Architecture Exploration for Tree-based Option Pricing Models

MEng Final Year Project Report

Qiwei Jin

Supervisor: Prof. Wayne Luk

2nd Marker: Dr. Oskar Mencer

Department of Computing
Imperial College London

June 2008
Abstract

This project explores the application of reconfigurable hardware and GPUs to the acceleration of financial computation using tree-based pricing models. Two parallel pipelined architectures have been developed for option valuation using binomial trees and trinomial trees, with support for concurrent evaluation of independent options to achieve high pricing throughput. Two highly optimised GPU implantations based on same models are developed to contrast the hardware results. The results show that in the best case the tree-based models executing on a Virtex 4 Field Programmable Gate Array (FPGA) at 82.7MHz with fixed-point arithmetic can run over 160 times faster than a Core2 Duo processor at 2.2GHz. The FPGA implementation is two times faster than the nVidia GeForce 7900GTX processor with 24 pipelines at 650MHz, and 35% slower than the nVidia GeForce 8600GTS processor with 32 Pipelines at 1450MHz. In a real scenario FPGA can run over 80 times faster than the reference AMD Opteron server processor at 1GHz and can run as fast as the nVidia GeForce 7900GTX processor. The FPGA implementation is about 50% slower than the nVidia GeForce 8600GTS processor in the real scenario.
Acknowledgements

First of all I would like to thank Prof. Wayne Luk who gives me the opportunity to do this piece of research. I couldn’t have finished this without his support and suggestions. I thank Dr. Oskar Mencer for his comments and suggestions.

Secondly I would like to thank David Thomas for his help on the *Handel-C* language and HyperStreams. He also have given me a lot of inspiring suggestions about my work.

I would also like to thank the following:

- Benjamin Cope for helping me get start with GPU programming on GLSL and answering numerous questions from me thereafter.
- Gary Chow for kindly let me use his computer to test CUDA programmes.
- Lee Howes and Jay Cornwall who kindly helped me on CUDA problems.
- Geoff Bruce for setting up a CUDA machine for me in the Lab quickly.

Finally I would like to thank my family, for their material and moral support over the past five years when I am studying in UK.
## Contents

1 Introduction

1.1 Project Inspiration .................................................. 1
1.2 Why not Monte-Carlo Simulation? .............................. 2
1.3 Objectives and Contributions .................................. 3
1.4 Published Work ..................................................... 4

2 Background

2.1 Options and American Options ................................. 5
2.2 The Binomial Tree Model .......................................... 6
2.3 The Greek Letters ................................................... 8
2.4 The Trinomial Tree Model ........................................ 9
2.5 FPGA and HyperStreams ........................................ 10
   2.5.1 The FPGA Hardware ........................................ 11
   2.5.2 Reconfigurable Computing .................................. 11
   2.5.3 HyperStreams ................................................. 12
   2.5.4 The DSM Library ............................................. 12
2.6 GPU and CUDA ...................................................... 12
   2.6.1 Computing in GPU .......................................... 12
   2.6.2 CUDA ......................................................... 13
2.7 Comparing FPGA and GPU ........................................ 15
2.8 Summary ........................................................... 15

3 Design Methodology .................................................. 17

3.1 Properties of the Tree Model .................................. 17
3.2 A Software Approach ............................................. 18
3.3 The Hardware Approach ......................................... 19
   3.3.1 Binomial Tree .............................................. 22
   3.3.2 Trinomial Tree ............................................ 23
   3.3.3 C-Slow .................................................... 23
   3.3.4 Parallel Replications ..................................... 24
3.4 The GPU Approach ................................................. 24
3.5 Summary ........................................................... 25

4 Design and Implement The Step Function ..................... 27

4.1 The Binomial Valuation Core: A Naive and An Improved Approach ........................................ 27
4.2 The Trinomial Valuation Core ................................. 29
4.3 Real Numbers in Hardware ..................................... 30
4.4 HyperStreams Vs Pipelined Floating Point Library .......... 31
Chapter 1

Introduction

Definition: **Hardware Assisted Acceleration** - is the use of hardware to perform some function faster than is possible in software running on the normal (general purpose) CPU [32].

A imaginary scenario:

**7:59AM:** In the Data Center control room of Investment Bank *FakeBank*, Chief System Administrator A is staring at his monitors and waiting anxiously for the Exchange to open, he wants to make sure everything will run smoothly in the morning peak time.

**8:00AM:** The Exchange opened sharply on time, hundreds of stock price updates are received and the corresponding derivative prices are updated accordingly. However, the system is soon fully loaded and all the pricing systems are experiencing delays between 1 to 3 seconds. A can not help thinking: we have just built a larger Data Center equipped with better computer facilities, why is the price updating still so slow?

You might think I am exaggerating the situation, let’s look at some numbers I cited from the Chicago Board of Options Exchange (CBOE)’s website. CBOE is one of the world’s largest options exchanges (www.cboe.com). At between 9:00AM and 9:30AM EST on 16th of June 2008, 1.24 million options are traded within half an hour. This means 690 options are being traded each second. This haven’t even counted the numerous over-the-counter options traded privately within the financial institutions. The options being traded will need to be priced. With such trading volume, the demand for fast and accurate option pricing can be imagined to be huge. I make a simple estimation, a modern PC is able to valuate an option in \(2.31 \times 10^{-3}\) second if the 150-Step binomial model is adopted, I get the number from the testing case in Chapter 8. If CBOE only have one PC, the it takes the PC 1.6 seconds finish pricing all the options being traded in a second. This means infinite delay in pricing, as the options being traded now are expected to have its price 1.6 seconds later, and the options being traded in the next second are expected to be priced 3.2 seconds later, etc.

The existing solution is to build large data centers, or computer farms to cope with the computational demand. Unfortunately this solution brings in two other problems:
• Massive power consumption, as both computers and cooling facilities need to be powered.

• Huge Space consumption, as we need a place to put the computers and cooling facilities.

On the other hand, hardware like Field Programmable Gate Arrays (FPGAs) and Graphics processing units (GPUs) provides an alternative route to deal with such problems. Satisfactory acceleration can be achieved with very little power and space consumption, if appropriate architectures are applied.

1.1 Project Inspiration

Previous work on hardware acceleration of financial simulation has focused on Monte Carlo methods. The nature of Monte-Carlo simulations allow simple mapping to hardware with potentially unlimited parallelism, as each run of simulation is independent to others. Three examples are given below. First, a stream-oriented FPGA-based accelerator with higher performance than GPUs and Cell processors has been proposed for evaluating European options [20]. Second, an automated methodology has been developed that targets high-level mathematical descriptions of financial simulation to produce optimised pipelined designs with thread-level parallelism [27]. Third, an architecture with a pipelined datapath and an on-chip instruction processor has been reported for speeding up the Brace, Gatarek and Musiela (BGM) interest rate model for pricing derivatives [38]. All three approaches result in designs based on Monte Carlo methods. However, many financial simulations have closed-form solutions, for which techniques such as binomial and trinomial trees will be more effective.

Meanwhile many interesting researches projects have been carried out to compare FPGAs and GPUs on different applications, with the intention to study their characteristics and exploit their resources for suitable applications. For example, a matched filter is implemented on FPGA, Cell processors and GPU in search of an optimal solution [1]. A comprehensive comparison between FPGA and GPU has been made to study their performance and characteristics on three diverse computing-intensive applications: Gaussian Elimination, Data Encryption Standard (DES), and Needleman-Wunsch [8].

When the project is started almost no work has been done investigating possibilities to accelerate the tree-based models, and I hoped to try and change this by exploring possible designs based on both FPGAs and GPUs.

1.2 Why not Monte-Carlo Simulation?

Tree-based pricing models are relatively simple compared with Monte-Carlo Methods. The most widely used tree-based pricing model in finance applications is the binomial model [15], since it is simple, efficient and importantly, can handle certain types of options that are difficult to price using Monte-Carlo methods. Another example is the trinomial option pricing model. It is an alternative to the binomial model which requires fewer tree nodes (computation steps) to achieve the same level of accuracy. As the trinomial model involves more complex computations in one step, it is used less often than the binomial
model for simple option valuations. However trinomial models are more widely adopted to evaluate interest rate derivatives [17], since it offers additional freedom in the model that cannot be achieved by binomial models, for example, to represent features of the interest rate process such as mean reversion [15]. The tree-based models are often used to provide prices to a trader, but increasingly is also used as a component of larger applications, where the application may use the model to value hundreds or thousands of options.

Pricing a single option using the tree-based model such as the binomial tree and the trinomial tree is relatively fast, and can typically be performed in within a second on a modern general-purpose processor. However, when huge numbers of options need to be valued, for example if the tree-based pricing model is embedded in a Monte-Carlo simulation, or if a huge number of options are being revalued are being revalued in real-time using live data-feeds, for example hundreds of options are being valued in a second-by-second basis, the pricing model can become the main computational bottleneck. This project studies how Field Programmable Gate Arrays (FPGAs) can provide a viable method of accelerating tree-based pricing computation, and how my proposed approach can be mapped effectively onto reconfigurable hardware (FPGAs). In addition, I also seek for the possibilities to map the same models to Graphical Processing Units (GPUs) and exploit the internal parallelism of the models to achieve acceleration. The FPGA and GPU implementations are compared and their characteristics are studied.

1.3 Objectives and Contributions

The main objective of this project is to explore possible designs for tree based models on both FPGAs and GPUs and study their characteristics to find an optimal solution for real applications.

Having completed the project I consider the above objective completed successfully. The main contributions in this project is listed as the following:

1. The design of two parallel pipelined architectures based on binomial tree and trinomial tree models, and the GPU models - Chapter 3
   The properties of the tree-based models are analysed and optimised software prototypes of the binomial and trinomial models are proposed. Based on the properties of the trees and the software prototypes two parallel pipelined architectures and two GPU models are designed for binomial models and trinomial models respectively. These design methodologies are later adopted in the implementations and can generally be applied for similar applications.

2. The implementations of fully pipelined Evaluation Cores on FPGA for the tree models- Chapter 4
   Evaluation Core is considered to be the most complex component in the system which the overall performance of the system will depend on. Evaluation Cores for both binomial and trinomial models are greatly highly optimised and fully pipelined to make sure a high performance is achieved. The implementations of Evaluation Cores also allows easy adoption of
different number representations and easy portability to different FPGA models.

3. **An optimised solution for tree valuations on FPGAs** - Chapter 5
   Two straightforward designs to model binomial trees and trinomial trees are proposed. The binomial design is modified to value Greeks with almost no additional overhead introduced. To implement an high throughput solution to valuate trees I tried several ways to optimise my implementation, for example reducing the number of memory reads from three to one in the inner-most loop; and pipelining the *Evaluation Core* to allow higher throughput. I also proposed an replicated architecture that is able to valuate multiple options simultaneously.

4. **Implementations for the tree models based on two different GPUs** - Chapter 6
   A GLSL design for the binomial tree model is illustrated, such implementation is tested on a nVidia GeForce 7900GTX GPU. Two designs for the binomial tree model and the trinomial tree model are developed based on nVidia’s new CUDA technology. The designs use tree partitioning and double buffering in high-speed caches to achieve higher parallelism and avoid possible data loss.

5. **Comparison of the FPGA and GPU implementations and study of their characteristics** - Chapter 7
   Evaluated the implementation of the *Evaluation Core* based on comparisons to two reference PCs and two GPUs, the upper bounds of speed-ups for the FPGA implementations are acquired; and the strength and weaknesses of FPGAs and GPUs are addressed. The result shows in an ideal case the FPGA can run two times faster than a GPU if appropriate architecture is used. To evaluation the FPGA tree models under real-world scenario, different tests are carried out to valuate the actual speed-ups of the FPGA implementations. The result shows those speed-ups can potentially be close to their corresponding upper bounds.

### 1.4 Published Work

During the project period I am lucky enough to be able to finish a conference paper on the same topic for ARC 2008 (the International Workshop on Applied Reconfigurable Computing) with the help of David, Wayne and Ben [16]. Some of the material in the report have ready been presented in ARC and a revised version of the paper is lately submitted to a special issue of ACM TRETS.
Chapter 2

Background

In this project I try to explore a viable way to accelerate tree-based models for option valuations using Field Programmable Gate Arrays (FPGAs) and contrast it with same models based on Graphics Processing Units (GPUs). This covers a wide range of issues from financial engineering to computing. Therefore to begin with the main body of the report, I will give a brief overview of some essential backgrounds that are necessary to understand: which areas are covered in this project; what resources are available for me and what problem I am trying to solve. In particular the following topics will be covered:

- The concept of financial options and in particular American Options; in Section 2.1
- Details of how the binomial tree model can be used to valuate American Put Options; in Section 2.2
- What are the Greek Letters and how to estimate them using the binomial tree model; in Section 2.3
- How the trinomial tree model is used to price American Put Options; in Section 2.4
- The concept of reconfigurable computing, the FPGA platform available for me and the latest models of FPGA boards that could potentially be used in this project. In addition, the latest pipelined streaming library from Celoxica; in Section 2.5
- Introduction to the concept of GPU programming in general and the latest CUDA technology from nVidia; in Section 2.6
- A list of existing studies based on applications and comparisons between FPGAs and GPUs; in Section 2.7
- A few final comments about the topics I will cover in this project and the state-of-the-art of similar areas; in Section 2.8
2.1 Options and American Options

Options are financial instruments that convey the right, but not the obligation, to engage in a future transaction on some underlying security \[33\]. Options are now traded all over the world in many exchanges. There are two basic types of options, call option and put option. A call option gives the holder the right to buy the underlying asset for a certain price at some particular time. A put option is the same as the call option except that it gives the holder the right to sell. The price stated in the contract is the exercise price or strike price; the date in the contract is the exercise date or maturity \[15\].

I’ll explain in detail the concept in terms of an American put option. We know that a put option gives party A the right to sell some asset \(S\) to party B at a fixed price \(K\) (called the strike price). Noting that the option provides a right, not an obligation: party A can choose whether or not to exercise that right (i.e. to sell asset \(S\) at price \(K\)).

In general the put option will only be exercised if \(K > S_t\), i.e. the strike price \(K\) is greater than the current price of the stock \((S_t)\), party A can buy the asset from the market at a lower price and immediately sell the asset to realise a profit of \(K - S_t\). If \(K < S_t\) then party A will choose to leave the option to expire and will neither gain nor lose money. In contrast party B has no control over the option, so in the first case B will lose \(K - S_t\), and in the second case B will neither gain nor lose. Because party A only stands to gain, and B only stands to lose, B must be offered some kind of compensation. The point of an option pricing model is to determine how much A should pay B in order to create the option contract, or equivalently how much A can charge a third party for the option at a later date.

An American option is one where party A can exercise the option at any time up until the option expires at time \(T\). In contrast, a European option is one where the option can only be exercised at a particular time \(T\). All else being equal, an American option must be worth more than a European option with the same parameters, since party A has more flexibility. With the flexibility come more opportunities for profit, which translates to greater possible losses for party B, so more compensation is required for the option contract.

The American option is very common, but it presents some difficulties in pricing due to the freedom to exercise the option before the expiry date. In particular it becomes very difficult to determine the option price using Monte-Carlo methods, another common method of option pricing mentioned earlier \[27\]. In contrast, tree-based techniques are able to accurately price both European and American options.

2.2 The Binomial Tree Model

The binomial model can be seen as a discrete-time approximation to the Black-Scholes continuous-time model \[3\]. The binomial model works by discretising both time and the price of underlying asset \(S\), and mapping both onto a binary tree. Each step from the root towards the leaves increases time by one step, and at each node one of the branches leads to an increase in \(S\), while the other branch leads to a decrease in \(S\). This is shown in Figure \[2.1\] with time along the horizontal axis, and asset price along the vertical axis.
At each node the upper branch increases the asset price by a factor $u$, while the lower branch decreases the price by a factor $d$. At the root of the tree the asset price is $S_0$, which is the current asset price. At the leaves of the tree are the possible asset prices at time $T$, which are defined by $S_0$ and the path through the tree to the leaf. For example, the highest price in Figure 2.1 is reached by taking only upper branches from the root, so the asset price at that node is $S_0u^3$. Note that the asset price can only take a fixed number of values, shown as horizontal dashed lines. The tree also recombines, so the leaf node with value $S_0u$ can be reached through three paths ($uud$, $udu$, or $duu$).

The idea behind binomial tree techniques is that the put option is worth $\max(K - S_T, 0)$ at the leaves of the tree. Knowing the value at all the leaves of the tree enables us to work backwards to previous time steps, until eventually the root of the tree is reached. The right-hand side of Figure 2.1 gives a simplified example over just one node update in one time step. The node asset prices are already known (shown at the top of each node label), so the option values at the leaves (shown as $v_u$ and $v_d$) can immediately be determined. Each node within this step is updated in this way before we go to the next step. To work back to $v_0$ we require another piece of information, which is the probability ($p$) that the asset price will move up. Given $p$, the expected value of the option at the first node can then be calculated.

Two further considerations are needed for practical use. The first is that interest rate evolution means that money earned in the future is worth less than money earned now. We handle this consideration by applying a discount factor $r$ (where $r < 1$) to option values as we move backwards up the tree. The second is that at some nodes, early exercise may offer a better return than future exercise; so at each node we need to choose the higher of the discounted future payoff versus the payoff from early exercise.

From the above discussion, the pricing model can be described as:

\begin{align*}
v_{T,i} &= \max(K - S_{T,i}, 0) \\
v_{t,i} &= \max(K - S_{t,i}, r(pv_{t+1,i+1} + (1-p)v_{t+1,i-1})) \\
S_{t,i} &= \begin{cases} S_0u^i, & \text{if } i \geq 0 \\ S_0d^i, & \text{otherwise} \end{cases}
\end{align*}

where $i$ is an integer indicating the number of jumps up or down from the initial asset price.
asset price, and $t$ is an integer indicating the number of time steps away from
the root of the tree (or which step we are currently on), with the leaves having
time step $t = T$. All other values are real numbers. The inputs to the model
are $T, S_0, K, p_u, p_d$ and $r$, and the output from the model is $v_{0,0}$, which is
the estimated price for the option. **Noting that in the implementation $t$ is
usually referred as $n$, the number of steps.**

The model can be implemented in computational form as a recursive func-
tion; however a direct implementation of this function is inefficient unless memo-
isation is used. An efficient solution can be formulated in an iterative form, with
an outer loop stepping $t$ backwards from $T$ to 0, and an inner loop calculat-
ing the price for each $i$ at level $t$ in the tree. A temporary array holds the
intermediate values, and can be updated in place.

### 2.3 The Greek Letters

The Greek letters, or Greeks are used to measure the risk in an option position.
Each Greek letter measures a different dimension to the risk. Greek letters
are used when the option has been tailored and does not correspond to the
standardised products traded by exchanges. Traders manage Greek letters to
make sure all the risks are acceptable. There are five Greek letters: Delta
($\Delta$), theta ($\Theta$), gamma ($\Gamma$), vega ($\nu$) and rho ($\rho$).

Greek letters can be estimated from the binomial model, in this section I
explain the definition of each Greek letter and how to estimate them using the
binomial option pricing model. A binomial tree example is shown in Figure 2.2.

![Binomial Tree Example](image)

Figure 2.2: A binomial tree example, noting $f$ stands for the option price, for
example, $f_{11}$ is the option price at the node when the underlying asset price is
$S_0u$.

Delta ($\Delta$) is defined as the rate of change of option price with respect to the
price of underlying asset. In general,

$$\Delta = \frac{\partial c}{\partial S} \quad (2.4)$$

where $c$ is the price of the option and $S$ is the stock price. Delta can be estimated
in the binomial model as:

$$\Delta = \frac{f_{11} - f_{10}}{S_0u - S_0d} \quad (2.5)$$

Theta ($\Theta$) is defined as the rate of change of the value of the option with
respect of the passage of time, provided that all else remain the same. Theta is
sometimes referred to as the time decay of the option. The valuation of Theta is quite complex, but it can be estimated a relatively easy way:

$$\Theta = \frac{f_{21} - f_{00}}{2\Delta t}$$ (2.6)

where $\Delta t$ is the time step.

Gamma ($\Gamma$) of an option is the rate of change of the option’s Delta with respect to the underlying asset. Namely,

$$\Gamma = \frac{\partial^2 c}{\partial S^2}$$ (2.7)

Gamma can be estimated as:

$$\Gamma = \frac{[(f_{22} - f_{21})/S_0^2 - S_0] - [(f_{21} - f_{20})/(S_0 - S_0d^2)]}{0.5(S_0u^2 - S_0d^2)}$$ (2.8)

where $S_0$ is the price of the underlying asset at time 0.

Vega ($\nu$) is the rate of change of the value of the option with respect to the volatility of the underlying asset.

$$\nu = \frac{\partial c}{\partial \sigma}$$ (2.9)

where $\sigma$ is the volatility of the underlying asset. Vega can be obtained by 2 valuations of the option with a small change in to the underlying asset volatility $\sigma$ and everything else being the same:

$$\nu = \frac{f^* - f}{\Delta \sigma}$$ (2.10)

where $f$ and $f^*$ are the estimates of the option price from the original and new tree.

Rho ($\rho$) is the rate of change of the value of the option with respect to the interest rate:

$$\rho = \frac{\partial c}{\partial r}$$ (2.11)

where $r$ is the risk free interest rate. Rho can be estimated in a similar way as Vega, instead of changing the volatility, a small change to the interest rate $r$ is made and the option price is valuated before and after the change:

$$\rho = \frac{f^* - f}{\Delta r}$$ (2.12)

The details of implementation to cope with Greeks in binomial tree is covered in Section 5.2.

Noting that the content of this section is mainly based on Hull’s book [15]

2.4 The Trinomial Tree Model

The trinomial model is a variant of the finite difference method [15]. It can be considered as an alternative to the binomial model. The trinomial model is initially proposed by Boyle [4] and later proved by Brennan and Schwartz [5] to
be equivalent to the explicit finite difference method [15], another method for American option evaluation. The trinomial model extends the binomial model, by allowing the price to increase or decrease as before, but by also allowing the price to stay the same. The main advantage of a trinomial tree is that it provides an extra level of freedom, making it easier for the tree to represent features of the interest rate process such as mean reversion [15]. This extra level of freedom is very useful for modelling interest rate derivatives such as bond options [17].

Generally a trinomial tree will have more nodes than a binomial tree with the same number of steps; therefore it is considered more accurate and will give the same result as a binomial model in fewer number of steps [24]. Typically, an N step binomial tree has \((N+1)(N+2)/2\) nodes whereas an N step trinomial tree has \((N+1)^2\) nodes.

A three-step trinomial tree is shown on the left-hand side of Figure 2.3 with time along the horizontal axis, and asset price along the vertical axis. On the right-hand side of Figure 2.3, \(p\) and \(q\) indicate the probability that the asset price will go up and go down respectively, and \(m\) is the probability for the asset price to remain unchanged. Given \(p\) and \(q\), the expected option price can be calculated.

The trinomial pricing model for American put option can be described as:

\[
v_{T,i} = \max(K - S_{T,i}, 0) \tag{2.13}
\]

\[
v_{t,i} = \max(K - S_{t,i}, r(pv_{t+1,i+1} + mv_{t+1,i} + qv_{t+1,i-1})) \tag{2.14}
\]

\[
S_{t,i} = \begin{cases} 
S_0u^i, & \text{if } i > 0 \\
S_0, & \text{if } i = 0 \\
S_0d^i, & \text{otherwise}
\end{cases} \tag{2.15}
\]

It can be observed that Equation 2.14 requires much more computations than Equation 2.2 however we are able to implement the trinomial model in a similar way to the binomial one by iterating over an array with nested for-loops.

### 2.5 FPGA and HyperStreams

One of the main objectives of this project is to investigate whether the tree-based model is suitable for FPGA acceleration and explore possible area/speed/accuracy trade-offs.
FPGAs also have programmable interconnects. One can programme Logic blocks and make them perform the function of basic logic gates such as AND, and XOR, or more complex combinational functions like decoders and simple mathematical functions. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory \[30\]. There are two main families of FPGAs (Virtex and Stratix) from two major manufacturers: Xilinx and Altera. The latest Virtex 5 and Stratix III FPGAs adopt 65nm technology, which gain significant performance improvement over the previous models, for example Virtex 4 and Stratix II which are based on 90 nm technology. Hardware description languages like VHDL and *Handel-C* can be used to programme FPGA hardware.

The FPGA platform available for me is a RCHTX high performance computing (HPC) board (Figure 2.4). RCHTX is based on a Virtex 4 xc4vlx160 FPGA, which has 67,584 slices and 24MB QDR SRAM on board. The host is a HP Proliant DL145 G3 Server running 64bit Redhat4 Linux operation system.

### 2.5.2 Reconfigurable Computing

Having mentioned FPGAs I can not avoid mentioning reconfigurable computing. Reconfigurable computing has become an active study area for a long time, the aim is to combine some of the flexibility of software with the high performance of hardware \[34\] by programme hardware fabrics like FPGAs using Hardware description languages. Many studies have been carried out to identify the design methodology and trend of reconfigurable computing. In particular: A survey study have been carried out to explore modern reconfigurable
system architectures and design methods [28]. Another study is presented to explore both hardware and software aspects for reconfigurable computing machines from single chip architectures to multi-chip systems together with runtime configurations [9].

2.5.3 HyperStreams

HyperStreams is a high-level abstraction based on the Handel-C language. It supports automatic optimization of operator latency at compile time to produce a fully-pipelined hardware implementation. This feature is useful for rapid development of complex algorithmic implementations on FPGAs. In addition, HyperStreams also provides means to connect to FPGA resources such as block RAMs.

HyperStreams is still a novel technique in the reconfigurable computing area; however it has already been applied to financial computations such as European option pricing using Monte Carlo method [20]. Experiments has shown that HyperStreams is very useful for fast prototyping. I have verified in this project that HyperStreams can produce better result than a straight forward pipelined floating point implementation in Handel-C. Details can be found in Chapter 4.4

In this project, HyperStreams is used to implement the Evaluation Core (the computational part) and Handel-C is used to implement control logic.

2.5.4 The DSM Library

The DSM library is used in this project to handle hardware-software communication. DSM stands for Data Stream Manager, it provides easy and portable means for hardware and software communication by provide independent, unidirectional data streams between hardware and software [6]. A DSM stream, or DSM channel can typically provide a bandwidth of 300MBps. Sending and receiving data from DSM streams are based on simple DSMWrite() and DSMRead() calls. DSM reads and DSM writes on hardware-side are circuit based hence do not suffer from any delay, however the DSM calls on software-side involve a number of library calls to check the low-level hardware handle status hence will incur some overhead.

2.6 GPU and CUDA

In this project, GPU is used as an alternative approach to contrast the performance of the FPGAs solutions. Two GPUs from different generations are used, one is an nVidia Geforce 7900GTX, with 512MB of on board RAM; the second one is a Geforce 8600GTS with 256MB of on board RAM, which supports the latest CUDA technology.

2.6.1 Computing in GPU

A recent trend has rise to deploy the enormous computational power that GPU have to treat computational intensive problems. This is referred to as "General-purpose computing on graphics processing units" (GPGPU). The addition of programmable stages and higher precision arithmetic to the rendering pipelines
in modern GPU has allowed software developers to use GPUs for non graphics related applications. By exploiting GPUs’ extremely parallel architecture using stream processing approaches many real-time computing problems can be sped up considerably.

Despite the power consumption and the need for cooling device, GPU based solutions can be considered as the main competitor of FPGA based solutions in financial computation. Elder generation GPUs like Geforce 7900GTX has mainly three types of processors, Vertex processors, Texture and Fragment Processors and Z-compare and Blend processors. Together they form the graphics pipeline which is able to map pixels on screen based on a list of geometric primitives. This approach is mentioned as parallelism in space, as the data is fed directly into next stage after finished processing in the previous stage. Among which the Vertex processors and Texture and Fragment Processors are the part that has the main computational power hence the conventional GPGPU program usually deploys them for parallel data processing. These two components can be programmed with user-specified programs to run on each vertex and fragment.

There are lots of APIs or libraries developed to make use of the programmable components in GPUs; GLSL (OpenGL Shading Language) is the one that I used to implement the binomial tree model on Geforce 7900GTX. An example of study based on the general GPU model is to accelerate a C++ image processing library with a GPU. Within which a source-to-source parser is used to analyse and translate the C++ source code so that the inherent parallelism in the complex C++ algorithm is detected and exposed. The confirmed parallelisable loops are then translated to equivalent code for the GPU, based on GLSL language.

However this model have a major disadvantage of load balancing, for example, If the vertex program is more complex than fragment program, overall throughput will dependent on the performance of the vertex program. The unified shader architecture seeks to overcome this problem. In the unified shader architecture all programmable units in the pipeline share a single programmable hardware unit. As the programmable parts of the pipeline are responsible for more and more computation within the graphics pipeline, the architecture of the GPU is migrating from a strict pipelined task-parallel architecture to one that is increasingly built around a single unified data-parallel programmable unit.

The unified shader architecture typically allows new generation GPGPU languages to emerge, such as AMD’s compute abstraction layer (CAL) and nVidia’s CUDA.

2.6.2 CUDA

The latest CUDA technology developed by nVidia has shed a light on GPGPU. It allows better memory control that is normally not allowed by old GPGPU means and in addition, better level of parallelism. The most marvelous contribution of CUDA might be that it no longer requires the data to be processed to be sealed into an image anymore. The CUDA technology is supported by nVidia’s latest GeForce 8800 class GPUs like GeForce 8600GTS. The latest generations of GPUs use different architecture. It adopts stream scalar processors instead of 4-vector processors like GeForce 7900GTX. The GeForce 8600GTS GPU has 32
stream processors while Geforce 7900GTX has 24 vector processors. I use both of the GPUs to implement the tree-based models. CUDA introduced the concept \textit{Multiprocessor}, which is essentially a 32-vector processor based on several stream processors. Geforce 8600GTS has 16 \textit{Multiprocessors}. There are some new terms in CUDA’s thread batching scheme, listed below:

- **Thread** is extremely light-weighted comparing with the treads on PCs.

- **Warp** A warp consist of 32 threads and can be viewed as 32 way SIMD instructions. The \textit{Multiprocessor} is able to process a “half-warp” in one clock cycle, hence it takes 2 clock cycles to process a single warp instruction.

- **Thread Block** is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses \cite{21}. All the threads in the same thread block will be processed by the same \textit{Multiprocessor} and is processed in warp-by-warp manner.

- **Grid of Thread Blocks**: is by definition a grid consists many thread blocks. A device may run all the blocks of a grid sequentially if it has very few parallel capabilities, or in parallel if it has a lot of parallel capabilities, or usually a combination of both \cite{21}.

Figure \ref{fig:thread-batching} shows the relationship between Threads, and Thread Blocks and Grids. I use CUDA to implement the binomial tree model and the trinomial tree model on Geforce 8600GTS.
### 2.7 Comparing FPGA and GPU

The comparison of FPGA and GPU for different applications has been addressed more and more often recently. While FPGAs allow flexible reconfigurability, GPUs convey maximum parallel computational power. In the interest of study what comparisons have been made, what applications are covered and which devices are tested I carried out a little survey and the results are listed in Table 2.1. It can been seen that most of the works are based on computing intensive works such as video processing, data encryption and financial applications. The implementations are mainly based on Xilinx Virtex FPGAs and nVidia GPUs. In some testing cases other devices such as Cell BE are involved in the comparison, they are excluded from the table as they are not considered within the current project scope. However future works can be carried out to involve the comparison to Cell BEs.

### 2.8 Summary

Hardware assisted accelerations for financial computations have been popular for a while. However previous studies have been focused on Monte-Carlo methods and almost no attention have been paid to the tree-based models, why? Hardware constraint can be one of the reasons. Earlier generation FPGAs have relatively limited on-chip resources and is not suitable for the models that involve complex calculations, heavy control logic and high memory requirement; like tree-based models. The emerge of new generation FPGAs have changed the situation.

On the other hand, there have been many ongoing research of GPU based accelerations as well, like accelerating C++ image processing library with GPU [12]. Some attention have been paid for the binomial tree models since the emerge of CUDA technology [24]. However the study is relatively shallow and only for simple European style options. In Chapter 6 I propose a more sophisticated way to price American style options using tree-based models.

It is worth nothing that apart from FPGA and GPU implementations, other parallel software solutions are also being developed. For example, a library is being developed to provide metadata to characterise data accesses, dependence constraints and allow aggressive inter-component loop fusion to be supported in

<table>
<thead>
<tr>
<th>Application</th>
<th>FPGA</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matched Filter [1]</td>
<td>Xilinx Virtex 2 Pro</td>
<td>NVIDIA GeForce 7900 GTX</td>
</tr>
<tr>
<td>Gaussian Elimination [8]</td>
<td>Xilinx Virtex 2 Pro</td>
<td>NVIDIA GeForce 8800 GTX</td>
</tr>
<tr>
<td>Data Encryption Standard [8]</td>
<td>Xilinx Virtex 2 Pro</td>
<td>NVIDIA GeForce 8800 GTX</td>
</tr>
<tr>
<td>Needleman-Wunse [8]</td>
<td>Xilinx Virtex 2 Pro</td>
<td>NVIDIA GeForce 8800 GTX</td>
</tr>
<tr>
<td>Map-reduce Programming Model [17]</td>
<td>Xilinx Virtex 2 Pro</td>
<td>NVIDIA GeForce 8800 GTX</td>
</tr>
<tr>
<td>Video Processing Algorithms [10]</td>
<td>Xilinx Virtex 2 Pro</td>
<td>NVIDIA GeForce 6800 GTX</td>
</tr>
<tr>
<td></td>
<td>Xilinx Spartan 3</td>
<td>NVIDIA GeForce 6600 GT</td>
</tr>
<tr>
<td>Monte Carlo simulation [20]</td>
<td>Xilinx Virtex 4 LX160</td>
<td>NVIDIA GeForce 7900 GTX</td>
</tr>
<tr>
<td></td>
<td>Xilinx Virtex 4 SX55</td>
<td>NVIDIA GeForce 7900 GTX</td>
</tr>
</tbody>
</table>

Table 2.1: A list of FPGA and GPU comparisons.
a representation generated at runtime. Such implementation, if run on multi-
core PCs, can generally provide 3 times to 4 times speed-ups with less memory
consumption, compared to the un-optimised version [13]. These methods can
also be adopted to accelerate tree-based models if satisfactory result can be
achieved.

In next chapter I describe the general methodology I used to implement
tree-based models in software, hardware and GPUs.
Chapter 3

Design Methodology

Mapping a tree-based model to FPGA will involve a number of complex tasks that require careful planning. The architectural design to implement the model is crucial to the success of this project. The exploration nature of this task requires a robust design that is both efficient and extensible.

In this chapter I propose a software model, a hardware architecture and a GPU approach based on properties of the tree-based models. In particular I will cover:

- The properties of the tree-based models that can be used to save memory usage and achieve parallelism; in Section 3.1
- Software implementations of such models, which are used as blueprints for both hardware implementations and GPU approaches. The software implementations also run on reference PCs as benchmarks for the hardware and GPU approaches; in Section 3.2
- The proposed hardware architecture to map tree based hardware to FP-GAs; how C-Slow approach can be deployed in the architecture and how replications can be done in hardware to achieve parallelism; in Section 3.3
- The central design methodology I use for the GPU implementations, described in more details later in Chapter [1] in Section 3.4

3.1 Properties of the Tree Model

A straight forward mapping of the tree model to hardware is easy, however finding an efficient way is not so straight forward. Therefore I start with analyse both the binomial and trinomial model and seek to exploit the models’ inherent parallelism.

Option pricing using tree-based models can be viewed as a two phase process:

- Phase 1: Walk forward to construct a tree of underlying asset prices.
- Phase 2: Walk backward to calculate and trace the option price based on the underlying asset prices in the tree.
Phase 1 and be done together with Phase 2 as the underlying asset price can be calculated on the fly based on the position of the node in the tree (as shown in Equations 2.3 and 2.15). Figure 3.1 shows the dependency relationships in a trinomial tree model.

If \( N \) is the total number of steps (the tree depth), \((N + 1)(N + 2)/2\) nodes calculations will be needed for the binomial model and \((N + 1)^2\) nodes calculations for the trinomial model. We can see that the nodes belong to the step in the tree are independent and therefore can be calculated in parallel. We can also observe that the number of computations required at each step to calculate the option prices decreases linearly from leaves to the root of the tree. For example, for a binomial tree it takes order \( n \) computations to calculate the nodes at the last step (the leaf nodes). And we only need one computation to calculate the option price at the root. This property means that if I try to achieve full parallelism in the tree, I will waste half of the computational power as it gradually reach the root node. For example, if I have a binomial tree like in the left-hand side of Figure 2.1, at step 3 I will need 4 evaluation units to calculate the nodes in parallel, but at the root node I’ll only need one; 3 evaluation units will be idle in this case. The situation is similar for step 1 and 2. A trinomial tree will have the same problem. This problem is discussed further in Section 5.5.

We can also observe that the nodes at step \( n \) is used only once to calculate the option prices for the nodes at step \( n - 1 \). This means at any step \( n \) we only need to keep node values from the previous step \( n + 1 \). Therefor at most \( N + 1 \) nodes will need to be remembered for in binomial model and \( 2N + 1 \) nodes in the trinomial model (at the leaf of the trees), where \( N \) is the total number of steps. This property is deployed in the software approach in next section.

3.2 A Software Approach

It is good practise to have a clear mind of the programming model before I start mapping the model on hardware. So I started with a software approach. The existing implementations of tree-based models tends to be tedious and inefficient in memory usage. For example, a binomial tree with \( N \) steps will typically need...
\[ S_0^N \quad S_0^{(N-1)} \quad \ldots \quad S_0 \quad \ldots \quad S_0^{(N-1)} \quad S_0^N \]

Table 3.1: The asset price lookup table.

a \((N+1)\times(N+1)\) array to store all the intermediate option values in the tree. However only half of the array elements will be utilised as a binomial tree only have \((N+1)(N+2)/2\) nodes; memory is not utilised efficiently this way. The trinomial tree implementation can also exhibit this problem. To enforce efficient memory use, a single array of \(N+1\) elements is used, and nested for-loops are applied around the array, with an outer loop stepping backwards from \(N\) to 0, and an inner loop calculating the option price for each node in the tree. Note that the underlying The new option values constantly overwrite the existing ones once they are not needed.

A binomial example is shown in Figure 3.2. The first for-loop is used to calculate the option prices at the leaf nodes; and the second nested for-loop is used to traverse with in the tree. Array \(c\) stores temporary option prices, variable \(strike\) is the strike price for the option, \(discount\) is the discount value to cope with the effect of interest rate in one time step, \(pd, pu\) are probabilities for the underlying asset price to go up, or go down respectively. \(s\) is the Asset Price Lookup Table, which will be mentioned later. The Step function essentially calculates Equation 2.2 with appropriate input.

A trinomial example is shown in Figure 3.3. The code structure is almost the same as the binomial model. \(pd, pm, pu\) are probabilities for the underlying asset price to go up, stay the same or go down respectively. The Step function essentially calculates Equation 2.14 with appropriate input.

It can also be observed in the left-hand side of Figure 2.1 that the same underlying asset value can appear multiple times in the tree (as they appear on the same dashed line). For example, asset price \(S_0d\) appears twice in the tree. To avoid recreative calculations of asset price, a lookup table, which appears in Figure 3.3 as array \(s\), is used to store all possible asset prices in the tree. The lookup table is organised in the way shown in Table 3.1 noting \(S_0\) is the price of the underlying asset at time 0, and \(N\) is the total number of steps (tree depth); the lookup table is the same for the binomial model and the trinomial model.

The software models are implemented with C++ and are later used as benchmarks running on PC. The hardware mappings are mainly based on the software implementations with hardware level optimisation.

3.3 The Hardware Approach

The high level view of the hardware approach for tree-based option valuation is straightforward and can be seen as a three-step procedure:

1. The software side send request to hardware with appropriate parameters.
2. The hardware side calculates the result based on the parameters and send it back.
3. The software side collects the results.
double Step(double discount, double strike, double pd, double pu, double upOptionValue, double downOptionValue, int currentAssetPrice)
{
    return c[j] =
        max(discount*(pd*downOptionValue + pu*upOptionValue), strike - currentAssetPrice);
}

//calculate values for leaf nodes
for(int i = -N; i<= N; i++)
    c[i+N] = max((strike-s[i+N]),0.0);

//Iterate within the tree
for(int i = N; i>0; i--){
    int baseoffset = 1-N;
    for(int j = 0; j<i; j++){
        assetPriceOffset = baseoffset+2*j;
        c[j] = Step(discount, strike, pd, pu, c[j], c[j+1],
                    s[assetPriceOffset]);
    }
}
double Step(double discount,
          double strike,
          double pd, double pm, double pu,
          double upOptionValue,
          double midOptionValue,
          double downOptionValue,
          int currentAssetPrice
    ){
    return c[j] =
        max(discount*(pd*downOptionValue +
                      pm*midOptionValue +
                      pu*upOptionValue),
            strike - currentAssetPrice);
}
//calculate values for leaf nodes
for(int i = -N; i<= N; i++){  
    c[i+N] = max((strike-s[i+N]),0.0);
}
//Iterate within the tree
for(int i = N; i>0; i--){  
    int baseoffset = N -i +1;
    for(int j = 0;j<=2*(i-1);j++){  
        assetPriceOffset = baseoffset+j;
        c[j] = Step(discount, strike,
                    pd, pm, pu, c[j], c[j+1], c[j+2],
                    s[assetPriceOffset]);
    }
}
The software side is relatively simple since the logic only involves send and receive. However the hardware side requires carefully design to achieve both speed and efficiency.

### 3.3.1 Binomial Tree

In mapping the binomial model described in Section 2.2 into hardware, two central assumptions are made:

- The trees use a non-trivial number of time-steps, so the amount of I/O per tree is small compared to the number of nodes that must be evaluated. The number of parameters needed for transfer is of order $x$, where $x$ is the number of time-steps; this overhead is insignificant when compared with the number of computations, which is of the order the quadratic of $x$. In our case I/O can be pipelined to take place concurrently with computation, hence further reducing the overhead. A further improvement is to compute the Lookup Table on the fly so that we can process more trees in a batch, reducing the effect of start-up overheads of I/O. This is discussed in more detail in the trinomial example.

- Requests for option valuations are received concurrently, so many individual trees can be valued in parallel.

The first assumption means that we only need to consider evaluation when it is computationally bound, so we can largely ignore the performance of any software to hardware communications channels. The second assumption allows us to use high-latency pipelined functional units to achieve high clock rates while still achieving high throughput, by using the C-Slow approach [29].

Figure 3.4 shows the architecture for mapping the binomial tree model into hardware. On the left is a bank of parameter sets, each of which describes a binomial tree which is currently in the process of being evaluated. In the center is a large pipelined block which takes two previously calculated option values and calculates the value of the parent node, this is referred later as the Evaluation.
To manage temporary storage, a set of buffers (shown to the right) are used; ideally they should be FIFO stream buffers which hold the option values until they are needed again.

In the project I use block RAMs to implement a lookup table for $S_{t,i}$ (see Equation 2.2), which is initialized at the beginning of each tree-evaluation run, to get around expensive exponential calculations in hardware.

### 3.3.2 Trinomial Tree

The trinomial tree model in hardware shares the same main assumptions as the binomial model. It differs from the binomial model in the following aspects:

- the trinomial model is more computationally intensive as the number of computations compared to the binomial model is double.
- it requires one extra multiplication and one extra addition within each step.
- it requires twice the memory space to store intermediate values.

The proposed architecture for mapping the trinomial tree model into hardware is similar to what is shown in Figure 3.4 except that the control logic and the Evaluation Core will need to be re-designed. More details to design the control logic can be found in Section 5.1, and more details of the Evaluation Core design can be found in Chapter 4.

### 3.3.3 C-Slow

The concept of C-Slow approach is first proposed by N. Weaver, Y. Markovskiy, Y. Patel and J. Wawrzynek in year 2003 [29]. The essence of C-Slow is to interleave data streams from different independent computations, we continuously feeding the interleaved stream into the pipeline while continuously getting result from the other end.

The idea is illustrated in Figure 3.5: the trivial approach would push one data to be processed into the pipeline, and wait for the result to come out. The problem is if the pipeline has $N$ stages, we typically need to wait $N$ clock cycles to get the result. The pipeline is not fully utilised while we are waiting, as only one stage of the pipeline will be doing useful work at any clock cycle. If C-Slow method is adopted, all the stages in the pipeline will be utilised at any clock cycle and the pipeline is able to produce one result per clock cycle. The throughput of the pipeline is increased by a factor of $N$. The only possible delay incurs is when the first data passes through the pipeline where there is no data in front of it to be processed.

C-Slow operation can be achieved by modeling multiple tree nodes in parallel: we continuously provide parameters into the pipeline to evaluate other tree nodes while we are waiting for the results required for the next iteration of the current tree. The stream buffers are carefully designed for this approach. A controller manages the overall timing of the system, ensuring that the intermediate values are stored and retrieved correctly, and that the correct parameter set is selected on each cycle.
3.3.4 Parallel Replications

To achieve parallelism in hardware we need to replicate the corresponding logic so that multiple data can be processed at the same time. For the tree models in general I consider two levels of parallelism:

- With in each tree, the Evaluation Cores can be replicated to accelerate the valuation procedure for a single option. Using this approach multiple nodes at the same step will be valuated in parallel, which essentially reduces the time required for valuation.

- The whole tree valuation logic, namely the proposed architecture shown in Figure 3.4 can be replicated so that multiple trees will be valuated simultaneously.

The main idea is demonstrated in Figure 3.5. The number of replications of Evaluation Cores within a tree and the number of replications of the tree valuation logic should be determined according to the use case. If the number of steps of the trees to be valuated is generally large, the multiple Evaluation Core approach can be adopted to achieve higher speed-up. On the other hand, if a large number of options are being evaluated at the same time then the multiple tree valuation method should be adopted to allow many trees to be evaluated at the same time. If both of the two cases are true then we’ll probably need to have multiple tree valuation logics with multiple Evaluation Cores. The on-chip resources available on an FPGA should certainly be considered in first place. Details of how the parallel replications are done can be found in Chapter 5.

3.4 The GPU Approach

Unlike FPGAs which provides flexibility for reconfiguration, GPUs provide maximum parallelism. Therefore to explore the inherent parallelism in the tree models becomes the prime consideration. I mentioned in Section 3.3.1 that the inherent parallelism in the tree can be exploited by processing the nodes at the same step in the same tree in parallel. However this will not be efficient as the size of the tree to be evaluated eventually shrinks to one, the utilisation of GPU resources will reduce as the tree shrinks as the number of nodes to be processed...
at each step reduces. New technologies like CUDA from GeForce may have instruction level optimisation to solve this problem. The "gap" can be filled with computations to valuate other trees. Parallel processing of nodes at the same step exhibits one problem: as we only use a one-dimensional array to store temporary option values, old data is overwritten when new result is produced. If the procedure is sequential we can be sure that the data being overwritten is not needed any more. However it is not the case when the array is processed in parallel. In a parallel processing procedure, data can be randomly overwritten in the array and possible data loss may occur. Double buffering can be used to overcome this problem. Essentially, two arrays (A and B) instead of one are used, array A is initialised in the beginning. In the first iteration data is read from array A and the result is written to array B, and in the next iteration data is read from array B and result is written to array A, etc. Using this approach we can avoid data loss while processing the tree model in parallel.

Devices like GPUs usually do not have large-size high speed caches. If the tree to be valuated is large we will not be able to fit the whole tree in caches. However we are able to can cut the tree into small trunks to fit the trunks into the cache. Then we view each trunk as a smaller tree and process the trunks separately. This is referred as tree cutting. Tree cutting allows high speed caches to be used with the price of processing redundant nodes in the tree.

More details about the GPU approach is implemented can be found in Chapter 6.

3.5 Summary

In this Chapter I analysed the properties of the tree-based models, listed as following:

- A binomial tree has \((N + 1)(N + 2)/2\) nodes and a trinomial tree has \((N + 1)^2\) nodes, where \(N\) is the number of steps (tree depth).
- The value of nodes at the same step can be calculated in parallel, in expense of lower utilisation of hardware. However for devices like GPUs which are designed to achieve massive parallelism, this property should be deployed for full utilisation of the device.
• The tree evaluation process can be viewed as a nested for-loop iterating over a one-dimensional array of size $x$; where $x$ is $N + 1$ for binomial trees and $2N + 1$ for trinomial trees, this property is applied to both the FPGA and GPU approaches.

First of all I implement an efficient pure software implementation. Based on the software version I suggest a fully pipelined hardware architecture for binomial and trinomial models. C-Slow method is adopted and multiple tree nodes can be evaluated in parallel. High throughput can be expected using this approach despite that the absolute pipeline delay might be high.

In the GPU approach I suggest that double-buffering to be used to overcome the problem caused by parallel processing; and tree cutting to be used to utilise the high speed caches on GPU that are relatively small in size.

In next chapter I explain the design and implementation of the Evaluation Core in Figure 3.4.
Chapter 4

Design and Implement The Step Function

So far the Evaluation Core would be the largest component in the hardware architecture. The Evaluation Core calculates the next node value as specified by Equation 2.2. In the asymptotic case I would expect the overall performance of the hardware implementation to be dominated by the size and speed of this block, as the other components consist of some memory blocks and selection logic. In this chapter I present my design and implementation of the Evaluation Core in detail. The following topics are covered:

- A straightforward pipeline design and an improved pipeline design that saves a multiplier and an adder, for the binomial Evaluation Core; in Section 4.1

- A fully pipelined design for the trinomial Evaluation Core with Table Generator reduce hardware-software communication; in Section 4.2

- A list of real number representations supported in hardware, and the pros and cons if my implementations are based on such representations; in Section 4.3

- An experiment is done to compare the HyperStreams library with the pipelined floating point library, based on maximum clock frequency achieved and amount of on-chip resources occupied; in Section 4.4

- Two fully pipelined HyperStreams implementations for both the binomial model and the trinomial model are illustrated; in Section 4.5

4.1 The Binomial Valuation Core: A Naive and An Improved Approach

Figure 4.1(a) illustrates a straightforward hardware implementation of the core evaluation pipeline. Two adders and three multipliers are required to implement Equation 2.2. If float or double data types are used in the implementation,
Figure 4.1: Binomial Model: hardware design for the block Calculate Node Value in Fig. 3.4. The solid black boxes denote registers and the dotted grey boxes denote pipeline balancing registers.

Multipliers can occupy significant amount of on-chip resources. The design can be improved if we re-arrange Equation 2.2 to:

\[
v_{t,i} = \max(K - S_{t,i}, rp_{t+1,i+1} + r(1 - p)v_{t+1,i-1})
\]  

(4.1)

where \( rp \) and \( r(1 - p) \) are calculated first then multiplied to \( v_{t+1,i+1} \) and \( v_{t+1,i-1} \) respectively, recall that \( p \) is noted as noted as \( p_u \) in Figure 4.1(a) which stands for the probability for the underlying asset price to go up, and \( 1 - p \) is noted as \( p_d \) which stands for the probability for the underlying asset price to go down. Using this method it is then possible to use one fewer multiplier if, instead of feeding in \( p_u, p_d \) and \( p \) separately, \( rp_u \) and \( rp_d \) are used as two inputs. The last multiplier can be omitted as the discount factor \( r \) is taken into account in the first two multiplications. \( rp_u \) and \( rp_d \) can be transferred directly from software.

Figure 4.1(b) shows an example of the improved hardware design of the Evaluation Core. For each tree it evaluates, it takes in a set of parameters that are provided by the controller from the tree parameters table in Figure 4.1(b). To optimise performance, a lookup table is initialised with all possible asset strike prices. The architecture takes from the stream buffers three parameters: the two previous tree node values, and \( i \), the price offset. Using the price offset \( i \) the current strike price \( S_{t,i} \) can be retrieved from the lookup table. I test the method by mapping the improved Evaluation Core to a Virtex 4 xc4vsx55 device, around 6% of total slices are saved if double precision operators are used.

With all the parameters ready, the Algorithmic Core in Figure 4.1 computes the option price \( v_{t,i} \) for the current tree node. The result is then sent back to the stream buffers for later use. The C-Slow method can be implemented here if:

- The outside controller is able to provide a set of correct parameters per clock cycle.
- The lookup tables are correctly initialised.
- The controller is able to store the result into the correct buffer.
4.2 The Trinomial Valuation Core

Figure 4.2 shows a hardware implementation of Equation 2.14. One more adder and one more multiplier are used when compared with the design in Figure 4.1(b). However, the pipeline depth only increases by one adder; it is therefore expected that there will be some rise in resource requirement and little increase in pipeline delay.

The box above the Asset Price Lookup Table in Figure 4.2 shows the logic to generate the Lookup Table on the fly. By using an extra multiplier, we are able to avoid using expensive exponential operators in hardware. The idea is to start from \( S_0 \) in the middle of the Lookup Table and accumulatively multiply \( u \) to it; write the result to the Lookup Table until we reach one end of the price table. Then do the same to the other half of the Lookup Table. If the memory is dual-ported, the two procedures can be done simultaneously. This approach allows us to cache only tree parameters instead of caching large lookup tables, hence allows us to transfer trees in a batch by batch manner from software. The tree parameters in the cache can be fetched by the control logic to generate lookup tables for later use. This reduces the communication overhead further. The Table Generator runs in parallel with the core evaluation logic to reduce the generate overhead. Extra memory cache is needed to store the generated lookup tables. A detailed discussion about whether the Table Generator will really work can be found in Section 5.3.
4.3 Real Numbers in Hardware

Having the design in front of me the problem now is how to implement it on hardware. In this project the step function is the only place that requires real number calculations. Unlike other applications on hardware, financial based hardware acceleration requires the result to be very accurate. Generally double precision point numbers are used in the applications in industry, but is double precision really necessary?

Before start the actual implementation I considered four possibilities:

- Use double precision in hardware. This has the advantage of being very accurate and make the hardware accelerated version identical to the original software version, allows easy portability with no potential side-effects. The only choice to for implementation is to use the HyperStreams library which supports double precision operators on FPGA. However it can be expected that the double precision operators will occupy a lot of on-chip resources and will be slower than floating point and fixed point operators.

- Adopt single precision floating point algorithmic on hardware, using the Handel-C pipelined floating point library. By Adjusting the software side accordingly to cope with single floating point numbers. As single precision numbers only have 32 bits instead of 64 bits, we are able to save half of the storage space in RAM for temporary values and the asset price lookup table. The communication between hardware and software will potentially be halved. More importantly, single precision operators runs faster than double precision operators.

- Using the HyperStreams floating point library. This will have the same advantage as the previous one, in addition, the HyperStreams floating point version can be easily converted to a double precision version or fixed point version.

- Implement a fixed point version, using either the Handel-C fixed point library or the HyperStreams library. Using this approach I can either convert the representation to fixed point in software, then transfer it to hardware or convert the floating point numbers to fixed point in hardware on the fly. Generally speaking the fixed point operators runs the fastest and requires least on-chip resources in hardware. Although fixed point numbers may exhibit some accuracy problems, the problem will not be significant as the prices of options tends to lie in a small range (from 0 to 1000).

All four options above are viable so I decide to implement all of them. However HyperStreams is a new library which has barely been used in practice, it is worth checking its performance by comparing it to an existing library. In particular, the comparison is based matrices such as resource usage and execution time. In next section I examine the quality of the HyperStreams library.
4.4 HyperStreams Vs Pipelined Floating Point Library

The HyperStreams library is very easy to use, for it provides automatic pipeline balancing feature therefore user do not need to cope with pipeline delays in the control logic. However the performance of the HyperStreams library is yet to be determined. In this section I compare a HyperStreams implementation with a straight forward pure Handel-C pipelined floating point implementation.

The implementation is based on the naive design described in Section 4.1, which uses two adders, three multipliers and a comparator. All the operators are based on 8-24 single precision floating point numbers. The two implementations are synthesised to EDIF using the Celoxica DK5 and Xilinx ISE 9.2i to place and route the design to the target device which is a xc4vsx55 FPGA. The result show that the HyperStreams version used 3,805 slices and can achieve a maximum clock frequency of 76MHz, and the pure Handel-C version used 4,574 slices and can achieve a maximum clock frequency of 68MHz. The results show that a pure Handel-C version uses 702 more slices and is 10% slower than the HyperStreams version. The saving of slices comes from the reuse of pipelined multiplier (use one instead of three) and adder (use one instead of two). I expect the HyperStreams library to use more efficient floating point operators to gain higher clock frequency.

In the testing case HyperStreams outperform the floating point library in terms of both performance and ease of use. In addition, HyperStreams can handle different data types very well, code conversion between double precision, single precision and fixed point implementations are more straight forward. Therefore I decide to use the HyperStreams library for this project.

4.5 The HyperStreams implementation

My FPGA implementation of the node evaluation logic to support the tree-based option pricing model is based on HyperStreams and the Handel-C programming language.

Figure 4.3(a) shows a fully pipelined FPGA implementation of the node evaluation logic indicated in Equation 2.2, while Figure 4.3(b) shows the implementation of Equation 2.14.

Each symbol shown in a “HyperStreams” block in Figure 4.3(a) refers to a HyperStreams operator: for example, \(\oplus\) for HsAdd, [RAMRead] for HsRAMRead and so on. Each arrow from DSM (Data Stream Manager), the interface used for hardware-software communication, indicates a stream data element received as an unsigned integer. The inputs are cast to desired internal representation, for example HS_DOUBLE, at the top of the HyperStreams block. Once all the computations are finished, the output stream is then cast back to the desired output format using the HsCast operator. HyperStreams is a device independent library so the implementation can theoretically be targeted to any device easily.

The control logic, which is used to send and retrieve data from pipelines, is written in the Handel-C language and is discussed in detail next chapter.
4.6 Summary

In this chapter I describe the design and implementation of the key component in the hardware system: the Evaluation Core. Two different versions of Evaluation Cores are proposed, one of them is used to valuate the binomial model, the other is used to valuate the trinomial model. As this component contains complex algorithmic logic, it potentially become the bottle neck of the entire system. Both of the Evaluation Cores are carefully designed and optimised. The performance of the two cores are later examined in Section 7.1. After that possible implementations based on different real number representations are discussed and the pros and cons of each one are addressed. A comparison between the HyperStreams library and the pipelined floating point library is made. The result show that the HyperStreams library is is able to achieve a higher clock frequency with less on-chip resource occupancy. HyperStreams library is then used to implement my designs of the binomial and trinomial Evaluation Cores. The implementations are flexible so that they can adopt different real number representations easily. The HyperStreams library is device independent therefore the implementations can be targeted to different devices without any difficulty. In next chapter I utilise the Evaluation Cores to try to valuate a tree-based model in hardware efficiently.
Chapter 5

Tree Valuation in hardware

Having the Evaluation Core ready, I plan for a full implementation is hardware. The overall design is based on my proposal in Section 5.3. I do this in the following stages:

- A straight forward mapping to hardware from the software prototype is illustrated, the skeleton for a full implementation is outlined; in Section 5.1.

- The mapping is modified to cope with Greek valuations. The tasks are carefully delegated to both the hardware side and the software side so that almost no extra timing overhead is introduced; in Section 5.2.

- The question "whether the Asset Price Lookup Table should be transferred from software or generated in the hardware" is argued and numbers are listed to support my view; in Section 5.3.

- Optimisations are applied to reduce memory reads from three reads per iteration to only one read per iteration; the Evaluation Core is pipelined fully to achieve maximum throughput and utilisation throughout the option valuation procedure; in Section 5.4.

- Two possible routes to achieve parallelism in the design are discussed, their strengths and weaknesses are addressed; in Section 5.5.

- The target device – a RCHTX platform is introduced; in Section 5.6.

5.1 Design and Implement the control logic

I start with a straight forward mapping from software to Handel-C, by utilising the Evaluation Core from Section 4.1, based on HyperStreams. There are four key components related to the control logic, listed as the following:

- The Tree Array, a one dimensional array to hold temporary option prices. The size required for the array depends on the number of steps of a particular option. However we can expect the size to be large as we will valuate options with non-trivial number of steps. This component should be implemented in block RAMs as otherwise it will acquire too many slices on chip.
Figure 5.1: A straightforward map from the software version described in Section 3.2, noting CM is the Communication Module.

- A *Asset Price Lookup Table* this should be implemented in block RAMs as well, as in a binomial model the size of *Asset Price Lookup Table* will be twice the size of *Tree Array* and in trinomial case the same.

- A *Communication Module* (CM) to communicate to the software, get requests and send results back. This component is implemented using the DSM library, just because it’s the only way I am aware of by then.

- A central control module that determines how many iterations are left; feed in correct data into the *Evaluation Core*; and extract the result when it is ready. This part is implemented by pure *Handel-C*.

Figure 5.1 shows the work flow of the design. The software side send request to hardware via the communication interface. At the hardware side the *DSM Module* receives the request from software, set up parameters for the *Control Logic* and initialise the *Asset Price Lookup Table*. The control logic then use the *Evaluation Core* to initialise the *Tree Array* and use the *Tree Array* to calculate the result based on the parameters set by the *Communication Module* (CM). When the option valuation is finished the result is sent back to software via the communication Module.

It is worth noting that the central control module, unlike the software version which has three nested for-loops, the hardware implementation uses while-loops instead. As for-loops generally takes 2 clock cycles to execute the loop-head itself, therefore not efficient. A sample control logic for binomial tree is shown in Figure 5.2. Note that this version is not pipelined, an improved version can be found in Section 5.4.

With everything else being the same, the central control module is different for binomial and trinomial implementations. In particular:

- The trinomial version the inner-most loop need to iterate twice as much element over the array compared to the binomial version, this can be easily adjusted by the control variable counter2 in Figure 5.2.

- The trinomial version need three memory reads in each iteration, while binomial only need two. The number of memory reads can be reduced to one for both binomial and trinomial models with proper optimisation, more details to be discussed in Section 5.4.
At the end of this section I think it is worth addressing some lessons I have learned in implementing the control logic. It is intuitive to think as a fresher to *Handel-C* that the main control logic should be implemented by HyperStreams as well, since it provides conditional statements and loops in the pipeline. I find out later that the HyperStreams "control logic" is tedious and extremely hard to maintain; and more importantly, inefficient compared to pure *Handel-C*. It did take me quite some time to realise this.

5.2 Dealing with Greeks

This section seeks an efficient way to accelerate Greeks valuations. The definition of Greeks are listed in Section 2.3. The calculation of Greeks involves simple subtraction, multiplication or division, hence could generally be done in software. The problem is how to get the parameters to calculate them. I consider categorising Greeks into two categories:

- **Delta (Δ), theta (Θ) and gamma (Γ)**, which require the value of option prices at Step 1 and Step 2, in particular: \( f_{00}, f_{10}, f_{11}, f_{20}, f_{21}\) and \( f_{22} \); and corresponding underlying asset prices (\( S_0, S_0u, S_0d, S_0u^2\) and \( S_0d^2 \)).

- **vega(ν) and rho (ρ)**, which require the re-valuation of the entire tree with modified parameters, namely \( \sigma \) and \( r \).

For the Greeks in the first category, the corresponding underlying asset prices can be calculated in software easily with all the parameters ready. However the intermediate option prices need to be transferred from hardware, as other wise we need to recalculate the entire tree in software. A straight forward approach to get the intermediate option prices is to simply set some *if statements* in

```c
unsigned 10 counter;  
unsigned 10 counter2; 
...
counter = tree_size;  
while(counter>0){
    ...
    while(counter2 <counter){
        Read_from_Memory;  
        Call_Evaluation_Core;  
        Write_Result_to_Memory;  
        counter2++;  
    }
    ...
    counter--;  
}
```

Figure 5.2: The binomial central control logic.
the inner-most for-loop within the Control Module and extract the values when necessary. This can not be a choice as a if statement takes 1 clock cycle to execute, inserting several if statements in a nested for-loop will bring down the performance significantly.

A more efficient way is to:

- Make use of the useful information in Tree Array.
- Extract values in parallel with other statements in the innermost for-loop without if statements.

First look at the Tree Array after the tree valuation has finished, shown in Figure 5.3. Although most of the intermediate values in the Tree Array is overwritten in the consecutive step in the for-loop, one will be left over as the number of nodes we are dealing with decrements at each step. In my implementation \( f_{00}, f_{11} \) and \( f_{22} \) will not be overwritten and can be retrieved at the end of the tree valuation. \( f_{10}, f_{20} \) and \( f_{21} \) can be seen as the "down option price" in the last three valuations in the tree. For example, to calculate \( f_{11} \) we need \( f_{22} \) as the "up option price" and \( f_{21} \) as the "down option price"; to calculate \( f_{10} \) we need \( f_{21} \) as the "up option price" and \( f_{20} \) as the "down option price", etc. A three element array is used as a FIFO buffer to store the three most recent "down option price"s and make sure \( f_{10}, f_{20} \) and \( f_{21} \) are in the buffer after the tree valuation. No almost overhead is added in this way as only an additional assignment operator is needed, which can be done in parallel with other statements in the for-loop. All the intermediate option values are then sent back to software via the Communication Module.

The Greeks in the second category can be calculated without any hardware side modification. It just require the software to make two consecutive requests with different parameters. This procedure can be accelerated if two trees can be valued in parallel.

### 5.3 Possible Acceleration: Asset Price Lookup Table

The Asset Price Lookup Table is used to get around expensive exponential calculations in hardware. In Section 4.1 I build the Asset Price Lookup Table in software first and transfer it to hardware. In Section 4.2 I describe a way to calculate the Asset Price Lookup Table on the fly in hardware efficiently. Which way is better? In this section I try to answer this question. The main overhead in the hardware accelerated architecture I described in Section 5.1 is caused by the Communication Module in software (in my implementation the overhead is the DSM calls such as DSMRead and DSMWrite). Typically the software side need to communicate to hardware twice to valuate a single tree.
• Send the parameters over, noting all the parameters can be batched in a single array and send in once.

• Get the result back from hardware.

To send over the Asset Price Lookup Table, the software side need to generate the table first and append the table to the array to be sent. Recall the size of the Asset Price Lookup Table is \( N + 1 \) where \( N \) is the number of steps (tree depth). The worst case is when the hardware has a double precision Evaluation Core, in which case additional \( 4(N + 1) \) bytes will be sent to hardware. However this is negligible as it takes only \( 4(N + 1)/(3 \times 10^8) \) seconds to complete. The time to generate the Asset Price Lookup Table is also negligible on a modern computer.

On the other hand, if the table is generated by the hardware, it will typically use an extra multiplier and some control logic. If only a single tree is valued, with almost the same overhead from the software size and the time we need to generate the table on hardware, the time can be excessive. In the case when a large number of trees are valued concurrently, storing all the lookup tables on-chip is imply not feasible; and the hardware-software communication may become an overhead. Only under this condition can Table Generator be used to run concurrently with the tree valuation process reduce memory usage and communication time significantly. The problem is the tree valuation procedure is computationally bounded, whether the hardware-software communication will be an overhead need to be determined in the experiments. (This answer to this question is actually ”no”, according the Section 7.2.)

5.4 Reducing Memory Access and Pipelining the Evaluation Core

So far the implementation has been a straight forward mapping from the software version. In this section I seek possible ways to optimise the code by reducing memory access and pipelining the design. In my implementation the Tree Table is stored in single-ported synchronous on-chip block RAMs. A straight forward implementation will take two memory reads to feed in as inputs to the Evaluation Core for binomial trees. This is shown in Figure 5.4, which takes 2 clock cycles to finish as the block RAM allows only exclusive access at any clock cycle.

A improved version is shown in Figure 5.5. I exploit the memory access sequence and make use of the previous value read from memory to reduce the number of memory access by one. Namely, \( upOptionValueR \) is initialised together with \( i \) before entering the loop; and with in the loop I read a new value from the memory to \( upOptionValueR \) and simultaneously pass the old value of \( upOptionValueR \) to \( downOptionValueR \) so that the previous \( upOptionValueR \) becomes the new \( downOptionValueR \). The trinomial implementation can adopt the same method, Figure 5.6 shows a fraction of improved code from my trinomial implementation. We can see that before entering the loop the values for \( upOptionValueR \) and \( midOptionValueR \) are initialised and within each iteration in the loop we only read a new \( upOptionValueR \) and reuse the previous values of \( upOptionValueR \) and \( midOptionValueR \).
unsigned 10 i;
unsigned 10 j;
...
i = 0;
while(i...){
    par{j = i+1;
    downOptionValueR
        = Tree[i];}
    upOptionValueR
        = Tree[j];
...
i++;
}

Figure 5.4: Straight forward Code.

unsigned 10 i;
signal unsigned 10 j;
...
par{i=0;
upOptionValueR = Tree[0];}
while(i...){
    par{
        j=i+1;
        downOptionValueR =
            upOptionValueR;
        upOptionValueR =
            Tree[j];
    }
...
i++;
}

Figure 5.5: Improved memory access
code in control logic for binomial tree.

unsigned 10 i;
signal unsigned 10 j;
...
par{i=0;
upOptionValueR = Tree[0];}
midOptionValueR = Tree[1];
while(i...){
    par{
        j=i+2;
        downOptionValueR =
            midOptionValueR;
        midOptionValueR =
            upOptionValueR;
        upOptionValueR =
            Tree[j];
    }
...
i++;
}

Figure 5.6: Improved memory access code in control logic for Trinomial tree.
for(i..){
    while(on){
        par{
            b = RAM[i];
            HsWrite (&a, b);
            ...
        }
    }
    while(on){
        par{
            HsRead (&Out, &c);
            RAM[turn1][i] = c;
        }
        par{
            turn0 = turn1;
            turn1 = turn0;
        }
    }
}

Figure 5.7: Straight forward Code.

Figure 5.8: Pipelined Code.

In the straightforward implementation I feed in one set of inputs to the pipeline and read the result out in sequential order. This means only one set of data is processed in the pipeline at any clock cycle. To fully utilise the pipeline, double buffering is used to get around the FPGA memory access limitation, for it has lower memory bandwidth than GPUs and CPUs [11] and allows exclusive access at any clock cycle. I give a simple example to illustrate.

Figure 5.7 shows some code feeding values to a HyperStreams pipeline. It waits until the result to come out from the pipeline and then feed another value. The pipeline is not fully utilised as only one pipeline stage will be effectively working at any time. Figure 5.8 shows an improved approach. Instead of waiting for the result to come out, inputs are read from one memory cache and constantly fed into the pipeline; while results are write into another memory cache simultaneously. Once the evaluation is finished, the result is sent back to software via the DSM interface.

Another improvement to the current design is to make the DSM module running in concurrent with the evaluation process, so that we can receive other requests from the software side while waiting for the result of the current valuation. As I consider the problem to be computationally bounded, the DSM module will fill the Asset Price Lookup Table much faster than the evaluation process. Therefore I use one extra bank of memory as a buffer for the Asset Price Lookup Table to fully utilise the Evaluation Core at any clock cycle.

Figure 5.9 shows the design of a fully pipelined Tree Valuation Core in hardware.

The Tree Valuation Core is designed in a way that it is easily replicated, therefore can be extended to handle multiple tree valuations on a single FPGA easily. Next section I discuss possible ways to achieve parallelism.
5.5 Multiple Trees Vs Multiple Evaluation Cores

There are two ways to scale up the implementation and achieve parallelism in hardware: use multiple Evaluation Cores to achieve internal parallelism within a tree and use multiple Tree Valuation Cores to handle multiple trees in parallel. Tree Valuation Cores with single Evaluation Core allow most efficient utilisation of hardware, while multiple Evaluation Cores allow certain speedup but potentially waste on-chip resources by introducing additional control logic, increase memory accesses and can potentially waste one or more core at near-root steps.

Figure 5.10 shows a design to handle multiple trees by replicating Tree Valuation Cores with multiple Evaluation Cores. Each Tree Valuation Core has a DSM port that connects to the software. The software sends option valuation requests concurrently to Tree Valuation Cores and reads the result concurrently from the hardware.

However I consider multiple Evaluation Cores in a Tree Valuation Core not efficient with the current FPGA technology. The reasons are listed below:

- **Multiple Evaluation Cores** entail multiple memory access at a single clock cycle, this is not supported by the single-ported block memory I used. Memories like Duo-ported memory allows multiple access within a single clock cycle, but they are very resource expensive in nature and not suitable for large arrays like the Tree Array.

- **Multiple Evaluation Cores** require more complex control logic, which will bring down the performance significantly. For example, if three Evaluation Cores are used we need to check in the inner-most loop to make sure no
core reads any out-of-bound value in the array.

- Although tree can be partitioned with the method I used for GPU implementation in Section 6.2, and each partition can be assigned to a Evaluation Core, the logic itself is too complex to be mapped to FPGAs.

Therefore in the implementation I only replicated Tree Valuation Cores, and each Tree Valuation Core have exactly one Evaluation Core.

5.6 Running on RCHTX

Having the implementation ready and running successfully on the hardware simulator, it is essential to test it on a real FPGA. The FPGA device available to me is a RCHTX high performance computing (HPC) board with a Virtex 4 xc4vlx160 FPGA, it has 67,584 slices and 24MB QDR SRAM on board. The host is a HP Proliant DL145 G3 Server running 64bit Redhat4 Linux operation system. Figure 5.11 show the local bus layout of RCHTX. The RCHTX board supports the DSM interface so that my code can be ported to real hardware very easily.

The tool flow is as follows. Handel-C source code is syntheised to EDIF using the Celoxica DK5 suite which supports HyperStreams. Xilinx ISE 9.2i project navigator is used to place and route the design.

5.7 Summary

I use this chapter to explain how my hardware implementation is designed to cope with the tree valuation procedure. The straightforward mappings from software to hardware provide an overview of how to build the skeleton of implementation. The main components in the system is then addressed based on the straightforward mapping. The design is then extended to cope with Greek calculations, almost no additional overhead is introduced to the system by:
• Utilise the useful data in the Tree Array.

• Adopt a three element FIFO buffer.

Having established a complete design I try to optimise the design by discussing possible ways to initialise the Asset Price Lookup Table; to reduce memory access in the inner most for-loop and to pipelining the Evaluation Core. I consider all the approaches to be effective and is able to enhance the overall performance significantly. Having a optimised Tree Valuation Core ready I then seek to achieve parallelism by replicating the Tree Valuation Core and the Evaluation Core. However the replication of Evaluation Core within a Tree Valuation Core can potentially introduce large overheads therefore the approach is abandoned. Only Tree Valuation Core is replicated and within each Tree Valuation Core exists only one Evaluation Core.

In next chapter I present my designs based on GPUs.
Chapter 6

Tree Valuation in GPU

In this project the GPU implementations play an important role to contrast the FPGA implementations. The first GPU design is implemented on an nVidia GeForce 7900GTX device with 512MB of on board RAM. GeForce 7900GTX device is based on same 90nm technology as the Virtex 4 FPGA that I used in this project, therefore I consider them to be broadly comparable.

The second GPU design is based on nVidia’s new CUDA platform [21] and running on an GeForce 8600GTS with 256MB of on board RAM. The GPU is a Geforce 8800 class GPU which adopts 80nm technology. The reason I choose this device is to compare my design on FPGA with the latest GPU technology and identify the possible trend of FPGA and GPU design for tree-based models. The full comparison can be found in the evaluation chapter in Section 7.1.

I use this chapter to show in how the models are implemented on GPUs and how the inherent parallelisms within the tree models are exploited, in particular:

• A straightforward implementation based on the GLSL library on Geforce 7900GTX device that does not support CUDA; in Section 6.1
• A CUDA based implementation on Geforce 8600GTS which exploits inherent parallelism of the tree model; in Section 6.2

6.1 GLSL Implementation

As the GeForce 7900GTX device does not support CUDA technology, I use the GLSL (OpenGL Shading Language). GLSL library allow graphical cards to deal with data-based computations based on pixel-based computations. This implementation is relatively straightforward compared to my CUDA implementation. The main idea is to seal the data required into a large image in a pixel by pixel manner; copy it to the GPU memory and use the customised kernel function to process it. I give a example to implement the binomial model using GLSL. The image is organised in the way shown in Figure 6.1.

The left-hand side of Figure 6.1 shows a $N \times N$ pixel GLSL image that will be processed by GPU. each row in the image stores temporary values for a single option. The right-hand side of Figure 6.1 shows how a pixel is utilised as a 4-float vector. The initially all the elements are used to store the temporary
vec4 Step(float baseOffset, vec4 ivalue, vec4 temp1, vec4 temp2)
{
    vec4 ival = vec4(baseOffset, baseOffset, baseOffset, baseOffset) + 2.0*ivalue;
    vec4 earlyvalue = vec4(strike_price, strike_price, strike_price, strike_price) - assetprice(ival);
    vec4 optionvalue =
        discountfactor*(m_up*temp1 + m_down*temp2);
    return max(earlyvalue, optionvalue);
}

option values, and each pixel essentially contains information for 4 independent options, and two pixels are processed at a time.

The main kernel function is the step function shown in Figure 6.2 in vector version, listed in Figure 6.2. 4-vector operators are used to utilise the 4-vector pipelines in the GeForce 7900GTX GPU.

### 6.2 CUDA Implementation

CUDA platform have a very different memory model from the traditional GPU memory architecture. The CUDA memory model is shown in Figure 6.3. We can see from the figure that all the threads within the same block shares a shared memory and all the threads globally (they do not need to be in the same block) shares a global memory. The shared memory is much faster than the global memory but is relatively small in size. According to the CUDA manual, shared memory is around 100 times faster than the global memory. In my implementation, each block will process a single binomial tree. It is ideal if we are able to fit the whole tree into shared memory first and then process everything based on the fast shared memory. However in cases when the tree is
Figure 6.3: The CUDA memory model [21].

large, for example the size of the binomial tree is over 500 steps, we won’t be able to put the whole tree in shared memory. A straight forward approach can be simply not use shared memory and do reads and write directly to the global memory. However, this approach encounter two major problems:

- According to the CUDA manual [21], The access to global memory is 100 times slower than to shared memory, frequent access to global memory brings down the overall performance of the implementation significantly.

- The CUDA model can be viewed asymptotically as a Parallel Random Access Machine (PRAM) based on the SIMD (Single Instruction Multiple Data) model. The order of threads scheduling is non-trivial and hence can be viewed as random. Random access to the array can introduce error in the evaluation procedure. I implement the CUDA version based on the software approach I show in Section 3.2 which uses one-dimensional array to store temporary variables in order to save memory. The approach however requires the array to be processed in ascending or descending order, as we need to make sure that the values we are overwriting is not used anymore. Randomly processing order can potentially erase the entries in the array that are still needed to evaluate other nodes. Figure 6.4(a) shows an example where the temporary option prices are updated from bottom to top in order, note that in the last step the top entry will not be updated, as we generally get less tree nodes approaching the root. Recall that the equation we use to update an binomial tree node is:

\[ v_{n,i} = \max(K - S_{n,i}, r(pv_{n+1,i+1} + (1-p)v_{n+1,i-1})) \] (6.1)

where \( i \) is an integer indicating the number of jumps up or down from the initial underlying asset price asset price \( (S_0) \), \( i \) is referred in the implementation as the variable asset price offset, and \( n \) is an integer indicating the number of steps away from the root of the tree (or which step we are
currently on). \(S_{n,i}\) indicates the underlying asset price at node \((n,i)\). \(S_{n,i}\) is \(S_{0}u^i\) if \(i \geq 0\), otherwise it is \(S_{0}d^i\).

Figure 6.4(a) shows the ordered update procedure while Figure 6.4(b) shows a random update at leaf nodes in a binomial tree, where the second node in the array from bottom is updated first, followed by the update of the bottom node in the array. We can see that the update is not correct as the step function is now taking \(v_i\) and \(v_{i-1}\) instead of two \(v_i\)s.

- **(a)** The ordered access.
- **(b)** If the update is not in order then we’ll get a wrong result.

Figure 6.4: The random access problem, noting \(v_i\) stands for the temporary option price at step \(i\) in a binomial tree.

A improved design is to partition the tree and process a trunk at a time and use double buffering to overcome the random access problem. The idea is inspired by Victor Podlozhnyuk’s binomial model proposal for European call options valuations [23]. I explain how I apply the tree partition method to price American Put Option using trinomial trees and achieve higher throughput on GPU.

To start with, I give a simple example. Imagine we now have a trinomial tree with 19 nodes, a graphical representation of such tree is illustrated in Figure 6.6(a). If we start reducing from top to bottom, we are able to reduce the array size by 2 each step. For example, if we are reducing 3 steps, i.e. from step 9 to step 6. The intuitive way to do it is to hold the entire array at step 9 and iterate through it for three times. Another approach is to partition the array into several parts. In the example shown in Figure 6.6(b) we partition the array into 2 parts, \(\text{Partition 1} \) and \(\text{Partition 2}\). As we are reducing from top to bottom, \(\text{Partition 2}\) can be reduced like a normal tree. However in order to reduce \(\text{Partition 1}\) for 3 steps we’ll need 6 additional nodes from \(\text{Partition 2}\). \(10 + 6 = 16\) nodes are required in total for \(\text{Partition 2}\). I call this piece of additional work to be done \(\text{Delta}\). In this example, \(\text{Delta}\) is 6. With \(\text{Delta}\) as the overhead we are able to partition the tree into 2 parts, this can be generalised if we try to partition the tree into \(y\) parts, in which case the total overhead will
be \((y − 1)\Delta\). Therefore if the size of the partition is \(x\), we will generally require a cache space of \(x + \Delta\) for each partition, and the number of steps can be done within this partition is \(\Delta/2\) for trinomial trees. The tree will be processed in a partition-by-partition manner in serial and the nodes within each partition will be processed in parallel.

The solid area in Figure 6.5 shows a tree partition in a larger trinomial tree scope. If the tree partition size is \(x\) and we want to allow \(n\) step reductions within the partition, an extra \(\Delta\) of \(2n\) nodes is required (indicated by shades boxes). This leads to a cache of size \(x + 2n\) to store the partition together with the \(\Delta\). For each partition, a base pointer \(c_{\text{base}}\) is used to indicate where the first element of the partition is within the whole tree. \(c_{\text{start}}\) and \(c_{\text{end}}\) are local counters for the shared memory, they are created to cope with CUDA’s thread ID and block ID algorithmic. In general, \(c_{\text{start}}\) means where the inner-most for-loop should terminate at its first run, \(c_{\text{end}}\) indicates where the inner-most loop should terminate at its last run.

Double buffering is used to overcome the random access problem [23]. Instead of read and update in the same cache array, 2 caches are used. We initialise cache \(A\) at the beginning; in the first iteration we read from \(A\) and write results to cache \(B\). Having finished updating cache \(A\), in the next iteration the position of cache \(A\) and cache \(B\) is swapped: data is read from cache \(B\) and the results are written to cache \(A\). We iterate over the array in this way till the termination condition is met. The implementation detail is demonstrated in Figure 6.7.

American options can be exercised at any time up to its expiration. Early exercise case needs to be coped with at each node in the pricing procedure. To calculate the early exercise price at each tree node, we must know the location of tree node in the tree. The tree node location is easy to retrieve when then whole tree is known to us. On the other hand, if a partition is processed, to get the global location of the local node within the partition is not that trivial. The location information can be calculated knowing \(c_{\text{base}}\), the current step \(n\) we are at. Details can be found in Figure 6.7 where the asset price offset, which indicates how far have the current underlying asset price jumped up or down from the initial asset price \(S_0\), is calculated. The code is equivalent to the software prototype shown in Figure 3.3.

![Figure 6.5: A single partition within a trinomial tree.](image-url)
(a) The normal trinomial tree reduce sequence.

(b) A partitioned reduce sequence.

Figure 6.6: Partitioning a tree: the concept.
//variable i indicates which time step we are at.
//dataA and dataB are high speed caches in shared memory
for(int k = c_start-1; k >= c_end;){
    __syncthreads();
    int baseOffset = 1-(i- (c_start -k)/2);
    for(int l = threadIdx.x; l < k; l += blockDim.x){
        int assetPriceOffset = baseOffset+l+c_base;
        float optionValue = PuByR * dataA[l + 2] +
            PmByR * dataA[l+1] +
            PdByR * dataA[l];
        float earlyExerciseValue =
            X-S*expf(upStepSize*assetPriceOffset);
        dataB[l] = fmaxf(earlyExerciseValue, optionValue);
    }
    k-=2;
//wait until all threads have finished processing dataA.
    __syncthreads();
    baseOffset = 1-(i- (c_start -k)/2);
    for(int l = threadIdx.x; l < k; l += blockDim.x){
        int assetPriceOffset = baseOffset+l+c_base;
        float optionValue = PuByR * dataB[l + 2] +
            PmByR * dataB[l+1] +
            PdByR * dataB[l];
        float earlyExerciseValue =
            X-S*expf(upStepSize*assetPriceOffset);
        dataA[l] = fmaxf(earlyExerciseValue, optionValue);
    }
    k-=2;
}

Figure 6.7: Process the tree using double buffering.
6.3 Summary

One major part of this project’s objective is to compare the FPGA implementations with the GPU implementations in seeking of a optimal solution. Hence both of the GPU implementations need to be highly optimised to be comparable. Being different from the FPGA implementation, the devices like GPUs require a large amount of data to be processed simultaneously to be fully utilised. For the tree based version, there are two possibilities to achieve this:

- To evaluate massive number of trees in parallel and process one element in a single trees at a time. The GLSL implementation followed this route. This route is straightforward but if there aren’t enough trees to be evaluated together the device utilisation will be low.

- My CUDA implementation seeks to exploit the inherent parallelism within the tree model. The fact that "in the same tree, the nodes at the same step can be processed in parallel" is used to achieve maximum parallelism. In cases then the tree is too large to fit into the high speed shared memory, I cut the tree into smaller trunks to fit them into the shared memory, hence reducing the latency caused by memory accessing. To avoid possible data loss in parallel operations, double buffering is used to make sure no useful data is overwritten.

Although the GLSL turned out to be not so exceptional, the CUDA implementations can achieve very impressive speedups. More details can be found in next chapter where the evaluation is done.
Chapter 7

Evaluations and Results

In this chapter my implementations are tested and evaluated. I divide the testing procedure into two phases:

- Testing for absolute speed-ups; in Section 7.1.
- Testing for speed-ups in context; in Section 7.2.

In the first phase I seek to find the maximum possible speed-ups that can be achieved by the following devices:

- The FPGA implementations, including double precision, single precision and fixed point implementations for both the binomial and trinomial models. All the FPGA implementations are mapped to a Virtex 4 xc4vsx55 device. In this case, only the EvaluationCore alone is being tested, therefore its maximum clock frequency is considered to be the maximum possible nodes that it can process in a second.

- The GPU implementations, including a GLSL implementation for the binomial model and two CUDA implementations for both the binomial and trinomial models. All implementations are adopt on floating point operators. The GLSL implementation runs on a Geforce 7900GTX and the CUDA implementations run on a Geforce 8600GTS.

- Two reference PCs running the software implementations, for both binomial and trinomial models. The software implementations run on two reference PCs as benchmarks: one laptop PC with Core2 Duo processor and one desktop PC with Pentium4 processor. The software versions are fully optimised to ensure optimality.

The teasing cases are set for GPU and PC implementations to achieve their maximum speed-ups. The different versions are compared and contrasted, their potential speed-ups are estimated and their limitations are addressed.

In the second phase I aim to find the actual speed-ups that can be archived for the FPGA implementations. The testing cases are set to gradually approach the limit of the FPGA’s processing power. A total number of 11 variations of FPGA implementations based on different real-number representations and different number of core duplications are being tested. The testing result is firstly compared to a reference PC then contrasted with the FPGA results and GPU.
results in phase one. The behavior difference exhibited by core duplications on FPGAs is also discussed in detail.

7.1 Absolute Speed-up Comparison

First I try to test absolute speed-ups of the FPGA Evaluation Cores against PC and GPU implementations. The absolute speed-up numbers provide an upper bound for possible accelerations, and provides a best-case scenario for further testing. In this test I only consider the speedup that can be achieved by the Evaluation Core without the full control logic. The FPGA testing is based on a xc4vsx55 device. This device is chosen because RCHTX is not available to me until near end of this project. The American put option benchmark has been calculated using three different FPGA implementations in different numerical representations as well as a reference PC implementation. All the FPGA implementations are compared to software implementations on a PC which provides the reference. The reference Intel PCl implementation is based on C++ code fully optimized for local hardware profile running on a 2.2GHz Core2 Duo processor with 2GB of RAM and Windows XP pro operating system. The reference Intel PC2 is a 3.2GHz Pentium4 processor with 1GB of RAM; this implementation involves fully optimised C++ code with Intel SSE3 enabled.

The FPGA device utilization figures are shown in Table 7.1. The results indicate that none of the FPGA device is fully occupied in all three cases involving double-precision, single-precision and fixed-point arithmetic for both binomial and trinomial models. Hence performance improvement can be achieved by replicating the Evaluation Core in a single device. Although only nodes on the same level of a binomial tree can be computed simultaneously, acceleration can be achieved by evaluating several trees in parallel.

The lower part of Table 7.1 shows the space occupancy/acceleration results for different precision implementations, including core replication that can be done on a single device to gain further performance. The Left-hand side part of the table shows results for the binomial implementation and the right-hand side gives data for the trinomial implementation.

Table 7.2 shows the data for two GPUs and two reference PCs. The speed benchmark for both reference PCs and GPU is to evaluate a $1 \times 10^3$ step binomial tree for $2^{20}$ times. The stated GPU clock rates in Table 7.2 are the peak rates specified by nVidia. Double-precision floating-point arithmetic is unavailable on Geforce 7900GTX [20] and is not yet supported by CUDA 1.1 for Geforce 8600GT.

First consider the acceleration of the FPGA over the Intel PC1 and PC2 benchmarks. From the results, it can be seen that, for the binomial implementation the 32-bit 16.16 fixed-point implementation offers a 11.5 times acceleration, while the 32-bit single precision floating-point and the 64-bit double precision version offer 10.6 times and 9.3 times speedup respectively. Not surprisingly, fixed-point arithmetic is faster and smaller than floating-point arithmetic in an FPGA. For instance, 3 cores can be implemented on a single xc4vsx55 device if double precision arithmetic is adopted, which leads to a 27.9 times speedup over optimized software running on a Core2 Duo processor. In contrast, 14 cores in fixed-point arithmetic can be implemented in an xc4vsx55 FPGA, indicating a 161 times acceleration for multiple binomial trees.
<table>
<thead>
<tr>
<th></th>
<th>FPGA Binomial</th>
<th>FPGA Trinomial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Double</td>
<td>Single</td>
</tr>
<tr>
<td>Slices</td>
<td>7,162</td>
<td>3,805</td>
</tr>
<tr>
<td></td>
<td>(29%)</td>
<td>(15%)</td>
</tr>
<tr>
<td>FFS</td>
<td>5,924</td>
<td>2,914</td>
</tr>
<tr>
<td></td>
<td>(12%)</td>
<td>(5%)</td>
</tr>
<tr>
<td>LUTs</td>
<td>6,020</td>
<td>3,238</td>
</tr>
<tr>
<td></td>
<td>(12%)</td>
<td>(6%)</td>
</tr>
<tr>
<td>BRAMs</td>
<td>20 (6%)</td>
<td>18 (5%)</td>
</tr>
<tr>
<td></td>
<td>(45%)</td>
<td>(24%)</td>
</tr>
<tr>
<td>FFs</td>
<td>5,924</td>
<td>2,914</td>
</tr>
<tr>
<td></td>
<td>(12%)</td>
<td>(5%)</td>
</tr>
<tr>
<td>LUTs</td>
<td>6,020</td>
<td>3,238</td>
</tr>
<tr>
<td></td>
<td>(12%)</td>
<td>(6%)</td>
</tr>
<tr>
<td>BRAMs</td>
<td>20 (6%)</td>
<td>18 (5%)</td>
</tr>
<tr>
<td></td>
<td>(45%)</td>
<td>(24%)</td>
</tr>
<tr>
<td>DSPs</td>
<td>32 (6%)</td>
<td>8 (1%)</td>
</tr>
<tr>
<td></td>
<td>(12%)</td>
<td>(5%)</td>
</tr>
<tr>
<td>MHz</td>
<td>67.3</td>
<td>76.0</td>
</tr>
<tr>
<td>Replication</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>(cores/chip)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processing Speed</td>
<td>67.3</td>
<td>76.0</td>
</tr>
<tr>
<td>(M nodes/sec)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean Error</td>
<td>0</td>
<td>$4 \times 10^{-4}$</td>
</tr>
<tr>
<td>Acceleration</td>
<td>9.3×</td>
<td>10.6×</td>
</tr>
<tr>
<td>(1 core)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acceleration</td>
<td>27.9×</td>
<td>63.6×</td>
</tr>
<tr>
<td>(replicated cores)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Binomial/Trinomial Performance/area results for xc4vsx55 FPGA. The percentage shows utilisation of a specific FPGA resource; note that acceleration is compared with the reference PC1.

<table>
<thead>
<tr>
<th></th>
<th>Binomial</th>
<th>Trinomial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU1</td>
<td>GPU2</td>
</tr>
<tr>
<td>MHz</td>
<td>Double</td>
<td>Single</td>
</tr>
<tr>
<td>Replication</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>(pipelines)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processing Speed</td>
<td>477</td>
<td>1476</td>
</tr>
<tr>
<td>(M nodes/sec)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean Error</td>
<td>$4 \times 10^{-4}$</td>
<td>$4 \times 10^{-4}$</td>
</tr>
<tr>
<td>Acceleration</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>(replicated cores)</td>
<td>66×</td>
<td>205×</td>
</tr>
</tbody>
</table>

Table 7.2: Binomial/Trinomial Performance Geforce 7900 (GPU1) and 8600 GPU (GPU2), Intel Core2 Duo (PC1) and Pentium4 (PC2) processors; note that acceleration is compared with the reference PC1.
evaluated in parallel.

The trinomial implementation is generally around 1.5% slower than the binomial implementation. The small timing overhead is introduced by longer pipeline and the associated control logic. The trinomial implementation is also 40% larger than the binomial implementation. The significant increase in space is due to the additional multiplier and adder in the Algorithmic Core and the multiplier in Table Generator. We discussed in Section 2.4 that the trinomial model will produce the same result as a binomial model in fewer number of steps. Up to \( x/2 \) fewer steps is required for the trinomial model compared to the binomial one, where \( x \) is the number of steps required for the binomial model to get the same result. The trinomial model also provides an effective means to model interest rate derivatives \cite{17}, such as American bond options \cite{15}. These properties compensate the drawbacks of the trinomial implementation.

The floating-point implementations on GPU are faster than the corresponding single precision implementations on FPGA with replicated cores in both binomial and trinomial implementations. However it is worth noting that the difference between FPGA and GPU1 is within a factor of 2. This is because both GPU1 and the FPGA approaches are based on straightforward implementations without including further parallelism and optimisation.

On the other hand, the implementation on GPU2 is based on CUDA and seeks to exploit full utilisation of GPU resources by:

- Scheduling options to be evaluated concurrently on different multiprocessors.
- Allow multiple threads to evaluate a single step simultaneously.
- Use double buffering in local shared memory to avoid high-delayed access to global memory \cite{23}.

It is interesting to see that GPU2 is three times faster than the FPGA in both binomial and trinomial implementations. We only have 2% latency increase to evaluate a trinomial tree on FPGA, and asymptotically doubled the computational time when model it on CPUs and GPUs having the extra multiply and add operations. This is because in the FPGA case we trade space for speed. The logic for trinomial model is asymptotically twice as large as the binomial one, hence we are not able to map as many cores on a single FPGA as in the binomial case. The other possible reason is that CUDA has instruction level parallelism that is able to reduce computational time for complex expressions. It is not surprising to see that GPU2 is over 3 times faster than GPU1. GPU2 has more pipelines than GPU1 (24 pipelines, used here to implement scalar operations, versus 32 stream scalar processors), which is 1.5 times more performance than GPU1; and the shader core used for computational purpose on GPU2 runs two times faster than GPU1 (from 650MHz to 1450MHz). The difference in on board memory of GPU1 and GPU2 is irrelevant since our problem is not data-bound but computation bound. We believe that the Virtex 4 FPGA and the Geforce 7900GTX GPU are broadly comparable, since both are based on 90 nm technology.

Additional performance improvement can be expected if the latest Geforce 8800 class GPU is adopted: a 4 times speedup can be gained from increased parallelism (128 versus 32 stream scalar processors) and some further speedup.
from clock speed increment (from 1450MHz to 1600MHz). However, the 8800 class GPUs adopt 80nm technology, and they should be compared with the latest FPGA technology such as Virtex 5 from Xilinx and Stratix III from Altera. Geforce 8600GTS is used as a benchmark to indicate how CUDA can be used to exploit resources for financial computations in similar areas.

If a larger Virtex 4 or even Virtex 5 device is used which has 4 times or more slices than that on our xc4vsx55 and with higher basic clock frequency, then at least 4 times speedup can be achieved without further optimisation.

On the other hand, single precision operators in FPGAs can run at a clock rate of up to 322MHz \[36\]; our current implementation at 76.0MHz has much scope for improvement. Furthermore, if we are able to reduce device utilization to half of the original size with further optimisation, that would enable us to put twice as many Evaluation Cores on the FPGA, producing a further 2 times speedup.

The speed benchmark is purely for the purpose of measuring maximum evaluation speed, hence we choose to measure 1024^2 trees with same depth to achieve maximum parallelism. In reality, it is rarely the case that 1024^2 options will be changing price at the same time and all of them will share the same depth. The requirement of varying tree depths for neighboring nodes would map inefficiently to the GPU SPMD (single program multiple data) programming model. It can be expected that the one which has the largest number of steps will become the bottleneck in GPU. An FPGA implementation can be designed with data path flexibility to alleviate this restriction.

From experience there is a tradeoff when using HyperStreams between the development time and the amount of acceleration that can be achieved. Although we are able to implement complex algorithms easily in FPGAs with HyperStreams, the highest possible performance and utilisation of FPGA resources is not guaranteed. The balance between development time and performance needs to be explored with further research and experiment. However our HyperStreams implementation still gives a satisfactory result with significant acceleration over the software implementations. Hence HyperStreams is useful particularly for producing prototypes rapidly to explore the design space; once promising architectures are found, further optimisations can be applied.

### 7.2 Speed-up In Context

In this section I evaluate the Tree Valuation Cores based on a imaginary real world scenario. The aim is to find out the actual speed-ups that can be achieved based on my design and address possible improvement that can be made. The testing scenario is to valuate 1000 options with different steps (from 25 steps to 800 steps), the time taken to valuate all the options is recorded and the corresponding speed in million nodes/second is derived.

The test cases are grouped into three categories based on different precision operators used, within each category, different versions are tested based on number of duplications of the Tree Valuation Cores on a single FPGA. In particular:

- Double precision valuation; three tests are taken to evaluate the single core version, double core version and triple core version.
• Single precision valuation: four tests are taken to evaluate the single core version, double core version, triple core version and quad core version.

• Fixed point valuation: same four tests are taken as the Single precision valuation.

To simplify the description, I give short names to different implementations, listed below:

1. SW: The software implementation, running on the server.
2. D1: Double precision version with one Tree Valuation Core.
3. D2: Double precision version with two Tree Valuation Cores.
4. D3: Double precision version with three Tree Valuation Cores.
5. FL1: Single precision version with one Tree Valuation Core.
6. FL2: Single precision version with two Tree Valuation Cores.
7. FL3: Single precision version with three Tree Valuation Cores.
8. FL4: Single precision version with four Tree Valuation Cores.
9. FX1: Fixed point version with one Tree Valuation Core.
10. FX2: Fixed point version with two Tree Valuation Cores.
11. FX3: Fixed point version with three Tree Valuation Cores.
12. FX4: Fixed point version with four Tree Valuation Cores.

There isn’t a four Tree Valuation Core version for double precision as the RCHTX board could only fit three double precision cores. Single precision and fixed point only have up to four cores as I consider the four cores case is general enough to cover the higher number multi-core cases.

The host is an HP Proliant DL145 G3 Server with three Dual-Core AMD Opteron(tm) Processors runs at 1000MHz and 10GB of RAM. The reference software implementation is running on the host to make fair comparison to the hardware version. In general, I expect the server to grand me asymptotically the same resource as PC1 to run my implementations.

The target device is a RCHTX board with Virtex 4 xc4vlx160 planted in the host. The RCHTX board DSM interface can provide a bandwidth of 300MBps for hardware-software communication. I consider the speed is fast enough for look-up table transferring. Typically, if double precision numbers are used to evaluate an \( n \) step tree, the size of the lookup table will be \( 4 \times (n + 1) \) bytes, the transferring time will be small unless \( n \) is significant compared to \( 10^6 \). The only overhead is the DSM start-up time on software side. This overhead occurs each time DSMRead or DSMWrite is called, as the software needs to check hardware handles and issue library calls.

In all the testing cases, exactly one DSMWrite and one DSMRead call is used for each option. To achieve pseudo parallelism in software, I utilised the DSM buffer by using a loop which does asynchronous reads and writes to different instances. A simple example code is given in Figure 7.1.
void ProcessLotsOfOptions(unsigned n,
   OptionInput *inputs, OptionOutput *outputs)
{
    unsigned wi=0, ri=0;
    unsigned contexts[NumInstances] = {-1};
    while(ri<n){
        for(i=0;i<NumInstances;i++){
            if(contexts[i]!=-1){
                if(ReadResult(i, outputs[contexts[i]])){
                    contexts[i]=-1;
                    ri++;
                }
            }
        }
        if((wi<n) && (contexts[i]!=-1)){
            if(WriteInput(i, inputs[wi])){
                contexts[i]=wi;
                wi++;
            }
        }
    }
}

Figure 7.1: Pseudo Parallelism in Software.

First we look at testing cases for D1, D2 and D3. Nine testing scenarios is used to valuate from 25-step options to 800-step options for 1000 times. D1, D2 and D3 are tested based on these testing scenarios. The results are shown in Table 7.3, Table 7.4 and Figure 7.2.

Table 7.3 displays the time taken to valuate 1000 options with different steps in hardware and software, Figure 7.2(a) shows a corresponding plotted chart. The chart indicates that the software version has a near-exponential growth in time taken when the number of steps of the option increases. Recall the number of nodes in a binomial tree is \((N+1)(N+2)/2\), where \(N\) is the number of steps of the tree. A quadratic growth in number of nodes can be expected as \(N\) grows. If the server runs the software has a linear processing speed, then a none-linear growth in time taken will be observed; the detail can be found in Table 7.4 discussed later. On the other hand, all the three hardware version indicate some pseudo linear growth in time taken in Figure 7.2(a). However it can be observed that the slope of the "linear" lines get steeper when the number of steps is jumping from 500 steps to 800 steps, this means the full power of the hardware has exhibited. It is not surprising to see that the software implementation is doing better than all the hardware implementations when the number of steps of the options are small. Recall that all the hardware versions have DSM overheads, the overhead is significant when the total processing time is small. The overhead becomes less significant when number of steps grows, we can see that in 7.3 that the hardware versions outperform the software version when the number of steps is around 75. This matches the central assumptions I made in Section 5.3.1 that a tree need have a none-trivial number of steps for
hardware implementation to be effective.

It can also be observed in the figure that, in general, the more *Tree Valuation Core* we have in the implementation, the less time is taken to finish the processing. The difference in time taken is not significant when number of steps of the option to be valued is small, however in cases when the number of steps is large (for example 800), it can be observed that 2 Core (D2) version takes roughly half time and D3 takes roughly a third of time to finish the valuations compared with D1.

We now examine the processing speeds of D1, D2 and D3, measured in million nodes/s. The data is shown in Table 7.4 and plotted in Figure 7.2(b). First we look at Figure 7.2(b) which shows that SW (the software version) is exhibiting a constant processing speed along the x-axis; this explains the quadratic growth of time taken with number of steps in the previous paragraph. The software version experience some turbulence in processing speed due to the inaccuracy of the system timer. We can see that initially the software version has a faster processing speed than the hardware versions. However the hardware versions quickly catch up at around step 75: the speed of all hardware versions grows very quickly between steps 25 to 250, which is indicated in the figure as steep slopes. After step 250 the growth slows down significantly and eventually stalls. I call this phenomenon as the hardware eventually "saturates". This phenomenon happens because the DSM overhead is significant compared to the time taken for pricing when the number of steps is small. The DSM overhead eventually becomes less significant when total time takes for pricing increases with number of steps. If the number of steps keeps increasing, the DSM overhead is eventually negligible – this is when the speed growth stalls. Therefore I measure the full speed of all the hardware versions at step 800. It can be seen easily that at step 800, D3 is asymptotically three times faster than D1, and D2 is asymptotically 2 times faster than D1. The reason why N-core version is not exactly N times faster than the one core version is that the software is not able to send and receive multiple requests in true parallelism.

Secondly we look at the testing cases for single floating point versions FL1, FL2, FL3 and FL4. The results are shown in Table 7.5, Table 7.6 and Figure 7.3. The data shows some very similar pattern as the double precision versions D1, D2 and D3, except that:

- FL1, FL2 are generally FL3 1.2 times faster than the corresponding D1, D2 and D3, measured at step 800. This number matches the result we get in Table 7.1 where absolute speed of the *Evaluation Core* is measured. It can be seen that the speed of different versions eventually converges proportionally with the speed of their *Evaluation Core*.

- FL4 at step 800 is asymptotically 4 times faster than FL1, this is the desirable result. However it can be observed that the difference between NT_N and T_1 is increasing with N, where N is the number of cores an implementation have, T_1 and T_N are the times taken for a single core version and an N-core version to finish the valuation respectively.

Thirdly we look at the testing cases for fixed point versions FX1, FX2, FX3 and FX4. The results are shown in Table 7.7, Table 7.8 and Figure 7.3. The data indicates similar patterns to the floating point versions data. However, several differences can be addressed:
FX1, FX2, FX3 and FX4 are generally 1.3 times faster than their corresponding single floating point versions. They are generally 1.6 times faster than the double precision versions respectively. The numbers obey the speeds of the Evaluation Cores in Table 7.1 where absolute speedup is measured.

The curve for the 2 core version in Figure 7.4(b) shows an unexpected jump at between step 500 and step 800. I think it is due to some inaccuracy in measurement, the host server might be busy when I test the 500-step case and becomes very available when the 800-step case is being tested.

Having seen the behaviors of different hardware implementations we now look at their speed-ups comparing to the software version (SW). Two cases are considered: speed-ups at 75 steps level, this is when the hardware version start to outperform the software version; and speed-ups at 800 steps level, this is when the hardware version saturates (when the DSM overhead is negligible). The data is shown in Table 7.9 and plotted in Figure 7.5(a) and Figure 7.5(b) respectively. It can been seen easily that in Figure 7.5(a) where we are valuating options with 75 steps, all the speed-ups of different precision implementations grow linearly with number of Tree Valuation Cores. The same can be observed in Figure 7.5(b) when options with 800 steps are valuated. This means that in the general case, the hardware implementations can achieve asymptotical linear speed-ups by replicating Tree Valuation Cores provided that the software-side is able to send enough requests concurrently to the hardware, regardless what the number of steps of a underlying option is.

It can also be observed in Table 7.9 that the speed-ups in context is slower than the absolute speed-ups measured in Table 7.1. I speculate three main reasons:

- The full implementation of tree model involves complex control logic with longer path than the pipelined Evaluation core. It becomes the new bottleneck and brings down the overall performance. Most likely the longer path is created by frequent block memory accesses.
- The DSM overhead is not negligible when the actual speed-ups are compared to absolute speed-ups.
- As the procedure to synthesis bitmaps from EDIF generally take over 5 hours, I set a very low target clock frequency (50MHz instead of 100MHz) to speed up the procedure. The builder is not generating the fastest bitmaps as the target clock frequency is low.

Asymptotically 2 times more speed-ups can be achieved with longer synthesis time if the highest possible target clock is set. That will make the difference between the speed-ups in context and the absolute speed-ups within factor of 2, which is a close match.

According to the relative speed-up result, I suggest that trees with trivial number of steps (trees with less than 75 steps) to be valuated in software directly and trees with none-trivial number of steps to be accelerated by software. The software side should be about to identify when trees should be sent to hardware for acceleration.
Now we look at the resource occupancy for different hardware implementa-
tions, shown in Table 7.10. The Slices utilisation figures for different implementa-
tions are plotted in a chart in Figure 7.6. We can see that number of slices occupied by the double precision, single precision and fixed point implementa-
tions all increase in a linear pattern when I try to increase the number of cores to map on the FPGA. It is interesting to see that the floating point and fixed point single core version have similar number of slice occupancy rates. However they soon become different as I duplicate more Evaluation Core 
s on the FPGA. If I am duplicating the entire Tree Valuation Logic, I would expect the slice occupancy rates for the two versions to be exactly the same, what goes wrong? To find out the reason I use the logic estimator provided by DK5. It turns out that the compiler is automatically optimising the design by reusing the DSM logic instead of duplicate it. For the implementations with more than one core, a single DSM module is shared between the cores.

Linear pattern can also be observed in Figure 7.7 and Figure 7.8 where data about utilisations of BRAMs and DPSs are plotted. This is the expected result, as no additional logic in added when the Tree Valuation Cores are duplicated, based on the design I described in Section 5.5, each Tree Valuation Core is independent communicate to the software on different channels.

We can see from the data indicated in Table 7.10 that for the double precision and fixed point implementations the dominant resource occupied is DSPs. Especially for the fixed point implementation, the number of replications can be done is determined by the rate of DSP occupancy. For the single precision implementa-
tion the dominant resource occupied is the number of slices. In general, three double precision cores can be mapped to the RCHTX board and achieve a expected maximum speed-up of 18 times; while 8 single precision cores can be mapped to the RCHTX board and obtain a maximum speed-up of 52 times and 12 fixed point cores can be mapped to get a maximum speed up of 84 times.

Comparing to the GPU results stated in Table 7.2, the expected speed-ups in context result for FPGA implementations seems to be 50% slower. However it is worth noting that the figures shown in Table 7.2 are the maximum speed-ups that GPU can achieve. The GPU implementations require a large number of options with same number of none-trivial steps to be valued simultaneously to achieve the maximum speed-up. In real scenarios options will have different steps, the number of steps of an option can some times be small, in that case the GPU performance will depend on the time taken to value the option with the largest number of steps in that batch. On other cases, the number of options to be valued may vary from time to time, if the number is relatively low comparing to the batch size, the GPU will not be utilised fully. The CUDA technology might have solutions for the problems stated above, for example efficient batch scheduling schemes can be deployed.

On the other hand, the FPGA solution only requires the options to have none-trivial number of steps (number of steps \( \geq 75 \)), which can be easily satisfied in real scenarios. Apart from the flexibility, FPGA outperforms the GPU solution in power consumption as well. Typically a Geforce 8600GTS requires a minimum power supply of 71 Watts while three-core double precision FPGA implementation requires just over 2 Watts (the FPGA power consumption is estimated by Xilinx Xpower Estimator and the GPU power consumption is based on the data sheet provided by nVidia). The power consumption of the FPGA implementation is 225 times smaller than the GPU, which is a very significant
saving in power.

Unfortunately due to the excessively long synthesis process mapping the binomial implementations to the FPGA on RCHTX (over 5 hours to for placing and routing each) I failed to get enough time to test the trinomial implementations under context. But I would expect the trinomial implementations to have very similar behavior to the binomial ones. In particular:

- I will expect the trinomial software version to be about 1/3 slower due to more complex algorithmic in the step function (described in Section 3.2). The trinomial hardware version should run at roughly the same speed as the binomial version, as the trinomial Evaluation Core in running asymptotically the same speed as the binomial Evaluation Core (described in Table 7.1) and the control module have similar number of stages as the binomial one (described in Section 5.4). Therefore higher speed-ups can be expected for the trinomial implementations compared to the numbers from the binomial ones.

- The hardware-software communication overhead will remain the same, as the Asset Price Lookup Table to be transferred is the same for binomial model and trinomial model with same number of steps. The only difference is the trinomial valuation requires one extra variable to describe the probability for the underlying asset to stay the same, and this is negligible given that the communication interface can provide a bandwidth of 300MBps.

- The number of Tree Valuation Core replications drop. I can expect the trinomial Tree Valuation Core to be asymptotically 40% larger than the binomial one, according to the size of their Evaluation Cores.

### 7.3 Power Consumption Estimation

Power consumption of the FPGA implementations is taken into account as well. As for the FPGA implementations power consumption is an important factor that can be used to estimate the power supply requirements and the working temperature of the chip. Therefore I tested the power consumptions of all the
<table>
<thead>
<tr>
<th>No.Steps</th>
<th>1 Core (D1)</th>
<th>2 Core (D2)</th>
<th>3 Core (D3)</th>
<th>S.W.</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1.25</td>
<td>1.35</td>
<td>1.60</td>
<td>3.90</td>
</tr>
<tr>
<td>50</td>
<td>4.23</td>
<td>5.17</td>
<td>6.03</td>
<td>3.79</td>
</tr>
<tr>
<td>75</td>
<td>8.78</td>
<td>9.86</td>
<td>10.97</td>
<td>4.12</td>
</tr>
<tr>
<td>100</td>
<td>10.88</td>
<td>14.05</td>
<td>17.17</td>
<td>5.05</td>
</tr>
<tr>
<td>125</td>
<td>12.00</td>
<td>17.02</td>
<td>22.64</td>
<td>5.72</td>
</tr>
<tr>
<td>150</td>
<td>12.94</td>
<td>19.45</td>
<td>26.08</td>
<td>4.84</td>
</tr>
<tr>
<td>250</td>
<td>14.96</td>
<td>25.50</td>
<td>31.63</td>
<td>5.27</td>
</tr>
<tr>
<td>500</td>
<td>16.78</td>
<td>31.18</td>
<td>41.92</td>
<td>5.53</td>
</tr>
<tr>
<td>800</td>
<td>17.49</td>
<td>33.58</td>
<td>48.18</td>
<td>5.58</td>
</tr>
</tbody>
</table>

Table 7.4: Binomial: Speed to valuate 1000 options using double precision floating point operators, measured in million nodes/sec.

<table>
<thead>
<tr>
<th>No.Steps</th>
<th>1 Core (FL1)</th>
<th>2 Core (FL2)</th>
<th>3 Core (FL3)</th>
<th>4 Core (FL4)</th>
<th>S.W.</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0.23</td>
<td>0.21</td>
<td>0.18</td>
<td>0.14</td>
<td>0.09</td>
</tr>
<tr>
<td>50</td>
<td>0.26</td>
<td>0.21</td>
<td>0.18</td>
<td>0.15</td>
<td>0.35</td>
</tr>
<tr>
<td>75</td>
<td>0.28</td>
<td>0.25</td>
<td>0.22</td>
<td>0.20</td>
<td>0.71</td>
</tr>
<tr>
<td>100</td>
<td>0.39</td>
<td>0.30</td>
<td>0.25</td>
<td>0.20</td>
<td>1.02</td>
</tr>
<tr>
<td>125</td>
<td>0.55</td>
<td>0.39</td>
<td>0.29</td>
<td>0.22</td>
<td>1.40</td>
</tr>
<tr>
<td>150</td>
<td>0.73</td>
<td>0.49</td>
<td>0.36</td>
<td>0.27</td>
<td>2.37</td>
</tr>
<tr>
<td>250</td>
<td>1.75</td>
<td>1.02</td>
<td>0.83</td>
<td>0.59</td>
<td>6.00</td>
</tr>
<tr>
<td>500</td>
<td>6.19</td>
<td>3.33</td>
<td>2.48</td>
<td>1.87</td>
<td>22.72</td>
</tr>
<tr>
<td>800</td>
<td>15.17</td>
<td>7.91</td>
<td>5.51</td>
<td>4.20</td>
<td>57.55</td>
</tr>
</tbody>
</table>

Table 7.5: Binomial: Time taken (in seconds) to valuate 1000 options using single precision floating point operators.

<table>
<thead>
<tr>
<th>No.Steps</th>
<th>1 Core (FL1)</th>
<th>2 Core (FL2)</th>
<th>3 Core (FL3)</th>
<th>4 Core (FL4)</th>
<th>S.W.</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1.52</td>
<td>1.63</td>
<td>1.93</td>
<td>2.51</td>
<td>3.90</td>
</tr>
<tr>
<td>50</td>
<td>5.12</td>
<td>6.25</td>
<td>7.29</td>
<td>9.04</td>
<td>3.79</td>
</tr>
<tr>
<td>100</td>
<td>13.17</td>
<td>17.00</td>
<td>20.78</td>
<td>25.76</td>
<td>5.05</td>
</tr>
<tr>
<td>125</td>
<td>14.52</td>
<td>20.60</td>
<td>27.40</td>
<td>36.37</td>
<td>5.72</td>
</tr>
<tr>
<td>150</td>
<td>15.66</td>
<td>23.54</td>
<td>31.56</td>
<td>41.99</td>
<td>4.84</td>
</tr>
<tr>
<td>250</td>
<td>18.11</td>
<td>30.86</td>
<td>38.27</td>
<td>53.30</td>
<td>5.27</td>
</tr>
<tr>
<td>500</td>
<td>20.31</td>
<td>37.73</td>
<td>50.72</td>
<td>67.37</td>
<td>5.53</td>
</tr>
<tr>
<td>800</td>
<td>21.17</td>
<td>40.63</td>
<td>58.30</td>
<td>76.48</td>
<td>5.58</td>
</tr>
</tbody>
</table>

Table 7.6: Binomial: Speed (million nodes/s) to valuate 1000 options on RCHTX and on software using single precision floating point operators.
Figure 7.2: Charts for double precision valuation based on binomial model.
Figure 7.3: Charts for single precision valuation based on binomial model.
<table>
<thead>
<tr>
<th>No. Steps</th>
<th>1 Core (FX1)</th>
<th>2 Core (FX2)</th>
<th>3 Core (FX3)</th>
<th>4 Core (FX4)</th>
<th>S.W.</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0.22</td>
<td>0.20</td>
<td>0.17</td>
<td>0.13</td>
<td>0.09</td>
</tr>
<tr>
<td>50</td>
<td>0.25</td>
<td>0.19</td>
<td>0.17</td>
<td>0.13</td>
<td>0.35</td>
</tr>
<tr>
<td>75</td>
<td>0.28</td>
<td>0.22</td>
<td>0.20</td>
<td>0.17</td>
<td>0.71</td>
</tr>
<tr>
<td>100</td>
<td>0.36</td>
<td>0.28</td>
<td>0.23</td>
<td>0.18</td>
<td>1.02</td>
</tr>
<tr>
<td>125</td>
<td>0.49</td>
<td>0.35</td>
<td>0.27</td>
<td>0.20</td>
<td>1.40</td>
</tr>
<tr>
<td>150</td>
<td>0.67</td>
<td>0.45</td>
<td>0.33</td>
<td>0.25</td>
<td>2.37</td>
</tr>
<tr>
<td>250</td>
<td>1.46</td>
<td>0.94</td>
<td>0.76</td>
<td>0.54</td>
<td>6.00</td>
</tr>
<tr>
<td>500</td>
<td>5.12</td>
<td>3.33</td>
<td>2.27</td>
<td>1.71</td>
<td>22.72</td>
</tr>
<tr>
<td>800</td>
<td>11.67</td>
<td>6.33</td>
<td>5.05</td>
<td>4.00</td>
<td>57.55</td>
</tr>
</tbody>
</table>

Table 7.7: Binomial: Time taken (in seconds) to valuate 1000 options on RCHTX and on software using fixed point operators.

<table>
<thead>
<tr>
<th>No. Steps</th>
<th>1 Core (FX1)</th>
<th>2 Core (FX2)</th>
<th>3 Core (FX3)</th>
<th>4 Core (FX4)</th>
<th>S.W.</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1.61</td>
<td>1.78</td>
<td>2.10</td>
<td>2.71</td>
<td>3.9</td>
</tr>
<tr>
<td>50</td>
<td>5.38</td>
<td>6.81</td>
<td>7.95</td>
<td>9.95</td>
<td>3.79</td>
</tr>
<tr>
<td>100</td>
<td>14.35</td>
<td>18.53</td>
<td>22.65</td>
<td>28.85</td>
<td>5.05</td>
</tr>
<tr>
<td>125</td>
<td>16.41</td>
<td>22.66</td>
<td>29.87</td>
<td>39.28</td>
<td>5.715</td>
</tr>
<tr>
<td>150</td>
<td>17.23</td>
<td>25.65</td>
<td>34.40</td>
<td>46.18</td>
<td>4.84</td>
</tr>
<tr>
<td>250</td>
<td>21.73</td>
<td>33.64</td>
<td>41.71</td>
<td>58.10</td>
<td>5.27</td>
</tr>
<tr>
<td>500</td>
<td>24.57</td>
<td>37.73</td>
<td>55.28</td>
<td>73.43</td>
<td>5.53</td>
</tr>
<tr>
<td>800</td>
<td>27.52</td>
<td>50.78</td>
<td>63.54</td>
<td>80.30</td>
<td>5.58</td>
</tr>
</tbody>
</table>

Table 7.8: Binomial: Speed (million nodes/s) to valuate 1000 options on RCHTX and on software using fixed point operators.

<table>
<thead>
<tr>
<th>Cores</th>
<th>75 Steps</th>
<th>800 Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Double</td>
<td>Single</td>
</tr>
<tr>
<td>1</td>
<td>2.13</td>
<td>2.58</td>
</tr>
<tr>
<td>2</td>
<td>2.39</td>
<td>2.90</td>
</tr>
<tr>
<td>3</td>
<td>2.66</td>
<td>3.22</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>3.94</td>
</tr>
</tbody>
</table>

Table 7.9: Relative Speed-ups in hardware comparing to the software version.
Figure 7.4: Charts for fixed point valuation based on binomial model.
(a) Speed-up against number of Tree Valuation Cores, to valuate 75-Step binomial tree, based on different precision versions.

(b) Speed-up against number of Tree Valuation Cores, to valuate 800-Step binomial tree, based on different precision versions.

Figure 7.5: Charts for relative speed-ups compared to the software implementation.
<table>
<thead>
<tr>
<th></th>
<th>1 Core</th>
<th>2 Core</th>
<th>3 Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slices</td>
<td>11411 (27%)</td>
<td>37849 (56%)</td>
<td>55418 (82%)</td>
</tr>
<tr>
<td>BRAMs</td>
<td>30 (10%)</td>
<td>58 (20%)</td>
<td>87 (30%)</td>
</tr>
<tr>
<td>DSPs</td>
<td>32 (33%)</td>
<td>64 (66%)</td>
<td>96 (100%)</td>
</tr>
<tr>
<td>Single</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slices</td>
<td>7340 (15%)</td>
<td>15817 (23%)</td>
<td>24183 (35%)</td>
</tr>
<tr>
<td>BRAMs</td>
<td>25 (8%)</td>
<td>46 (15%)</td>
<td>68 (23%)</td>
</tr>
<tr>
<td>DSPs</td>
<td>8 (8%)</td>
<td>16 (16%)</td>
<td>24 (25%)</td>
</tr>
<tr>
<td>Fixed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slices</td>
<td>6023 (8%)</td>
<td>8535 (12%)</td>
<td>11181 (16%)</td>
</tr>
<tr>
<td>BRAMs</td>
<td>25 (8%)</td>
<td>46 (15%)</td>
<td>68 (23%)</td>
</tr>
<tr>
<td>DSPs</td>
<td>8 (8%)</td>
<td>16 (16%)</td>
<td>24 (25%)</td>
</tr>
</tbody>
</table>

Table 7.10: The resource occupation rate for different implementation versions. The percentage shows utilisation of a specific FPGA resource.

Figure 7.6: Number of Slices utilised against Number of Core replications, based on Table 7.10.
Figure 7.7: Number of BRAMs utilised against Number of Core replications, based on Table 7.10.

Figure 7.8: Number of DSPs utilised against Number of Core replications, based on Table 7.10.
implementations I mentioned in Section \(\text{7.2}\). In the testing cases, I try two different kinds of multiplier implementations:

- On-chip slice resources.
- Digital signal processors (DSPs).

The results are shown in Table \(\text{7.11}\).

Figure 7.9 shows the corresponding chart based on data from Table \(\text{7.11}\). We are able to see that the DSP multiplier implementations generally consume less power than their corresponding Slice implementations. The reason is that DSPs contains highly optimised dedicated 18×18 two’s complement multipliers which are more power efficient than the direct implementation based on Slices.

To find out whether this power saving comes for free, I checked the clock frequency for different double precision implementations. The result is shown in Table \(\text{7.12}\). We can see that while the DSP implementations are more power saving than the Slice implementations, they are also slower in clock frequency; I call this the "Power-Speed Trade-off". Slice implementations are generally 5% faster but consumes about 2% more energy.

To test whether FPGA or GPU is more power efficient I run another test. This time exactly one option is processed and the processing speed per unit power is measured; noting that the fastest 1 Core fixed point FPGA implementation is used. The result is shown in Table \(\text{7.13}\).
<table>
<thead>
<tr>
<th></th>
<th>Slice</th>
<th></th>
<th>DSP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 Core</td>
<td>2 Cores</td>
<td>3 Cores</td>
<td>1 Core</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>3.246</td>
<td>4.482</td>
<td>5.719</td>
<td>3.093</td>
</tr>
<tr>
<td>Clock Freq (MHz)</td>
<td>161</td>
<td>160</td>
<td>160</td>
<td>153</td>
</tr>
</tbody>
</table>

Table 7.12: Power consumption (measured in Watts) VS Clock frequency (measured in MHz).

<table>
<thead>
<tr>
<th>No. of Steps</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>125</th>
<th>150</th>
<th>250</th>
<th>500</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>0.073</td>
<td>0.228</td>
<td>0.404</td>
<td>0.636</td>
<td>0.894</td>
<td>1.070</td>
<td>1.720</td>
<td>2.802</td>
<td>3.038</td>
</tr>
</tbody>
</table>

Table 7.13: Power efficiency (measured in Million Nodes/sec/Watt).

Figure 7.10: Power efficiency against No. of Steps, based on data from Table 7.13.
Figure 7.10 shows the chart based on the data from Table 7.13. It can be observed that in all the cases the FPGA implementation have a higher power efficiency than the GPU implementation; and the difference in power efficiency between the FPGA and GPU implementations become larger as the number of steps increases.

7.4 Summary

For the FPGA implementations I mainly concern two issues:

- The maximum processing speed that can be achieved by the Evaluation Core. I concern this because it is the most complex component in the overall architecture, and the performance of the system will be depend on it.
- The real speed-up that the system can achieve. I concern this as this can be used to measure how useful the system is for real world applications. I believe the performance of the system will depend on both the Control Module logic that manipulates the Evaluation Core and the Evaluation Core itself.

Based on these understandings I try to optimise my design and build the testing cases around them. The first testing case aim to compare the maximum speed-ups of the FPGA implementations with the corresponding GPU and PC implementations. In this testing case I only consider the maximum processing speed that the Evaluation Core can achieve, this can be derived from the clock frequency it is running at. Namely, the Evaluation Core will produce one result each at each clock cycle. On the other hand for the GPU and PC implementations I setup a testing scenario that allows them to run at maximum speeds. The result turns out the FPGA Evaluation Core are doing very well to be only 27% –35% slower than the newer generation GPU – the Geforce 8600GTS which based on the 80nm technology. The FPGA Evaluation Core can run 2 times faster than the Geforce 7900GTX GPU which is based on the same 90nm technology as itself. Compared to the fastest reference PC, a speed up of over 160 times can be achieved by our FPGA Evaluation Core.

Being sure that the FPGA Evaluation Cores are good enough, I then try to test how fast the FPGA system can really run under real-world scenarios. Tree Valuation Cores are physically duplicated to test the parallelism that can be achieved under real-world testing cases. The result turns out that the Evaluation Cores are generally running 2.5 times faster than correspondingly full-tree implementations. This is more or less the expected result due to the software-hardware communication overhead and possible delays in control logic. The fact that the FPGA is not running at its fastest speed due to the low target clock rate also needs to be taken into consideration. Fixed point implementations are generally the fastest. This time a one core fixed point FPGA implementation can achieve a maximum speed-up of 5 times. If the cores are duplicated it can achieve a potential speed-up of 84 times. which is broadly acceptable. However the result is still 50% slower than the GPU implementation on the Geforce 8600GTS processor. It is worth noting that the FPGA implementations still have unique advantages of extremely low power consumptions, and I believe this aspect will be taken into consideration in a real-world scenario.
Chapter 8

Conclusions and Further Work

I began this project with the aim to explore possible designs for tree based models on both FPGAs and GPUs and study their characteristics to find an optimal solution for real applications. In this chapter I review the project and address any further work that can be done on this topic.

8.1 Project Review

At the beginning of the project I studied the properties of the tree-based models. I seek to exploit inherent parallelism for tree-based models and tried possible optimisations to save memory usage. Optimised software prototypes of the binomial and trinomial models are proposed based on the properties of the tree model I learned. The software prototype is later used as a blueprint to develop the FPGA and GPU implementations. Two parallel pipelined architectures and two GPU models are designed based on both the properties of the tree models and the software prototype.

The designs are later used for actual implementations, they are considered to be generally adoptable for problems in similar domains.

Secondly I propose the implementations of fully pipelined Evaluation Cores on FPGA for the tree models. A binomial design and a trinomial design is illustrated; both of them are highly optimised and fully pipelined. The implementations of the designs are later evaluated by a comparison to 2 reference PC and 2 GPU implementations of the same models. The reason I am paying so much attention to the Evaluation Core is because it is considered to be the most complex component and is most likely to become the performance bottleneck of entire system. I also considered the portability in the implementations of the Evaluation Cores. The core logic can adopt easily different real number representations (single, double and fixed point) by changing only two variable types. The implementation is also based on platform-independent libraries, which means it can be targeted to different FPGA platforms easily.

Thirdly I start with implementing a straightforward mapping to hardware from the software prototype, in order enable the tree valuation procedure in hardware. In the mapping procedure all the for-loops are changed to equivalent
while-loops to increase efficiency. Then I modified the binomial design to cope with Greeks valuation with almost no additional overhead. After that I tried several means to optimise the tree valuation procedure, in particular I managed to reduce number of memory reads from three to one in the inner-most loop of the trinomial tree model. This is a significant improvement as the inner loop is iterated $O(N^2)$ times, where $N$ is the number of steps of the options to be valued. Another optimisation is to pipeline the Evaluation Core, making data continuously feed into the pipeline and read out simultaneously. This guaranteed a higher throughput. In particular, is the number of stages in the pipeline is $P$, the throughput can be improved by a factor of $P$. I also design and implement a replicated architecture to allow multiple options to value in parallel. The design is later proved to provide linear growth of speed up with the number of replications of cores in the evaluation chapter.

Fourthly I propose architectures to valuate tree models in GPUs. Two different designs are illustrated. One is based on GLSL and is run on a Geforce 7900GTX to valuate binomial models. The other are two CUDA implementations which support binomial and trinomial valuations respectively. I optimised the CUDA version by partitioning a tree into smaller trunks so that the double buffering method can be adopted in high speed shared memory. These two approaches exploit maximum inherent parallelism in the tree models and allow significant speed-ups without possible data loss caused by parallel operations.

Lastly I evaluated my implementation of the Evaluation Core based on comparisons to two reference PCs and two GPUs. I called the speed-ups achieved by the Evaluation Core the "absolute speed-up" and use it as the upper bounds of speed-up for all possible FPGA implementations are acquired. The strength and weaknesses of FPGAs and GPUs are also addressed in this stage as all the testing cases are aiming exploit maximum throughput on device. The result shows that in an ideal case the FPGA can run two times faster than a Geforce 7900GTX GPU if appropriate architecture is used. However the FPGA is around 50% slower than the Geforce 8600GTS GPU under a real scenario.

8.2 Project Remarks

The testing results shows that the FPGA implementations outperform the PC implementation when the number of steps of the options to be valuated is greater than 75. Therefore that I suggest the software side to check the number of steps of the options to be valued before sending requests to hardware. If the the number of steps is smaller than 75 steps then the valuation should be done in software to assure maximum throughput. It can also be observed that since the speed-ups against number of cores obey a linear pattern, we are able to replicate many cores on a single FPGA without compromising efficiency significantly, provided that there is enough on-chip resources and the software is able to sent enough requests to keep the FPGA busy.

It can also be seen from the result that the actual speed-ups can potentially be close to the speed-up upper bounds, but they are now around 2.5 times slower. Under real-world scenarios the single core fixed point FPGA implementation can achieve a maximum speed-up of 5 times and if duplicated it can achieve a speed-up of 84 times. This speed-up is broadly good, but still 50% slower than the GPU implementation on the Geforce 8600GTS processor.
On the other hand, the FPGA implementations have extremely low power consumptions compared to the GPU ones. A 3-core double precision FPGA implementation requires just over 2 Watts of power supply while the Geforce 8600GTS GPU needs over 71 Watts. Therefore I believe depending on the actual requirements, for example, the option pricing volume demand or the power supply in the data center, an appropriate choice can then be made between FPGAs and GPUs to guarantee an optimal solution.

8.3 Further Work

- **Apply run-time reconfiguration to the FPGA designs.**
  Most of the latest FPGAs are based on SRAM technology that allows run-time reconfiguration. Studies have shown that run-time reconfigurations can improve performance [25], increase functional density [35] and reduce power dissipation [18] [26]. The unused part on FPGA can be switched off to save power and make space for a new module [2].

  One of the possible implementations of run time reconfiguration in my project scope is to control number and precision of Evaluation Cores. Evaluation Cores can be switched off if not needed, and lower precision cores can be used for tasks with tighter timing constraint and lower demand for precision.

- **More sophisticated power consumption analysis.**
  So far the power consumption analysis is based on the Xilinx Xpower Estimator, the numbers obtained may not be accurate enough to give a detailed specification of the exact power consumption rate. Such specification can be crucial for real applications. I expect more accurate power estimation to be done by the Xpower tool provided with Xilinx ISE.

- **Design tool generate both FPGA and GPU implementations.**
  Ideally there should be a design tool to generate both FPGA and GPU implementations automatically based on a special description language (SDL) that is easy to use. C style syntax should be adopted to this SDL to allow easy adoptability and potential portability. Such language is already available for FPGA implementations, for example, a solution is given to allow C style code to describe both behavior level and cycle-accurate level details [14]. As the GPU programming languages are mostly C-based, C or C++ code can be transformed to GPU-supported GLSL code by source-to-source translators automatically [12]. However new generation GPU programming languages like CUDA might require a effective task scheduler in order to run efficiently. How to obtain such task scheduler automatically and how to combine the FPGA and GPU code generators will be an interesting area to study.

- **Comparisons to AMD/ATI GPUs, Altera FPGAs and the Cell Broadband Engine.**
  Comparisons to other devices such as AMD/ATI GPUs, Altera FPGAs and the Cell Broadband Engine might exhibit interesting results and provide better performance. These can be done if the devices are available.
• FPGA-GPU collaborated work.
In the project scope I haven’t consider the possibility to deploy mote-carlo methods to valuate American style options.
American style options can not be priced easily by Mote-Carlo methods, but it does not mean it is impossible. The problem can be solved by the least square method [19]. First an $N \times T$ Monte-Carlo simulations are done to sample $N$ paths with $T$ steps. The result is stored in an $N \times T$ two-dimensional matrix. Then we walk through the matrix column by column. At each column we use least square curve fitting method to obtain a function that determines the value of each node within the column. There are already existing studies to accelerate Monte-Carlo simulations based on FPGAs [27]. The problem is how to store the $N \times T$ matrix when $N$ is a few millions and $T$ is also large. FPGAs do not have large enough RAMs to store a matrix of that size. Even they do, how can such matrix be accessed and processed efficiently is another problem. FPGA platforms like RCHTX support the HyperTransport (HTX) interface and have direct access to the entire host system memory space [7]. This allows us to store the matrix in the system memory. Monte-Carlo simulations can be done in FPGA and results can be written directly to the system memory to build the matrix.

Another problem is how to implement the lease square curve fitting. Even the simplest least square method involves a multiplication of an $N \times 3$ matrix and a $3 \times N$ matrix. Such matrix multiplication is possible to be done in FPGA, but will typically involve sophisticated buffering hence hard to design. On the other hand, matrix multiplications can be done easily by GPUs. It should be able to implement a design that allows the matrix multiplication to be done in a GPU feed the result back to FPGA for further processing. Though it can be expected that moving the data from/to memory will be the major overhead and whether such hybrid system will work is yet to be determined by further experiments.

8.4 Final Remarks
Having done this project I would like review the opening scenario. What difference could it make if CBOE adopts my GPU solution? If we assume that CBOE is using a PC that has a RCHTX development board with my double precision 3-Core FPGA implementation running on it. Then it can finish all the option pricings within 0.12 seconds.

If the PC CBOE have is equipped with a Geforce 8600GTS GPU which runs my CUDA implementation, we can expect a higher speedup. As the number of options to be valued is large (690 options/s), GPU can be fully utilised to achieve a maximum speed up of 250 times compared to the software version. The options that previously take 1.6 seconds to price can now be priced in 0.03 second. We are able to turned a infinite delayed system to a delay-free one with the help of a GPU that costs less than 50 pounds.
Bibliography


