AMD Phenom 9650 quad-core CPUs is used as the comparison reference. It can be seen that a speedup of 21.8 times is achieved by the xc5vlx330t FPGA. For the GPU, a speedup of 15.7 times

is achieved by the Tesla C1060. For the collaboration with FPGA, GPU and 2 CPUs, a speedup of 33.8 times is achieved using Linear-Incremental policy with $TS_{init} = 1000$.

The collaborative computation time results of using FPGA, GPU and CPU kernels in one node with different scheduling policies are shown in Fig. 5.6. The Constant-Size, Linear-Incremental and Exponential-Incremental policies are used in the MC distributor with different TS_{init} values. From the figure, when TS_{init} is small, the Constant-Size policy suffers from large overhead and thus long computation time. For large TS_{init}, all policies suffer from reduced performance due to the waiting of the completion of the slowest kernel. The shortest computation time achieved is 11.8 seconds when Linear-Incremental policy is used with $TS_{init} = 1000$, and it is used as the result for Table 5.2.

The theoretical upper bound of the collaborative computation time is defined by assuming no communication overhead between devices such that the aggregated throughput is the sum of the throughput of each device. It is defined with the following equation:

$$t_{tc} = \left(\sum t_i^{-1}\right)^{-1} \tag{5.6}$$

where t_{tc} is the theoretical upper bound of the collaborative computation time, t_i is the computation time using device i. The theoretical upper bound of the collaborative computation time using all computational devices is 10.4 seconds for the pricing of Asian option. In our experiments, our best timing achieved is 11.8 seconds which is within 14% of the theoretical upper bound. This best timing is achieved by using a Linear-Incremental scheduling policy. Linear-Incremental scheduling policy is an example policy aiming at slowly increasing the allocated task sizes for the computing devices in order to match their throughput ratio. Therefore, we expect that a collaborative computation time closer to the theoretical upper bound could be achieved if the scheduling policy can be further optimised such that the allocated task sizes can match the device throughput quicker and the communication messages can be reduced.

In this Asian option pricing application, maximum performance is achieved when $TS_{init} = 1000$ under Linear-Incremental policy. However, other applications may achieve the maximum performance under different policy and with different variables. It is because each application has its particular

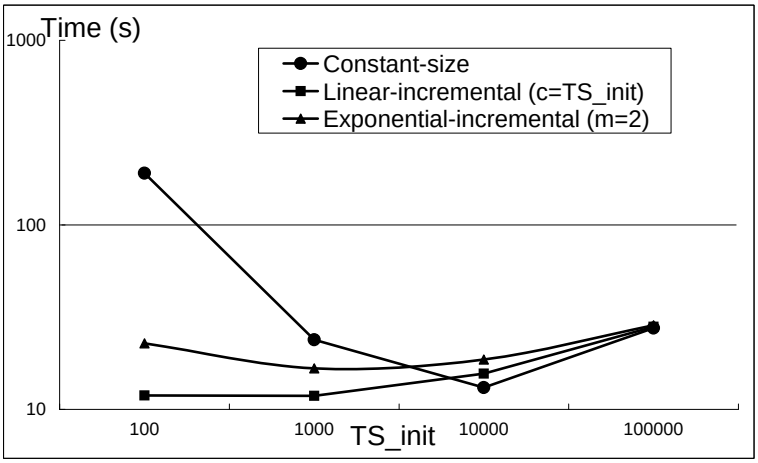

Figure 5.6: The performance comparison for different scheduling policies.

set of parameters, and different communication overhead. Fig. 5.6 is just a demonstration of how the performance can differ with different starting task size under different dynamic scheduling policies.

5.6.2 Performance, energy and efficiency analysis of accelerator allocation of a cluster

Acceleration performance versus energy consumption is an important factor when considering the efficiency of an accelerator. As a result, it is also one of the main concerns in our evaluation of the proposed framework and we use the GARCH asset simulation application for the evaluation. We study 5 different methods for allocating computational devices in the cluster for collaborative computation:

- CPUs only: use two Phenom CPUs in each node
- FPGA only: use one xc5vlx330t FPGA in each node
- GPU only: use one Tesla C1060 GPU in each node
- FPGA and GPU: use one xc5vlx330t FPGA and one Tesla C1060 GPU together in each node
- FPGA, GPU and CPUs: use one xc5vls330t, one Tesla C1060 and two Phenom CPUs together in each node

We measure the additional power consumption for computation (APCC) with a power monitor. APCC is defined as the power usage during the computation time (run-time power) minus the power usage

Table 5.3: Performance of the GARCH asset simulation of different accelerators and number of collaborative nodes

Using 2 CPUs per node only				
Number of nodes	1	2	4	8
Time (ms)	1,162,725	660,687	360,129	162,018
APCC (W)	49	78	146	300
AECC (J)	56,973.53	51,533.58	52,587.83	48,605.40
Using FPGA per node only				
Number of nodes	1	2	4	8
Time (ms)	38,969	19,691	10,458	5,418
APCC (W)	5	11	21	43
AECC (J)	194.85	216.60	219.62	232.97
Using GPU per node only				
Number of nodes	1	2	4	8
Time (ms)	64,299	32,308	16,310	8,252
APCC (W)	97	192	350	676
AECC (J)	6237.00	6203.14	5708.50	5578.35
Using FPGA and GPU per node				
Number of nodes	1	2	4	8
Time (ms)	24,706	12,822	6,825	3,636
APCC (W)	102	203	392	683
AECC (J)	2520.01	2602.86	2675.40	2483.40
Using both FPGA, GPU and 2 CPUs per node				
Number of nodes	1	2	4	8
Time (ms)	24,595	12,884	7,167	4,391
APCC (W)	130	270	506	908
AECC (J)	3197.35	3478.68	3626.50	3987.03

at idle time (static power). The static power of each cluster node is approximately 210W. In other words, APCC is the dynamic power consumption for that particular computation. The additional energy consumption for computation (AECC) is defined by the following equation:

$$AECC = APCC \times Total\ Computational\ Time. \tag{5.7}$$

Therefore, AECC measures the actual additional energy consumed for that particular computation.

The speed and power consumption of the GARCH asset simulation for different accelerator combination in the multi-accelerator cluster is studied. The number of Monte-Carlo simulations is 100,000,000 and one asset is simulated. Linear-Incremental scheduling policy is employed on each MC distributor of the cluster node with $TS_{init} = 1000$. Constant-Size scheduling policy is employed at the higher

level MC distributor in the user node with $TS_{init} = 100M$, $50M$, $25M$ and $12.5M$ for a cluster with 1, 2, 4 and 8 nodes. The computation time, APCC and AECC results are shown in Table 5.3.

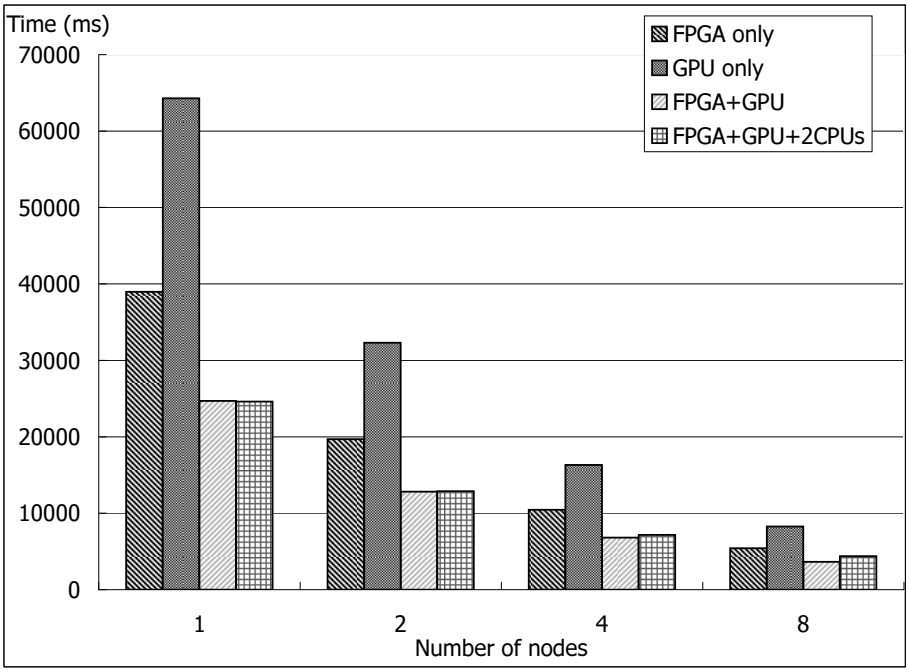

Figure 5.7: The computation time of GARCH asset simulation.

As expected, an increase in the number of active nodes generally decreases the time for computation. From the results, we can see that the cluster activating 8 FPGAs and 8 GPUs as MC worker processes is the fastest (3.6s) even when compared with the cluster activating all 8 FPGAs, 8 GPUs and 16 CPUs as MC worker processes. This can be explained by the fact that activating CPUs as MC worker processes decreases the system response time. Therefore, the computational performance gain of using CPUs as MC worker processes is offset by the decrease in response time of the MC distributer process, reducing the overall performance in this application. The cluster activating 8 xc5vlx330t FPGAs and 8 Tesla C1060 GPUs is 44 times faster than the cluster activating 16 AMD Phenom 9650 CPUs. A graphical summary about the computational time is shown in Fig. 5.7.

The increased number of active nodes increases the APCC proportionally. However, the AECC remains roughly the same level as the computation time is decreased proportionally at the same time. We can see from the result that the cluster using a single FPGA has the lowest AECC. A graphical summary about the AECC is shown in Fig. 5.8.

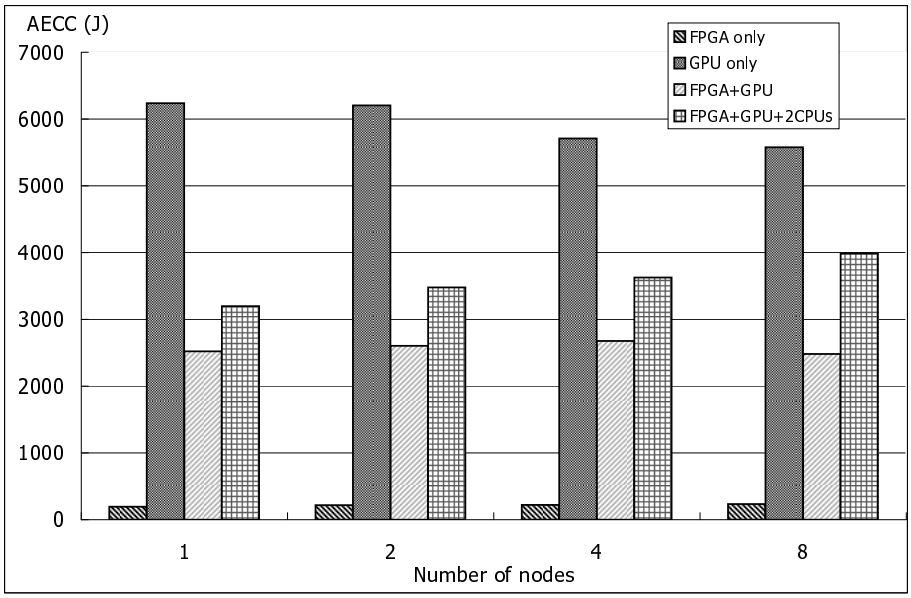

Figure 5.8: The AECC of GARCH asset simulation.

We used an approach for identifying speed and energy efficient accelerator allocation, called Efficient Allocation Line (EAL). A scatter plot graph is firstly constructed with the computation time versus energy consumption for all accelerator allocation combinations. The EAL is then constructed by drawing a line linking the leftmost and bottommost allocations. The allocations of computational devices along the EAL are called "efficient" compared with the other allocations, as they are either energy efficient (the lowest energy consumption at a given computational time budget), or speed efficient (the lowest computational time at a given energy budget). In other words, the allocations of computational devices along the EAL are the Pareto optimum points. Fig. 5.9 shows the computation time versus the energy consumption (AECC) of different accelerator allocations for the GARCH asset simulation in our 8-node cluster. The solid line is the EAL.

In this GARCH asset simulation application, FPGA is both faster and more energy efficient than the other two computational devices (GPU and CPU). We can simply allocate as many FPGAs as possible in the cluster. However, in the case of one accelerator is more speed efficient, but less energy efficient than the others, identifying the optimized device allocation will be much more challenging. The EAL can then be used for optimizing accelerator allocation. A dynamic scheduling policy based on the EAL could also be developed such that it allocates the tasks to the accelerators based on a certain energy budget or time budget which can vary during run time.

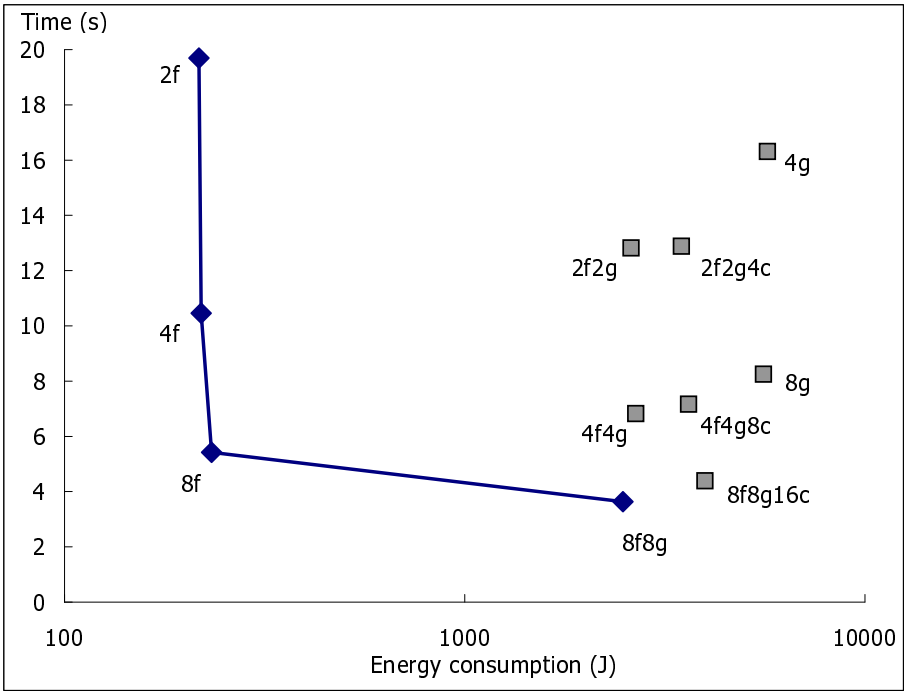

Figure 5.9: The computation time and energy consumption for GARCH asset simulation in our cluster. The solid line is the Efficient Allocation Line (EAL). 2f2g4c denotes a design with 2 FPGAs, 2 GPUs and 4 CPUs.

5.7 Summary

In this chapter, we propose a dynamic scheduling Monte-Carlo framework for collaborative computation in a multi-accelerator heterogeneous cluster. The load balancing process is automated by employing dynamic scheduling policies using the proposed framework. The framework is scalable and extensible for a variety of dynamic scheduling policies. We have shown that the proposed framework is viable by mapping two applications involving financial computation.

From our results, the overall performance of a Monte-Carlo simulation can be improved by allowing heterogeneous accelerators to work collaboratively. We explore different schemes of scheduling the workloads to the processing units to better utilize the computing resources. We also explore the speed and energy consumption trade-off for different accelerator allocation, and we propose the Efficient Allocation Line (EAL) as a method to identify the most efficient accelerator allocations.

We shows that pricing an Asian option using an FPGA, a GPU and two CPUs collaboratively on a single node under our proposed framework is 33.8 times faster than using two CPUs only. We also shows that a cluster using 8 FPGAs and 8 GPUs is 44 times faster than a cluster using 16 CPUs for the

GARCH asset simulation problem under our proposed framework. There is no comparable related work or related performance result as we known. The closest application is a N-body simulation problem described in [94]. They demonstrated that a cluster using 16 FPGAs and 16 GPUs is 22 times faster than a cluster using 32 CPUs in a N-body simulation using manual task partitioning. They used the same types and same ratio of accelerators as in this chapter. We shows that we can achieve a better speedup figure (44 times faster) than them (22 times faster) using our proposed dynamic task partitioning and scheduling scheme. However, N-body simulation and Monte-Carlo simulation are two different types of problem and should not be compared directly. Another related work on multi-accelerator heterogeneous cluster is the Quadro Plex (QP) Cluster presented in [93]. Their cluster consists of both FPGAs and GPUs but there is no application performed using collaborative computing. A cosmology data analysis application running on 8 FPGAs is 6.3 times faster than 8 CPUs in their cluster.

Chapter 6

Optimising Performance of Monte-Carlo Methods with Mixed Precision

6.1 Motivation

The ability to support customizable data-paths of different precisions is an important advantage of reconfigurable hardware. Reduced-precision data-paths usually have higher clock frequencies, consume fewer resources and offer a higher degree of parallelism for a given amount of resources compared with full precision data-paths. In Chapter 3, we presented the design and optimisation techniques to use FPGA as an accelerator for option pricing with control variate Monte-Carlo method. In this chapter, we aim at increasing the performance further by exploiting the precision flexibility of reconfigurable hardware.

This chapter introduces a novel mixed precision methodology for accurate Monte-Carlo simulations. The key difference between the proposed methodology and previous FPGA Monte-Carlo designs lies in the way finite precision errors are handled. Instead of keeping the output error within certain tolerance, the FPGA data-path is initially constructed with an aggressively reduced precision. This produces a result with finite precision error exceeding a given error tolerance. An auxiliary sampling process using both a high precision reference and the reduced precision is then used to correct the error. The output accuracy of the proposed technique is not limited by the precision of the data-paths.

The proposed methodology can also exploit the synergy between different processors in a reconfigurable accelerator system. Reference precision computations required in the auxiliary sampling can be carried out by a Central Processing Unit (CPU) in a host PC, while reduced precision computations target customized data-paths on the FPGA. This allows different processors to work in precisions for which they are specialised, leading to higher overall performance.

The major contributions of this chapter are:

- error analysis that separates finite precision error and sampling error for reduced precision Monte-Carlo simulations, and a novel mixed precision methodology to correct finite precision errors through auxiliary sampling (Section 6.2 and Section 6.3).
- techniques for partitioning workloads of different precisions for auxiliary sampling to a reconfigurable accelerator system consisting of FPGA(s) and CPU(s) (Section 6.4).
- optimisation method based on an analytical model for the execution time of a Monte-Carlo simulation on a reconfigurable accelerator system, and Mixed Integer Geometric Programming to find optimal precision for the FPGA's data-paths and optimal resource allocation (Section 6.5).
- evaluation of the proposed methodology using four case studies, with performance gains of 2.9 to 7.1 times speedup over FPGA only designs using double precision arithmetic. The mixed precision designs are also 44 to 106 times faster and 41 to 104 times more energy efficient compared with software design on a quad-core CPU (Section 6.6 and 6.7).

6.2 Error Analysis

This section provides an error analysis for Monte-Carlo simulations. The total error ϵ_{total} of a Monte-Carlo simulation can be divided into two components: Sampling error ϵ_S and finite precision error ϵ_{fin}. Sampling error ϵ_S is the error due to having a finite number of samplings and finite precision error ϵ_{fin} is due to non-exact arithmetic. The finite precision error ϵ_{fin} is accumulated in a datapath due to truncating or rounding of the number representation after each operation. It is assumed that when a

sufficiently accurate precision, such as IEEE-754 double precision, is used, the finite precision error is negligible. We call this value the **reference precision**. Let us recall some background knowledge from Chapter 2, Section 2.2.1 and begin with sampling error. For a sequence of mutually independent, identically distributed random variables, X_i from a MC simulation, If, $\text{Sum}_N = \sum_{i=1}^{N} X_i$, and the expected value, I, exists, the Weak Law of Large Numbers states that if $p(x)$ is the probability of x, for $\epsilon > 0$, the approximation approaches the mean for large N [46],

$$\lim_{N\to\infty} p\left(|\frac{\text{Sum}_N}{N} - I| > \epsilon\right) = 0 \tag{6.1}$$

Moreover, if the variance σ^2 exists, the Central Limit Theorem states that for every fixed a,

$$\lim_{N\to\infty} p\left(\frac{\text{Sum}_N - NI}{\sigma\sqrt{N}} < a\right) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{a} e^{-z^2/2}dz \tag{6.2}$$

that is, the distribution of the standard error is normal.

In practice, we must deal with finite N. If the sampling function f represents a mathematical expression defining the quantity being sampled, $\vec{x_i}$ is the input vector of length s from a uniform distribution [1] $[0,1)^s$, N is the number of sample points and $\langle f_H\rangle_N$ is the sampled mean value of the quantity, the conventional MC sampling process[2] can form an approximation to I,

$$I \approx \langle f_H\rangle_N = \frac{1}{N}\sum_{i=1}^{N} f_H(\vec{x_i}) \tag{6.3}$$

Thus a sampling error $\epsilon_S(\langle f_H\rangle_N) = I - \langle f_H\rangle_N$ with approximately normal distribution is introduced:

$$\epsilon_S(\langle f_H\rangle_N) \sim \mathcal{N}(0, \sigma_{f_H}^2/N) \tag{6.4}$$

Equation 6.4 shows that the bound of the sampling error can be constructed as a confidence interval. Given the same confidence level, the interval is proportional to the standard deviation of the sam-

[1] Some MC simulations require non-uniformly distributed $\vec{x}$ values, for example in many option pricing simulations normally distributed $\vec{x}_i$ are required.

[2] Throughout the chapter, we use the subscript H and L to denote quantities evaluated with the reference precision arithmetic and the reduced precision arithmetic respectively. We use $\langle X\rangle$ to denote the sampled mean value of a random variable X and $\langle X\rangle_N$ to denote the sampled mean value of X calculated by N samples.

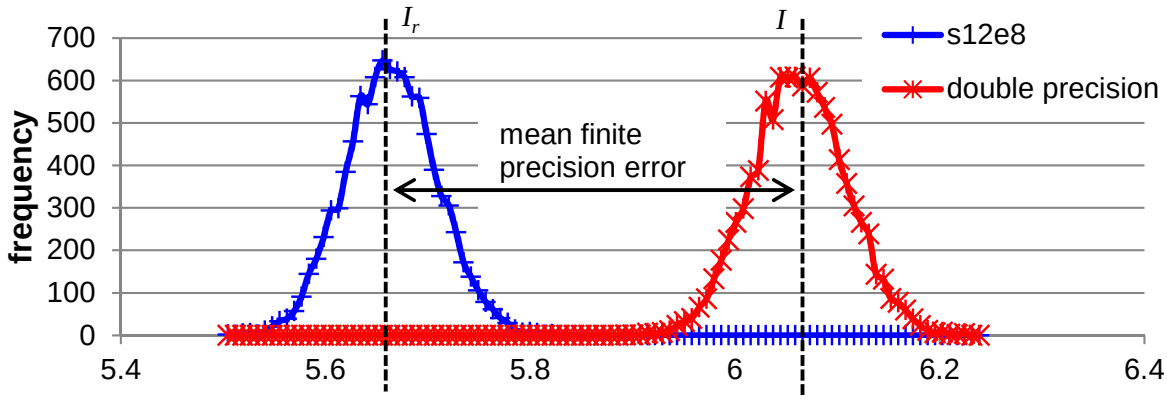

Figure 6.1: Distribution of 10k runs of a reduced precision and a double precision Monte-Carlo.

pling function, σ_{f_H}, and inversely proportional to the square root of the number of sample points, N. Hence quadrupling the number of sample points halves the confidence interval of the sampling error $\epsilon_S(\langle f_H \rangle_N)$. We assume there is no precision error associated with the sampling error. In FPGA designs, the sampling function f is usually evaluated using a low reduced precision, f_L, compared to the high reference precision, f_H. The reduced precision design is smaller and faster, at the expense of higher error. However, reduced precision increases the error. We call the difference between a reference precision computation and a reduced precision computation, $f_H(x) - f_L(x)$, the finite precision error.

6.3 Mixed precision methodology

Our novel mixed precision methodology is motivated by two ideas. First, we can correct the finite precision error when both its magnitude and sign are known. Second, in Monte-Carlo simulations, we are only interested in the finite precision error in the final result but not the finite precision errors of individual sample points.

When a reduced precision data-path is used in a Monte-Carlo simulation, the reduced precision expected value I_r is approximated by the following equation, where N_L is the number of sample points:

$$I_r \approx \langle f_L \rangle_{N_L} = \frac{1}{N_L} \sum_{i=1}^{N_L} f_L(\vec{x_i}) \tag{6.5}$$

Due to the effect of finite precision error, the reduced precision sample mean $\langle f_L \rangle_N$ cannot be used to approximate the expected value I directly as I might not equal to I_r. We define the difference of the two expected means as the mean finite precision error, $\mu_{\epsilon_{fin}}$, where

$$\mu_{\epsilon_{fin}} = I - I_r \tag{6.6}$$

Figure 6.1 shows the distributions of Monte-Carlo simulations using a reduced precision (s12e8) data-path and a double precision data-path of for pricing Asian options. The reduced precision floating operators are from Xilinx core generator which employs round-to-nearest rounding mode. In each MC simulation, N = 32,768 sample points are used and each of the reduced and double precision MC simulation is repeated for 10,000 times with different random seeds. As shown in the figure, the magnitude of the mean finite precision error $\mu_{\epsilon_{fin}}$ between the expected value of I and I_r is significant. The error bound of an MC simulation using this reduced precision data-path would be at least $2\mu_{\epsilon_{fin}}$, and cannot be improved by increasing the number of sample points. This is the fundamental limit of conventional reduced precision MC simulations.

To find both the magnitude and the signs of the mean finite precision error $\mu_{\epsilon_{fin}}$, we define an auxiliary sampling function $f_a(\vec{x})$:

$$f_a(\vec{x}) = f_H(\vec{x}) - f_L(\vec{x}) = \epsilon_{fin}(\vec{x}) \tag{6.7}$$

where ϵ_{fin} is the finite precision error for each $\vec{x}$. Therefore, with an sufficient large sample size N_a, we can approximate the mean finite precision error $\mu_{\epsilon_{fin}}$:

$$\mu_{\epsilon_{fin}} \approx \langle f_a \rangle_{N_a} = \frac{1}{N_a} \sum_{i=1}^{N_a} f_a(\vec{x_i}) \tag{6.8}$$

The sampling error of this auxiliary sampling $\epsilon_S(\langle f_a \rangle_{N_a}) = \mu_{\epsilon_{fin}} - \langle f_a \rangle_{N_a}$ is approximately normal distributed:

$$\epsilon_S(\langle f_a \rangle_{N_a}) \sim \mathcal{N}(0, \sigma^2_{f_a}/N_a) \tag{6.9}$$

Finally, we can approximate the true mean I by two sets of sampling:

$$I_{mixed} = \langle f_L \rangle_{N_L} + \langle f_a \rangle_{N_a} \tag{6.10}$$

$$\begin{aligned} E(I_{mixed}) &= E(\langle f_L \rangle_{N_L}) + E(\langle f_a \rangle_{N_a}) \\ &= I_r + (I - I_r) = I \end{aligned} \tag{6.11}$$

As shown in Equation 6.11, the expected value of the auxiliary sampling is $I - I_r$. Hence the expected mean of the mixed precision approximation I_{mixed} is exactly the same as the expected mean I computed in reference precision. Equation 6.10 can thus be viewed as the reduced precision sample mean plus the correction for the mean finite precision error.

Since two samplings are used in the proposed mixed precision methodology, there are two sampling errors in the result and they can be found using Equation 6.13 and 6.14. As both sampling errors are approximately normally distributed, their sum is also approximately normally distributed and has a variance equal to the sum of their individual variances as shown in Equation 6.15 if two sets of uncorrelated random numbers are used. By using the proposed mixed precision methodology, we effectively replace the finite precision error of reduced precision data-paths by the sampling error of the auxiliary sampling. A confidence interval can also be constructed using the combined variance.

$$\epsilon_S(I_{mixed}) = \epsilon_S(\langle f_L \rangle_{N_L}) + \epsilon_S(\langle f_a \rangle_{N_a}) \tag{6.12}$$

$$\epsilon_S(\langle f_L \rangle_{N_L}) \sim \mathcal{N}(0, \sigma^2_{f_L}/N_L) \tag{6.13}$$

$$\epsilon_S(\langle f_a \rangle_{N_a}) \sim \mathcal{N}(0, \sigma^2_{f_a}/N_a) \tag{6.14}$$

$$\epsilon_S(I_{mixed}) \sim \mathcal{N}(0, \sigma^2_{f_L}/N_L + \sigma^2_{f_a}/N_a) \tag{6.15}$$

Although the proposed mixed precision methodology is analysed mathematically, we also show its

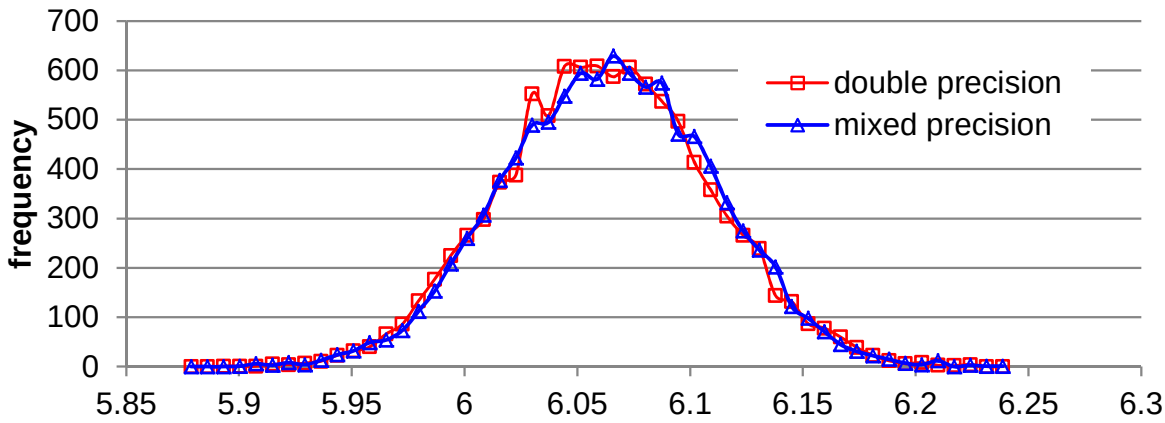

Figure 6.2: Distribution of 10k runs of a mixed precision and a double precision Monte-Carlo.

desired effect through experiments. Using Equation 6.15, we find that a mixed precision MC run using a precision of s12e8 with N_a = 1078 and N_L = 33,773 should yield the same error as a double precision sampling with N = 32,768. We repeat both the mixed precision and the double precision MC for 10,000 times using different random seeds, and their distributions are shown in Fig. 6.2. Note that both distributions have roughly the same variance and the same mean. The result agrees with our mathematical model and no finite precision error exists between the double precision Monte-Carlo and our mixed precision Monte-Carlo runs.

The proposed mixed precision methodology provides several advantages over previous FPGA designs.

1. The final result is adjusted with an approximated mean finite precision error $\mu_{\epsilon_{fin}}$. This is a novel approach which enables us to obtain a probably more accurate result by adjusting the reduced precision result instead of passively finding the error bound.

2. Since there are only sampling errors in the output, we can achieve a more accurate result by increasing the number of sample points N_L and N_a.

3. The methodology is independent of the function f. Therefore, it could be applied to other Monte-Carlo simulation problems directly without performing accuracy analysis of the function.

Although the proposed mixed precision methodology enables us to aggressively exploit reduced pre-

cision data-paths while maintaining the accuracy of the final result using auxiliary sampling, each auxiliary sampling still requires a costly evaluation of the sampling function f at the reference precision.

The effectiveness of the proposed technique depends heavily on how resources are allocated among the reduced precision hardware and auxiliary sampling hardware. To find the optimal resource allocation, we should consider a number of factors such as the cost of evaluating f_L and f_H, the area available on the FPGA, the bandwidth between the FPGA and CPU, and the reduced precision values being used.

In the next section, we propose different schemes for partitioning workloads. An analytical model is developed in Section 6.5 based on the partitioning schemes which enables us to find the optimal resource allocation and optimal reduced precision using mixed integer geometric programming.

6.4 Workload partitioning

Central Processing Units (CPUs) are optimised for standard precisions such as IEEE-754 single/double precision. CPUs can also employ reduced precision via multiple precision software libraries such as MPFR [111]. Multiple standard precision instructions are required to complete a reduced precision computation even if the reduced precision format has a smaller wordlength. Hence, it is usually not cost effective to use CPUs for reduced precision computations. On the other hand, FPGA data-paths are customizable. Lower precision are usually preferred over higher precision ones because they usually have higher clock frequency, consume less resources and allow higher degrees of parallelism given the same amount of resources. It is thus better to perform reduced precision computations on the FPGA and leave reference precision computations to the CPU.

Since the sampling of $\langle f_L \rangle_{N_L}$ involves only reduced precision evaluations of f, we assume it is achieved by using reduced precision sampling data-paths on FPGA as shown in figure 6.3. A seed is fed into the random number generator from the CPU. The random numbers are converted into the reduced precision format and scaled to the sampling domain. Although only a small fraction of bits generated by the RNG are used in reduced precision sampling, we keep the bit-width of the RNG the

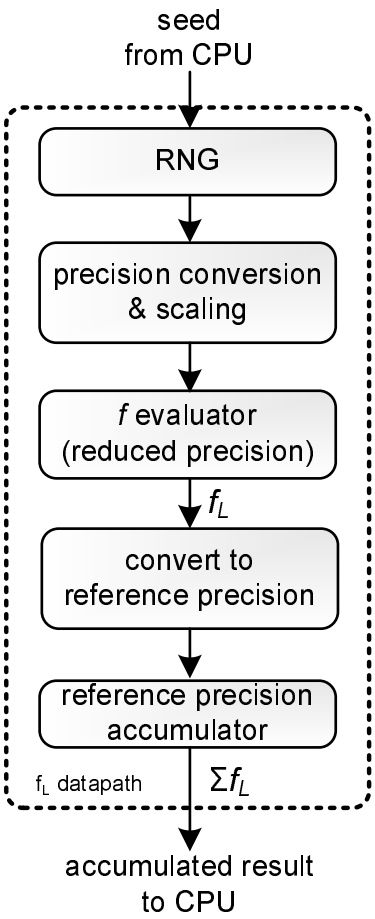

Figure 6.3: Reduced precision sampling data-path.

same as that for reference precision sampling. The scaled random number is then evaluated by the reduced precision sampling function evaluator. The accumulation is performed in reference precision to avoid lost of accuracy due to insufficient dynamic range in the accumulator. Finally, the accumulated result is sent back to the CPU. Multiple reduced precision sampling data-paths can be used with different seeds, and the averaging of the final results is done in the CPU.

Figure 6.4 shows the workload partitioning of the auxiliary sampling. It consists of 4 main stages: (1) random number generation, (2) evaluation of the sampling function f in reference and reduced precision, (3) computing the difference e between f_L and f_H in reference precision, and (4) accumulate the difference. Since the auxiliary sampling is the process to figure out the mean finite precision error ($\mu_{\epsilon_{fin}}$) between the reduced and the reference precision data-paths under the same set of random inputs, we decided to implement the random number generator using the FPGA and sent results back to the CPU. This method utilise highly efficient RNG generation on FPGAs since they are usually an

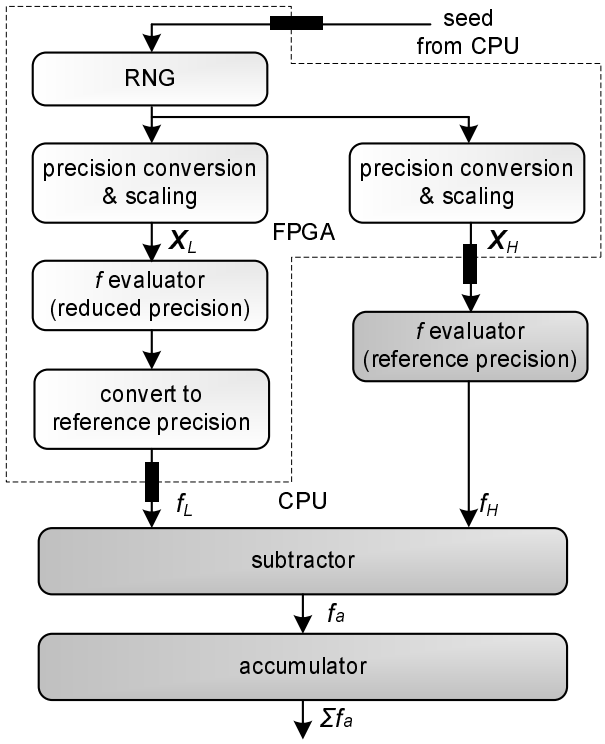

Figure 6.4: Workload partitioning of the auxiliary sampling. Operations in CPU are shaded.

order of magnitudes faster than CPU based RNGs [112]. The trade-off for this partitioning method is the increased bandwidth. For each sample point of the auxiliary sampling, we need to transfer s reference precision random numbers and one reference precision evaluation result from the FPGA to the CPU where s is the dimension of the sampling function.

6.5 Mixed precision optimisation

In this section, we develop analytical models for determining the required execution time of the proposed mixed precision method on a reconfigurable accelerator system. Figure 6.5 shows the system architecture for the reconfigurable system in our analytical model. The CPU is connected to an I/O hub (i.e. North Bridge) through a high bandwidth communication channel such as the Intel QPI or the AMD HyperTransport link. The FPGAs are connected to the I/O hub through another bus, usually PCI express. Thus communication between the CPU and the FPGA has to pass through the two kinds of communication link.

Table 6.1: Parameters in our analytical model.

Problem parameters	
σ_{tol}	output error tolerance, in terms of standard deviation of the output
s	dimension of the sampling function
σ_{f_L}	standard deviation of reduced precision sampling
σ_{f_a}	standard deviation of auxiliary sampling
L	the reduced precision data format, denoted as $sAeB$ where A is the number of significand (mantissa) bits and B is the number of exponent bits
Resource allocation parameters	
$p_L \in \mathbb{Z}$	number of reduced precision sampling data-paths
$p_{aux} \in \mathbb{R}$	**effective** number of auxiliary sampling data-paths
FPGA parameters (for each FPGA)	
A_{total}	total available area
A_{com}	cost of communication infrastructure
R_S	slack ratio
$freq$	clock frequency
c	number of clock cycles to compute a sample point
A_{red}	cost of a reduced precision sampling data-paths as shown in figure 6.3
A_{aux}	cost of auxiliary sampling data-path
CPU parameters	
T_{aux}	time required to compute a sample point
System parameters	
N_{core}	number of cores in each CPU
N_{cpu}	number of CPUs in the system
N_{fpga}	number of FPGAs in the system
BW_{cpu}	bandwidth between the CPU and I/O the hub (in terms of number of reference precision data / sec)
BW_{fpga}	bandwidth between each FPGA and I/O the hub
Output	
t	time required for the system to get output with specific error tolerance

Table 6.1 shows the parameters in our analytical model. It should be noted that all FPGA cost related parameters such as A_{total} and A_{red} should be applied to every kind of FPGA resource that is involved. For example, there will be 4 different A_{total} parameters for FPGA's look up table (LUT), registers, embedded DSP blocks and block memory respectively. Some other assumptions are made in the model. First, we assume a fixed amount of FPGA resources is used for the communication infrastructure between the FPGA and the I/O hub. Second, we assume the entire FPGA is running at a single clock frequency. Finally, we assume that a certain percentage of the FPGA's resource (the slack ratio) is left intentionally unused to avoid over-congestion in placement and routing.

Since the aggregated throughput of the auxiliary sampling on CPU does not always match the throughput of an auxiliary sampling data-path on the FPGA, we assume the effective number of auxiliary sampling data-paths can take fractional values. For example, $p_{aux} = 0.75$ means there is one aux-

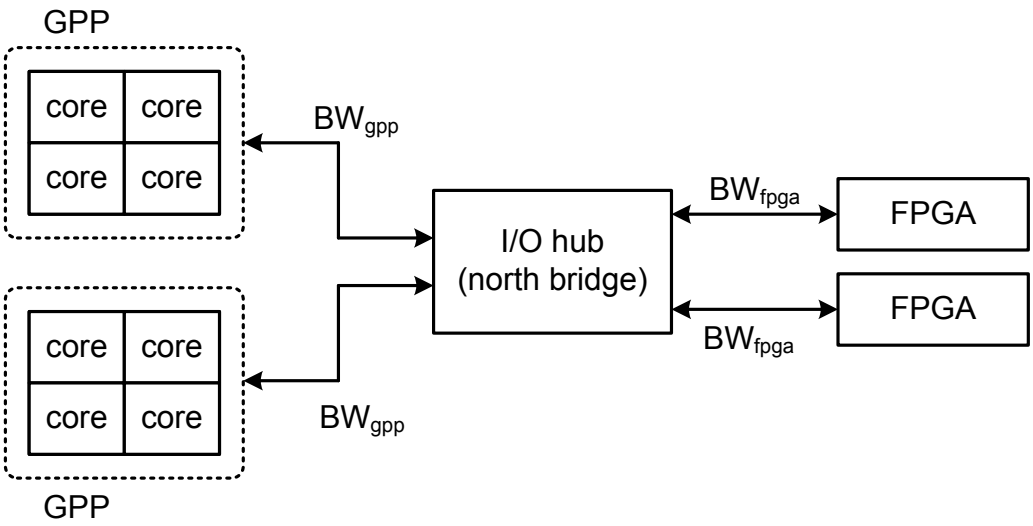

Figure 6.5: System architecture of the reconfigurable accelerator system in our analytical model.

iliary sampling data-path on the FPGA but only 75% of its outputs are computed by the CPU. The remaining 25% of the outputs are discarded.

Let TH_{red} and TH_{aux} be the aggregated throughput of reduced precision sampling and auxiliary sampling of the entire system. Using Equation 6.15, the required execution time for the system to produce an output with error equal to σ_{tol} can be found by Equation 6.16:

$$\begin{aligned} \sigma_{tol}^2 &= \frac{\sigma_{f_L}^2}{t \times TH_{red}} + \frac{\sigma_{f_a}^2}{t \times TH_{aux}} \implies \\ t &= \frac{\sigma_{f_L}^2}{\sigma_{tol}^2 \times TH_{red}} + \frac{\sigma_{f_a}^2}{\sigma_{tol}^2 \times TH_{aux}} \end{aligned} \tag{6.16}$$

The aggregated throughput of the reduced precision sampling and the auxiliary sampling of all FPGAs can be modelled as:

$$\begin{aligned} TH_{red} &= N_{fpga} \times p_L \times freq/c \\ TH_{aux} &= N_{fpga} \times p_{aux} \times freq/c \end{aligned} \tag{6.17}$$

The execution time for the mixed precision methodology is:

$$t(p_L, p_{aux}) = \frac{c}{\sigma_{tol}^2 \times N_{fpga} \times freq} \times \left(\frac{\sigma_{f_L}^2}{p_L} + \frac{\sigma_{f_a}^2}{p_{aux}} \right) \tag{6.18}$$

The following constraint should be applied to ensure the architecture described by the resource allo-

cation parameters can fit within the FPGA. We round p_{aux} to the next larger integer. The constraint (6.19) is transformed into two new constraints (6.20-6.21) using a new integer variable p_{aux_i} to avoid the ceiling function:

$$p_L \times A_{red} + \lceil p_{aux} \rceil \times A_{aux} \leq A_{total} \times (1 - R_S) - A_{com} \tag{6.19}$$

$$p_L \times A_{red} + p_{aux\_i} \times A_{aux} \leq A_{total} \times (1 - R_S) - A_{com} \tag{6.20}$$

$$p_{aux\_i}^{-1} \times p_{aux} \leq 1 \tag{6.21}$$

The number of auxiliary samplings that each CPU can perform is N_{core}/T_{aux} and the aggregated throughput of all CPUs is $N_{cpu} \times N_{cores}/T_{aux}$. Hence the effective number of auxiliary sampling data-paths on each FPGA is constrained by the following equation:

$$N_{fpga} \times p_{aux} \times freq/c \leq N_{cpu} \times N_{core}/T_{aux} \tag{6.22}$$

One evaluated value of f and s random numbers must be sent every cycle to the CPU to complete the subtraction and accumulation for each auxiliary sampling, hence the bandwidth constraints are:

$$p_{aux} \times freq/c \times (s + 1) \leq BW_{fpga} \tag{6.23}$$

$$N_{core}/T_{aux} \times (s + 1) \leq BW_{cpu} \tag{6.24}$$

The optimal resource allocation among the reduced precision sampling and the auxiliary sampling can be found by applying the following optimisation:

$$\min_{p_L \in \mathbb{Z}, p_{aux} \in \mathbb{R}, p_{aux\_i} \in \mathbb{Z}} t(p_L, p_{aux}, p_{aux\_i})$$

$$s.t. \text{ constraints (6.20)-(6.24) are satisfied}$$

Since the objective function $t(p_L, p_{aux})$ and all the constraints are posynomial, the optimisation can be solved using mixed integer geometric programming (MIGP) [113]. The globally optimal p_L, p_{aux}

values and the optimal precision can be found using enumeration from Algorithm 3, where L_{min} and L_{max} are the minimum and maximum choice of reduced precision in the system respectively.

Algorithm 3 Enumeration process for optimal reduced precision and optimal resource allocation.

1: $t_{global} \leftarrow$ huge_value
2: **for** $L = L_{min} \rightarrow L_{max}$ **do**
3: apply MIGP on $t(p_L, p_{aux})$ for the minimum execution time t_{min} in precision L
4: **if** $t_{min} < t_{global}$ **then**
5: $t_{global} = t_{min}$
6: $p_{aux}(global) = p_{aux}, p_L(global) = p_L$
7: **end if**
8: **end for**

6.6 Case studies

6.6.1 Asian option pricing

The first case study for our mixed precision methodology is an arithmetic Asian call option pricing problem. It has no closed-form solution and the payoff is calculated based on the arithmetic mean of the assets price at all observation points. The payoff equation of an arithmetic Asian call is shown in Section 2.1 as Equation 2.8. In Chapter 3, we have already presented the FPGA accelerated designs of Asian option pricer based on control variate Monte-Carlo method. In this chapter, we use an Asian option pricer based on pure Monte-Carlo as an application using our mixed precision methodology. The pricing algorithm of an Asian option based on pure Monte-Carlo is presented as Algorithm 2 in Chapter 3.

6.6.2 The GARCH volatility model

Our second case study is for pricing of a fixed strike lookback call option under the GARCH model. This option pays the owner $max(S_{ceil} - K)$ at maturity, where K is the strike price and S_{ceil} is the maximum day closing price of the underlying asset within the lifetime of the option.

In the original Black-Scholes model, the volatility of an asset is assumed to be constant. However, this assumption may not be realistic. A solution is to employ a stochastic volatility model such as the generalised autoregressive conditional heteroskedasticity (GARCH) model proposed by Bollerslev [114]. It is presented in Section 2.1.3 and the volatility σ_i in each time step i is simulated according to Equation 2.9. with an additional Gaussian random number generator and price an European option accordingly. The implementation of lookback option pricing is similar to the Asian option pricer, except that $drift$ and $vsqrtdt$ are updated every time step with the updated σ_i. An additional random number source is also required.

6.6.3 Numerical integration

Our last case study is multi-dimensional integral evaluation using the Monte-Carlo integration method. Multi-dimensional integrals arise in many areas such as engineering, biology, chemistry and physics modellings and they are not always solvable with analytical methods. Equation 6.25 shows multi-dimensional integration where a_i and b_i are the lower and upper bounds of the integration domain of the ith dimension. To evaluate the integral using Monte-Carlo simulation, random input vectors are generated within the integration domain and the average value of the integration function f is sampled. The approximated value for the integration value can be found by multiplying the average with the hypercube V of the integration domain as shown in Equation 6.26. The Monte Carlo integration method is preferable over quadrature based integration methods (Section 2.2.3) for high dimensional integrals, because MC integrations always converge with a rate of $\mathcal{O}(1/\sqrt{N})$ and the complexity of MC integration does not increase exponentially as quadrature based numerical integration methods as we described in Chapter 4. Typically, an integration with the number of dimensions larger than four will be evaluated use Monte-Carlo integration.

$$I = \int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 \cdots \int_{a_n}^{b_n} dx_n f(x_1, x_2, \cdots x_n) \tag{6.25}$$

$$I \sim V \times \langle f \rangle$$

$$V = \prod_{i=1}^{n} (b_i - a_i) \tag{6.26}$$

Table 6.2: Parameters of the current system and other hypothetical systems.

	current	I	II	III
N_{cpu}	1	8	1	1
N_{fpga}	1	1	4	8
$BW_{cpu}(GB/s)$	2	2	4×2	8×2
$BW_{fpga}(GB/s)$	2	8×2	2	2

We picked Genz's "Discontinuous" multi-dimensional integral in this case study (6.27). It is a common test integral being used in evaluation of different numerical integration methods. In our tests we use n = 8 as the dimension and an integration domain $[0, 1)^8$. Fully parallelised designs are used in our FPGA implementations and the data-paths can compute a single sample point per clock cycle, with constants c_i and w_i:

$$f_{dis} = \begin{cases} 0 & \text{if } x_0 > w_0 \text{ or } x_1 > w_1 \\ exp(\sum_{i=1}^{n}(c_i \times x_i)) & \text{otherwise} \end{cases} \tag{6.27}$$

6.7 Evaluation

6.7.1 Reconfigurable accelerator system

We use the MaxWorkstation reconfigurable accelerator system from Maxeler Technologies for our evaluation. It has a MAX3424A card with a Xilinx Virtex-6 SX475T FPGA. The card is connected to an Intel i7-870 CPU through a PCI express link with a measured bandwidth of 2 GB/s. The Intel CPU has 4 physical cores.

An important advantage of having an analytical model for our mixed precision methodology is that system designers can predict the performance of a hypothetical system based on parameters of the current system and the analytical model. Table 6.2 shows the parameters for our current system and three hypothetical systems. The hypothetical systems are constructed in such a way that the aggregated computational power of the FPGAs or the CPUs are 4 or 8 times higher than the current system, and the bandwidth is scaled proportionally.

The Intel Compiler (ICC) and the Intel Math Kernel Library are used in our software implementations.

We use the SFMT random number generator and the Box-Muller transformation in the Intel Vector Statistical Library (VSL) for the random number generation. Every effort has been made to ensure the software implementations are optimised, and the comparisons are fair and accurate. For the FPGA implementations, we use the MaxCompiler as our development system, which adopts a streaming programming model similar to [115] and supports customisable data formats so that floating point can be exploited with different precisions. All the FPGA results reported in this chapter are post place and route results.

The error tolerance σ_{tol} of the three financial case studies is set to 2.5e-3 such that 99.99% (4σ) of the time the error is less than a cent, given that the pricing is in dollars. For the numerical integration case study, the tolerance is set to 2.5e-4 since most scientific applications require high accuracy.

6.7.2 Applying optimisation

There are a few steps to apply Algorithm 3 in Section 6.5 in order to use the proposed mixed precision methodology optimally on a reconfigurable accelerator system. The first step is to find the system parameters such as N_{core}, N_{cpu}, N_{fpga}, BW_{cpu} and BW_{fpga}. These parameters can usually be found in the specification of the reconfigurable accelerator system.

The second step is to collect application specific FPGA and CPU parameters. In the MaxCompiler system, we describe the precision of the entire sampling data-path using a global variable and scripts are used to automatically generate data-paths with varying number of significand bits. Figure 6.6 shows the place and routed result of reduced precision data-paths of the Asian option problem. It is clearly shown that all the resource requirement increase with precision. Moreover, due to the function approximator in the exponential function, the block memory usage increases exponentially with the precision. The figure shows the A_{red} parameters of different precisions used in our model. Other FPGA parameters such as the A_{aux} parameters can be found using a similar method. The cost of communication infrastructure A_{com} is assumed to be constant. We also estimate the CPU parameter T_{aux} by writing a software benchmark program, which implements the data-flow in Figure 6.4 for certain iterations, and the average time required for an iteration is used as T_{aux}.

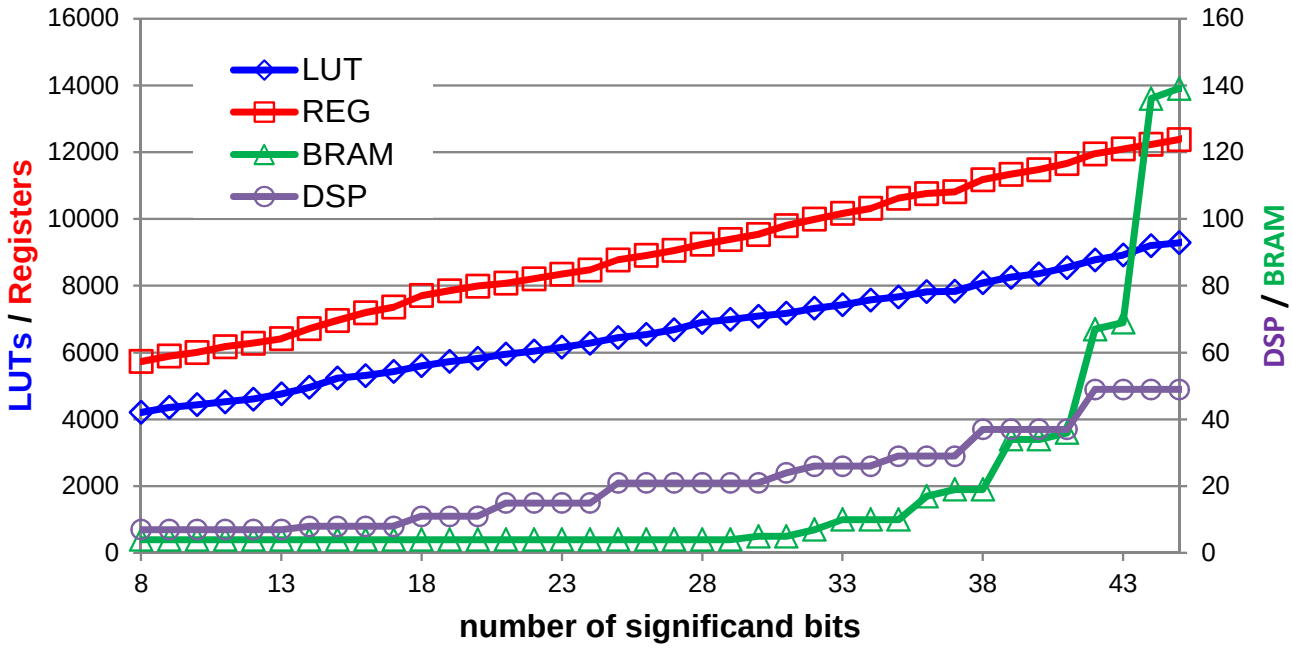

Figure 6.6: Cost of reduced precision sampling data-paths of the Asian option problem.

The next step of using the proposed method is the estimation of the standard deviations for reduced precision sampling, σ_{f_L}, and for auxiliary sampling, σ_{f_a}. An FPGA bit-stream with auxiliary sampling data-paths with different precisions is loaded and the results of the sampling function evaluations in different reduced precisions are sent back to the host PC. Using these results and the reference precision sampling function evaluations result from the CPU. The two standard deviations can be estimated using the two-pass algorithm on the host PC [116]. Figure 6.7 shows how the two standard deviations change with different precisions in the Asian option problem, using the parameters ($S_0 = K = 100, T = 1, v = 0.2, r = 0.05, steps = 360$).

It is interesting to note that the standard deviation of the auxiliary sampling σ_{f_a} decreases exponentially with increasing precision. However the standard deviation of the reduced precision sampling σ_{f_L} is low when the precision is low; it increases rapidly to reach a constant maximum value with further increases in precision. The same pattern is also observed in the standard deviations of other case studies. A possible explanation is that when the reduced precision is low, different values are compressed to the same numerical representation and hence the standard deviation is reduced. The standard deviation grows with reduced precision because there are more possible representations, and will finally converge to a value where the value is the same as the standard deviation of the reference

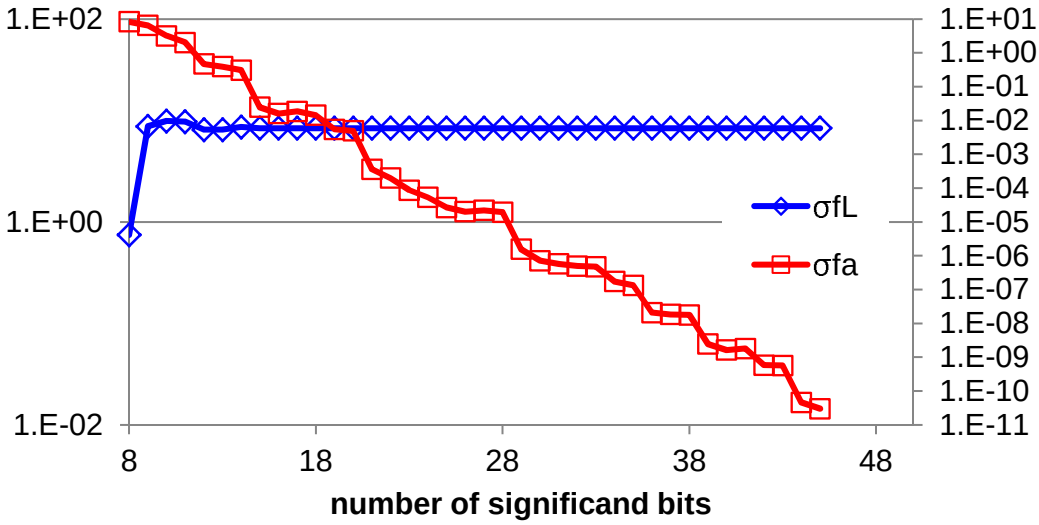

Figure 6.7: The standard deviations of the reduced precision sampling and the auxiliary sampling verse different precisions.

precision sampling (i.e. σ_{f_H}). The observed exponential reduction of σ_{f_a} could be explained by the fact that finite precision error decreases exponentially with the number of significand bits in floating point formats.

Using parameters collected in the previous steps, we can apply geometric programming to find the optimal precision and resource allocation. A major assumption of this flow is that the two standard deviations do not change with input parameters (e.g. strike price of an option). If this assumption is not valid, we can profile the common σ_{f_L} and σ_{f_a} combinations and generate an optimal bit-stream for each of these combinations. When the input parameters change, we profile the two standard deviations again, run the geometric programming solver and load the bit-stream closest to the optimal configuration.

It is important to note that the choice of error tolerance σ_{tol} affects the execution time of our mixed precision methodology as shown in Equation 6.16. However, the optimal reduced precision and resource allocation do not change with the error tolerance. Hence when a new error tolerance is applied, we just update the required execution time according to Equation 6.16 – there is no need for applying geometric programming again.

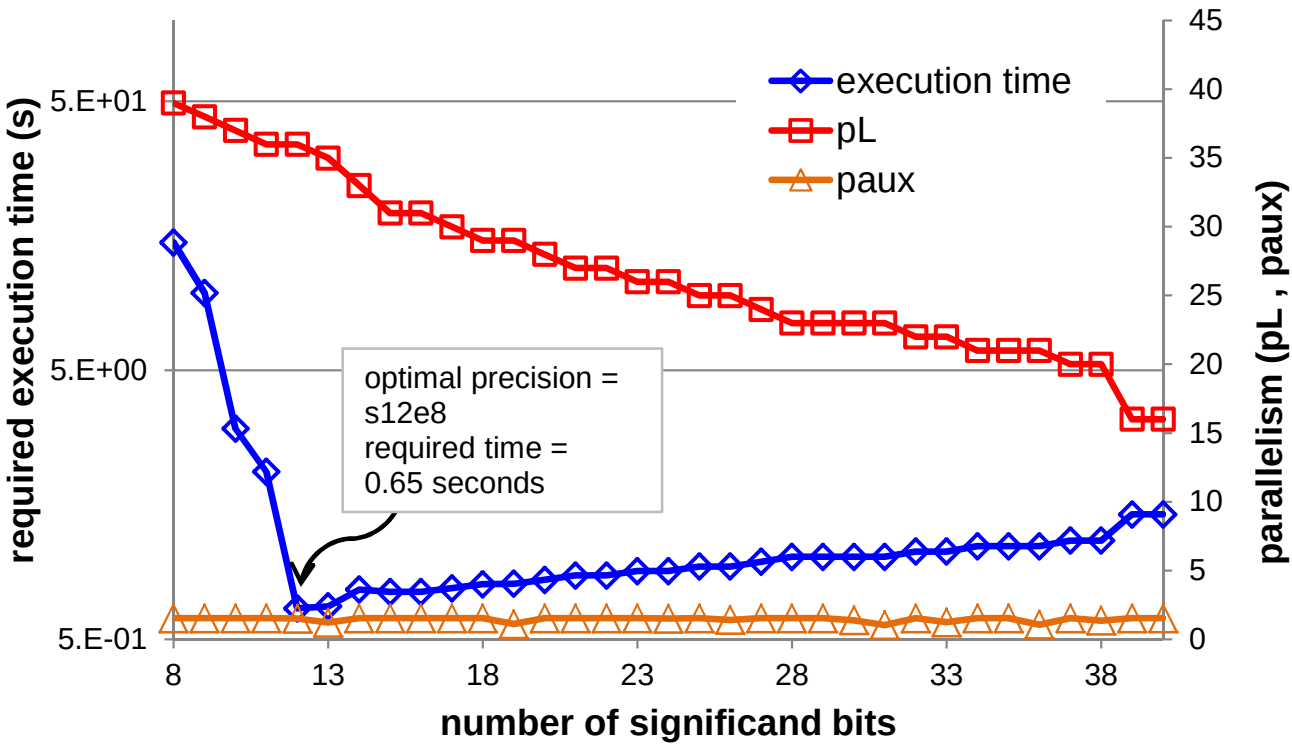

Figure 6.8: Results of Asian option pricing versus different number of significand bits.

Table 6.3: Execution time, optimal reduced precision and the p_L/p_{aux} ratio of the same Asian option pricing under different system parameters.

	current	I	II	III
execution time (s)	0.65	0.65	0.19	0.09
optimal precision	s12e8	s12e8	s15e8	s16e8
p_L/p_{aux}	23.9	13	73.9	147.9

6.7.3 Performance: parallelism versus precision

Figure 6.8 shows the execution time and the degree of parallelism of the Asian option pricing problem for different reduced precision in the current system as evaluated by our analytical model. The optimal reduced precision in this benchmark is s12e8. The performance curve and the optimal point can be explained by considering Figures 6.6 and 6.7. If the reduced precision is lower than the optimal one, the auxiliary sampling error σ_{f_a} is high, and more computations must be included. This will take a longer time even though parallelism is increased due to smaller data-paths. If the reduced precision is higher than the optimal one, the decrease of the auxiliary sampling error is marginal and cannot offset the disadvantage of reduced parallelism.

We also investigate the relationship between the optimal reduced precision and system parameters. Table 6.3 shows that when the aggregated FPGA computational power is increased (II and III), the optimal reduced precision will increase because the system can perform more reduced precision sampling (i.e. higher p_L/p_{aux} ratio), and can thus afford a higher σ_{f_L}. Investing for higher aggregated CPU computational power (I) seems to have marginal effect in this benchmark as the sampling error in reduced precision already dominates the sampling error in the auxiliary sampling.

6.7.4 Comparison: CPU/FPGA double precision

Table 6.4 shows comparisons of the 3 MC case studies running on a CPU only system with double precision arithmetic, an FPGA only system with double precision arithmetic, and a reconfigurable accelerator system with our proposed mixed precision methodology. All designs are run for a specific time such that the 3 systems have the same accuracy. As shown in the table, the mixed precision methodology requires 5-11 % additional sample function evaluations but only 1-4 % of total evaluations are computed in reference precision. This clearly shows the trade-offs between number of computations and the contribution of each computation in increasing the accuracy of the final result. Using the mixed precision methodology, we achieve 2.9 to 7.1 times speedup over the double precision FPGA designs and 44 to 106 times speed up over the **quad-core** CPU designs.

We also compare the energy efficiency of the three settings. The average power consumption is measured using a remote power measuring socket from Oslon® electronics with an measuring interval of 1 second. As shown in Table 6.4, although the mixed precision designs using both the FPGA and the CPUs have the highest power consumption compared with the CPU only or the FPGA only settings, they consume the least total energy to achieve the required accuracy because the execution times are significantly reduced thanks to our technique for workload partitioning. Our mixed precision methodology achieves 1.4 to 3.1 times energy saving compared with the FPGA only designs with double precision, and 41 to 104 times energy saving compared with CPU only designs, while meeting the same output accuracy requirement.

Table 6.4: Comparison of MC simulations using CPU only system (SW), double precision FPGA only system (FP) and mixed precision methodology using both CPU and FPGA (Mixed).

	Asian option			GARCH			Numerical integration		
	SW	FP	Mixed	SW	FP	Mixed	SW	FP	Mixed
clock freq. (GHz)	2.93	0.175	0.175[1]	2.93	0.175	0.175[1]	2.93	0.175	0.16[1]
num. of cores[2]	4	5	36/1.5	4	5	24/0.9	4	5	16/0.18
num. of f_L evaluations (M)	0	0	12	0	0	321	0	0	2320
num. of f_H evaluations (M)	11.3	11.3	0.47	317	317	11.6	2230	2230	26.8
num. of total evaluations (M)	11.3	11.3	12.5	317	317	333	2230	2230	2347
additional evaluation (%)	-	-	10.6	-	-	4.8	-	-	5.2
evaluations in reference precision (%)	100	100	3.8	100	100	3.5	100	100	1.1
execution time (sec.)	29	4.7	0.66	1560	131	26.6	95.8	2.6	0.9
normalised speedup	1x	6.2x	**44x**	1x	12x	**59x**	1x	37x	**106x**
mixed precision gain	-	1x	**7.1x**	-	1x	**4.9x**	-	1x	**2.9x**
power consumption (W) [3]	183	85	192	179	90	181	184	90	189
energy consumption (kJ)[4]	5.3	0.4	0.13	280	11.8	4.8	17.6	0.23	0.17
normalised energy	**41x**	3.1x	1x	**58x**	2.5x	1x	**104x**	1.4x	1x

[1] Only the FPGA clock frequencies are reported and the 4 CPU cores are all running at 2.93 GHz.
[2] For the mixed precision design, all the 4 CPU cores are used and the number of reduced precision sampling and auxiliary sampling data-paths (pL/p_{aux}) are shown.
[3] The idle power consumption of the system is $80W$.
[4] Energy consumption = power consumption $\times$ execution time.
[5] The optimal precision of the 4 mixed precision designs is **s12e8**.

6.7.5 Comparison: GPU

We also compare our mixed precision methodology on reconfigurable accelerator system with a graphical processor unit (GPU). Table 6.5 compares the execution time and power consumption of our mixed precision methodology with a NVIDIA Tesla C2070 GPU in the Asian option pricing problem. The GPU has 448 cores running at 1.15 GHz and has a peak double precision performance of 515 GFlops. Using our mixed precision methodology, an Virtex-6 ST475X FPGA and an i7-870 CPU are able to out-perform the GPU by 4.6 times. We also achieved 5.5 times energy saving compared with the GPU.

6.8 Summary

This chapter proposes a novel mixed precision methodology for Monte-Carlo simulation in reconfigurable accelerator systems. The methodology covers any Monte-Carlo applications and exploits the

Table 6.5: Comparison with CPU and GPU.

	CPU only	GPU	FPGA only	FPGA + CPU
precision	double	double	double	mixed
execution time (s)	29	3	4.7	0.65
power (W)	183	236	85	192
energy (kJ)	5.3	0.71	0.4	0.13
normalised speedup	1x	9.7x	6.2x	44.6x
normalised energy	40.7x	**5.5x**	3.1x	1x

synergy between FPGA and CPU to produce results as accurate as users would require. An analytical model and an optimisation method is developed for locating the optimal precision and optimal resource allocation. Experimental results on three realistic case studies show that auxiliary sampling would only require 5 % to 11 % additional evaluations and less than 4 % of total evaluations are computed in the reference precision (Table 6.4). We demonstrate that reconfigurable accelerator system using our methodology can be up to 4.6 times faster than state-of-the-art GPU, 7.1 times faster than a baseline FPGA design using double precision, and 106 times faster than optimised software running on a quad-core CPU. It can also be up to 5.5 times more energy efficient than a GPU and 104 times more energy efficient than a quad-core software design.

There is no comparable performance result from previous work for this novel mixed precision methodology. The closest previous work is in [16], where an FPGA accelerated Monte-Carlo simulation on BGM interest rate model with optimised precision is presented and shows a speedup of 25 times over a comparable CPU. Although the reference FPGA and CPU used in [16] are different from the reference FPGA and CPU used in this chapter, both works used a comparable set of FPGA and CPU at the time of research and both works used precision optimisation techniques for a Monte-Carlo problem. The speedup figure achieved in this chapter using our mixed precision methodology (106 times faster) is far more than the speedup figure achieved in [16] (25 times faster).

Chapter 7

Optimising Performance of Quadrature Methods with Reduced Precision

7.1 Motivation

Using quadrature methods to price a single simple option is fast and can typically be performed in milliseconds on desktop computers. However, quadrature methods can become a computational bottleneck, for example, when a huge number of complex options are being revalued overnight under many different scenarios for risk management. Moreover, energy consumption of computation is a major concern when the computation is performed 24 hours a day, 7 days a week.

In Chapter 4, we presented the design and optimisation techniques to use FPGA as an accelerator for option pricing with quadrature method. In this chapter, we aim at increasing the performance and energy efficiency further by exploiting the precision flexibility of reconfigurable hardware.

The ability to support customisable precision is an important advantage of reconfigurable hardware as it could be exploited to provide additional speedup. The use of reduced precision affects the accuracy of the numerical results. However, with a higher throughput capacity using reduced precision, the integration grid spacing could be reduced which might actually increase accuracy. This chapter introduces a novel optimisation methodology for determining the optimal combination of operator

precision and integration grid spacing in order to maximize the performance of quadrature method on reconfigurable hardware. The major contributions of this chapter include:

- optimisation modeling based on a step-by-step accuracy analysis and performance model. A discrete moving barrier option pricer is used as an example to graphically illustrate the analysis and to provide empirical evidence for the model (Section 7.2);
- a methodology and algorithms to determine the optimal mantissa bit-width and the integration grid density for a given integration problem by finding the Pareto frontier satisfying the error tolerance level (Section 7.3);
- case studies of two financial applications and one benchmark quadrature problem using the proposed methodology, namely a discrete moving barrier option pricer, a 3-dimensional European option pricer, and a discontinuous integration benchmark (Section 7.4);
- performance comparison of the optimised FPGA implementation versus GPU and CPU. Our results show that the bit-width optimisation increases performance by around 4x, resulting in a total speed-up over double-precision software of 15.1x while maintaining the same error level. The optimised FPGA designs over a comparable GPU design is 1.2 times faster and 42.2 times more energy efficient. (Section 7.5).

7.2 Optimisation Modeling

The proposed optimisation objective function is based on a step-by-step accuracy analysis and performance modeling. A barrier option pricer is used as an example to illustrate the relationship between accuracy, throughput, integration grid density and the precision of floating-point operations.

7.2.1 Accuracy Analysis

There are two sources of error affecting the "accuracy" of the integration result, namely integration error ϵ_{int} and finite precision error ϵ_{fin}. The total error ϵ_{total} is a function of both error sources.

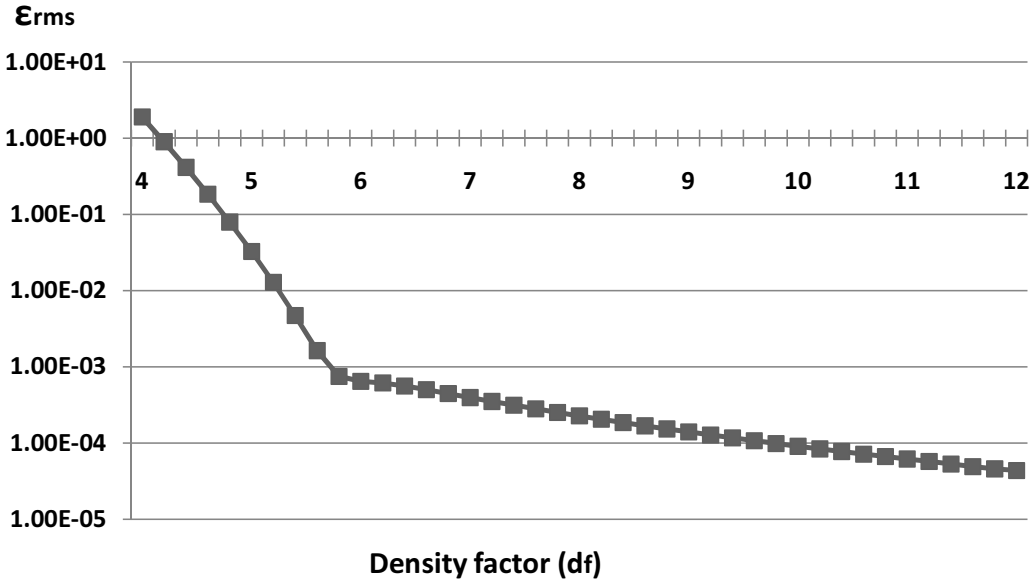

Figure 7.1: The ϵ_{rms} for different d_f at m_w=53.

Integration error ϵ_{int} is the error due to having a finite number of integration points within an integration interval. Finite precision error ϵ_{fin} is the error due to non-exact floating-point arithmetic. Floating-point number representation in computer has a finite significant bit-width. The rounding of the intermediate or final result leads to precision loss. We define grid density factor $\boldsymbol{d_f}$ as a variable which is inversely proportional to the integration grid spacing and we define $\boldsymbol{m_w}$ as the width of the mantissa (significant bits). Therefore, we have $\epsilon_{int}(d_f)$ and $\epsilon_{fin}(m_w)$ respectively to represent their relationships.

To measure $\epsilon_{total}(m_w, d_f)$, the root-mean-squared error $\boldsymbol{\epsilon_{rms}(m_w, d_f)}$ comparing with a set of reference results is used as the proxy. The set of reference results are computed using a large value of m_w and d_f.

We investigate the result of $\epsilon_{rms}(m_w, d_f)$ by computing a portfolio of 30 barrier options using different mantissa bit-width and density factor. The computed option values are compared with a set of reference values using $m_w = 53$ (double precision) and $d_f = 20$. Fig. 7.1 shows the graph of $\epsilon_{rms}(53, d_f)$. We can see that the total error is decreasing with respect to d_f. One interesting observation is that the error decrease rapidly at the beginning and then decrease slowly after around $d_f = 5.8$. One reason behind is that the function has discontinuous points, therefore a misalignment of integration grid points lead to a large error. When the density increase, the effect of misalignment

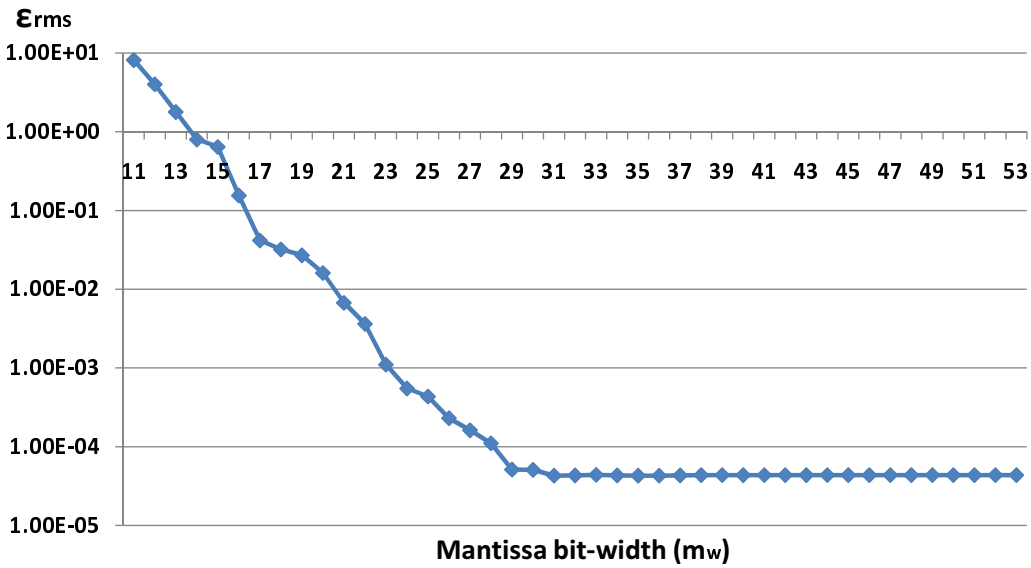

Figure 7.2: The ϵ_{rms} for different m_w at d_f=12.

diminishes.

Fig. 7.2 shows the graph of $\epsilon_{rms}(m_w, 12)$. This figure shows that with a sufficient large density factor, the total error of the result decreases with increasing mantissa bit-width. In addition, this figure also indicates that at $d_f = 12$, increasing mantissa bit-width for more than 33 would not increase the accuracy significantly. It is because the ϵ_{total} is dominated by ϵ_{int} but not ϵ_{fin} after m_w reached 33. Therefore, using more than 33 bits of mantissa is consuming unnecessary resources.

Fig. 7.3 shows the contour plot of $\epsilon_{rms}(m_w, d_f)$ at different error levels for the barrier option pricer and provides an overview of the total error using different m_w and d_f combinations.

7.2.2 Performance Modeling

The performance of the system is defined with the following equation:

$$\phi_{int}(m_w, d_f) = \frac{\phi_{pt}(m_w)}{N_{pt}(d_f)} \tag{7.1}$$

ϕ_{int} is the throughput in aggregated integrations per second per FPGA, ϕ_{pt} is the throughput in aggregated number of integration points per second per FPGA and N_{pt} is the number of integration points per integration. Furthermore, we define p_L as the degree of parallelism (number of replicated cores)

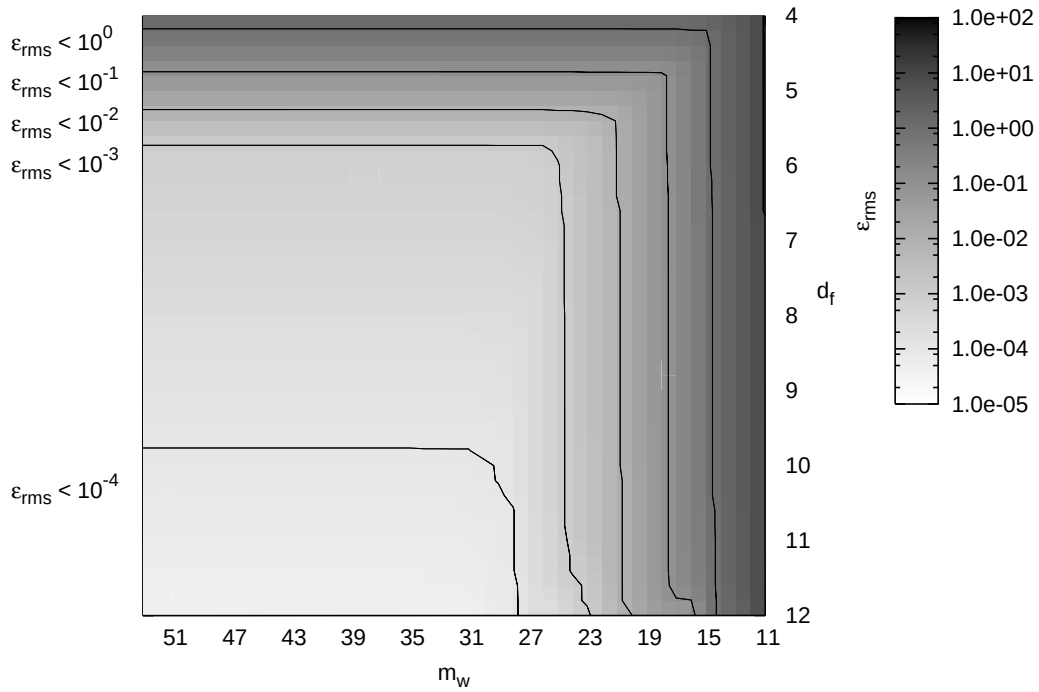

Figure 7.3: The contour plot of ϵ_{rms} of barrier option pricer for different m_w and d_f.

and $freq$ as the clock frequency of the FPGA. With multiple replicated and fully pipelined integration cores running in parallel, ϕ_{pt} is defined as:

$$\phi_{pt}(m_w) = p_L(m_w) \cdot freq \tag{7.2}$$

because each core can process one integration point per clock cycle. ϕ_{pt} and p_L is monotonically decreasing with m_w. A higher m_w leads to a larger core, so fewer cores will fit in the FPGA, reducing degree of parallelism p_L and lower aggregated integration points throughput ϕ_{pt}. ϕ_{pt} is also monotonically decreasing with d_f, as a higher d_f leads to more integration points per integration N_{pt} and less integrations could be computed per second. Therefore, we have the following inequalities:

$$\phi_{int}(m_{w_x}, d_f) \geq \phi_{int}(m_{w_y}, d_f), \forall m_{w_x} < m_{w_y} \tag{7.3}$$

$$\phi_{int}(m_w, d_{f_x}) \geq \phi_{int}(m_w, d_{f_y}), \forall d_{f_x} < d_{f_y} \tag{7.4}$$

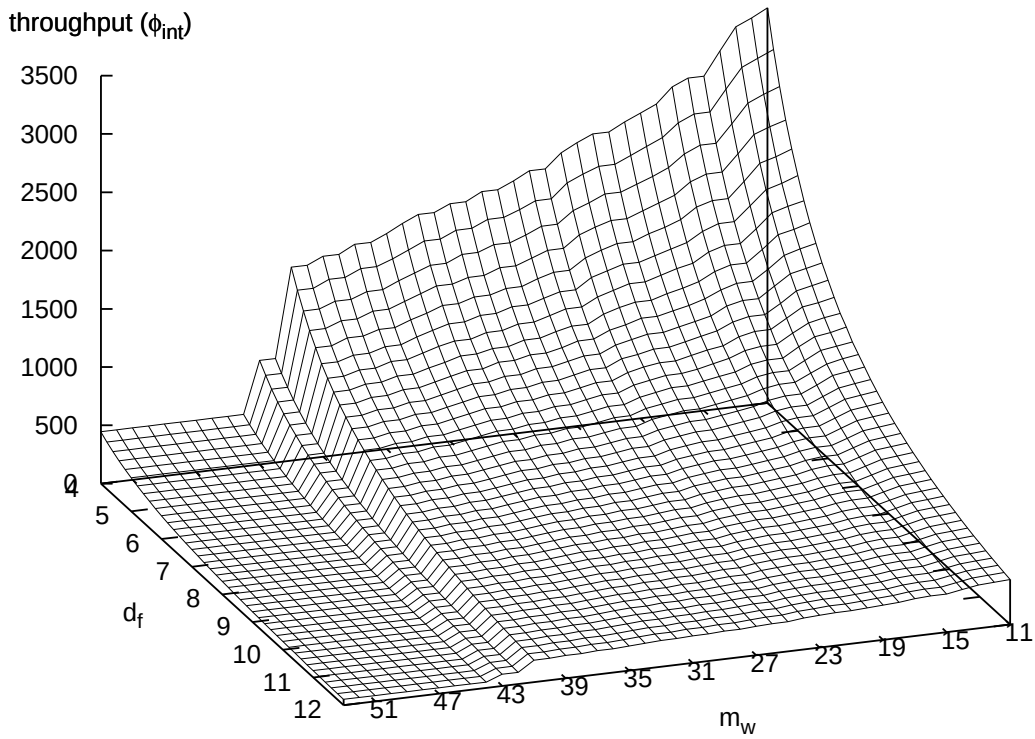

Figure 7.4: The aggregated FPGA throughput.

Fig. 7.4 shows the 3D graph of aggregated FPGA throughput $\phi_{int}(m_w, d_f)$ of the barrier option pricer which is consistent with the above inequalities.

7.2.3 Optimisation Objective Equation

Our objective is to determine the set of (m_w, d_f) which produces the design with optimal performance while maintaining the same level of accuracy. We define $\boldsymbol{\epsilon_{tol}}$ as error tolerance level. With the results from Equation 7.1 and 7.2, the following 2-dimensional optimisation problem can be formulated:

$$\max_{m_w, d_f} \left(\frac{p_L(m_w) \cdot freq}{N_{pt}(d_f)} \right), m_w \in \mathbb{Z}^+, d_f \in \mathbb{R}^+, \epsilon_{rms}(m_w, d_f) < \epsilon_{tol} \tag{7.5}$$

For example, Fig. 7.5 and Fig. 7.6 show the 3D plots of the optimisation result of barrier option pricer at $\epsilon_{tol} = 10^{-4}$ and $\epsilon_{tol} = 10^{-3}$ respectively by using the result of Fig. 7.3 and Fig. 7.4. We can see from the figures that the optimal aggregated throughputs are 350 and 1078 integrations per second. The corresponding (m_w, d_f) sets are (31,9.8) and (26,5.8).

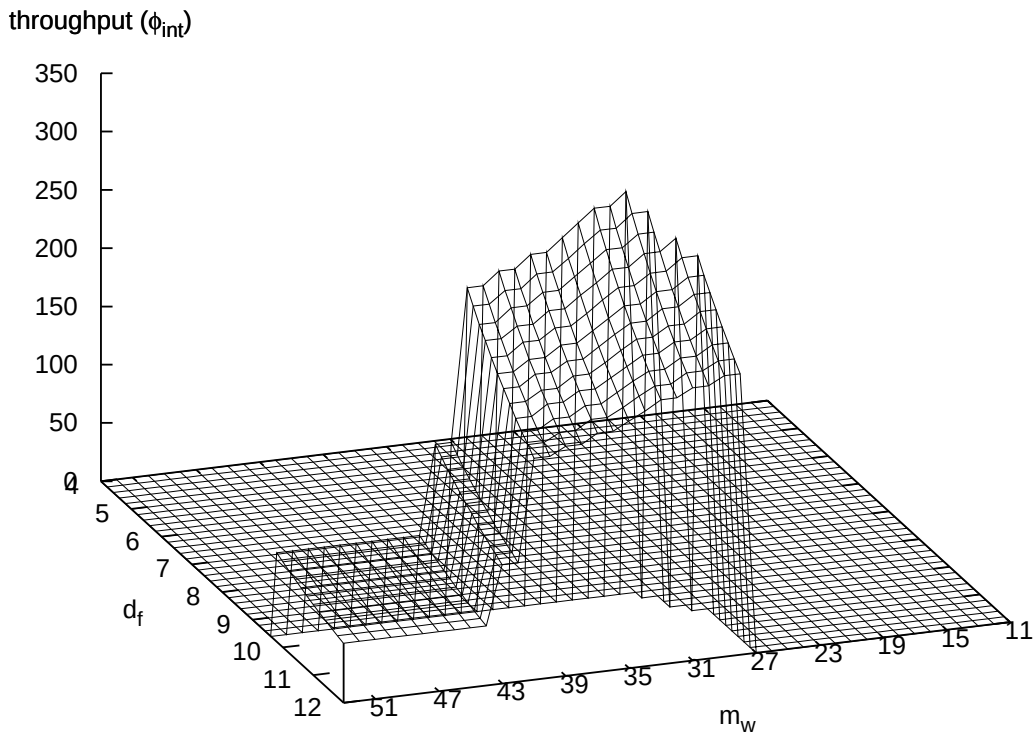

Figure 7.5: The aggregated FPGA throughput satisfying $\epsilon_{rms}(m_w, d_f) < 10^{-4}$.

7.3 Optimisation Algorithm and Methodology

This section provides the algorithms and a systematic way to apply the precision optimisation technique for a quadrature problem. The optimisation algorithm uses the property that the throughput of the integration decreases monotonically with respect to both m_w and d_f as shown in inequalities (7.3) and (7.4). The optimal throughput will only occur at the Pareto frontier points (Pareto set $\mathbf{S}$) of (m_w, d_f) satisfying $\epsilon_{rms} < \epsilon_{tol}$ and, therefore, it is not necessary to obtain the ϵ_{rms} values for all (m_w, d_f) combinations. Fig. 7.7 shows the Pareto frontier of a barrier option pricer and the corresponding throughput as an illustration. The point $(m_w, d_f) = (26, 5.8)$ is the optimal point as it produced the maximum aggregated throughput with a high enough degree of parallelism and a low enough number of grid points while satisfying the minimum accuracy requirement.

The detailed steps of our proposed one-pass optimisation process are:

1. Prepare a set of sample inputs.

2. Evaluate the results of the sample inputs as reference values.

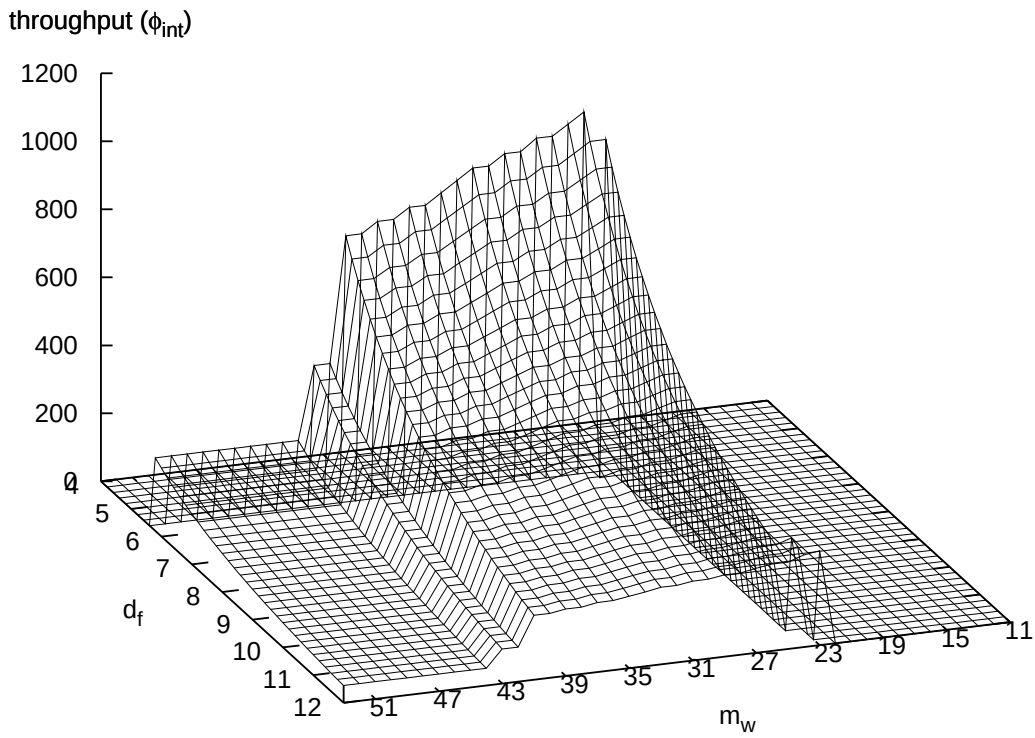

Figure 7.6: The aggregated FPGA throughput satisfying $\epsilon_{rms}(m_w, d_f) < 10^{-3}$.

3. Apply Algorithm 4 to obtain a Pareto set **S**.

4. Apply Algorithm 5 on **S** to obtain the optimal ϕ_{int}.

In step 2, we will typically use double precision (m_w=53) and a sufficiently large d_f to obtain the reference values such that the reference values are known to be accurate. In step 4, the algorithm requires the values of function $N_{pt}(d_f)$ and $pL(m_w)$ in order to compute ϕ_{int}. The function $N_{pt}(d_f)$ could easily be determined with the knowledge of the integration problem. The parameters pL can be either obtained directly after the full FPGA implementation, or estimated using the resource usage of a single core FPGA design. Fig. 7.8 shows our estimation of pL and the resource usage of a single-core barrier option pricer.

The whole optimisation process is completely automated in our case studies by designing the hardware implementation as a parametric template, with m_w and pL as parameters. The hardware implementations for different values of m_w are generated, placed and routed automatically from the template for the use of step 3 and 4.

Algorithm 4 Algorithm for obtaining Pareto set **S**

1: $S \leftarrow \emptyset$
2: **for** $m_w \in m_w^{min}..m_w^{max}$ do **do**
3: perform binary search for $\min d_f$ s.t $\epsilon_{rms}(m_w, d_f) < \epsilon_{tol}$
4: **if** found **then**
5: add tuple (m_w, d_f) to S
6: **end if**
7: **end for**

Algorithm 5 Algorithm for determining the optimal precision and density factor

1: $\phi^{max} \leftarrow 0$
2: **for** $(m_w, d_f) \in$ **S** do **do**
3: **if** $\phi_{int}(m_w, d_f) > \phi^{max}$ **then**
4: $\phi^{max} \leftarrow \phi_{int}(m_w, d_f)$
5: $S_{optimal} \leftarrow (m_w, d_f)$
6: **end if**
7: **end for**

7.4 Case Studies

The hardware architectures of two financial applications and one benchmark integration problem are designed. Simpson's rule is used in all three case studies. The optimal combination of (m_w, d_f) is determined using the optimisation methodology and algorithms as described in the previous two sections with $\epsilon_{tol} = 10^{-3}$. As the range of the floating-point numbers is known to be small, the exponent size of the floating-point operators is set to 8. The accumulation is performed in double precision ($m_w = 53$) to minimise the loss of accuracy due to insufficient dynamic range in the accumulator.

7.4.1 Discrete Moving Barrier Option pricer

The first case study is the pricing of discrete barrier options, which is a real-world pricing problem for which there is no closed-form solution. The pricing equation of an barrier option using quadrature methods is derived from the Black and Scholes partial differential equation [117]. For an option with an underlying asset following geometric Brownian motion and having continuous dividend yield:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r - D_c) S \frac{\partial V}{\partial S} - rV = 0 \tag{7.6}$$

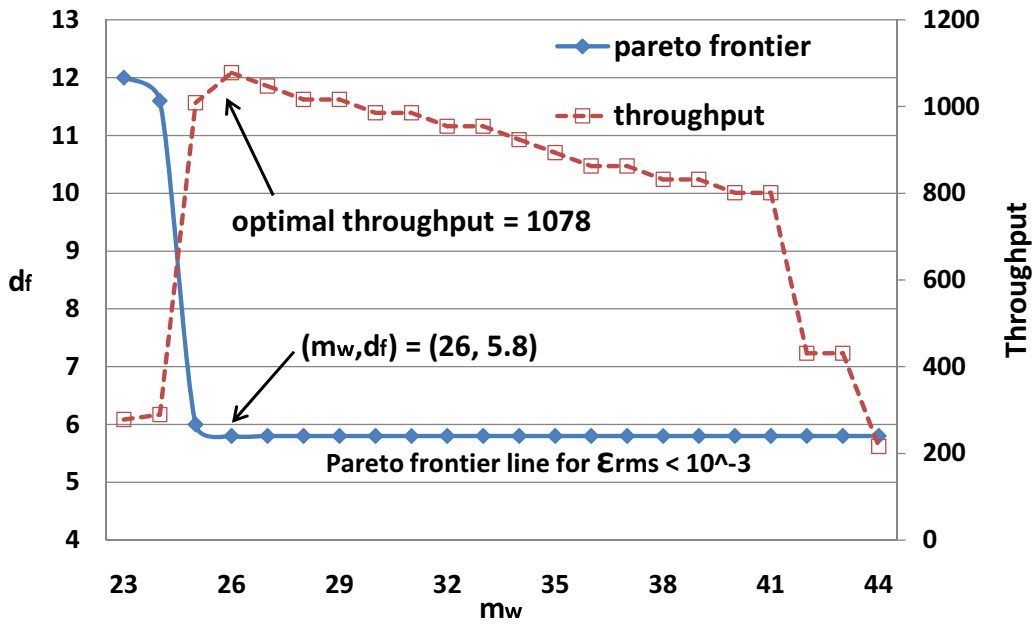

Figure 7.7: The Pareto frontier line of barrier option pricer when $\epsilon_{tol} = 10^{-3}$.

where $V(S,t)$ is the price of the option, S is the value of the underlying asset, t is time, r is risk-free interest rate, σ is volatility of the underlying asset, K is exercise price, and D_c is continuous dividend yield.

The following standard transformations

$$x = \log(S_t/K), \qquad y = \log(S_{t+\Delta t}/K)$$

give us the solution of $V(x,t)$ as:

$$V(x,t) = A(x) \int_{-\infty}^{+\infty} E(x,y) V(y, t+\Delta t) dy \tag{7.7}$$

where

$$A(x) = \frac{1}{\sqrt{2\sigma^2 \pi \Delta t}} e^{(-kx/2)-(\sigma^2 k^2 \Delta t/8)-r\Delta t} \tag{7.8}$$

$$E(x,y) = e^{(yC_2-(x-y)^2 C_1)}, \quad C_1 = \frac{1}{2\sigma^2 \Delta t}, \quad C_2 = \frac{2(r-D_c)-\sigma^2}{2\sigma^2} \tag{7.9}$$

Since C_1 and C_2 will not change during the whole pricing process, they can be precomputed in software. Eq. (7.7) is the basic building block for quadrature option pricing. To price a down-and-out

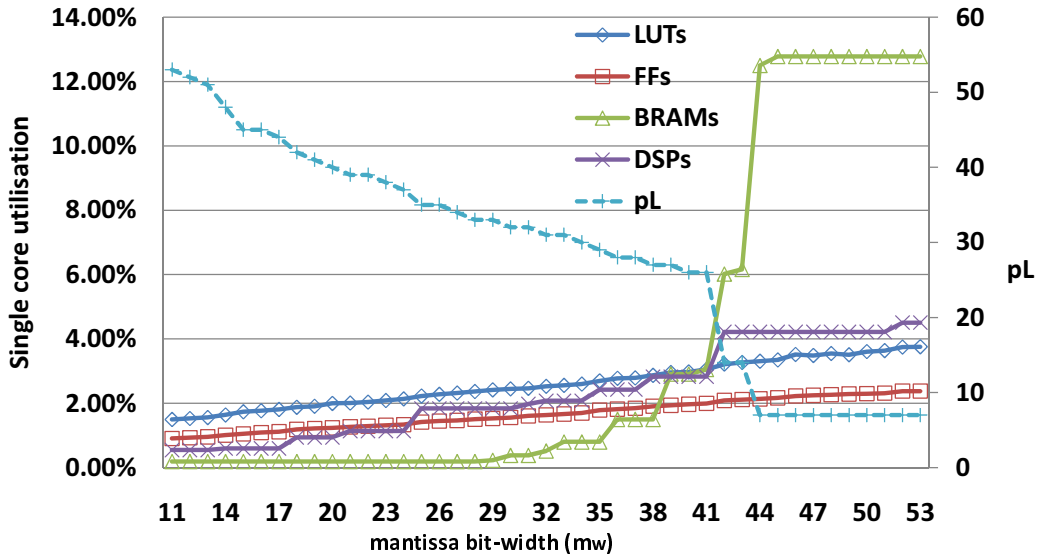

Figure 7.8: pL estimation and the single core resource utilisation of barrier option pricer.

discrete moving barrier option with m time steps and B_m as the barrier price at time step m, we define the transformed position of b_m as:

$$b_m = log(B_m/K), \tag{7.10}$$

then the option price V_m at time step m can be computed using the equation:

$$V_m(x, t_m) \approx A(x) \int_{y_{min_m}}^{y_{max_m}} E(x, y) V_{m+1}(y, t_{m+1}) dy, \tag{7.11}$$

where

$$y_{max_m} = x + 10\sigma\sqrt{t_{m+1} - t_m} \tag{7.12}$$

$$y_{min_m} = \max(b_m, x - 10\sigma\sqrt{t_{m+1} - t_m}) \tag{7.13}$$

The barrier option value is calculated iteratively backward from the expiry date to present date as shown in Fig. 7.9. The main data-path for the hardware barrier core is shown in Fig. 7.10.

As the change of price exhibits a Brownian motion, the value of y fluctuates proportional to $\sqrt{\Delta t}$. Therefore, the size of δy should also be defined proportional to $\sqrt{\Delta t}$. We define the grid density factor d_f as $\frac{\sqrt{\Delta t}}{\delta y}$. Using the methods from Section 7.3, the optimal (m_w, d_f) is found to be (26,5.8).

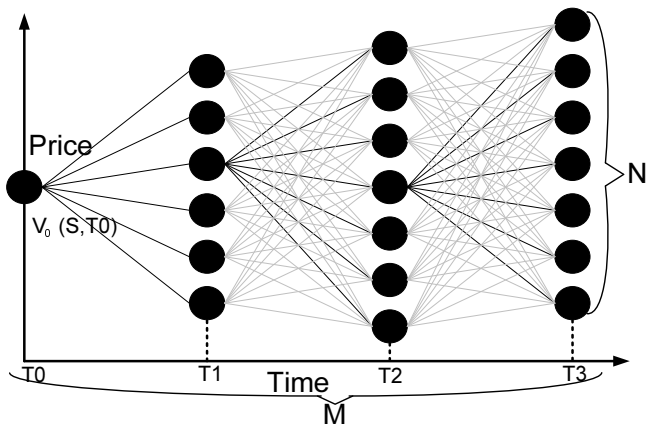

Figure 7.9: The backward barrier option iteration process.

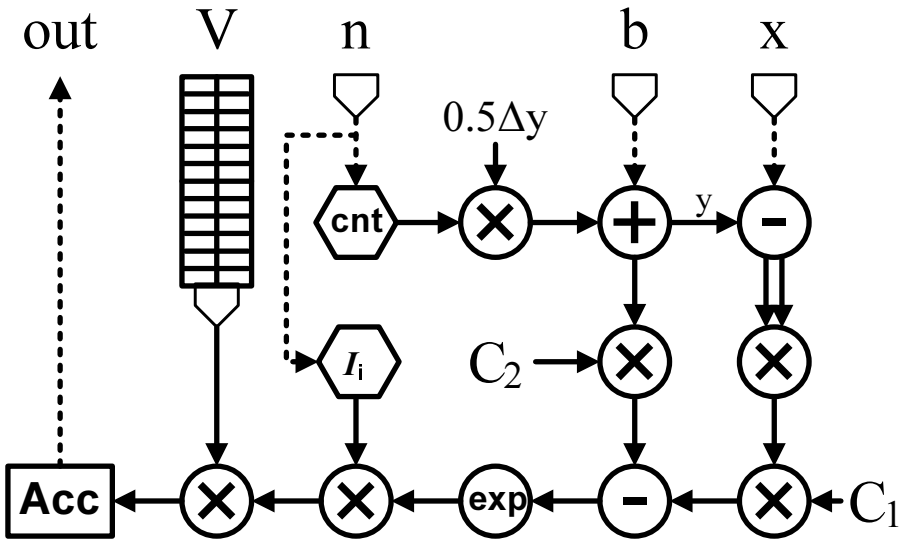

Figure 7.10: The hardware barrier option pricing core.

7.4.2 Multi-dimensional European Option pricer

The second application is a multi-dimensional European option pricer. The option pricing equation with multiple underlying assets using quadrature methods is based on the multi-asset version of Black and Scholes partial differential equation [39]:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}\sigma_i\sigma_j\rho_{ij}S_iS_j\frac{\partial^2 V}{\partial S_i \partial S_j} + \sum_{i=1}^{d}(r - D_i)S_i\frac{\partial V}{\partial S_i} - rV = 0 \tag{7.14}$$

where r is the risk-free interest rate, d is the number of underlying assets, S_i are the underlying asset values, σ_i and D_i are the corresponding volatilities and dividend yields, and ρ_{ij} is the correlation coefficient between underlying asset values S_i and S_j. Note that $|\rho_{ij}| \leq 1$, $\rho_{ii} = 1$ and $\rho_{ij} = \rho_{ji}$. We

make the logarithmic transformations

$$x_i = \log(S_i), \quad y_i = \log(S_i)$$

to be the chosen nodes at t and $t + \Delta t$. Let R be the matrix such that element $R(i,j) = \rho_{ij}$.The solution is:

$$V(x_1, \ldots, x_d, t) = C \int_{-\infty}^{+\infty} \ldots \int_{-\infty}^{+\infty} V(y_1, \ldots, y_d, t + \Delta t)$$
$$E(x_1, \ldots, x_d, y_i, \ldots, y_d) dy_1 \ldots dy_d \tag{7.15}$$

$$C = e^{-r\Delta t}(2\pi\Delta t)^{-n/2}(|R|)^{-1/2}(\sigma_1\sigma_2 \ldots \sigma_d)^{-1}, \tag{7.16}$$

$$E(x_1, \ldots, x_d, y_i, \ldots, y_d) = exp(-\frac{1}{2}\alpha^T R^{-1}\alpha), \tag{7.17}$$

$$\alpha_i = \frac{x_i - y_i + (r - D_i - \frac{\sigma_i^2}{2})\Delta t}{\sigma_i(\Delta t)^{1/2}} \tag{7.18}$$

We define the grid density factor d_f as $\frac{\sqrt{\Delta t}}{\delta y}$ such that it is inversely proportional to the grid spacing. The optimal (m_w, d_f) is found to be (20,23).

7.4.3 Genz's "Discontinuous" benchmark integral

Our last case study is Genz's "Discontinuous" benchmark multi-dimensional integral(7.19). It is a common test integral being used in evaluation of different numerical integration methods. This benchmark integral has been used as an application in the mixed precision Monte-Carlo framework in Chapter 6. In the tests in this chapter, we set the number of dimensions to four as a higher dimension would be better handled using Monte-Carlo integration. The integration domain is $[0, 1)^4$. Fully parallelised designs are used in our FPGA implementations and the data-paths can compute a single

sample point per clock cycle, with constants c_i and w_i:

$$I = \int\int\cdots\int f_{dis}(x_1, x_2, \cdots x_n)dx_1dx_2\cdots dx_n \tag{7.19}$$

$$f_{dis} = \begin{cases} 0 & \text{if } x_0 > w_0 \text{ or } x_1 > w_1 \\ exp(\sum_{i=1}^{n}(c_i \times x_i)) & \text{otherwise} \end{cases} \tag{7.20}$$

In this problem, we define $d_f = N$ since its grid density should depends on the number of grid points only. The optimal (m_w, d_f) is found to be (11,96).

7.5 Result and Evaluation

We use the MaxWorkstation reconfigurable accelerator system from Maxeler Technologies for our evaluation. It has a MAX3424A card with a Xilinx Virtex-6 SX475T (xc6vsx475t) FPGA. The xc6vsx475t FPGA has a total of 297,600 LUTs, 595,200 FFs, 1,064 DSPs and 2,016 BRAMs. We set the target clock frequency at 100MHz ($freq$). The card is connected to an Intel i7-870 CPU through a PCI express link with a measured bandwidth of 2 GB/s. The Intel CPU has 4 physical cores.

The Intel Compiler (ICC) is used in our software implementations with optimisation flag `-fast` and SSE4.2 enabled. The software implementation is manually optimised in order to achieve the maximum throughput. Multiple processes are launched simultaneously in order to utilise all 4 physical cores of the quad-core i7-870 CPU.

For the FPGA implementations, we use the MaxCompiler as our development system, which adopts a streaming programming model similar to [115] and supports customisable data formats so that floating-point calculations can be performed with different mantissa bit-widths. The hardware implementations are synthesized, placed and routed using Xilinx 13.1 ISE.

For the GPU performance result, we use NVIDIA Tesla C2070 GPU to measure the performance of our 3-dimensional European option pricer. The GPU has 448 cores running at 1.15 GHz and has a

Table 7.1: Comparison of different applications using i7-870 quad-core CPU, NVIDIA Tesla C2070 GPU, double precision xc6vsx475t FPGA and reduced precision optimised xc6vsx475t FPGA.

	Discrete barrier option			3D European option				Genz's benchmark		
	CPU	FPGA		CPU	GPU	FPGA		CPU	FPGA	
arithmetic	double	double	optimised	double	double	double	optimised	double	double	optimised
clk freq. (GHz)	2.93	0.1	0.1	2.93	1.15	0.1	0.1	2.93	0.1	0.1
num. of cores	4	7	35	4	448	5	18	4	6	36
exec. time (sec.)	313	86.3	22.3	145	11.45	34.5	9.6	328.56	169.98	28.33
norm. speedup	1x	3.6x	**14.0x**	1x	12.7x	4.2x	**15.1x**	1x	1.9x	**11.6x**
opti. gain	-	1x	**3.9x**	-	-	1x	**3.6x**	-	1x	**6.0x**
APCC(W) [1,2]	89	13	16	69	117	4	5	81	4	4
AECC(J) [3]	27857.0	1121.9	356.8	10005.0	2026.7	138.1	48.0	26613.4	679.9	113.3
norm. energy	**78.1x**	3.1x	1x	**208.4x**	**42.2x**	2.9x	1x	**234.9x**	**6x**	1x

[1] APCC = run-time power consumption - idle power consumption.
[2] The idle power is $80W$ for FPGA and CPU system, and $154W$ for GPU system.
[3] AECC = APCC $\times$ execution time.
[4] In all applications, $\epsilon_{tol} = 10^{-3}$.

peak double precision performance of 515 GFlops.

The experiments are performed to compute a portfolio of 100 barrier options, a portfolio of 576 3D-European options, and a set of 1120 Genz's benchmark integrals.

7.5.1 Performance Comparison

Table 7.1 shows comparisons of the implementations running on a CPU with double precision arithmetic, an FPGA with double precision arithmetic, and an FPGA with optimised precision using our proposed methodology. The GPU result of the multi-dimensional European option pricer is also presented. The computed results of all designs are all optimised for $\epsilon_{tol} = 10^{-3}$ and have the same accuracy level. The measured execution time includes the data transfer time, which means the speedup figures are measured end-to-end.

Using the reduced precision optimisation techniques with xc6vsx475t FPGA, we achieve 3.6 to 6.0 times speedup gain over the original double precision FPGA designs. These optimised FPGA designs running on xc6vsx475t are 11.6 to 15.1 times faster than multi-threaded software designs running on a **quad-core** Intel i7-870, and 1.2 times faster than a GPU design running on a Tesla C2070.

7.5.2 Energy Comparison

We also compare the energy efficiency of the three applications on different devices. The average power consumption is measured using a remote power measuring socket from Oslon® electronics with an measuring interval of 1 second. Additional power consumption for computation (APCC) is defined as the power usage during the computation time (run-time power) minus the power usage at idle time (static power). In other words, APCC is the dynamic power consumption for that particular computation. Since the dynamic power consumption fluctuates a little, we take the average value of dynamic power to be the APCC. The additional energy consumption for computation (AECC) is defined by the following equation:

$$AECC = APCC \times Total\ Computational\ Time. \tag{7.21}$$

Therefore, AECC measures the actual additional energy consumed for that particular computation.

As shown in Table 7.1, the precision optimised FPGA designs demonstrate the greatest energy efficiency over both CPU and GPU. It is 78.1 - 234.9 times more energy efficient than Intel i7-870 quad-core CPU, and 42.2 times more energy efficient than Tesla C2070 GPU.

7.6 Summary

We presented a precision optimisation methodology for the generic quadrature method using reconfigurable hardware. Our novel methodology optimises the performance by considering both integration grid density and mantissa bit-width of the floating-point operators. Increasing the integration grid density reduces integration error but increases the required amount of computation, while increasing the mantissa bit-width improves precision but decreases the computation speed, due to reduced parallelism. Our proposed algorithm allows us to identify the optimal balance between the number of integration points and the precision of the floating-point operator, such that the throughput is maximised while the accuracy remains in a given error tolerance level.

Our three case studies demonstrate that using our proposed optimisation methodology, the reduced

precision FPGA designs are up to 6 times faster than comparable FPGA designs with double precision arithmetic. They are up to 15.1 times faster and 234.9 times more energy efficient than an i7-870 quad-core CPU, and are 1.2 times faster and 42.2 times more energy efficient than a Tesla C2070 GPU.

We are unable to compare this chapter with other previous works as there is no similar work which accelerates quadrature methods in option pricing with reduced precision using FPGA. If we compare this chapter with Chapter 4 which is about FPGA-accelerated option pricing using quadrature methods, we can see that the performance has been increased significantly using our reduced precision methodology as the speedup of FPGA over a comparable CPU has been improved from 4.6 times to 15.1 times.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

The research in this thesis has contributed to reconfigurable financial computing and satisfied our three main objectives: *Generic architecture for derivatives pricing*, *Automated management* and *Precision optimisation*. There are three main achievements.

- First, we proposed novel reconfigurable designs for generic derivative pricing using both Monte-Carlo and quadrature methods. Optimisation techniques such as control variate method and automation of multi-dimensional designs generation are proposed and explored. Significant speedups and energy savings are achieved using our reconfigurable designs over both CPU and GPU.

- Second, we proposed a scalable distributed framework for financial computing on multi-accelerator heterogeneous clusters. In this framework, different computational devices including FPGAs, GPUs and CPUs work collaboratively on the same financial problem based on dynamic scheduling policy. The speed and energy consumption trade-off of different accelerator allocation is explored and analysed.

- Finally, we proposed a mixed precision methodology and a reduced precision methodology for optimising the performance of Monte-Carlo and quadrature methods. These methodologies en-

able us to obtain the optimal throughput of the reconfigurable design by using reduced precision datapath while maintaining the same accuracy of the results.

Table 8.1: Summary of the key results

Research Area	Key results
Monte-Carlo Methods for Option Valuation (Chapter 3)	• The FPGA design under the proposed novel parallel hardware framework using the control variate Monte-Carlo method for pricing exotic options is 24 and 2.4 times faster than comparable CPU and GPU designs. • The FPGA design using control variate Monte-Carlo method is 2 times faster than the FPGA design using pure Monte-Carlo method for a given accuracy. Therefore, the reduced number of cores in FPGA for the control variate method is more than compensated by the benefit of reduced variance.
Quadrature Methods for Option Valuation (Chapter 4)	• The FPGA design under the proposed novel parallel architecture using quadrature methods is 4.6 and 1.8 times faster than comparable CPU and GPU designs. • The FPGA design is up to 25 times more energy efficient than comparable CPU and GPU implementations.
Distributed Financial Computing in Heterogeneous Cluster (Chapter 5)	• Two applications are implemented using our proposed scalable framework on multi-accelerator heterogeneous cluster with three sample dynamic policies. A speedup of 33.8 times is achieved by using both FPGA, GPU and CPUs collaboratively in one node of the multi-accelerator heterogeneous cluster. • There is a trade-off between performance and energy consumption for different accelerator allocations in a cluster. The Efficient Allocation Line (EAL) approach can be used to determine the most efficient accelerator allocation for our given objective.
Optimising Performance of Monte-Carlo Methods with Mixed Precision (Chapter 6)	• A reconfigurable accelerator system using our proposed mixed precision methodology for Monte-Carlo simulation is 4.6 times faster than a comparable GPU system, 7.1 times faster than an original FPGA system using double precision, and up to 106 times faster than a software design running on a comparable CPU system. • The mixed precision FPGA design is also 5.5 times more energy efficient than a GPU design and 104 times more energy efficient than a software design running on a quad-core CPU.
Optimising Performance of Quadrature Methods with Reduced Precision (Chapter 7)	• The reduced precision FPGA designs using our proposed reduced precision optimisation methodology and technique are up to 1.2 times faster than a comparable GPU design, 6 times faster than the original FPGA designs using double precision, and 15.1 times faster than a software design running on a comparable CPU. • The reduced precision FPGA design is also 42.2 times more energy efficient than a GPU design and 234.9 times more energy efficient than a software design running on a quad-core CPU.

The key results of each chapter in this thesis are summarised in Table 8.1. These results are important for accelerating financial computing. Our designs, methodologies and techniques are available to be applied in the financial industry to enhance computing throughput and reduce energy consumption. Our experimental results also provide insights into the general comparisons of performance

and energy efficiency between different computational devices including FPGAs, GPUs and CPUs. These comparison results are key consideration aspects for financial institutions to make decision on designing their derivative pricing infrastructure.

8.2 Impact

There are three main impacts of this thesis. First, the reconfigurable designs of quadrature and Monte-Carlo option pricers described in this thesis have an impact on satisfying the high computational demand in the financial industry. Second, the optimisation techniques described in this thesis have an impact on providing optimisation techniques of using reconfigurable hardware as accelerator in financial applications. Lastly, the performance comparison results in this thesis have an impact on determining the right combination of accelerators in financial applications.

8.2.1 Satisfying high computational demand in the financial industry

Financial institutions continually invent new financial products in order to satisfy the needs of different investors as well as to diversify the risk. Many financial derivatives involve multiple underlying assets. The computation complexity increases exponentially with the number of underlying assets (i.e. the number of dimensions), finding ways to accelerate the option pricing computation becomes a significant challenge. In order to calculate the overall risk exposure, thousands of computing servers are required for financial institutions to test different risk scenarios. A huge amount of energy is consumed by these computing servers and the corresponding cooling equipments. In 2010, it is estimated that 41 million servers on the planet consumed around 18,118 billion kWh electricity each year when the energy for associated cooling and power distribution is included [3]. Therefore, each additional computing server consumes 442 thousand kWh of electricity each year in average. Accelerate financial computing in an energy efficient way is therefore one of our major goals.

It has been predicted that the performance of processors in 2024 will have only 7.9 times average speedup over the processors in 2008 [7]. In Chapter 7, we shown that our FPGA design of quadra-

ture option pricer using reduced precision optimisation methodology is 15.1 times faster and 234.9 times more energy efficient than a software design running on a comparable CPU. This would be a significant impact on the financial industry as it meets the high computation demand in the financial industry in a much more energy efficient way even when comparing with the predicted processors in 2024.

8.2.2 Providing optimisation techniques in financial application domain

Moore's Law [4] (the doubling of transistors on chip every 18 months) and Dennard scaling [5] has been the fundamental drivers of computing technology for the previous years. Since the end of Dennard scaling in 2005, processor designers have increased core counts to continue exploit Moore's Law scaling [6]. However, the increasing number of components on a chip, combined with decreasing energy scaling, is leading to the phenomenon of "Dark Silicon" [7], where chips have a too high power density to use all components at once. These challenges are changing the computer technology to emphasize on efficiency, and are forcing chips to use multiple different components, each carefully optimised to efficiently execute a particular type of task.

According to the Europe's premier organisation for coordinating research - HiPEAC (European Network of Excellence on High Performance and Embedded Architecture and Compilation), the solution to improved energy efficiency is to use application-optimised processors and accelerators [8]. By optimising these components for specific application, their energy efficiency can be increased by orders of magnitude. Therefore, "efficiency" of heterogeneous computing systems is one of the major research objectives in the roadmap of HiPEAC.

Optimisation techniques described in this thesis including the control-variate optimisation in Chapter 3, the mixed precision optimisation in Chapter 6 and the reduced precision optimisation in Chapter 7 are in line with the research roadmap of HiPEAC for enhancing the performance and energy efficiency of FPGA as accelerators in financial applications. The performance of these optimisation techniques is compared in detail with the original design in each of the chapter. These techniques have an impact on designing and optimising energy efficient financial application when using recon-

figurable hardware as accelerators in the future.

8.2.3 Determining the right combination of accelerators

According to the 2011/12 roadmap of HiPEAC (European Network of Excellence on High Performance and Embedded Architecture and Compilation) [8], determining the right mix or choice of processor and accelerators for a specific application domain in order to maximise the efficiency is one of the major research goals for computing systems.

In this thesis, the performance comparisons between FPGA, GPU and CPU for different types of financial applications including the use of quadrature and Monte-Carlo methods are presented which enable us to determine the right choice of accelerators for different types of financial applications. In addition, in Chapter 5, we proposed a methodology to dynamically partition and schedule the tasks and so to select accelerators depending on the optimisation goal. Therefore, this thesis has an impact on determining the right mix or choice of accelerators for financial applications, which is one of the main future challenges in this era of changing technology of computing systems.

8.3 Future Work

8.3.1 Quadrature methods in other problem domain

Accelerating quadrature methods with reconfigurable hardware can be applied to other problem domain apart from option pricing. In Chapter 4 and 7, we presented the architectures and techniques of hardware accelerated quadrature methods in option pricing. Similar architectures and techniques can be applied on other applications if the computations can be expressed in an integral form. Some of the possible applications include the solutions of electromagnetic problems [41], calculations involving photon distribution [42] and modeling of credit risk [40].

8.3.2 Accelerating adaptive quadrature methods

Adaptive quadrature method is a process in which the integral of a function $f(x)$ is approximated using quadrature rules on adaptively refined subintervals of the integration range. Adaptive quadrature methods are as efficient as traditional quadrature methods for "well-behaved" integrands, but are also effective for "badly-behaved" integrands for which traditional algorithms fail [118]. Adaptive quadrature algorithm uses an estimate of the error from calculating a definite integral. If the error exceeds a user-specified tolerance, the algorithm calls for subdividing the interval of integration in two and applying adaptive quadrature algorithm to each subinterval in a recursive manner.

In Chapter 4 and 7, we demonstrated the effectiveness of hardware accelerated quadrature methods using static integration interval only. One of the future work is to extend it to cover adaptive quadrature. There are two issues when designing hardware accelerated adaptive quadrature method. First, an additional control logic is required for estimating the error, subdividing the interval and recomputing the integral. How to effectively design this control logic such that the whole design is fully parallel and pipelined is one of the key challenge. Second, the run-time of the adaptive algorithm is non-deterministic. An additional protocol is needed for the hardware to notify the host when the computation is finished.

8.3.3 Monte-Carlo method in other problem domain

Hardware accelerated Monte-Carlo method could be applied to other problem domain apart from option pricing. Many existing problems are solved by Monte-Carlo methods, such as calculation of particles transport in Physics [119], Bayesian network learning in the field of Biology [120] and modeling evolution of galaxies in Astrophysics [121]. They could be implemented in reconfigurable hardware by designing a hardware simulation core.

In Chapter 3 and 6, we demonstrated the effectiveness of hardware accelerated Monte-Carlo methods for option pricing with optimisation techniques including control variate method and mixed precision methodology. The control variate techniques could also be applied in other applications as well provided that there is a control variable to be used as a proxy.

8.3.4 Interest rate derivative pricing

The derivatives priced in this thesis are mainly equity derivatives. The same techniques can also be applied to the pricing of interest rate derivatives with models such as BGM model [16] or HJM model [122]. The design complexity will be increased as those model involves non-constant interest rate. The critical difference is that in each Monte-Carlo path of interest rate derivatives, the interest rates in different time steps are also simulated using Monte-Carlo method. As a result, there is an additional dimension to simulate compared to equity derivatives.

8.3.5 Accelerating Quasi Monte-Carlo methods

Quasi Monte-Carlo (QMC) methods is similar to regular Monte-Carlo methods except that low-discrepancy sequences are used instead of random numbers [123]. QMC approach in pricing some types of financial derivatives is faster than regular Monte-Carlo method for a given accuracy [124].

In Chapter 3 and 6, we presented hardware accelerated Monte-Carlo methods for option pricing with optimisation techniques including control variate method and mixed precision. If the random number generators were changed to low-discrepancy sequence generators such as Sobol-sequence generators [45], we could get a hardware accelerated Quasi Monte-Carlo option pricer. However, the optimisation technique of control variate methods and mixed precision methodology are based on the statistical random nature of regular Monte-Carlo method. Therefore, they are not directly applicable on Quasi Monte-Carlo methods.

8.3.6 Other grid-based pricing methods

Quadrature methods presented in Chapter 4 and 7 are one of the grid-based option pricing methods. There are other grid-based numerical methods for option pricing such as finite-difference methods and tree-based methods as described in Chapter 2. Although they have been studied in other research [36, 38], optimisation techniques described in this thesis could be applied on those works such as the reduced precision methodology described in Chapter 7.

8.3.7 Sophisticated dynamic scheduling policies

In Chapter 5, we demonstrated that different computational devices can work collaboratively on the same problem with dynamic scheduling policies. There could be a further improvement by developing more sophisticated dynamic scheduling polices. For example, a policy that minimise communication overhead, a policy that minimise energy consumption and a policy that optimise for load-balancing across nodes could be designed and employed in the framework described in Chapter 5. Experiments of these more sophisticated dynamic scheduling policies could be carried in order to verify their effectiveness.

8.3.8 Algorithmic Trading

Algorithmic trading is a computer-based approach to execute buy and sell orders on financial instrument. Financial traders exercise investment strategies using autonomous high-frequency algorithmic trading by real-time market events. To take advantage of the timely market information, the algorithmic trading engine must be able to respond quickly. Existing pure software solutions are no longer able to provide low latency solutions. There is a need for hardware acceleration for the algorithmic trading engine and reconfigurable hardware is a highly desirable platform.

Research on algorithmic trading engine using reconfigurable hardware has been focusing on equity trading only. An implementation of "Participate" algorithms for trading equity orders in reconfigurable hardware is presented and shows a 133 times speedup over a software implementation [53]. In this thesis, we designed option pricing engine and presented performance optimisation techniques. The input parameters of the option pricing engine could be obtained from live market data-feed, and the pricing result could be used as an decision input of an algorithmic trading engine. Therefore, our work could be used for extending an algorithmic trading engine to support the trading of options.

Bibliography

[1] David Dorfman and Don Canning. The actuarys high-performance computing challenge. *Windows in Financial Services*, 2007.

[2] M.B. Haugh and A.W. Lo. Computational challenges in portfolio management. *Computing in Science Engineering*, 3(3):54 –59, may 2001.

[3] J.G. Koomey. Growth in data center electricity use 2005 to 2010. Technical report, Standord University, 2011.

[4] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82 –85, jan 1998.

[5] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Proceedings of the IEEE*, 87(4):668 –678, apr 1999.

[6] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *Solid-State Circuits Newsletter, IEEE*, 12(1):11–13, winter 2007.

[7] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, 2011.

[8] M. Duranton, D. Black-Schaffer, and K. De Bosschere S. Yehia. Computing systems: Research challenges ahead the HiPEAC vision 2011/ 2012.

[9] K. McLaughlin, N. O'Connor, and S. Sezer. Exploring cam design for network processing using fpga technology. In *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, page 84, feb. 2006.

[10] Liang Lu, J.V. McCanny, and S. Sezer. Reconfigurable motion estimation architecture for multi-standard video compression. In *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pages 253 –259, Jul 2007.

[11] K.H. Tsoi, D. Rueckert, C.H. Ho and W. Luk. Reconfigurable Acceleration of 3D Image Registration. In *Proc. Southern Programmable Logic Conference (SPL)*, pages 95–100, 2009.

[12] C.K. Wong and P.H.W. Leong. An FPGA-Based Electronic Cochlea with Dual Fixed-Point Arithmetic. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, 2006.

[13] Adrien Le Masle, Wayne Luk, Jared Eldredge, and Kris Carver. Parametric encryption hardware design. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5992 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin / Heidelberg, 2010.

[14] Melissa C. Smith, Jeffery S. Vetter, and Xuejun Liang. Accelerating scientific applications with the SRC-6 reconfigurable computer: Methodologies and analysis. In *IEEE International Parallel and Distributed Processing Symposium IPDPS '05*, pages 157b–157b, 2005.

[15] C.H. Ho, K.H. Tsoi, H.C. Yeung, Y.M. Lam, K.H. Lee, P.H.W. Leong, R. Ludewig, P. Zipf, A.G. Ortiz, and M. Glesner. Arbitrary function approximation in hdls with application to the n-body problem. In *Proc. International Conference on Field Programmable Technology (FPT)*, pages 84–91, Dec. 2003.

[16] G.L. Zhang, P.H.W. Leong, C.H. Ho, K.H. Tsoi., Dong-U Lee, C.C.C. Cheung, R.C.C. Cheung, and W. Luk. Reconfigurable acceleration for Monte-Carlo based financial simulation. In *Proc. Int. Conf. on Field-Programmable Technology*, pages 215–224. IEEE, 2005.

[17] David B. Thomas and Wayne Luk. Credit risk modelling using hardware accelerated monte-carlo simulation. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008.

[18] Stephen Weston, Jean-Tristan Marin, James Spooner, Oliver Pell, and Oskar Mencer. Accelerating the computation of portfolios of tranched credit derivatives. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–8, Nov. 2010.

[19] Stephen Weston, James Spooner, Sebastien Racaniere, and Oskar Mencer. Rapid computation of value and risk for derivatives portfolios. *Concurrency and Computation: Practice and Experience*, 2011.

[20] A. Lee, C. Yau, M. B. Giles, A. Doucet, and C. C. Holmes. On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. *Journal of Computational and Graphical Statistics*, pages 769 –789, 2010.

[21] Ari D. Andricopoulos, Martin Widdicks, Peter W. Duck, and David P. Newton. Universal option valuation using quadrature methods. *Journal of Financial Economics*, 67(3):447–471, March 2003.

[22] Anson H. T. Tse, David B. Thomas, K. H. Tsoi, and Wayne Luk. Efficient reconfigurable design for pricing asian options. *SIGARCH Comput. Archit. News*, 38:14–20, Jan 2011.

[23] Anson H. T. Tse, David B. Thomas, K.H. Tsoi, and Wayne Luk. Reconfigurable control variate monte-carlo designs for pricing exotic options. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, 2010.

[24] A. H. T. Tse, D. Thomas, and W. Luk. Design exploration of quadrature methods in option pricing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (Accepted for publication)*, 2012.

[25] Anson H. T. Tse, David B. Thomas, and Wayne Luk. Accelerating quadrature methods for option valuation. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2009.

[26] A. H. T. Tse, D.B. Thomas, and W. Luk. Option pricing with multi-dimensional quadrature architectures. In *Proc. International Conference on Field Programmable Technology (FPT)*, pages 427 –430, dec 2009.

[27] A. H. T. Tse, D.B. Thomas, K.H. Tsoi, and W. Luk. Dynamic scheduling Monte-Carlo framework for multi-accelerator heterogeneous clusters. In *Proc. International Conference on Field Programmable Technology (FPT)*, pages 233 –240, dec. 2010.

[28] G. C. T. Chow, A. H. T. Tse, Jin Q., D.B. Thomas, P. Leong, and W. Luk. A mixed precision Monte Carlo methodology for reconfigurable accelerator systems. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA) (Accepted for publication)*, 2012.

[29] A. H. T. Tse, G. C. T. Chow, Jin Q., D.B. Thomas, and W. Luk. Optimising performance of quadrature methods with reduced precision. *Proc. International Symposium on Applied Reconfigurable Computing (Accepted for publication)*, 2012.

[30] Robert James Elliott and P.E. Kopp. *Mathematics of financial markets*. Springer, 2005.

[31] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

[32] Robert C Merton. On the pricing of corporate debt: The risk structure of interest rates. *Journal of Finance*, 29(2):449–70, May 1974.

[33] A. G. Z. Kemna and A. C. F. Vorst. A pricing method for options based on average asset values. *Journal of Banking & Finance*, 14(1):113–129, March 1990.

[34] Tim Bollerslev. Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31(03):307–327, 1986.

[35] John C. Cox, Stephen A. Ross, and Mark Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3):229 – 263, 1979.

[36] Qiwei Jin, David B. Thomas, Wayne Luk, and Benjamin Cope. Exploring reconfigurable architectures for binomial-tree pricing models. In *Proceedings of the 4th international workshop on Applied Reconfigurable Computing*, pages 245–255. LNCS 4943. Springer-Verlag, 2008.

[37] J.C. Hull. *Options, Futures and Other Derivatives*. Prentice Hall, 6th edition, 2005.

[38] Qiwei Jin, D.B. Thomas, and W. Luk. Exploring reconfigurable architectures for explicit finite difference option pricing models. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pages 73 –78, 2009.

[39] Ari D. Andricopoulos, Martin Widdicks, David P. Newton, and Peter W. Duck. Extending quadrature methods to value multi-asset and complex path dependent options. *Journal of Financial Economics*, 83(2):471 – 499, 2007.

[40] Mark H. A. Davis and Juan Carlos Esparragoza-Rodriguez. Large portfolio credit risk modeling. *International Journal of Theoretical and Applied Finance*, 10(04):653–678, 2007.

[41] Alexandre Masserey, Jacques Rappaz, Roland Rozsnyo, and Marek Swierkosz. Numerical integration of the three-dimensional green kernel for an electromagnetic problem. *Journal of Computational Physics*, 205(1):48 – 71, 2005.

[42] T. Humphries, A. Celler, and M.R. Trammer. Improved numerical integration for analytical photon distribution calculation in spect. *Nuclear Science Symposium Conference IEEE*, 5:3548–3554, 2007.

[43] D.B. Thomas, J.A. Bower, and W. Luk. Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations. In *Int. Conf. on Application-Specific Systems, Architectures and Processors*. IEEE, 2007.

[44] G.W. Morris and M. Aubury. Design space exploration of the European option benchmark using hyperstreams. In *Proc. Int. Conf. on Field Programmable Logic and Applications*. IEEE, 2007.

[45] Xiang Tian and K. Benkrid. American option pricing on reconfigurable hardware using least-squares monte carlo method. In *Proc. Int. Conf. on Field-Programmable Technology*, pages 263 –270, 2009.

[46] George S. Fishman. *Monte Carlo Concepts, Algorithms, and Applications*. Springer, 1995.

[47] L. Le Cam. The Central Limit Theorem Around 1935. *Statistical Science*, 1(1):78–91, 1986.

[48] S.F. Arnold. *Mathematical statistics*. Prentice-Hall International editions. Prentice-Hall International, 1990.

[49] John Hull and Alan White. The use of the control variate technique in option pricing. *Journal of Financial and Quantitative Analysis*, 23(03):237–251, September 1988.

[50] Endre Sueli and David F. Mayers. *An Introduction to Numerical Analysis*. Cambridge University Press, 2006.

[51] Katherine Heires. Budgeting for latency: If i shave a microsecond, will i see a 10x profit? *Security Industry*, 2010.

[52] G.W. Morris, D.B. Thomas, and W. Luk. Fpga accelerated low-latency market data feed processing. In *Proc. IEEE Symposium on High Performance Interconnects*, pages 83 –89, aug 2009.

[53] Stephen Wray, Wayne Luk, and Peter Pietzuch. Exploring algorithmic trading in reconfigurable hardware. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 325 –328, Jul 2010.

[54] S. Wray, W. Luk, and P. Pietzuch. Run-time reconfiguration for a reconfigurable algorithmic trading engine. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 163 –166, Sep 2010.

[55] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proc. VLDB Endow.*, pages 1525–1528, Sep 2010.

[56] R. Pottathuparambil, J. Coyne, J. Allred, W. Lynch, and V. Natoli. Low-latency fpga based financial data feed handler. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93 –96, may 2011.

[57] Lei Pan, Lixu Gu, and Jianrong Xu. Implementation of medical image segmentation in cuda. *Proc. Int. Conf. on Technology and Applications in Biomedicine*, pages 82–85, May 2008.

[58] Honghoon Jang, Anjin Park, and Keechul Jung. Neural network implementation using cuda and openmp. *Computing: Techniques and Applications, 2008. DICTA '08.Digital Image*, pages 155–161, Dec. 2008.

[59] J.E. Stone, D. Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66 –73, may-jun 2010.

[60] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *CoRR*, 2010.

[61] Jianbin Fang, A.L. Varbanescu, and H. Sips. A comprehensive performance comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216 –225, sept. 2011.

[62] L. Lu, J.V. McCanny, and S. Sezer. Reconfigurable system-on-a-chip motion estimation architecture for multi-standard video coding. *Computers Digital Techniques, IET*, 4(5):349 –364, Sep 2010.

[63] K.H. Leung, K.W. Ma, W.K. Wong, and P.H.W. Leong. FPGA implementation of a microcoded elliptic curve cryptographic processor. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 68 –76, 2000.

[64] K.K. Ting, S.C.L. Yuen, K.H. Lee and P.H.W. Leong. An FPGA Based SHA-256 Processor. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 577–585, 2002.

[65] K.H. Tsoi, K.H. Leung and P.H.W Leong. High Performance Physical Random Number Generator. *Computers and Digital Techniques, IET*, 1(4):349–352, July 2007.

[66] P.H.W. Leong and C.K. Chung. A FPGA Based Runtime Configurable Clause Evaluator for SAT Problems. *Electronics Letters*, 35(19):1618–1619, 1999.

[67] P.H.W Leong, C.W. Sham, W.C. Wong, H.Y. Wong, W.S. Yuen and M.P. Leong. A Bitstream Reconfigurable FPGA Implementation of the WSAT algorithm. 9(1):197–201, Feb 2001.

[68] K.H. Tsoi, C.H. Ho, H.C. Yeung and P.H.W. Leong. An Arithmetic Library and Its Application to the N-body Problem. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 68–78, 2004.

[69] L. Musa. FPGAS in high energy physics experiments at CERN. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 2–2, 2008.

[70] George A. Constantinides. Word-length optimization for differentiable nonlinear systems. *ACM Trans. Des. Autom. Electron. Syst.*, 11:26–43, 2006.

[71] Ki-Il Kum and Wonyong Sung. Combined word-length optimization and high-level synthesis of digital signal processing systems. 20(8):921 –930, 2001.

[72] Claire F. Fang, Rob A. Rutenbar, and Tsuhan Chen. Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs. In *IEEE/ACM international conference on Computer-aided design*, pages 275–282, 2003.

[73] Altaf Abdul Gaffar, Oskar Mencer, Wayne Luk, and Peter Y. K. Cheung. Unifying bit-width optimisation for fixed-point and floating-point designs. In *FCCM*, pages 79–88, 2004.

[74] Dong-U Lee, Altaf Abdul Gaffar, Oskar Mencer, and Wayne Luk. Minibit: bit-width optimization via affine arithmetic. In *DAC*, pages 837–840, 2005.

[75] Dong-U Lee, Altaf Abdul Gaffar, Ray C. C. Cheung, Oskar Mencer, Wayne Luk, and George A. Constantinides. Accuracy-guaranteed bit-width optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):1990–2000, 2006.

[76] William G. Osborne, Ray C. C. Cheung, José Gabriel F. Coutinho, Wayne Luk, and Oskar Mencer. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 617–620, 2007.

[77] W.G. Osborne, J.G.F. Coutinho, R.C.C. Cheung, W. Luk, and O. Mencer. Instrumented multi-stage word-length optimization. In *Proc. International Conference on Field Programmable Technology (FPT)*, pages 89 –96, 2007.

[78] A.B. Kinsman and N. Nicolici. Finite precision bit-width allocation using SAT-Modulo theory. In *Proc. Design automation and test in Europe (DATE)*, pages 1106 –1111, 2009.

[79] D. Boland and G.A. Constantinides. Automated precision analysis: A polynomial algebraic approach. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 157 –164, 2010.

[80] G. C. T. Chow, K.W. Kwok, W. Luk, and P. Leong. Mixed precision processing in reconfigurable systems. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17 –24, May 2011.

[81] Maya Gokhale, Janette Frigo, Christine Ahrens, and Ron Minnich. Monte Carlo radiative heat transfer simulation on a reconfigurable computer. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 95–104, 2004.

[82] Alexander Kaganov, Asif Lakhany, and Paul Chow. FPGA acceleration of multifactor CDO pricing. *ACM Trans. Reconfigurable Technol. Syst.*, 4:20:1–20:17, 2011.

[83] Akila Gothandaraman, Gregory D. Peterson, G.L. Warren, Robert J. Hinde, and Robert J. Harrison. FPGA acceleration of a quantum Monte Carlo application. *Parallel Computing*, 34(4-5):278 – 291, 2008.

[84] Ramon Edgar Moore. *Interval arithmetic and automatic error analysis in digital computing*. PhD thesis, Stanford University, 1963.

[85] C.F. Fang, Tsuhan Chen, and R.A. Rutenbar. Floating-point error analysis based on affine arithmetic. In *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages II–561–4, 2003.

[86] Xiang Tian and K. Benkrid. Design and implementation of a high performance financial Monte-Carlo simulation engine on an FPGA supercomputer. In *Proc. International Conference on Field Programmable Technology (FPT)*, pages 81–88, 2008.

[87] Xiang Tian and Christos-Savvas Bouganis. A run-time adaptive FPGA architecture for Monte Carlo simulations. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[88] Dave Strenski. The Cray XD1 computer and its reconfigurable architecture. Technical report, Cray Inc., July 2005.

[89] R. Baxter et al. Maxwell – a 64 FPGA supercomputer. *Engineering Letters*, 16(3):426–433, 2008.

[90] nVidia. *nVidia CUDA Programming Guide v2.1*, 2008.

[91] Toshio Endo and Satoshi Matsuoka. Massive supercomputing coping with heterogeneity of modern accelerators. In *IEEE International Symposium on Parallel and Distributed Processing, IPDPS'08*, pages 1–10, 2008.

[92] Lokman A. Abbas-Turki, Stephane Vialle, Bernard Lapeyre, and Patrick Mercier. High dimensional pricing of exotic European contracts on a GPU cluster, and comparison to a CPU cluster. In *International Parallel and Distributed Processing Symposium*, pages 1–8, 2009.

[93] M. Showerman et al. QP: A heterogeneous multi-accelerator cluster. In *10th LCI International Conference on High-Performance Clustered Computing*, Boulder, Colorado, March 2009.

[94] Kuen Hung Tsoi and Wayne Luk. Axel: a heterogeneous cluster with FPGAs and GPUs. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 115–124, 2010.

[95] Celoxica. *Handel-C Language Reference Manual*.

[96] Celoxica. *Hyperstreams User Manual*.

[97] Celoxica. *DSM API Reference Manual*.

[98] Maxeler Technologies. *MaxCompiler Manual*.

[99] David B. Thomas and Wayne Luk. A framework for development and distribution of hardware acceleration. *Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications, Proceedings of SPIE*, 4867:60–70, 2002.

[100] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distrib. Comput.*, 15(2):109–126, 2002.

[101] J.G.F. Coutinho, J. Jiang, and W. Luk. Interleaving behavioral and cycle-accurate descriptions for reconfigurable hardware compilation. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 245–254, April 2005.

[102] Anthony L. Slade, Brent E. Nelson, and Brad L. Hutchings. Reconfigurable computing application frameworks. *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 0:251, 2003.

[103] W. Cescirio, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore socs. pages 789–794, 2002.

[104] V. Schaumont and I. Verbauwhede. A component-based design environment for esl design. *Design and Test of Computers, IEEE*, 23(5):338–347, May 2006.

[105] Satnam Singh and David J. Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 3–12, April 2008.

[106] Roberto Szechtman. Control variate techniques for monte carlo simulation. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, pages 144–149. Winter Simulation Conference, 2003.

[107] David B. Thomas and Wayne Luk. Non-uniform random number generation through piecewise linear approximations. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pages 1–6, 2006.

[108] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, Jan 1998.

[109] Gianluca Fusai and Maria Cristina Recchioni. Analysis of quadrature methods for pricing discrete barrier options. *Journal of Economic Dynamics and Control*, 31(3):826–860, March 2007.

[110] G. E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *Ann. Math. Statist.*, 29(02):610–611, 1958.

[111] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33, 2007.

[112] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 63–72, 2009.

[113] Stephen Boyd, Seung-Jean Kim, Lieven Vandenberghe, and Arash Hassibi. A tutorial on geometric programming. *Optimization and Engineering*, 8:67–127, 2007.

[114] Tim Bollerslev. Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31(3):307–327, 1986.

[115] O. Mencer. ASC: a stream compiler for computing with FPGAs. 25(9):1603 –1617, 2006.

[116] Eric W. Weisstein. Sample variance computation. `http://mathworld.wolfram.com/SampleVarianceComputation.html`, accessed Sept. 2011.

[117] Fischer Black and Myron S Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–54, May-June 1973.

[118] John R. Rice. A metalgorithm for adaptive quadrature. *Journal of the ACM*, 22:61–82, Jan 1975.

[119] D Heifetz, D Post, M Petravic, J Weisheit, and G Bateman. A monte-carlo model of neutral-particle transport in diverted plasmas. *Journal of Computational Physics*, 46(2):309 – 327, 1982.

[120] Michael D. Linderman, Robert Bruggner, Vivek Athalye, Teresa H. Meng, Narges Bani Asadi, and Garry P. Nolan. High-throughput bayesian network learning using heterogeneous multicore computers. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 95–104, 2010.

[121] A. Mucke, Ralph Engel, J.P. Rachen, R.J. Protheroe, and Todor Stanev. Monte Carlo simulations of photohadronic processes in astrophysics. *Computer Physics Communications*, 124(2-3):290 – 314, 2000.

[122] Robert Jarrow and Yildiray Yildirim. Pricing treasury inflation protected securities and related derivatives using an hjm model. *Journal of Financial and Quantitative Analysis*, 38(02):337–358, 2003.

[123] Xiaoqun Wang and Kai-Tai Fang. The effective dimension and quasi-monte carlo integration. *Journal of Complexity*, 19(2):101 – 124, 2003.

[124] N.A. Woods and T. VanCourt. FPGA acceleration of quasi-Monte Carlo in finance. In *Proc. International Conference on Field Programmable Technology (FPT)*, pages 335 –340, Sept. 2008.