University of Crete

Master Thesis

A reconfigurable architecture for Black-Scholes Option pricing using finite-difference schemes

Author:
Chatziparaskevas Georgios

Supervisor:
Dr. Papaefstathiou Ioannis

Examining Board: Dr. Dollas Apostolos, Dr. Papaefstathiou Ioannis, Dr. Pnevmatikatos Dionysios

May 6, 2010

Acknowledgements

The current thesis is the distillate of arduous efforts that began in March 2008. Throughout this period I was embraced by all the current and past members of the Microprocessors and Hardware Laboratory. This is a great opportunity to express my appreciation to all of them for their acceptance and nice moments we have spent together. I would also like to thank my supervisor for the confidence he showed to my decisions. The current thesis wouldn't have been accomplished without his guidance and financial endorsement. I am also pleased to thank the other two members of the dissertation committee Dr. Apostolos Dollas and Dr. Dionysis Pnevmatikatos for the time spent on the evaluation of the current thesis. Finally it couldn't be possible to forget the unsung heroes of this project, my family and my girlfriend Christodoulou Andri for their support and tolerance.

Contents

1 Options in finance **3**
- 1.1 Definition . . . 3
- 1.2 Types of options . . . 3
- 1.3 Option pricing . . . 3
  - 1.3.1 The Black-Scholes model . . . 4
  - 1.3.2 The Binomial model . . . 5
  - 1.3.3 Monte Carlo methods . . . 5

2 Option Pricing with the Black-Scholes model **6**
- 2.1 The Black-Scholes equation . . . 6
- 2.2 General boundary conditions . . . 6
- 2.3 Conversion to the heat diffusion equation . . . 8
- 2.4 Analytical solution . . . 8
- 2.5 Numerical solution . . . 9

3 Finite differences **10**
- 3.1 Introduction . . . 10
- 3.2 Finite differences for the Black-Scholes equation . . . 10
  - 3.2.1 Grid definition . . . 11
  - 3.2.2 Boundary conditions . . . 12
  - 3.2.3 Selection of the best-fitting boundary conditions . . . 13
  - 3.2.4 Explicit scheme . . . 14
  - 3.2.5 Implicit scheme . . . 17
  - 3.2.6 Crank-Nicholson scheme . . . 19
  - 3.2.7 Parallelization of finite-difference schemes . . . 22

4 Solving Tridiagonal systems **24**
- 4.1 LU-Decomposition . . . 24
- 4.2 Cyclic Odd-Even Reduction . . . 25
  - 4.2.1 Traditional odd-even reduction . . . 27
  - 4.2.2 The new odd-even reduction variant . . . 28
  - 4.2.3 Comparison of tridiagonal system solution algorithms 29

5 Towards the proposed reconfigurable architecture 31
5.1 Retrospection on the attacked problem 31
5.2 Related work 32
5.2.1 Related work on high performance Option Pricing . . 32
5.2.2 Motivation from other approaches 33
5.3 The initial approach 33
5.3.1 Algorithmic level 33
5.3.2 Implementation level 34

6 The proposed architecture 37
6.1 Architecture of a single core 37
6.1.1 Local memory - Data organization 37
6.1.2 Memory controller 39
6.1.3 Floating point unit 41
6.2 Top-level architecture 44
6.2.1 Interconnection network 45
6.2.2 Input-Output 46

7 Results and Evaluation 47
7.1 Resource utilization 47
7.1.1 Estimation methodology 47
7.1.2 Analysis 48
7.2 Speed comparison 49
7.2.1 Methodology 49
7.2.2 Analysis 50

8 Conclusions and future Work 53

A Core instruction set 54

Abstract

Option pricing is a fundamental problem in modern economics that entails rigorous calculations. The current thesis presents a reconfigurable system for option pricing that exploits the capability of FPGAs to execute massively parallel calculations to speedup the process. The system is based on a parallel architecture that can accommodate the basic finite-difference methods used for option pricing with the Black-Scholes model. The explicit and Crank Nicholson schemes for Black-Scholes option pricing were implemented. The designed system consists of a variable number of interconnected processing units able to achieve a fair performance gain over an up-to-date dual-core CPU.

Introduction

As financial markets grow bigger and more complex the volume of calculations is escalating. Derivative markets in particular have seen an outstanding growth since the 1970's when the first types of option contracts were openly traded. Figure 1 shows the annual volume of traded option contracts at the Chicago Options Board Exchange and is indicative of the development of derivative markets. Meanwhile the mathematical models used to price such financial instruments are becoming more and more complex and computationally intensive in order to address sophisticated derivative types. Greater performance must be achieved on the other hand in terms of speed and power efficiency. Financial institutions turned to high performance computing (HPC) solutions such as clusters, grids and recently GPUs and FPGAs to gain speed edge. FPGAs have proven their ability to speed up processing in various scientific fields such as bioinformatics and image processing. They are also power efficient, occupy minimal space and their acquisition and maintenance cost is low.

An FPGA-based high performance solution for option pricing is proposed in the current thesis. It consists of a parallel reconfigurable architecture suitable for finite-difference schemes used extensively in financial derivatives pricing. The implemented schemes include the explicit and Crank-Nicholson methods. Other schemes can also be implemented easily.

The parallel architecture includes a number of interconnected cores ordered in ring topology. Each core has a local memory, a programmable Von-Neumann memory controller, and a hardwired dataflow-based floating point unit. The implementation of an algorithm over this architecture requires the programming of the memory controller with assembly-like instructions and the "rewiring" of the floating point unit. This model tried to combine the flexibility of a control-flow architecture for memory addressing and inter-core communication with the speed of the dataflow approach for the floating point operations. Transferring the algorithmic complexity to software level results in simpler core architecture and reduced resource utilization. The Mapping of an algorithm to as series of instructions though cannot exploit possible parallelization at instruction level. The reduced resource utilization per core allowed for massive process-level parallelization, compensating for no instruction level parallelism: More than 64 cores can

Figure 1: Annual volume of traded options in millions at the Chicago Options Board Exchange (source: Chicago Board Options Exchange 2008 Annual Report)

be fitted in high-end reconfigurable devices such as the Virtex^{TM} 5 family achieving up to eigh-fold speed-up compared to an up-to-date dual-core CPU.

Great effort has also been focused on the selection of the implemented option-pricing algorithms. The strictly economical techniques have been rejected in favor of finite differences that can be applied in a broad range of scientific fields. The usability of the proposed reconfigurable architecture can be extended well beyond option pricing in this way. In the context of the implemented Crank-Nicholson finite-difference scheme we also had the chance to explore methods for solving tridiagonal systems efficiently over parallel architectures, contributing a variant of cyclic odd-even reduction (a parallelizable method for solving tridiagonal systems) that requires smaller machine precision than the original method without compromising speed of execution.

Chapter 1

Options in finance

1.1 Definition

An option is a contract which entitles its buyer (holder) with the right but not the obligation to buy or sell an asset (underlying asset) at a predefined price (strike price) and at some future moment before or at a predefined date (expiry date). The seller (issuer) of the contract is obliged to sell or buy the asset when the holder chooses to exercise his right. Options for buying assets are named call options and those for selling, put options.

A simplified example of the usage of an option is that of a producer of wheat who wants to fix a price for his future crop buying the right to sell it to a wheat trader at a specified price when it is ready. If the price of wheat falls, exercising his right shall be profitable. If the price rises on the other hand, he can sell his crop elsewhere at higher price trading off the price he has paid for the option. No matter what happens he has mitigated the risk of falling wheat prices. In other words options are used for offsetting the exposure to price fluctuations and other risks (hedging).

1.2 Types of options

The most important option types are European and American. European options can be exercised only at the expiry date of the option. American options can be exercised anytime up to the expiry date. Other types can be exercised only at specific dates before or on expiry (Bermudan), or when the value of the underlying asset reaches a specified price (Barrier). Options not included in these categories are characterized as exotic.

1.3 Option pricing

Options are financial instruments whose value derives from the value of the underlying asset. Their price depends on various factors, including the

current price of the underlying asset (stock price), the time remaining until the expiration date, the price volatility of the asset and the strike price. The way these drivers affect the price of an option is determined using option pricing models. The most important are the Black-Scholes and the Binomial model. Options can be priced with these models or with Monte-Carlo techniques. Option pricing and especially the Black-Scholes model is the main issue addressed by the current thesis. General information about the other methods are given below in order to get an intuition about all Option pricing methods available.

1.3.1 The Black-Scholes model

It was introduced by Fischer Black and Myron Scholes in 1973 [1] (a primitive version is attributed to [2]) and is founded on the following assumptions:

- The asset price follows a lognormal random walk (geometric Brownian motion) or equivalently the return (change of price) of the asset underlying the option is modelled as a Wiener process. The return consists of two components, a deterministic called drift and a random with constant volatility. The former quantifies the tendency of the prices to get higher and the latter represents the unexpected change of the asset price in the course of time. The random component is Markovian implying that all history of changes is contained in the current asset price and price is adapted instantaneously to new market conditions (efficient-market hypothesis) [3].

- There are no arbitrage opportunities, that is to say no opportunities to make instantaneous risk-free profit. To gain intuition on this assumption consider an option and an underlying asset whose values depend on the same source of uncertainty. We can form a portfolio consisting of the asset and the option which eliminates this source of uncertainty (in other words it becomes "riskless"). The Black-Scholes model determines a fair price for the option so that this portfolio does not yield any profit (risk-free profit).

Other conditions include the divisibility of the asset and the absence of transaction costs and dividents payment by the underlying asset. The Black-Scholes model considers that the value V of an option is a function of the time t and the asset price S ($V = f(S, t)$) that satisfies a second order partial differential equation known as the Black-Scholes formula. This equation can be solved analytically or approximated with numerical methods. Details on its solution are presented in the following chapters.

The main advantage of the Black-Scholes model is its speed. It allows the calculation of option prices quicker than the Binomial model. Its main limitation on the other hand is that it cannot provide a closed form expression for the price of American style and exotic options.

1.3.2 The Binomial model

It was introduced by Cox, Ross and Rubinstein in 1979 [4] and belongs to the family of Lattice methods. In essence it is a numerical technique for pricing options that calculates a tree of possible option values as time progresses. The time to expiry date is divided in discrete time intervals. At each time step it is assumed that the stock price shall increase or decrease with some certain probability. This procedure produces a binomial tree of stock values, upon which the corresponding option values are calculated afterwards starting from the expiry date and moving backwards. The value of the option is the one corresponding to the root of the binomial tree of stock prices.

The binomial model is based upon the assumption of risk neutrality as the Black-Scholes model and the modelling of the movement of the stock price as discrete random walk. Its main advantage over the Black-Scholes model is that it can be used to accurately price American options and handle complex conditions, albeit it is far slower.

1.3.3 Monte Carlo methods

Monte Carlo option pricing Considers that the price of the asset underlying the option follows geometric Brownian motion as the Black Scholes and binomial model. A simplified approach of Monte Carlo methods generates large numbers of random paths for the underlying asset price. Afterwards it calculates the price of the option over them as the exercise price of the option at that moment (equal to the payoff function of the option). The option price is calculated as the average of these prices and discounted to the present moment at risk-free interest rate, as the initial prices were calculated for the expiry date of the option.

The accuracy of Monte-Carlo methods depends on the number of generated asset price paths. When other sources of uncertainty exist apart from the asset price, they must be modelled as random processes and new paths must be generated for them for each path of the asset price. Thus Monte-Carlo methods are vary adaptive to options or other derivatives with various sources of uncertainty.

Chapter 2

Option Pricing with the Black-Scholes model

2.1 The Black-Scholes equation

The Black-Scholes model is summarized in the following equation:

$$rV = \frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} \tag{2.1}$$

where:

S : Stock price ($0 \leq S < \infty$).

r : Risk-free rate of interest. It represents the theoretical rate of return of a risk-free investment such as government bonds over a period of time.

σ : Volatility of the stock. It quantifies the option price fluctuation till the expiry date.

t : Time interval after the signing of the option ($0 \leq t \leq T$, T: expiry date). $T - t$ is the time remaining until the expiration date.

Parameters r and σ are considered constant until the expiry date T of the option. Equation 2.1 is a second order partial differential equation that is produced upon the assumptions mentioned in the previous chapter. The way it is produced exceeds the purposes of the current thesis and thus it is omitted.

2.2 General boundary conditions

The solution of the Black-Scholes equation is a boundary value problem, that is to say a possible solution must satisfy certain constraints called boundary

conditions along with the partial differential equation itself. The boundary conditions describe the behavior of the solution on the boundaries of the independent variables domain. These boundary conditions for the case of European call options and their real-world interpretations are given below:

- $V(0,t) = 0$

  The value of an option over zero price asset is zero.

- $V(S,T) = max(S-K,0)$

  The value of a call option on the expiry date is equal to the profit made by the holder if he exercises the option (buys the underlying asset) - this profit is equal to the price of the asset S minus the strike price K and constitutes the payoff function of the option. This condition is referred to as terminal condition as it concerns the expiry date of the option [3].

- $V(S,t) = S - Ke^{-r(T-t)}, S \to \infty$

  When S tends to infinity, the value of the option is the asset price minus the exercise price discounted by the risk-free interest rate (interest rate of a risk-free investment such as depositing the same amount of money K to a bank account) [5]. A simpler condition argues that $V(S,t) = S, S \to \infty$ as the influence of the strike price K on the option value diminishes when $S \to \infty$.

The above conditions concern the value of the solution at the boundaries of the domain and are known as Dirichlet conditions. Conditions that concern the value of the derivative of the solution on the boundaries are known as Von Neumann conditions and for the case of European call options are given below:

- $\partial^2 V/\partial S^2 = 0, S \to \infty$

  This condition comes from the Dirichlet conditions $V(S,t) = S$ and more precisely $V(S,t) = S - e^{-r(T-t)}$ when $S \to \infty$ (it is equivalent to $\partial V/\partial S = 1, S \to \infty$) [6], [7].

- $\partial V/\partial S = 0, S = 0$

  Equivalent to the Dirichlet condition $V(0,t) = 0$.

2.3 Conversion to the heat diffusion equation

The Black-Scholes equation can be converted to a simpler form known as the heat diffusion equation:

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2} \tag{2.2}$$

The above partial differential equation is used to model the spread of heat in a single dimension (x) inside a medium over time (τ). It is simpler than the Black-Scholes equation in the sense that the coefficients of the partial derivatives are constant resulting in more convenient numerical approximations. In order to transform the Black-Scholes PDE to the heat diffusion equation we make the following transformation of variables:

$$\begin{aligned} S &= K \cdot e^x \\ t &= T - \tau / \frac{1}{2} \cdot \sigma^2 \\ V(S,t) &= K \cdot e^{-\frac{1}{2}(k-1)x - \frac{1}{4}(k+1)^2 \tau} u(x,\tau) \end{aligned}$$

with $k = r/\frac{1}{2}\sigma^2$

Applying the above transformations, the BS equation is transformed to equation 2.2 with $-\infty < x < +\infty$ and $0 \leq \tau \leq \sigma^2 T/2$. The new Dirichlet boundary conditions shall be:

$$\begin{aligned} u(x,0) &= max(e^{\frac{1}{2}(k+1)x} - e^{\frac{1}{2}(k-1)x}, 0) \\ u(x,\tau) &= 0, x \to -\infty \\ u(x,\tau) &= (e^x - e^{r \cdot \tau / \frac{1}{2}\sigma^2}) \cdot e^{\frac{1}{2}(k-1)x + \frac{1}{4}(k+1)^2 \tau}, x \to \infty \end{aligned}$$

The above conditions derive from the Dirichlet conditions of the Black Scholes equation after making the transformations that lead to the heat diffusion equation.

2.4 Analytical solution

The Black-Scholes equation can be solved analytically only for European call and put options. When it comes to American or other exotic options there is no analytical solution and the option price can only be approximated with numerical methods. The current section provides a brief description of the analytical solution meant only for comparison with the finite-difference family of methods that is described afterwards.

The analytical solution of the Black-Scholes equation that calculates the value of a European call option has the following form:

$$C(S,t) = S \cdot N(d_1) - K \cdot e^{-r(T-t)} \cdot N(d_2)$$

with

$$d_1 = \frac{ln(\frac{S}{K}) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}}$$
$$d_2 = d_1 - \sigma\sqrt{T-t}$$

$N(x)$ is the cumulative distribution function of a standard normal random variable x. For the case of the European put option the analytical solution takes the following form:

$$P(S,t) = K \cdot e^{-r(T-t)} \cdot N(-d_2) - S \cdot N(-d_1)$$

The above closed form solution require complex computations involving logarithms and exponentials. Despite the fact that they can be implemented in software in a straightforward way, designing a reconfigurable system to implement these solutions is impractical not only in terms of complexity but also because only European put and call options will be handled.

2.5 Numerical solution

The most important class of methods for the numerical solution of the Black-Scholes equation is finite differences. There also exist other more sophisticated techniques such as the radial basis functions approximation [8] and finite elements [9] which are more complicated and are more suitable for software implementation over generic architectures. The next chapter is dedicated to the finite-difference methods implemented on the reconfigurable architecture proposed by the current thesis.

Chapter 3

Finite differences

3.1 Introduction

Finite difference methods are numerical techniques for approximating the solution of partial differential equations. They are very flexible as they can approximate the solution of partial differential equations that cannot be solved analytically and thus price virtually every financial derivative instrument whose value can be described by such an equation. The basic ideas behind this class of methods are:

- The independent variables found in the partial differential equation (PDE) are discretized. The discretization process generates a grid of points each one of which corresponds to a unique combination of values of the indepenent variables. After the discretization of the continuous space of the independent variables, the value of the function described by the PDE is calculated over the points of the grid.
- The partial derivatives inside the PDE are approximated with finite differences of the values of the function to be found over consecutive points divided by the finite differential of the respective independent variable.

The above concepts shall become more tactile to the reader during the description of finite differences as applied to the Black-Scholes PDE for the case of European call options.

3.2 Finite differences for the Black-Scholes equation

Finite differences were first applied to option pricing by Brennan and Schwartz [10]. This section describes the basic principles of these methods as applied to the European call options pricing with the Black-Scholes model.

The independent variables of the Black-Scholes equation are the stock price S and the time t that has elapsed after the option was signed. These two variables form a 2-D space, over which the function that describes the option price $V(S,t)$ forms a three dimensional curve. Instead of looking for a closed form expression for the function $V(S,t)$, finite-difference methods approximate its value over discrete points of this space that form a grid.

3.2.1 Grid definition

The time t takes values in the domain $[0,T]$ - time is zero at the moment when the option contract is signed and reaches T at the expiry date of the option. We shall substitute t with $\tau = T - t, 0 \leq \tau < T$ in order to transform the terminal condition $V(S,t) = max(S-K,0), t = T$ to the initial condition $V(S,\tau) = max(S-K,0), \tau = 0$. The necessity of this substitution shall be clarified during the description of the finite-difference schemes. The Black-Scholes PDE after this transformation shall be:

$$rV = -\frac{\partial V}{\partial \tau} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} \tag{3.1}$$

The asset price S takes values from the domain $[0,\infty]$ as an asset can have zero or positive price. In order to set a finite number of discrete points in the dimension of asset price we must make the hypothesis that the asset price takes a finite maximum value S_{max}.

Apart from the selection of an upper boundary value for S, another issue that arises is the degree of granulation of the grid, or in other words how many points should the grid include. These issues are addressed in different ways depending on the type of finite-difference method used. For now we shall consider that the space (S,τ) shall be divided in N_τ points in the time dimension and N_S points in the S dimension. The grid shall also be bounded to the domain $[0,S_{max}]$ for S and $[0,T]$ for τ.

Another issue to be addressed before the grid is thoroughly defined is whether the points shall have constant distance from each other or shall be positioned using some special pattern. Sophisticated grid patterns can increase accuracy and minimize the number of grid points for given accuracy goals, whereas a grid with its points placed at fixed distances (uniform mesh) has the advantage of algorithmic simplicity. As the proposed option pricing system shall be implemented on reconfigurable hardware simplicity was considered vital and the linear grid was selected.

For a uniform mesh of size $N_S \times N_\tau$ and amplitude $[0,T]$ for τ and $[0,S_{max}]$ for S the distance dS between two consecutive points shall be $dS = S_{max}/N_S$ in the S direction and $d\tau = T/N_\tau$ in the τ direction (See also Fig. 3.1). The value of the function calculated over the point (S_i,τ_j) shall be denoted as $V_{i,j}$ and shall approximate the real value of the function at this point: $V_{i,j} \approx V(S_i,t_j)$.

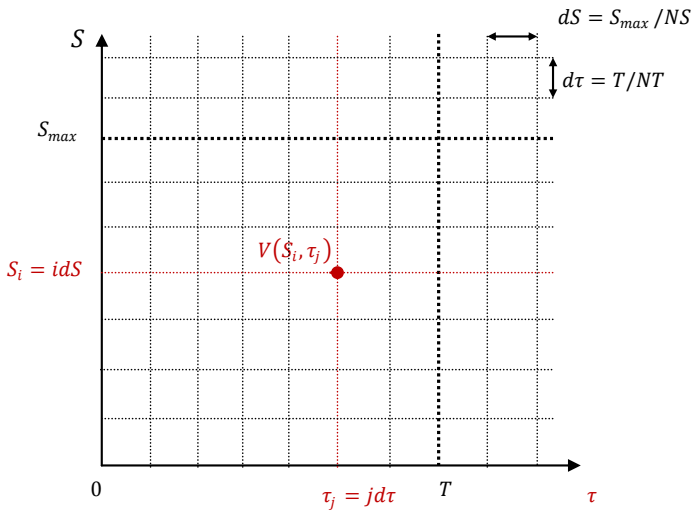


Figure 3.1: Finite grid over the independent variables space (S, τ) of the Black Scholes PDE.

The Black-Scholes equation over the discrete space shall have the following form:

$$rV_{i,j} = -\frac{\partial V}{\partial \tau} + rS_i\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S_i^2\frac{\partial^2 V}{\partial S^2} \tag{3.2}$$

The next step after defining the grid is to approximate the partial derivatives found in the PDE. The way they are approximated as finite difference quotients leads different algorithms of approximation. If we consider a function f over a point x and a small positive amount h there are three types of finite differences:

- Forward difference: $\Delta f = f(x+h) - f(x)$
- Backward difference: $\Delta f = f(x) - f(x-h)$
- Central difference: $\Delta f = f(x+h) - f(x-h)$

There are three main finite-difference methods (schemes): the explicit, the implicit, and the Crank-Nicholson which make different use of the above types of finite differences to approximate the partial derivatives of the PDE to be solved.

3.2.2 Boundary conditions

The general boundary conditions for the Black-Scholes equation defined in the previous chapter must be mapped onto the finite grid used by the finite differences. These conditions describe the behavior of the discretized Black-Scholes equation over the borders of the grid. The Dirichlet conditions for the discretized Black-Scholes equation are the following:

- $V_{i,0} = max(S_i - K, 0)$
  It constitutes the initial condition and is equal to the value at the

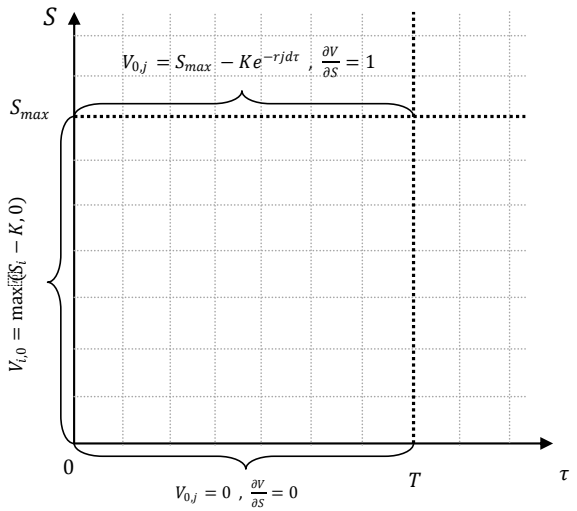


Figure 3.2: Boundary conditions on the borders of the finite grid.

expiry date ($\tau = 0$) is the payoff function of the call option. This is the terminal condition of the conntinuous Black-Scholes equation described in paragraph 2.2.

- $V_{N_S,j} = S_{max} - Ke^{-rjd\tau}$
  This condition determines the value of the option for the upper bound of S and maps the asymptotical option price for $S \to \infty$ of the continuous space (paragraph 2.2) to the finite grid. A simpler but less accurate boundary condition is $V_{N_S,j}) = S_{max}$.

- $V_{0,j} = 0$
  This is the lower boundary condition for $S = 0$. The intuition behind it is that for a zero asset price the call option shall definitely not be executed (it is "out of the money" in economic jargon), thus its price is zero!

The Von Neumann conditions (paragraph 2.2) that determine the value of the derivative of V on the boundaries have the same form but correspond to the finite boundaries $[0, S_{max}]$ (see also fig. 3.2).

3.2.3 Selection of the best-fitting boundary conditions

The implemented finite-difference methods make use of the Von Neumann conditions. This is done because no extra values have to be calculated for the boundaries. The Dirichlet conditions for S_{max} derive from a complex formula and implementing it on reconfigurable hardware is a problem of its own. A way to use these conditions is pre-calculating them on the host computer and sending them to the FPGA option pricer. Nevertheless this choice comes with an increased IO delay toll. Von Neumann conditions on the other hand are equivalent with Dirichlet ones as they are produced by them and can be subtly incorporated to the implemented method.

3.2.4 Explicit scheme

Introduction

The Explicit scheme uses a forward difference quotient to approximate the time derivative:

$$\left.\frac{\partial V}{\partial \tau}\right|_{S=S_i,\tau=\tau_j} \approx \frac{V_{i,j+1} - V_{i,j}}{d\tau} \tag{3.3}$$

It also uses central difference for both the first and second order derivatives in the asset price direction:

$$\left.\frac{\partial V}{\partial S}\right|_{S=S_i,\tau=\tau_j} \approx \frac{V_{i+1,j} - V_{i-1,j}}{2dS} \tag{3.4}$$

$$\left.\frac{\partial^2 V}{\partial S^2}\right|_{S=S_i,\tau=\tau_j} \approx \frac{V_{i+1,j} + V_{i-1,j} - 2V_{i,j}}{dS^2} \tag{3.5}$$

Analysis

The explicit scheme calculates the option values $V_{i,j+1}$ of the $j+1$-th time moment(next time step) over all the internal (not boundary) values of the asset price ($0 < i < N_S$) using the option values of the current time step $V_{i,j}$ 3.3. The option values for the first time step ($j = 0$) are known from the initial condition $V_{i,0} = max(S_i - K, 0)$.

Substituting equations (3.3) and (3.4) to the discretized Black-Scholes equation (eq. 3.2) we reach the following equation:

$$\begin{aligned} rV_{i,j} = &- \frac{V_{i,j+1} - V_{i,j}}{d\tau} + rS_i \frac{V_{i+1,j} - V_{i-1,j}}{2dS} \\ &+ \frac{1}{2}\sigma^2 S_i^2 \frac{V_{i+1,j} + V_{i-1,j} - 2V_{i,j}}{dS^2} \end{aligned} \tag{3.6}$$

and after executing the computations one reaches the following formula that calculates $V_{i,j+1}$ using the known $V_{i-1,j}$, $V_{i,j}$, $V_{i+1,j}$:

$$V_{i,j+1} = a_i \cdot V_{i-1,j} + b_i \cdot V_{i,j} + c_i \cdot V_{i+1,j} \tag{3.7}$$

The parameters a_i, b_i, c_i for $i < 0 < N_S$ remain constant in the course of time and have the following values:

$$\begin{aligned} a_i &= -\frac{1}{2}rid\tau + \frac{1}{2}\sigma^2 i^2 d\tau \\ b_i &= 1 - rd\tau - \sigma^2 i^2 d\tau \\ c_i &= \frac{1}{2}rid\tau + \frac{1}{2}\sigma^2 i^2 d\tau \end{aligned}$$

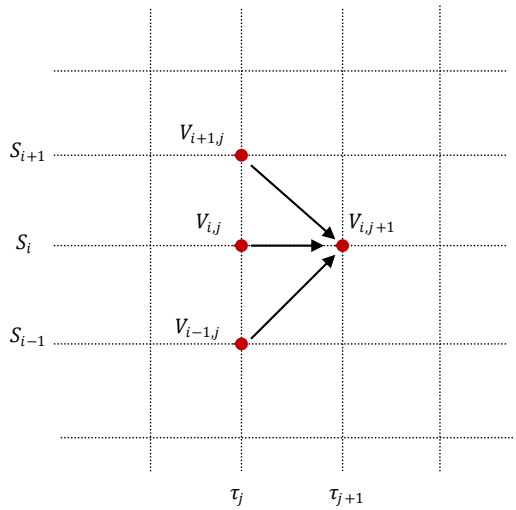


Figure 3.3: Explicit scheme: flow of option value calculations over the grid.

The parameters a_i, b_i, c_i on the boundaries $i = 0, i = N_S$ shall be calculated using the Von Neumann conditions. For the upper boundary ($i = N_S$) we have the condition:

$$\left.\frac{\partial V}{\partial S}\right|_{S=S_max} = 1 \tag{3.8}$$

$$\left.\frac{\partial^2 V}{\partial S^2}\right|_{S=S_max} = 0 \tag{3.9}$$

For the lower boundary the following conditions hold:

$$\left.\frac{\partial V}{\partial S}\right|_{S=0} = 0 \tag{3.10}$$

$$\left.\frac{\partial^2 V}{\partial S^2}\right|_{S=0} = 0 \tag{3.11}$$

Substituting the central difference 3.5 that approximates the second order derivative to the condition 3.9 we take:

$$rV_{N_S+1,j} = 2V_{N_S,j} - 2V_{N_S-1,j} \tag{3.12}$$

For the lower boundary we substitute 3.5 to equation 3.11 to acquire:

$$rV_{-1,j} = 2V_{0,j} - 2V_{1,j} \tag{3.13}$$

Consider the explicit formula (3.7) over the boundaries ($i = 0, N_S$). We replace the option values $V_{N_S+1,j}$, and $V_{-1,j}$ that appear in the upper and lower boundary expressions respectively from equations 3.12 and 3.13. The final form for the upper boundary shall be:

$$VN_S, j+1 = a_{N_S} \cdot VN_S - 1, j + b_{N_S} \cdot VN_S, j \tag{3.14}$$

with:

$$a_{N_S} = -rN_S d\tau$$
$$b_{N_S} = 1 - rd\tau + rN_S d\tau$$

For the lower boundary we shall have:

$$V_{0,j+1} = b_0 \cdot V_{0,j} + c_0 \cdot V_{1,j} \tag{3.15}$$

with:

$$b_0 = 1 - rd\tau$$
$$c_0 = 0$$

Reaching equations 3.14 and 3.15 we managed to eliminate the influence of $V_{N_S+1,j}$ and $V_{-1,j}$ in the calculation of $V_{N_S,j+1}$ and $V_{0,j+1}$ respectively.

Remarks

We have presented an explicit finite difference scheme implementation using the Von Neumann conditions in order to simplify the calculations on the boundaries. The explicit scheme is first order accurate in the time dimension. This happens because the time derivative is approximated with a forward finite difference that produces a local truncation (discretization) error of order $O(d\tau)$. It is also second order accurate in the asset price direction as the central differences used generate a truncation error of order $O(dS^2)$ Despite its simplicity compared to other schemes it is not always stable - small errors in the approximations used grow bigger over time. The necessary condition for stability is that the coefficients a_i, b_i, c_i must be non-negative for all is [10]. It can be proven that $(d\tau/d_S^2) \approx 1$ in order that $b_i > 0$. This condition says that the time steps should be equal to the square of the asset price points.

The explicit scheme applied to the heat diffusion equation

The Black-Scholes equation can be transformed to the heat diffusion equation 2.2 as mentioned in the previous chapter. The explicit scheme can be applied on this equation in a simpler way as the heat diffusion equation has constant coefficients.

The quantity $u(x, \tau)$ instead of $V(S, t)$ is approximated with:

$$x = ln(S/K)$$
$$\tau = \frac{1}{2}\sigma^2(T - t)$$
$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2}$$

with $k = 2r/\sigma^2$.

It is approximated over the grid (x_i, τ_j) with $x_i = idx$, $-N_x \leq i \leq N_x$, and $\tau_j = jd\tau, 0 \leq \tau \leq N_\tau$. The borders of the grid are $[-X, X]$ ($N_x = 2X/dx$) and $[0, \sigma^2/2T]$ ($N_\tau = \sigma^2/2T/d\tau$).
Using a forward difference for $\partial u/\partial\tau$ and a central difference for $\partial^2 u/\partial x^2$ we produce the following formula:

$$u_{i,j+1} = a \cdot u_{i-1,j} + (1 - 2a) \cdot u_{i,j} + a \cdot u_{i+1,j} \tag{3.16}$$

where: $a = d\tau/dx^2$
The boundary conditions are the following:

$$\begin{aligned} &u_{i,0} = max(e^{\frac{1}{2}(k+1)x_i} - e^{\frac{1}{2}(k-1)x_i}, 0), \\ &u_{-N_x,j} = 0, \\ &u_{N_x,j} = (e^X - e^{r\cdot\tau/\frac{1}{2}\sigma^2}) \cdot e^{\frac{1}{2}(k-1)X + \frac{1}{4}(k+1)^2\tau} \end{aligned}$$

In order to acquire the corresponding $V(S,t)$ values we have to make the following transform:

$$V(S,t) = K \cdot e^{-\frac{1}{2}(k-1)x - \frac{1}{4}(k+1)^2\tau} u(x,\tau)$$

It is easily noticed that the explicit scheme applied to the heat diffusion equation is far less complicated than that applied to the initial Black-Scholes equation as the coefficients of the explicit formula (eq. 3.16) are constant. This simplicity comes to the expense of the various transformations required. The explicit scheme for the heat diffusion equation has been implemented on the proposed FPGA architecture.

3.2.5 Implicit scheme

Introduction

The implicit scheme calculates three consecutive option values $V_{i-1,j}$, $V_{i,j}$, $V_{i+1,j}$ for $(0 < i < N_S)$ (internal values of the asset prices) of the j-th time step using the option value $V_{i,j-1}$ of the previous time step 3.4. The option values for the first time step ($j = 0$) are known from the initial condition $V_{i,0} = max(S_i - K, 0)$ as in the case of the explicit scheme.

Analysis

The implicit scheme uses central differences for the first and second order derivatives with respect to the asset price (eq. 3.4, 3.5) and a backward difference in the time direction:

$$\left.\frac{\partial V}{\partial \tau}\right|_{S=S_i,\tau=\tau_j} \approx \frac{V_{i,j} - V_{i,j-1}}{d\tau} \tag{3.17}$$

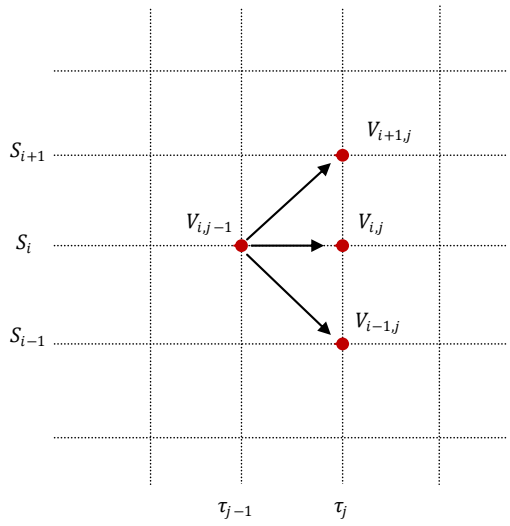


Figure 3.4: Implicit scheme: flow of option value calculations over the grid.

The substitution of the partial derivatives of equations 3.4, 3.5 and 3.17 to the discretized Black-Scholes equation (eq. 3.2) produces the following expression:

$$\begin{aligned} rV_{i,j} = &- \frac{V_{i,j} - V_{i,j-1}}{d\tau} + rS_i \frac{V_{i+1,j} - V_{i-1,j}}{2dS} \\ &+ \frac{1}{2}\sigma^2 S_i^2 \frac{V_{i+1,j} + V_{i-1,j} - 2V_{i,j}}{dS^2} \end{aligned} \tag{3.18}$$

Carrying out the computations we end up with the following formula:

$$a_i \cdot V_{i-1,j} + b_i \cdot V_{i,j} + c_i \cdot V_{i+1,j} = V_{i,j-1} \tag{3.19}$$

with:

$$\begin{aligned} a_i &= \frac{1}{2} rid\tau - \frac{1}{2}\sigma^2 i^2 d\tau \\ b_i &= 1 + rd\tau - \sigma^2 i^2 d\tau \\ c_i &= -\frac{1}{2} rid\tau - \frac{1}{2}\sigma^2 i^2 d\tau \end{aligned}$$

The parameters a_i, b_i, c_i as computed above apply only to the calculation of $V_{i,j}$ for $0 < i < N_S$. In order to calculate the boundary option values $V_{0,j}$, $V_{N_S,j}$ in each time step we use the Von Neumann conditions used previously in the Explicit scheme. Considering the implicit formula 3.19 for the upper boundary N_S and substituting the quantity $V_{N_S+1,j}$ from expression 3.12we obtain:

$$a_{N_S} \cdot V_{N_S-1,j} + b_{N_S} \cdot V_{i,j} = V_{i,j-1} \tag{3.20}$$

with:

$$\begin{aligned} a_{N_S} &= rN_S d\tau \\ b_{N_S} &= 1 + rd\tau + 3\sigma^2 N_S^2 d\tau \end{aligned}$$

In order to obtain the formula for the lower boundary we consider the implicit formula 3.19 for the lower boundary ($i = 0$) and substitute the quantity $V_{-1,j}$ from expression 3.13:

$$b_0 \cdot V_{0,j} + c_0 \cdot V_{1,j} = V_{0,j-1} \tag{3.21}$$

with:

$$b_0 = 1 + rd\tau$$
$$c_0 = 0$$

Equation 3.19 obtained for $0 < i < N_S$ along with the boundary formulae (eq. 3.20, 3.21) constitute a linear system of equations. The unknowns are the option values of the current time step ($V_{i,j}$). The right hand side of the system consists of the option values of the previous time step ($V_{i,j-1}$) which are considered known. This system shall have the following form in matrix notation:

$$\begin{pmatrix} b_0 & c_0 & 0 & \cdots & 0 \\ a_1 & b_1 & c_1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & a_{N_S-1} & b_{N_S-1} & c_{N_S-1} \\ 0 & \cdots & 0 & a_{N_S} & b_{N_S} \end{pmatrix} \begin{pmatrix} V_{0,j} \\ V_{1,j} \\ \vdots \\ V_{N_S-1,j} \\ V_{N_S,j} \end{pmatrix} = \begin{pmatrix} V_{0,j-1} \\ V_{1,j-1} \\ \vdots \\ V_{N_S-1,j-1} \\ V_{N_S,j-1} \end{pmatrix}$$

Remarks

The implicit scheme entails the solution of a series of tridiagonal systems equal in number to the multitude of time steps . The solutions produced by each system are used as the right hand side of the next to be solved. The increased complexity compared to the explicit scheme compensates for better stability. This scheme is unconditionally stable and as the explicit one it is first order accurate in the time direction and second order accurate in the asset price direction.

3.2.6 Crank-Nicholson scheme

Introduction

The Crank-Nicholson scheme can be perceived as the average of implicit and explicit schemes. It uses a central differences for both the asset price derivatives 3.4, 3.5 and the time derivative 3.3. The option value is also approximated over the point $(i, j + 1/2)$ as the average of the values at the points $(i, j + 1)$ and (i, j):

$$V(S_i, \tau_{j+1/2}) = \frac{V_{i,j+1} - V_{i,j}}{2} \tag{3.22}$$

All the partial derivatives are centered at the point $S_i, \tau_{j+1/2}$ too:

$$\begin{aligned}
\left.\frac{\partial V}{\partial \tau}\right|_{S=S_i,\tau=\tau_{j+1/2}} &= \frac{V_{i+1} - V_i}{d\tau} \\
\left.\frac{\partial V}{S}\right|_{S=S_i,\tau=\tau_j+1/2} &= \frac{V_{i+1,j+1} + V_{i+1,j} - V_{i-1,j+1} - V_{i-1,j}}{4dS} \\
\left.\frac{\partial^2 V}{S^2}\right|_{S=S_i,\tau=\tau_j+1/2} &= \frac{V_{i+1,j+1} + V_{i+1,j} + V_{i-1,j+1} + V_{i-1,j} - 2V_{i,j+1} - 2V_{i,j}}{2dS^2}
\end{aligned}$$

Analysis

Using the approximations mentioned above for the partial derivatives and the option values, the discretized Black-Scholes equation takes the following form:

$$\begin{aligned}
r\frac{V_{i,j+1} + V_{i,j}}{2} = &- \frac{V_{i,j+1} - V_{i,j}}{d\tau} + rS_i\frac{V_{i+1,j+1} + V_{i+1,j} - V_{i-1,j+1} - V_{i-1,j}}{4dS} \\
&+ \frac{1}{2}\sigma^2 S_i^2 \frac{V_{i+1,j+1} + V_{i+1,j} + V_{i-1,j+1} + V_{i-1,j} - 2V_{i,j+1} - 2V_{i,j}}{2dS^2}
\end{aligned} \tag{3.23}$$

Another way to come up with the above expression is to average the explicit formula (eq. 3.6) and implicit one (eq. 3.18) replacing the time step index j with $j+1$ in the latter. After rearranging the option value terms in equation 3.23 so that those of step $j+1$ move to the left side and those of step j go to the right, we obtain the following expression:

$$a_i V_{i-1,j+1} + b_i V_{i,j+1} + c_i V_{i+1,j+1} = d_{0,i} V_{i-1,j} + d_{1,i} V_{i,j} + d_{2,i} V_{i+1,j} \tag{3.24}$$

with:

$$\begin{aligned}
a_i &= \frac{rid\tau}{2} - \frac{1}{2}\sigma^2 i^2 d\tau, \\
b_i &= 2 + rd\tau + \sigma^2 i^2 d\tau, \\
c_i &= -\frac{rid\tau}{2} - \frac{1}{2}\sigma^2 i^2 d\tau, \\
d_{0,i} &= -a_i, \\
d_{1,i} &= 2 - rd\tau - \sigma^2 i^2 d\tau, \\
d_{2,i} &= -c_i,
\end{aligned}$$

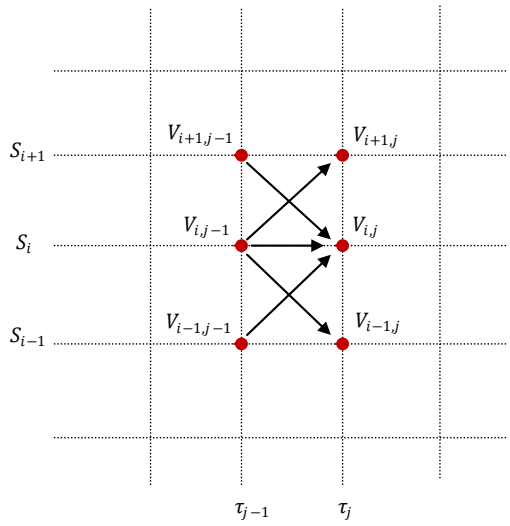


Figure 3.5: Crank-Nicholson: flow of option value calculations over the grid.

Equation 3.24 associates the unknown values of time step $j+1$ with those of time step j that are known. The way the option values are interwined can also be visualized in figure 3.5.

The above expressions for the parameters $a_i, b_i, c_i, d0, i, d1, i, d2, i$ are valid over non-boundary asset prices (0 ¡ i ¡ N_S). As for the boundaries, we shall resort once again to the Von Neumann conditions. For the case of the upper boundary we consider the Crank-Nicholson formula 3.24 for $i = N_S$ and substitute the values $V_{NS+1,j+1}$ and $V_{NS+1,j}$ that show up from expression 3.12 obtained for $j+1$ and j respectively. After executing the necessary computations to the derived formula we reach the following equation:

$$a_{N_S} \cdot V_{N_S-1,j+1} + b_{N_S} \cdot V_{N_S,j+1} = d_{0,N_S} \cdot V_{N_S-1,j} + d_{1,N_S} \cdot V_{N_S,j} \qquad (3.25)$$

with:

$$\begin{aligned} a_{N_S} &= rN_S d\tau, \\ b_{N_S} &= 2 + rd\tau - rN_S d\tau, \\ d_{0,N_S} &= -a_{N_S}, \\ d_{1,N_S} &= 2 - rd\tau + rN_S d\tau \end{aligned}$$

We obtain the lower bound formula in similar way:

$$b_0 \cdot V_{0,j+1} + c_0 \cdot V_{1,j+1} = d_{1,0} \cdot V_{0,j} + d_{2,0} \cdot V_{1,j} \qquad (3.26)$$

with:

$$\begin{aligned} b_0 &= 2 + rd\tau \\ c_0 &= 0, \\ d_{1,0} &= 2 - rd\tau, \\ d_{2,0} &= 0 \end{aligned}$$

A linear tridiagonal system is produced by equation 3.24 for $0 < i < N_S$ along with the boundary formulae 3.25 and 3.26 with unknowns the option values of time step $j+1$ using those of the j-th time step:

$$\begin{pmatrix} b_0 & c_0 & 0 & \cdots & 0 \\ a_1 & b_1 & c_1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & a_{N_S-1} & b_{N_S-1} & c_{N_S-1} \\ 0 & \cdots & 0 & a_{N_S} & b_{N_S} \end{pmatrix} \begin{pmatrix} V_{0,j+1} \\ V_{1,j+1} \\ \vdots \\ V_{N_S-1,j+1} \\ V_{N_S,j+1} \end{pmatrix} = \begin{pmatrix} e_{0,j} \\ e_{1,j} \\ \vdots \\ e_{N_S-1,j} \\ e_{N_S,j} \end{pmatrix}$$

with:

$$\begin{pmatrix} e_{0,j} \\ e_{1,j} \\ \vdots \\ e_{N_S-1,j} \\ e_{N_S,j} \end{pmatrix} = \begin{pmatrix} d_{1,0}V_{0,j} + d_{2,0}V_{1,j} \\ d_{0,1}V_{0,j} + d_{1,1}V_{1,j} + d_{2,1}V_{2,j} \\ \vdots \\ d_{0,N_S-1}V_{N_S-2,j} + d_{1,N_S-1}V_{N_S-1,j} + d_{2,N_S-1}V_{N_S,j} \\ d_{0,N_S}V_{N_S-1,j} + d_{1,N_S}V_{N_S,j} \end{pmatrix}$$

Remarks

The Crank-Nicholson scheme requires the solution of a tridiagonal system in each time step and the update of the right hand side with the new values for $e_i, 0 < i < N_S$. Therefore it is the most computationally intensive of the three schemes. Nevertheless the use of central differences for the approximation of the time derivatives permits 2nd order accuracy with respect to time. In the asset price direction the accuracy remains quadratic [11]. Moreover it is unconditionally stable. The Crank-Nicholson scheme is the last of the three finite difference methods implemented on the proposed FPGA architecture.

3.2.7 Parallelization of finite-difference schemes

Finite-differences schemes calculate the values of the unknown function over a grid of points. Its value on a given time step depends on values over the neighboring points of the previous step in the case of the explicit scheme (see fig. 3.3). For the implicit and Crank-Nicholson schemes the dependency is extended to neighboring points of the current time step (see fig. 3.4, 3.5). This locality of data dependencies allows for large scale parallelization. Ideally if a processing unit (core) is available for the calculation of a function value over a single point in the asset price direction, all the values of the same time step could be calculated simultaneously (at least in the case of

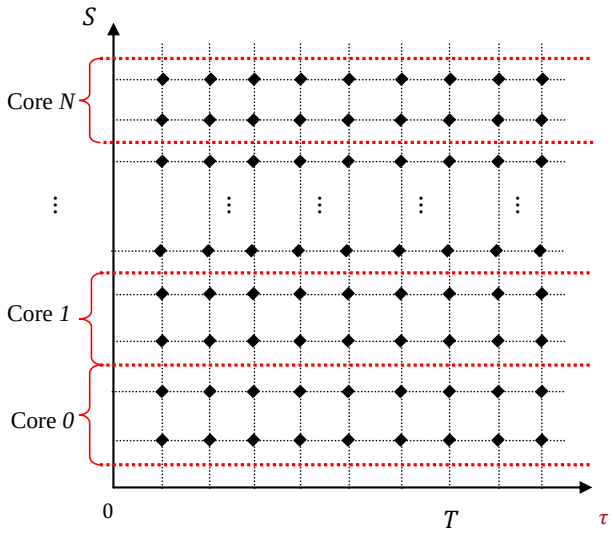


Figure 3.6: Allocation of grid points over the cores of a conceptual parallel architecture.

explicit finite differences). Given a parallel architecture with a fixed number of cores, if an equal number of grid points is allocated to each one of them, the process can be accelerated by a factor equal to the number of processors. The way the grid is partitioned can be seen in figure 3.6. It should be noted that accelerating the implicit and Crank-Nicholson scheme requires a parallelized way for solving the underlying system of equations. This is the subject of the next chapter.

Chapter 4

Solving Tridiagonal systems

The implicit and explicit schemes require the solution of a linear tridiagonal system in each time step. Such systems can be solved directly with more efficient algorithms than those used for general form linear systems such as Gaussian Elimination [12]. Two of these methods shall be presented in the current chapter. The first one is the LU-decomposition for tridiagonal systems and the other is the cyclic odd-even reduction.

4.1 LU-Decomposition

The idea behind LU-decomposition is to factorize the matrix M of a tridiagonal linear system $Mx = e$ in two matrices L, U ($M = LU$), the first lower triangular and the last upper triangular, and solve the pair of matrix equations $Lz = e$ and $Ux = z$ afterwards.

Consider a tridiagonal linear system of size N that has the following form:

$$\underbrace{\begin{pmatrix} b_0 & c_0 & 0 & \cdots & 0 \\ a_1 & b_1 & c_1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & a_{N-2} & b_{N-2} & c_{N-2} \\ 0 & \cdots & 0 & a_{N-1} & b_{N-1} \end{pmatrix}}_{M} \underbrace{\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-2} \\ x_{N-1} \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{N-2} \\ e_{N-1} \end{pmatrix}}_{e} \tag{4.1}$$

The first phase of LU-Decomposition is the factorization of M in the matrices L, U which is called decomposition or reduction. The second phase is finding the solution of the equation $Lz = b$ with forward substitution. The last phase includes the solution of $Ux = z$ with backward substitution (see also algorithm 1).

$$\underbrace{\begin{pmatrix} b_0 & c_0 & 0 & \cdots & 0 \\ a_1 & b_1 & c_1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & a_{N-2} & b_{N-2} & c_{N-2} \\ 0 & \cdots & 0 & a_{N-1} & b_{N-1} \end{pmatrix}}_{M} = \underbrace{\begin{pmatrix} 1 & 0 & \cdots & 0 \\ a'_1 & 1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & a'_{N-1} & 1 \end{pmatrix}}_{L} \cdot \underbrace{\begin{pmatrix} b'_0 & c_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & b'_{N-2} & c_{N-2} \\ 0 & \cdots & 0 & b'_{N-1} \end{pmatrix}}_{U}$$

Algorithm 1 LU-Decomposition

$b'_0 = b_0$
$e'_0 = e_0$
for $i = 1$ to $N - 2$ **do**
  {Decomposition}
  $a'_i = a_i / b'_{i-1}$
  $b'_i = b_i - a'_i \cdot c_{i-1}$
  {Forward substitution}
  $z_i = e_i - a'_i \cdot z_{i-1}$
end for
{Backward substitution}
$x_{N-1} = e_{N-1} / b'_{N-1}$
for i = N-2 to 0 **do**
  $x_i = (z_i - c_i \cdot x_{i+1}) / b'_i$
end for

The time complexity of the tridiagonal LU-decomposition is linear with respect to the system size. It does not require complex computations, nevertheless it cannot be fully parallelized without extensive modifications [13] as data dependencies exist in each loop iteration of the decomposition/forward substitution phase and during the backward substitution too. Moreover these two phases cannot be executed in parallel (see algorithm 1). Only the last two operations within the same loop iteration of the decomposition/forward substitution phase can be parallelized. This parallelization cannot escalate on massively parallel systems whatsoever.

4.2 Cyclic Odd-Even Reduction

Cyclic reduction is a family of methods for solving tridiagonal systems invented by G.H. Golub and R. W. Hockney in mid 1960's. The basic idea behind these methods is to apply a transformation on a set of linear equations that results in two (or more) sets of linear equations which are independent

of each other. In other words the initial linear system is decoupled in a number of linear systems that can be solved independently.

Odd-even reduction is a type of cyclic reduction that eliminates recursively half of the unknowns (the odd-indexed ones) and produces a new system with half size (consisting of the even-indexed ones) till reaching a unit size system which can be solved trivially. Starting from the calculated unknown of the unit size system and going backwards, the eliminated odd-indexed unknowns of each step are retrieved using the already calculated even-indexed ones.

The process of eliminating the odd-indexed unknowns is called forward elimination. As the size of the system reduces to half in each step, the total number of steps till reaching a unit size system is $\lceil \log N \rceil$. Consider the following tridiagonal system at the beginning of the algorithm (the first of the two indices of each coefficient which is zero denotes that the system refers to the initial step):

$$\begin{pmatrix} b_{0,0} & c_{0,0} & 0 & \cdots & 0 \\ a_{0,1} & b_{0,1} & c_{0,1} & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & a_{0,N-2} & b_{0,N-2} & c_{0,N-2} \\ 0 & \cdots & 0 & a_{0,N-1} & b_{0,N-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-2} \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} e_{0,0} \\ e_{0,1} \\ \vdots \\ e_{0,N-2} \\ e_{0,N-1} \end{pmatrix}$$

In the k-th step the initial system shall be reduced to the following system of size $N_k = \lceil N/2^k \rceil$:

$$\begin{pmatrix} b_{k,0} & c_{k,0} & 0 & \cdots & 0 \\ a_{k,1} & b_{k,1} & c_{k,1} & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & a_{k,N_k-2} & b_{k,N_k-2} & c_{k,N_k-2} \\ 0 & \cdots & 0 & a_{k,N_k-1} & b_{k,N_k-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_{2^k} \\ \vdots \\ x_{(N-2)/2^k} \\ x_{(N-1)/2^k} \end{pmatrix} = \begin{pmatrix} e_{k,0} \\ e_{k,1} \\ \vdots \\ e_{k,N_k-2} \\ e_{k,N_k-1} \end{pmatrix}$$

By the completion of $\lceil \log N \rceil$ steps, the system is reduced to a single equation with one unknown variable: x_0. After calculating x_0, the other unknowns are calculated with back substitution of the already calculated ones to the linear systems produced during forward elimination. Having calculated $x_{i \cdot 2^k}$, with $i = 0, \ldots, N_k - 1$ where k is the current algorithm step, $x_{i \cdot 2^k}$ with $i = 1, 3, \ldots, N_{k-1}$ are calculated. This process can be visualized in figure 4.1 for $N = 8$, where $p_k = (a_k, b_k, c_k, e_k)$ is the matrix of all the coefficients and the right hand side of the system of k-th step. Notice the order in which the unknowns are calculated during the backward substitution.

Two approaches of odd-even reduction are described in the following sections. The first one is the original version and the second is a new variant

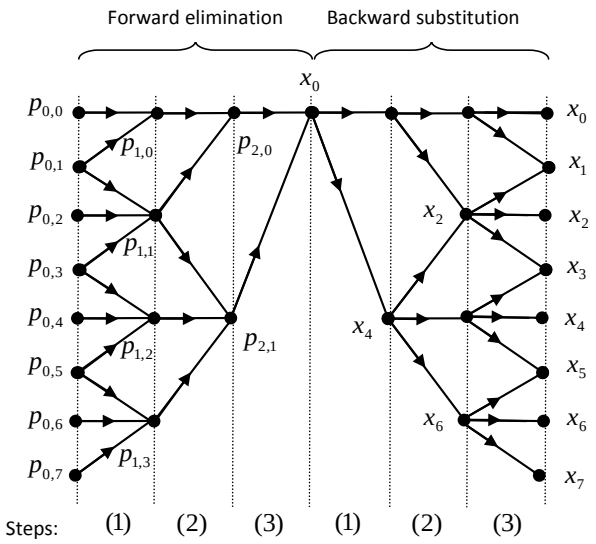


Figure 4.1: Cyclic odd-even reduction applied on a system of size $N = 8$ - Flow of computations.

- the one chosen for implementation on the proposed reconfigurable architecture. The implemented variant is based on the hypothesis that the main diagonal of the initial system is unit and applies a slightly modified forward elimination in order to produce systems with unit main diagonal too. These two approaches are presented in the following sections followed by a comparison of their performance.

4.2.1 Traditional odd-even reduction

The traditional odd-even reduction eliminates the odd-indexed unknowns of each step and produces the coefficients of the new half-size system by adding the even-indexed i-th equation with multiples of the odd-indexed $i-1, i+1$ ones using the formulae described in algorithm 1. The out-of-bounds coefficients of the first and last iteration are considered zero: $a_{-1} = c_{-1} = e_{-1} = 0.0$, $a_{N_k} = c_{N_k} = e_{N_k} = 0.0$

Algorithm 2 Forward Phase of traditional Odd-Even Reduction

for $i = 0$ to $N_k - 1$ **do**

$$
\begin{aligned}
a_{k+1,i/2} &= -a_{k,i-1}a_{k,i}b_{k,i+1} \\
b_{k+1,i/2} &= b_{k,i-1}b_{k,i}b_{k,i+1} - c_{k,i-1}a_{k,i}b_{k,i+1} - \\
&\quad b_{k,i-1}c_{k,i}a_{k,i+1} \\
c_{k+1,i/2} &= -b_{k,i-1}c_{k,i}c_{k,i+1} \\
e_{k+1,i/2} &= b_{k,i-1}e_{k,i}b_{k,i+1} - e_{k,i-1}a_{k,i}b_{k,i+1} - \\
&\quad b_{k,i-1}c_{k,i}e_{k,i+1}
\end{aligned}
$$

$i+ = 2$

end for

The back-substitution phase of the algorithm, during which the unknowns are calculated is described in algorithm 3.

It can be easily observed that fine grained parallelism can be achieved for both phases of the odd-even reduction as loop iterations are not data depen-

Algorithm 3 Backward Phase of traditional Odd-Even Reduction

for $i = 1$ to $N_k - 1$ **do**
  $x_{i2^k} = (e_{k,i} - a_{k,i}x_{(i-1)2^k} - c_{k,i}x_{(i+1)2^k})/b_{k,i}$
  $i+ = 2$
end for

dent. Odd-even reduction requires more operations than LU-decomposition but its performance can be leveraged with the use multiple processing units.

Despite its parallelization capabilities, odd-even reduction has a major valnerability. The products of the formula for the calculation of $b_{k+1,i}$ increase excessively with k and for a given finite computer arithmetic precision it is possible that they can not be represented. In order to address this issue we implemented a variant of this method that normalizes the main diagonal in each step avoiding the excessive growth of the coefficients [14].

4.2.2 The new odd-even reduction variant

This variant requires that the initial system has unit main diagonal. This prerequisite does not limit the generality of this approach as the main diagonal of a general form tridiagonal system can be normalized by dividing both sides of each equation with the corresponding main diagonal element, producing an equivalent system with the desirable property. The main diagonal of the half-size systems produced during forward elimination is kept unit applying slightly different operations. This strategy ensures that the produced coefficients do not grow excessively as in the case of traditional odd-even reduction.

The forward phase is summarized in algorithm 4. It is derived from algorithm 2 by substituting $b_{0,i} = 1$ initially (as we made the hypothesis that the initial system has unit main diagonal. Dividing all the produced coefficients with the quantity $1-c_{0,i-1}a_{0,i}-c_{0,i}a_{0,i+1}$ afterwards we normalize the produced main diagonal ($b_{1,i/2} = 1$). At step k the new coefficients are produced in the same way knowing in advance that $b_{k-1,i} = 1$.

Algorithm 4 Forward elimination of the implemented Odd-Even Reduction variant

for $i = 0$ to $N_k - 1$ **do**
  $temp = 1.0/(1 - c_{k,i-1}a_{k,i} - c_{k,i}a_{k,i+1})$
  $a_{k+1,i/2} = -temp \cdot (a_{k,i-1}a_{k,i})$
  $c_{k+1,i/2} = -temp \cdot (c_{k,i}c_{k,i+1})$
  $e_{k+1,i/2} = temp \cdot (e_{k,i} - e_{k,i-1}a_{k,i} - c_{k,i}e_{k,i+1})$
  $i+ = 2$
end for

The back substitution phase of the implemented variant (alg. 5) is de-

rived from the original back substitution (alg. 3) by substituting $b_{k,i}$ with unity, with $x_0 = e_{\lceil \log N \rceil, 0}$ initially.

Algorithm 5 Back substitution of the implemented Odd-Even Reduction variant

for $i = 1$ to $N_k - 1$ **do**
  $x_{i2^k} = e_{k,i} - a_{k,i} x_{(i-1)2^k} - c_{k,i} x_{(i+1)2^k}$
  $i+ = 2$
end for

4.2.3 Comparison of tridiagonal system solution algorithms

All the algorithms described above have linear execution time with respect to the system size. The loops of the forward and backward phase of LU-decomposition are executed $N - 1$ times each in total(see alg.1). So do the respective loops of the two variants of odd-even reduction. The number of operations inside these loops is illustrative of the relative speed of each method. This information is shown in table 4.1.

Table 4.1: Number of operations of a single Step for each method

	Forward step			**Backward step**		
Operation type	#mul	#add/ #sub	#div	#mul	#add/ #sub	#div
LU decomposition	2	2	1	1	1	1
Traditional odd-even Red.	16	4	0	2	2	1
Implemented odd-even Red.	9	4	1	2	2	0

The implemented odd-even reduction variant requires less operations than the original, eliminating at the same time its main weakness of producing quantities that may exceed the finite machine precision. Though slower in terms of total number of operations compared to LU-decomposition, it is more suitable for parallel architectures. Given such a multicore architecture, odd-even reduction can be almost fully parallelized by allocating a part of the initial system to each core (see also fig. 4.2).

As for memory requirements, LU decomposition is the most thrifty of the three, requiring space only for the three main diagonals and the right hand side. Odd-even reduction is recursive requiring double the necessary space of LU-decomposition in order to store the coefficients of the produced systems without overwriting the initial ones. The implemented odd-even reduction variant does not require the storage of the main diagonal, as it implies that it remains unit throughout the algorithm. At this point it should be mentioned

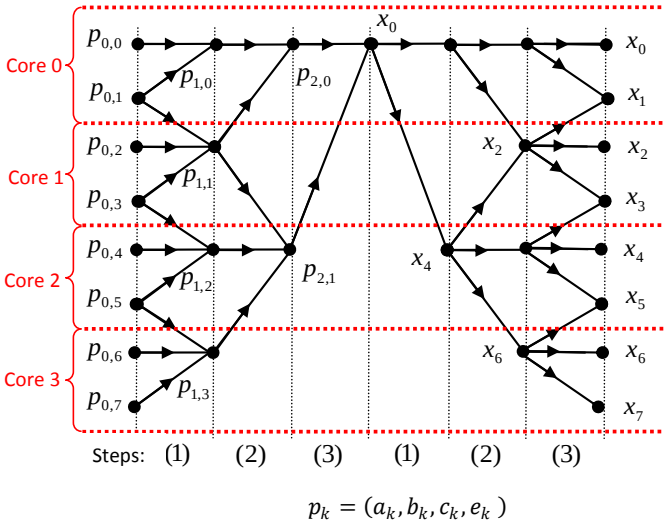


Figure 4.2: Cyclic odd-even reduction applied on a system of size $N = 8$ - Flow of computations.

that odd-even reduction as implemented in the proposed architecture cannot handle systems whose size is not a power of two. The same must apply to the number of cores of the architecture.

Despite the fact that no stability issues have arisen during the application of the above mentioned methods to Black-Scholes option pricing, it should be mentioned that LU-decomposition is of equivalent stability with Gaussian elimination without pivoting [15]. The odd-even reduction is essentially a special case of gaussian elimination were certain permutations are applied to the initial system in order to eliminate the odd-indexed unknowns first, thus it has equivalent stability with gaussian elimination on the permutated systems [16].

Chapter 5

Towards the proposed reconfigurable architecture

The current chapter provides information about the evolution process of the proposed reconfigurable system, detailing the necessities that dictated its final form. A number of already existent approaches is described alongside. At this point we have to take a retrospective view at the targeted problem itself.

5.1 Retrospection on the attacked problem

The problem we focused on in its most abstract formulation is financial option pricing. The mathematical tools available for its modelling are the option pricing models(techniques) presented in chapter 2. The Black-Scholes model is one of them and provides a rather compact way to describe the option pricing problem by means of an elegant differential equation. It reduces the initial problem to the solution of a partial differential equation, a problem which is quite familiar to engineers, can be solved in many ways and can describe a variety of real world problems. These observations consisted strong motives for the selection of the Black-Scholes model instead of others despite its ineffectiveness on pricing cutting-edge financial derivative products. The available methods for solving the partial differential equations have their own strong points and weaknesses and the selection of the most appropriate one is a problem of its own.

Given that the available platform for the implementation of the method to be chosen would be an FPGA device, the initial selection was based on three main drivers:

- Speed: The target of the proposed system is to price options at greater speeds than software implementations on commodity CPUs. An inherently fast algorithm implemented on software can outperform a slower even efficiently implemented on an FPGA device.

- Accuracy: It is crucial for financial computations to be accurate.
- Simplicity and regularity, in order to be implemented in a straightforward way on reconfigurable hardware.
- Ability to parallelize in order to be mapped efficiently on FPGAs and achieve greater speeds than software implementations.

The numerical techniques for the approximation of the Black-Scholes equation were favored instead of the analytical solution because they can be applied to every partial derivative equation and they entail simple computations.

Among the available numerical techniques finite differences are the most fundamental ones with scalable performance in terms of speed and accuracy. The basic finite-difference schemes were presented in chapter 3, with the so-called Crank-Nicholson outperforming the others in terms of stability and speed of convergence. In order to capitalize on these advantages the initial approach was to develop a reconfigurable system for the Crank-Nicholson scheme. Before proceeding to the description of the initial approach we shall refer to a couple of already existent ones.

5.2 Related work

Various approaches to the option pricing problem have been proposed in algorithmic and implementation level. The most commonly implemented methods are Monte-Carlo simulation and binomial pricing. The most prevalent platforms of high performance computing over which these methods are implemented are FPGAs and GPUs (at least at academic level).

5.2.1 Related work on high performance Option Pricing

Monte carlo option pricing is very popular due to its ability to parallelize and adapt to various types of financial derivative products. Two FPGA-based approaches of Monte-Carlo simulation are described in [17],[18]. with both of them achieving great performance boost. In [19] Monte-Carlo simulation was implemented over a hybrid supercomputer called Maxwell [20] consisting of a 32 CPUs and 64 Virtex-4 Xilinx FPGAs achieving more than 100x speedup. GPU implementations of Monte-Carlo Simulation option pricing achieve performance gains of the same order of magnitude as FPGA-based approaches. Two such approaches implemented over NVIDIA GPUs are described in [21] and [22]. The performance gain is based on the simultaneous production of multiple random paths of option prices.

The binomial model calculates one Option value at a time building a binomial tree of asset prices. An implementation of binomial option pricing implementation over FPGAs and GPUs is discussed in [23] achieving

triple-digit accelation over mainstream CPUs. The speedup is achieved by calculating option prices in parallel using multiple binomial trees.

High performance finite-difference approaches are not strictly focused on option pricing such as the explicit scheme described in [24] for the wave equation. An explicit scheme implementation for option pricing over FPGA and GPU is described in [25]. The maximum achieved speedup of the FPGA implementation is 12.2x while the GPU-based reaches 43.9x.

5.2.2 Motivation from other approaches

The plethora of Monte-Carlo based and binomial pricing implementations and their achieved speedups have discouraged us from designing a same method implementation. Monte-Carlo simulation generates random asset price paths over which calculates the option value as the average of the values calculated over these paths while the Binomial model creates a binomial lattice for the calculation of single option value. Despite their ability to parallelize they are inherently slower than finite differences that calculate multiple option prices at a time. A possible speed comparison of the above implementations should take this remark into consideration.

The inherent speed of finite differences along with their adaptability to a wide range of scientific problems has motivated us in exploring them. Moreover the lack of FPGA implementations for the Crank-Nicholson scheme (as far as we know) which is faster in terms of convergence than the implemented explicit ones has decisively influenced our decision to implement it on FPGA.

5.3 The initial approach

The initial approach consisted of a reconfigurable system for solving tridiagonal systems with the traditional odd-even reduction. The plan was abandoned for various reasons explained in the current section.

5.3.1 Algorithmic level

The Crank-Nicholson scheme was selected for our initial implementation for its robustness and performance. When it comes to be mapped on reconfigurable hardware, it requires an efficient implementation of a method to solve the entailed tridiagonal systems (see also section 3.2.6 and algorithm 6).

Setting aside the computation of the right hand side $\mathbf{e_{k+1}}$ for the system of the next time step, our problem is degenerated to the solution of a tridiagonal system. The traditional odd-even reduction has been selected for this purpose due to its ability to parallelize extensively.

Algorithm 6 Crank-Nicholson algorithm

for each time step k **do**
- Solve the tridiagonal system with main diagonals $\mathbf{a}, \mathbf{b}, \mathbf{c}$, unknown vector $\mathbf{V_k}$ and right hand side $\mathbf{e_k}$.
- Compute the right hand side $\mathbf{e_{k+1}}$.

end for

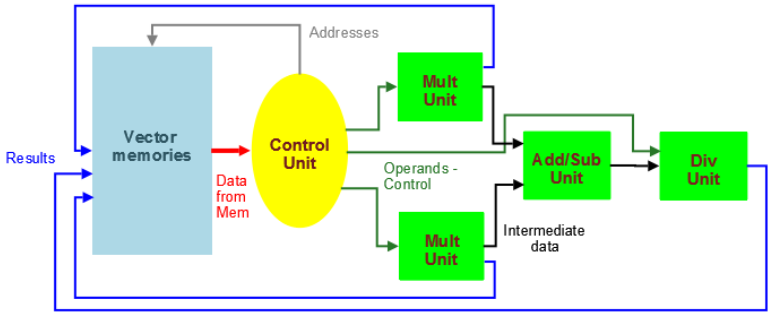


Figure 5.1: Top-level architecture of the initial processing unit for odd-even reduction

5.3.2 Implementation level

The initial implementation consisted of an independent dual port block RAM for each vector a, b, c, e, V, that appears in the system to be solved. When no data had to be written, two consecutive elements were read from each diagonal and the right hand side. At the i-th iteration these were: (a_i, a_{i+1}), (b_i, b_{i+1}), (c_i, c_{i+1}), (e_i, e_{i+1}). These elements were pumped to a control unit that decided which of them would be further dispatched to the floating point units. When results were ready to be written back to memory, the control unit paused reading to write the results at the second port of the appropriate memory. An abstract schematic for this core is shown in figure 5.1.

There are four floating point units: two for multiplication, one for addition and one for division. Each one of them is equipped with the corresponding floating point operator and three FIFOs for temporary data storage. A simplified schematic of an implemented floating point unit is shown in figure 5.2. Two floating point multiplication units were used in order to calculate simultaneously the products inside the forward and backward loops (see algorithms 2, 3). The products are passed to the adder/subtractor unit or returned immediately to memory.

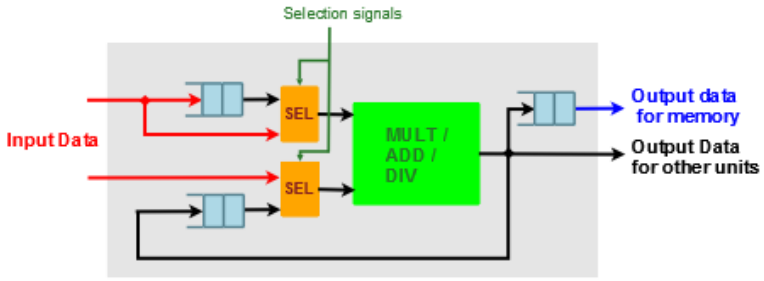


Figure 5.2: A floating point unit equipped with two FIFOs for temporary data storage.

Table 5.1: Resource Utilization of the initial single core architecture on a Virtex^{TM} 5 330T Board

Resource Type	Used	Utilization %	Available
Slice Registers	3,275	1%	207,360
Slice LUTs	3,889	1%	207,360
Occupied Slices	1,496	2%	51,840
LUT Flip Flop pairs	4,661	-	-
BlockRAM/FIFO	5	1%	324
DSPs	8	4%	192

This approach was relatively fast - approximately twice slower than LU decomposition and 6% slower than odd-even reduction with both executed on Pentium^{TM} 4 (see also 5.2). Moreover it was frugal in resource utilization (5.1) allowing the deployment of a number of instances on a high-end FPGA device (such as the Virtex^{TM} 5 family) capable of out-performing the software competitors. It was downloaded successfully on a Spartan^{TM}3E FPGA board. The architecture of the downloaded system consisted of one instance of the core attached to the PLB bus of microblaze^{TM} embedded processor. The latter was in charge of the data IO bridging the peripheral core with the host computer via the serial port (see also fig. 5.3).

Despite its virtues, this single-core architecture was extremely cumbersome to parallelize and incorporate the right-hand-side (RHS) update functionality. Apart from that, it inherited the excessive arithmetic precision requirements of the traditional odd-even reduction. In order to map the new variant, incorporate the (RHS) update and coordinate a number of these cores to run in parallel we had to design from the beginning at least the main control unit. Given these difficulties we decided to take a more flexible approach in order to be able to accommodate new functionalities that might be proved necessary during the development of the final system.

Table 5.2: Execution time for a single tridiagonal system: the initial reconfigurable core vs a Pentium™ 4 single-core CPU

	FPGA Virtex™ 5 330T	**Pentium™ 4**		**Pentium™ Core 2 Duo**	
Method	**Odd-Even Red.**	**Odd-Even Red.**	**LU-Dec.**	**Odd-Even Red.**	**LU-Dec.**
Clock freq.	144MHz	2.6GHz		2.0GHz	
system size	execution time (us)				
256	49	35	23	12	7
512	92	88	49	23	15
1024	178	235	104	44	32

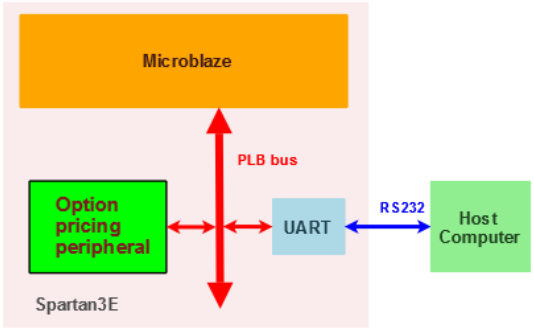


Figure 5.3: The initial option pricing system as peripheral of microblaze™ connected serially to host PC.

Chapter 6

The proposed architecture

The proposed architecture consists of a number of interconnected cores that run in parallel and are able to communicate with each other with message passing. The core architecture described in the previous chapter was abandoned as once instantiated multiple times, the instances needed extensive modifications in order to communicate with each other.

6.1 Architecture of a single core

The architecture of a single core consists of three main parts:

- The local memory: A dual port memory that stores all the necessary data - both integer and floating point data.
- The memory controller: Its main tasks is the resolution of addresses for reading and writing data, the synchronization of the floating point operations and sending data to other cores.
- The floating point unit: The part that executes the necessary floating point operations.

These three sub-units and the way they are connected can be seen in figure 6.1. Each one of them is described meticulously in the following sections.

6.1.1 Local memory - Data organization

The local memory of each core stores general information such as the size of the local grid partition, the IDs of the neighboring cores etc. that constitute integer data, along with the coefficients and initial conditions of each finite-difference scheme (see also fig. 6.2a).

These coefficients are stored immediately after the integer data and include the initial condition (option values of the initial time step $V_{i,0}$) and

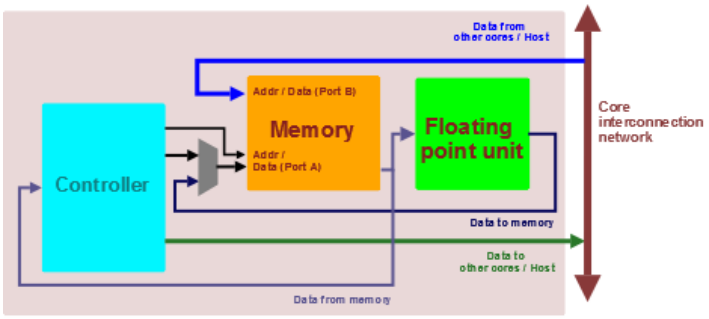


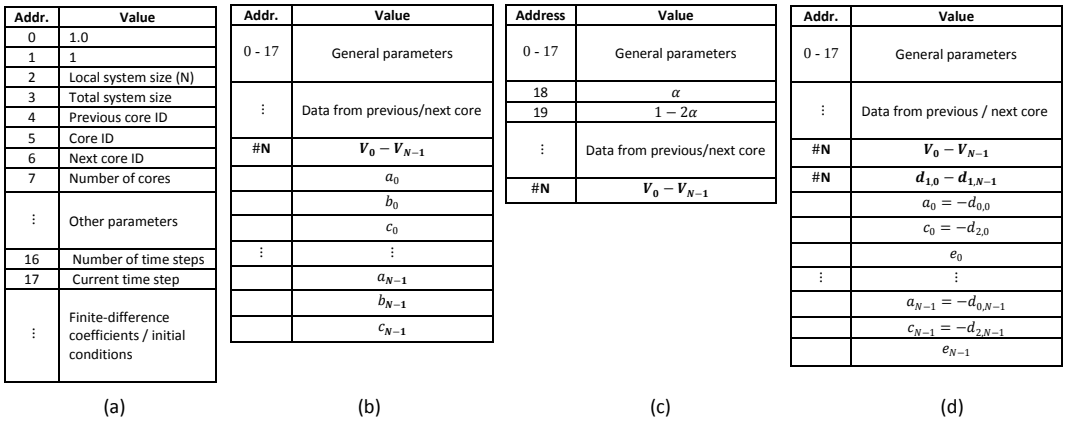
Figure 6.1: Architecture of a single core: Sub-units are arranged according to the pipeline stage they belong.

the parameters inside the formula of each scheme. These are the coefficient vectors a, b, c for the explicit scheme (fig. 6.2b) - the only coefficients when heat-diffusion transform is applied are α and $1 - 2\alpha$ (fig. 6.2c)- and the non-zero diagonals a, b, c of the implicit and Crank-Nicholson schemes. Using the proposed Odd-even reduction variant we can avoid storing the main diagonal b which is implied to be unit. For the Crank-Nicholson scheme we must also store the vectors: d_0, d_1, d_2 that participate in the right-hand-side update at each time step and the right hand side e itself. Hopefully $d_0 = -a$ and $d_2 = -c$ and no extra space is required for them (fig. 6.2d). Additional space is required for storing the vectors a_k, c_k, e_k of the k-th forward step of Odd-even reduction. As the system size is reduced by half its value at the k-th forward step is $N_k = N/2^k$ and the total extra space (data words) required for these vectors is equal to: $\sum_{k=1}^{\lceil logN \rceil N/2^k} = N$.

The local memory is 32 bits wide in order to store single precision floating point numbers (IEEE 754-1985 standard) used for the implemented algorithms. The memory controller datapath utilizes only the 24 less significant bits of integer data stored in the local memory. Data produced by the memory controller are sign-extended in order to be stored to the local memory. The 24-bit width allows the addressing of 2^{24} memory positions in total.

Implementation

The data memory was implemented as true dual port BRAM. The first port is used by the memory controller for reading and writing local data. The second is directly connected to the inter-core communication network and is used for writing the incoming data from other cores or the host computer. The memory requirements for each algorithm can be seen in figure 6.3(a). The available on-chip Block RAM memory of Virtex^{TM} 5 330T can easily



Addr.	Value
0	1.0
1	1
2	Local system size (N)
3	Total system size
4	Previous core ID
5	Core ID
6	Next core ID
7	Number of cores
⋮	Other parameters
16	Number of time steps
17	Current time step
⋮	Finite-difference coefficients / initial conditions

(a)

Addr.	Value
0 - 17	General parameters
⋮	Data from previous/next core
#N	$\boldsymbol{V_0 - V_{N-1}}$
	a_0
	b_0
	c_0
⋮	⋮
	a_{N-1}
	b_{N-1}
	c_{N-1}

(b)

Address	Value
0 - 17	General parameters
18	α
19	$1 - 2\alpha$
⋮	Data from previous/next core
#N	$\boldsymbol{V_0 - V_{N-1}}$

(c)

Addr.	Value
0 - 17	General parameters
⋮	Data from previous / next core
#N	$\boldsymbol{V_0 - V_{N-1}}$
#N	$\boldsymbol{d_{1,0} - d_{1,N-1}}$
	$a_0 = -d_{0,0}$
	$c_0 = -d_{2,0}$
	e_0
⋮	⋮
	$a_{N-1} = -d_{0,N-1}$
	$c_{N-1} = -d_{2,N-1}$
	e_{N-1}

(d)

Figure 6.2: Memory organization: (a) General information stored first (b) Stored coefficients for the explicit scheme (c) Stored coefficients for the explicit scheme with Heat-Diffusion transform (d) Stored coefficients for the Crank-Nicholson scheme.

Table 6.1: Memory requirements for the implemented algorithms with respect to the number asset price points N_S of the grid

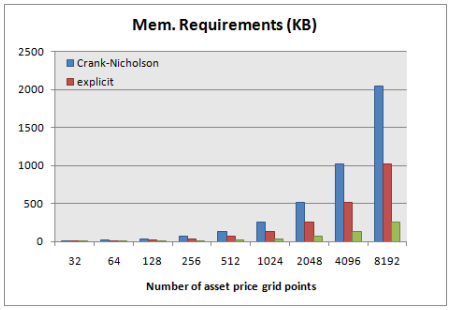
Method	Memory words	KB
Explicit	$4N_S$	$2^{-3}N_S$
Explicit/Heat Diffusion transform	$N_S + 2$	$2^{-5}(N_S + 2)$
Crank-Nicholson	$8N_S$	$2^{-2}N_S$

satisfy the needs of all the implemented algorithms (see also fig. 6.3(b)).

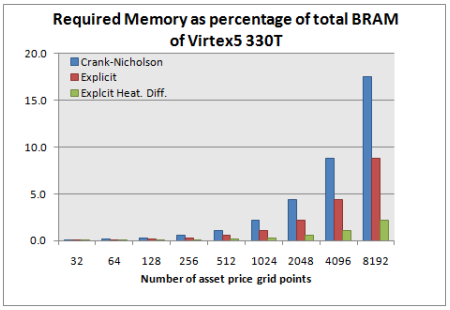
6.1.2 Memory controller

The memory controller is essentially a pipelined 24-bit integer datapath (fig. 6.4). It includes a small instruction memory, a register file, an adder/subtractor, a barrel shifter and a comparator. The pipeline stages are: instruction fetch - instruction decoding - execution - writing data to the register file or a floating point register. It utilizes a small instruction set that supports the following operations:

- Addition/Subtraction
- Shifting
- Branching and Jumps
- Memory reading/Writing
- Dispatching data to and from the floating point unit.



(a) Memory requirements in KB



(b) Memory requirements as percentage of Virtex^{TM} 5 330T BRAM resources

Figure 6.3: Memory requirements of the implemented schemes

- Sending data to other cores or the host computer

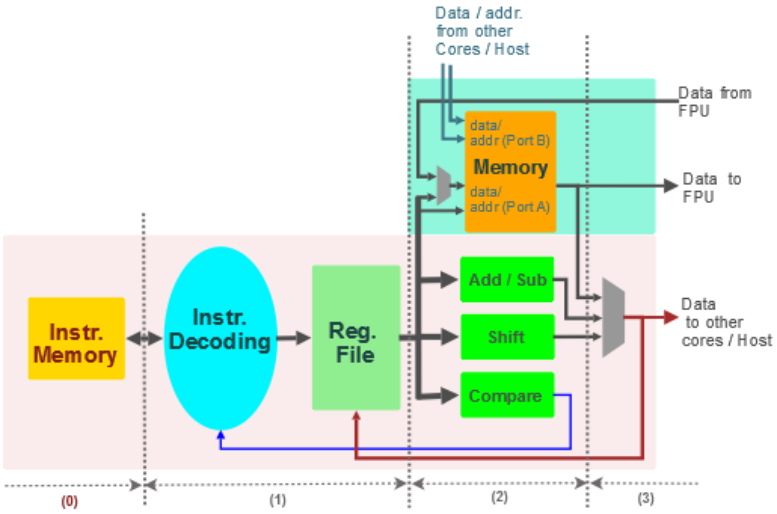


Figure 6.4: Architecture of the memory controller.

The instruction words are 32 bits long. They are decomposed in the operation code field (6 bits long), the result register field that can refer to either to a position in the register file or to a floating point unit (FPU) register ($\$r$), the two read positions from the register file ($\$a$, $\$b$) and the immediate operand field (imm) that occupies the last 12 bits (see also fig. 6.5). The available instructions, their encoding, and their operation are shown in table A.1 of appendix A. An interpreter has been written in Python in order to facilitate the programming of the memory controller. It translates the instructions from the encoding of table A.1 to machine code. The instructions

are preloaded to the instruction memory before downloading the design to the FPGA device.

Instruction code (op-code)	Write register ($r)	1st Read register ($a)	2nd Read register ($b)	Immediate operand (imm)
6 bits	6 bits	4 bits	4 bits	12 bits

Figure 6.5: The 32-bit instruction word of the memory controller and its fields.

The architecture of the memory controller is control-flow based [26]. The executed process is carried out as a sequence of instructions that specify an operation and the position of the participating operands, or transfer the control to another instruction. The availability of the operands is not guaranteed by the instructions themselves but from the way the execution flow is controlled. The floating point unit does not follow this model. It is based on dataflow rather than control-flow.

Implementation

This section discusses the implementation details of each component of the memory controller.

The instruction memory has been implemented as single port LUT-based ROM with depth 128 for the the explicit schemes. A single port Block ROM with capacity 272 words has been used for the more complex Crank-Nicholson scheme. Block ROM was selected in order to spare LUTs.

The register file has two read ports and one for writing and numbers 16 registers 24-bit long each.

All the integer arithmetic units handle 24-bit operands and are single-cycled. An integer adder was implemented for the execution of additions and subtractions. The comparator unit entails one comparator "less" and one "equal" for the evaluation of branch conditions. The shifting operations are carried out with a barrel shifter that can shift up to seven positions at a time (its second operand is 3-bit long) for minimal complexity. The explicit scheme implementations do not include a barrel shifter as no such operation was necessary.

6.1.3 Floating point unit

The floating point unit follows the dataflow model. There is no predefined sequence for the operations to be executed. The sequence is determined only by the availability of data [27]. As long as the operands are ready, and the corresponding operation unit is not busy, the operation is executed.

The architecture of the FPU shall be described by means of the following example. Consider the formula: $a \cdot b + c \cdot d + e$, the quantities a, b, c, d, e are read from memory and the result of the formula is stored there. Similar expressions appear in the implemented finite difference schemes and the odd-even reduction. This expression can be compiled to a dataflow graph like the one in figure 6.6(a).

Consider an architecture that is a precise mapping of the graph of figure 6.6(a) into operational(addition and multiplication) units (denoted by circles) and registers (denoted by rectangles). Now let us consider the iterative calculation of the above mentioned expression resembling the iterative calculations of the explicit formulae or the forward reduction and back substitution phases of odd-even reduction. If all the quantities are loaded at the same rate, then the quantity e is replaced before it is used. Replacing the corresponding register with a FIFO, this problem is solved to the extend that the FIFO does not overflow (see fig. 6.6(b)).

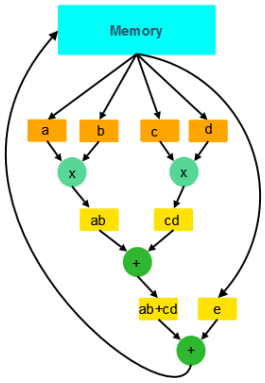
If the quantities a, b, c, e are read in the same clock cycle, the architecture can fully utilize all the operation units every clock cycle. The utilization drops when these quantities are read serially, as happens in our case, where the memory controller reads one data word at each clock cycle. In order to keep the resource utilization minimal the duplicate units can be removed. Grouping the storage elements for the operands of the same type of units afterwards, we end up with the reduced dataflow graph of figure 6.6(c).

The problem of an architecture based on the reduced graph is that when the operands of two or more operations of the same type become available simultaneously, only one of them shall be executed. Prioritizing these operations and executing them serially is the solution. When operands are available while operational units are busy, the possibility of replacing them increases.

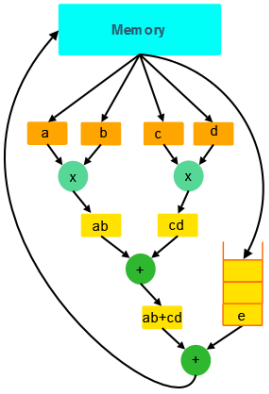
This problem is addressed using FIFOs and status flags indicating when the respective storage elements have valid data and the corresponding operation can be executed or are empty and new data can be loaded to them. The status flag of a register is set to busy each time new data arrives and to zero when the corresponding operation is executed. The status of a FIFO is the reverse of the "empty" flag from the side of the operational unit that drains its content and the full flag from the side of the content source.

This abstraction leads to a generic dataflow model for the FPU which is hardwired for a single dataflow graph but can be easily adapted for others too. It includes a number of operational units each one of which has its own group of storage elements that can be either registers or FIFOs. A priority encoder selects the operation to be executed according to the availability of data for each unit. Modifying the combination of data that trigger each operation and its priority, results in a different dataflow graph.

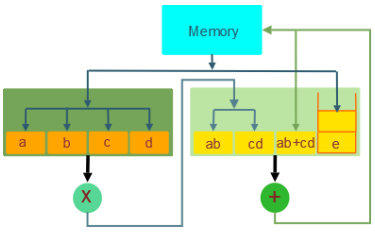
The FPU for the odd-even reduction based Crank-Nicholson scheme has one multiplier, one adder and one divider. The FPU for the explicit schemes



(a) The dataflow graph of $a \cdot b + c \cdot d + e$



(b) The dataflow graph of $a \cdot b + c \cdot d + e$ with a FIFO for the storage of e.



(c) The reduced dataflow graph after removing the duplicate operational units and grouping their storage elements.

Figure 6.6: Evolution of data flow graphs until the implemented form.

has no divider on the other hand, as no divisions are carried out (fig. 6.7). The results that are to be written in memory are stored temporarily in a result FIFO. When new results are produced, a signal is issued to the memory controller to store them to memory using a routine defined for this purpose.

Implementation

The implementation of the FPU was based on the IP cores for floating point operations provided by the Xilinx Core Generator^{TM} tool (Floating-Point Operator v3.0 IP [28]). The floating point multiplier was selected to make max usage of DSPs (3 DSP48E slices). The usage of DSPs can be set to full or medium or no at all depending on the available DSP resources of the target device and the number of implemented cores. The adder and divider

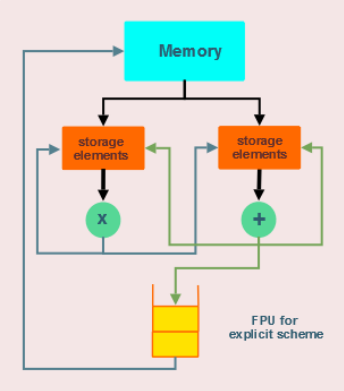


Figure 6.7: The reduced dataflow graph after removing the duplicate operational units and grouping their storage elements.

cores are thoroughly LUT-based (the latter was deployed only for the Crank-Nicholson scheme). Details about their characteristics can be found in table 6.2.

Table 6.2: Characteristics of the Xilinx Floating-Point Operator v3.0 IP cores used for addition, multiplication, addition

Unit type	DSP usage	Latency	Cycles per operation
Multiplier	max (3 DSP48E)	4	1
Adder	No	8	1
Divider	No	14	12

The storage elements of each operational unit were implemented either as registers or FIFOs obtained by the FIFO Generator v4.4 included in Core Generator^{TM}. A priority encoder was implemented for selecting the next operation to be executed according to the status bits of the two operands and the priority of the operation. Each storage element has also its own data source which is either an operational unit or the data memory.

6.2 Top-level architecture

The top-level architecture follows the model of interconnected peer units (cores) with different local memories that communicate through messages achieving process level parallelism [29]. The whole system acts as co-processor of a host computer where the option pricing tasks are offloaded. The host computer sends the initial parameters and triggers the co-processor and the produced results are sent back (client-server model) (see also fig. 6.8).

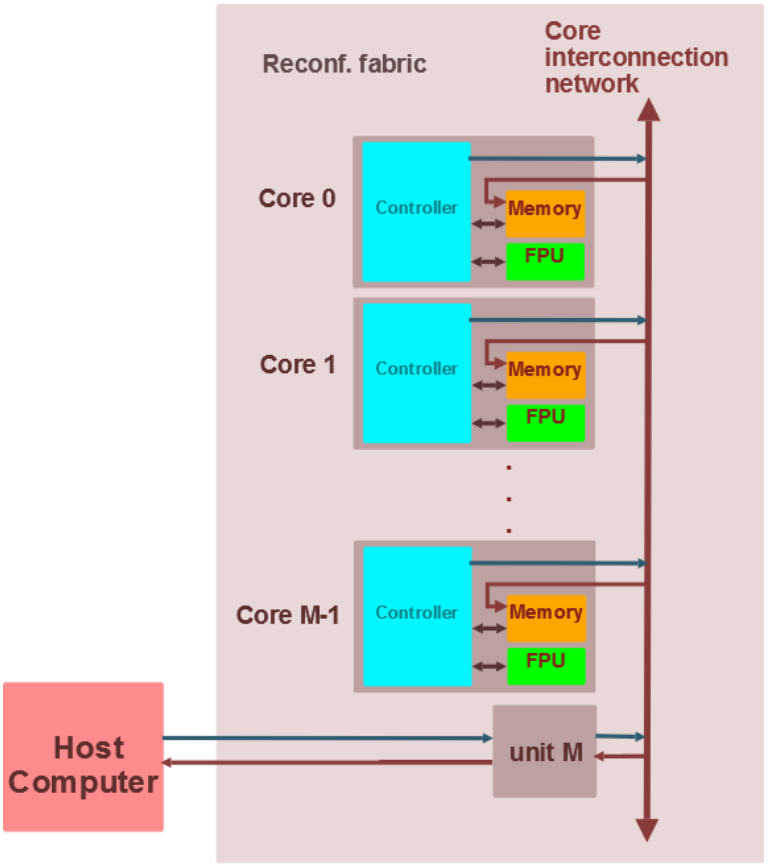
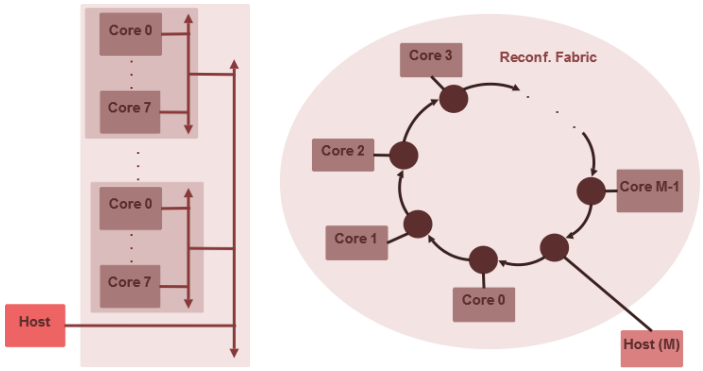


Figure 6.8: The top-level architecture of the FPGA co-processor connected to a host computer.

6.2.1 Interconnection network

The communication between cores is message-driven. Each message is accompanied by a destination header which is the ID of the destination core concatenated with the target local memory address. After each data shipping, an acknowledgement message is sent by the issuing core that has value equal to its ID. The recipient core is polling the local memory position where the acknowledgement is to be received when the local forthcoming operations are dependent on the exchanged data. The reception of messages is done through the second port of the local memory (fig. 6.8).

The first approach for the interconnection network was hierarchical, having the form of a tree topology. Eight cores were connected to a common bus. Each bunch of eight processors was connected to a higher level bus. A system comprised of 32 processors for instance had two hierarchical bus levels and a message required three "hops" in the worst case to reach its



(a) Hierarchical interconnection network.

(b) The implemented ring interconnection network

Figure 6.9: The tested interconnection networks

destination. Each bus used an arbiter permitting round-robin usage. This approach was incited by the odd-even reduction algorithm during which data exchanges form a tree structure. Despite that, it was abandoned as it required excessive resources and new hierarchical levels had to be added for extra cores.

The succeeding approach that was eventually implemented uses point-to-point connections between cores forming a ring topology. The last node of the network is used by the host computer and the messages flow in a single direction (see fig. 6.9(b)). This approach was selected as the majority of data exchanges in finite-difference schemes and odd-even reduction concern neighboring cores. Its resource utilization is less than that of the tree approach and can be easily generalized for arbitrary number of cores. Nevertheless the maximum number of "hops" for message forwarding, being equal to the number of cores, is greater than that of the tree structure.

6.2.2 Input-Output

The IO-operations between the FPGA co-processor that constitutes the proposed architecture and the host computer are limited. The initial parameters of the explicit schemes should be sent from the Host to the FPGA and upon the completion of calculations, the FPGA sends the results back to the Host. The IO-operations being limited to the beginning and the end of the algorithm are not critical for the performance of the FPGA co-processor, thus they do not require great performance. The best IO solution is the use of the PCI Express interconnect. An endpoint block of PCI express is required in the side of the FPGA, connected to the inter-core communication network.

Chapter 7

Results and Evaluation

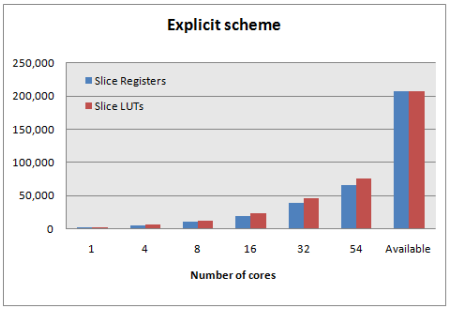
7.1 Resource utilization

7.1.1 Estimation methodology

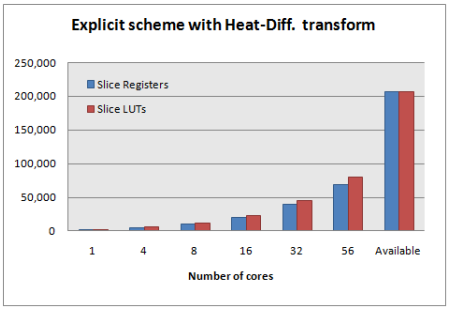
The resource utilization estimation has been done using Xilinx ISE design suite. The selected target device is the Virtex^{TM} 5 330T. The main criterion for its selection was the great number of built-in logic cells. Three versions of the designed system were placed and routed on the target device for resource usage estimation. One implementing the Crank-Nicholson scheme, the second the explicit scheme and the last the explicit scheme applying the heat-diffusion equation transform. The measurements were obtained for 1, 4, 8, 16 and 32 cores per system. Extra measurements for a 54-core explicit scheme system, and a 56-core heat-diffusion transform explicit one. Tables 7.3, 7.2, 7.1 provide more precise information on the number of occupied resources for a unit-core and a 32-bit core system of each version.

Table 7.1: Resource Utilization for explicit scheme on a Virtex^{TM} 5 330T Board

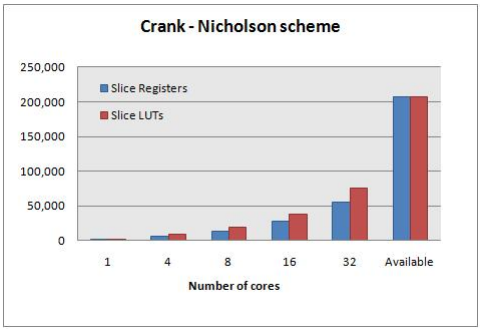
Number of cores	1		32		
Resource Type	Used	Util. %	Used	Util. %	Available
Slice Registers	1,051	1%	39,092	18	207,360
Slice LUTs	1,268	1%	22,584	21	207,360
Occupied Slices	530	1%	17,481	33	51,840
LUT Flip Flop pairs	1,607	-	56,187	-	-
BlockRAM/FIFO	5	1%	176	54	324
DSPs	3	1%	96	50	192



(a)



(b)



(c)

Figure 7.1: Number of used slice registers and slice LUTs

7.1.2 Analysis

The resource utilization of the Crank-Nicholson scheme is greater than the others as a result of the usage of a floating point divider in the FPU and a barrel shifter in the memory controller. These extra components also influence the achieved clock frequency which is slightly smaller in the case of the Crank-Nicholson scheme for given number of cores. Small differences in resource utilization and clock frequency between the two explicit schemes are attributed to the slightly different dataflows of their FPUs.

Figure 7.1 shows the number of slice registers and slice LUTs used by each version of the proposed system for various numbers of processor. Resource utilization increases proportionately to the number of cores in all the implemented schemes. Following this tendency, we can observe that a 64-core system for all schemes can be easily hosted in the target device. Resource measurements were not obtained in this case as memory because of the excessive memory requirements of the synthesizer.

Figure 7.2 shows the maximum achievable clock frequency of the three systems for various numbers of cores. As we can see the frequency is slightly declining as the number of cores increases. As the system increases in size

Table 7.2: Resource Utilization for explicit scheme with heat-diffusion equation transform on a Virtex™ 5 330T Board

Number of cores	1		32		
Resource Type	Used	Util. %	Used	Util. %	Available
Slice Registers	1,054	1%	39,188	18	207,360
Slice LUTs	1,278	1%	45,398	22	207,360
Occupied Slices	526	1%	17,299	37	51,840
LUT Flip Flop pairs	1,610	-	56,681	-	-
BlockRAM/FIFO	5	1%	176	54	324
DSPs	3	1%	96	50	192

Table 7.3: Resource Utilization for Crank-Nicholson scheme with heat-diffusion equation transform on a Virtex™ 5 330T Board

Number of cores	1		32		
Resource Type	Used	Util. %	Used	Util. %	Available
Slice Registers	1,564	1%	55,444	26	207,360
Slice LUTs	2,239	1%	76,274	36	207,360
Occupied Slices	931	1%	29,654	57	51,840
LUT Flip Flop pairs	2,735	-	92,006	-	-
BlockRAM/FIFO	9	1%	192	59	324
DSPs	3	1%	96	50	192

the routing becomes harder and has an negative impact on the maximum achievable clock frequency. At this point it should be mentioned that synthesis, placement and routing have been done using the default optimization strategy of ISE suite. A timing optimization strategy could have yielded slightly greater frequency.

7.2 Speed comparison

7.2.1 Methodology

Speed comparisons have been made against a Pentium™ Core 2 Duo CPU with 2 GHz clock frequency and 3Gbyte RAM. The software versions of the implemented algorithms were written in C++, compiled with the g++ compiler and executed in Windows Vista environment. The Crank-Nicholson scheme was tested against two software versions of itself: one using odd-even reduction and one LU-decomposition. The software execution time estimation was made with Vtune™ performance analyzer.

Soft versions were executed for systems of 4096 and 8192 size, five times each. The number of time steps depended on the method; For explicit schemes $N_\tau = N_S^2$ and for the Crank-Nicholson scheme $N_\tau = N_S$. The aver-

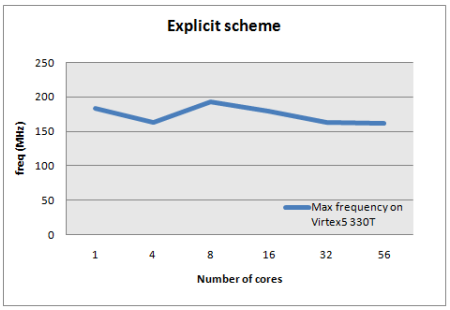
age of these five execution times for each system size was computed. These average times were divided by the number of time steps, in order to obtain the average execution time for a single time step. For the Crank-Nicholson scheme execution times were obtained using both LU-decomposition and odd-even reduction. For the latter we obtained measurements for each of the three phases. The execution time was measured in CPU_CLK_UNHALTED events. A metric that counts the processor's clock ticks for non-halt cpu state [30]. The execution times were obtained using the formula:

Execution_time = CPU_CLK_UNHALTED.TOTAL_CYCLES / Processor_Frequency / Number_of_Cores

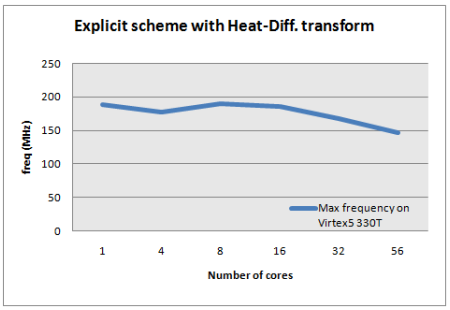
It should be pointed out that the execution time calculated in this way implies perfect load balancing on both cores of the Core 2 Duo architecture. Once the single-time step average execution times for system sizes 4096 and 8192 were obtained, they were divided by the FPGA execution times. The latter were measured in clock ticks by simulation on Modelsim Simulator for architectures with 4, 8, 16 and 32 cores and divided by the maximum achieved frequency for each architecture (see also fig. 7.2). Averaging two speedup values for each architecture - one for $N_S = 4096$ and the other for $N_S = 8192$ - we obtained the final speedup value for a single time step. These values are shown in figure 7.3. The last column of each graph that refers to 64 processors was not obtained by simulation but extrapolating the measured execution times and the maximum clock frequencies of the corresponding architectures and constitutes a projected value. The accommodation of 64 cores on the target device is feasible according to the resource utilization results (see also figure 7.1).

7.2.2 Analysis

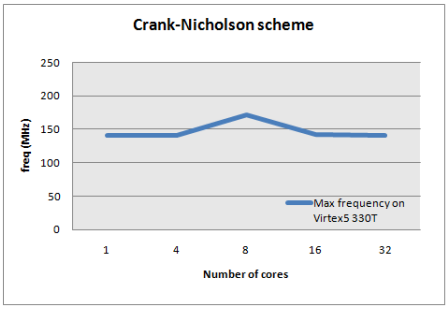
The comparison indicates that 8 cores on average equal the performance of Pentium^{TM} Core 2 Duo CPU. Each time the number of cores in the architecture is doubled, the performance of the reconfigurable system is increased nearly twofold. The highest performance is achieved by the explicit scheme, which is approximately 4x times faster than the CPU using 32 cores and can reach a projected 8x using 64 cores. As for the Crank-Nicholson scheme the performance gain is worse compared to software LU-decomposition as the latter is more efficient over CPUs with small scale parallelization, compared to the gain over software odd-even reduction (fig. 7.4(b)). As for the different phases of odd-even reduction (fig. 7.4(a)), the best performing one is the forward phase contrary to the back substitution phase that demonstrates the poorest performance. This happens because of the limited data throughput from the data memory to the FPU. This overhead is negligible compared to the bulk of computations of the forward reduction phase, but is significant compared to the limited computations of back substitution phase.



(a) Explicit scheme implementation

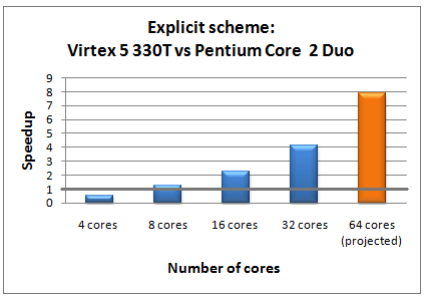


(b) Explicit scheme with heat-diffusion transform implementation

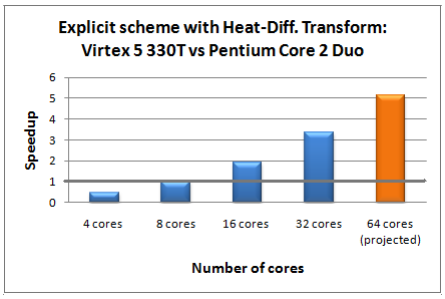


(c) Crank-Nicholson scheme implementation

Figure 7.2: Maximum clock frequency achieved for various number of cores

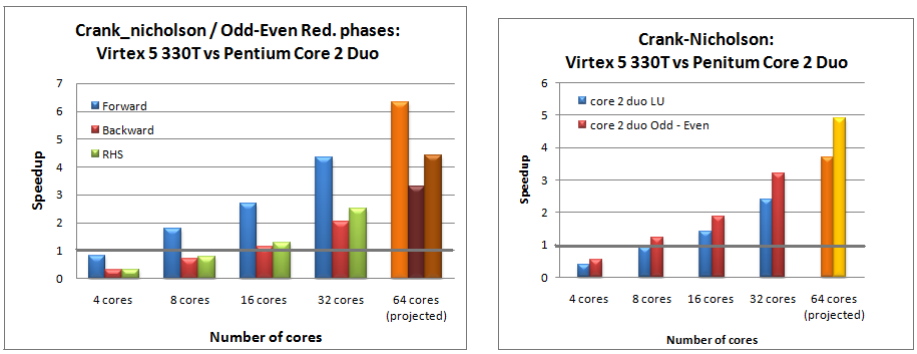


(a)



(b)

Figure 7.3: Average achieved speedup of explicit schemes implementations for various number of cores



(a) (b)

Figure 7.4: Average achieved speedup of Crank-Nicholson scheme implementation for various number of cores

Chapter 8

Conclusions and future Work

The reconfigurable architecture presented in the current thesis has achieved a fair speedup over the Pentium Core 2 Duo processor. The control-flow part of the memory controller of each core that was selected for increased flexibility is mainly responsible for the moderate performance. The presence of commodity CPUs with more than two cores intensifies the antagonism and the proposed architecture can compete them only if multiple FPGAs are utilized. Our architecture can be easily distributed across multiple FPGA devices. Apart from that it can be a good starting point for more specialized architectures that implement a single scheme. A primitive customization has already been made excluding the floating point divider and the shifter from the explicit scheme implementations. More extensive modifications in the control-flow datapath of the memory controller can yield the desirable performance boost.

Apart from the possible modifications targeting better performance of the already implemented algorithms, future efforts could be focused on the implementation of other option pricing techniques on the current platform. An interesting candidate is the binomial option pricing model, as it can be parallelized and entails simple computations.

Appendix A

Core instruction set

Table A.1: The instruction set of the memory controller

Instruction encoding	**Operation**
add / sub instructions	
iadd $\$r$ $\$a$ $\$b$	$\$r = \$a + \$b$
iaddi $\$r$ $\$a$ imm	$\$r = \$a + imm$
isub $\$r$ $\$a$ $\$b$	$\$r = \$a - \$b$
isubi $\$r$ $\$a$ imm	$\$r = \$a - imm$
shift instructions	
isll $\$r$ $\$a$ $\$b$	$\$r = \$a << \$b$
islli $\$r$ $\$a$ imm	$\$r = \$a << imm$
isrl $\$r$ $\$a$ $\$b$	$\$r = \$a >> \$b$
isrli $\$r$ $\$a$ $\$b$	$\$r = \$a >> imm$
load / store integer instructions	
ildin $\$r$ $\$b$	$\$r$ = mem[$\$b$]
ildini $\$r$ imm	$\$r$ = mem[imm]
iswin $\$a$ $\$b$	mem[$\$b$] = $\$a$
iswini $\$a$ imm	mem[imm] = $\$a$
load / store float instructions	
ildf $\$r$(f) $\$b$	$\$r$(f) = mem[$\$b$]
ildfi $\$r$(f) imm	$\$r$(f) = mem[$imm$]
ildfa $\$r$(f) $\$a$ $\$b$	$\$r$(f) = mem[$\$a$], $\$a = \$a + \$b$
ildfai $\$r$(f) $\$a$ imm	$\$r$(f) = mem[$\$a$], $\$a = \$a + imm$
ildfs $\$r$(f) $\$a$ $\$b$	$\$r$(f) = mem[$\$a$], $\$a = \$a - \$b$
ildfsi $\$r$(f) $\$a$ imm	$\$r$(f) = mem[$\$a$], $\$a = \$a - imm$
iswf $\$b$	mem[$\$b$] = FPU result
iswfi imm	mem[imm] = FPU result
iswfa $\$a$ $\$b$	mem[$\$a$] = FPU result, $\$a = \$a + \$b$
iswfai $\$a$ imm	mem[$\$a$] = FPU result, $\$a = \$a + imm$
iswfs $\$a$ $\$b$	mem[$\$a$] = FPU result, $\$a = \$a - \$b$
iswfsi $\$a$ imm	mem[$\$a$] = FPU result, $\$a = \$a - imm$
branch instructions	
ibeq $\$a$ $\$b$ imm	if $\$a == \$b$ jump to imm
ibne $\$a$ $\$b$ imm	if $\$a$! = $\$b$ jump to imm
iblt $\$a$ $\$b$ imm	if $\$a < \$b$ jump to imm
ibge $\$a$ $\$b$ imm	if $\$a >= \$b$ jump to imm
Jump / Stall instructions	
ij imm	jump to imm
iwst $\$r$(f)	wait for FPU position $\$r$(f) to be freed
ijcc	jump to the current execution context after signal handling
Instructions for sending data to other cores or the host	
isnm $\$r$ $\$b$	$\$r$ (destination core ID / memory address) = mem[$\$b$]
isnmi $\$r$ imm	$\$r$ (destination core ID / memory address) = mem[imm]
Other instructions	
isint imm	Set imm as the address of the signal handling routine
isintc imm	Set imm as the address of the signal handling routine/ clear the FPU storage elements.

Bibliography

[1] F. Black and M. Scholes, "The pricing of options and corporate liabilities," in *The Journal of Political Economy*, vol. 81, pp. 637–654, 1973.

[2] W. Hafner and H. Zimmermann, *Vinzenz Bronzin's Option Pricing Models: Exposition and Appraisal*. Springer, 2009.

[3] P. Wilmott, S. Howison, and J. Dewynne, *The Mathematics of Financial Derivatives: A Student Introduction*. New York, USA: Cambridge University Press, 1995.

[4] J. C. Cox, S. A. Ross, and M. Rubinstein, "Option pricing: A simplified approach," *Journal of Financial Economics*, vol. 7, pp. 229–263, 1979.

[5] C. Leentvaar, "Numerical solution of the black-scholes equation with a small number of grid points," Master's thesis, Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Section Applied Mathematics, Numerical Analysis Group, 2003.

[6] H. Windcliff, P. A. Forsyth, and K. R. Vetzal, "Analysis of the stability of the linear boundary condition for the black-scholes equation.," *Journal of Computational Finance*, vol. 8, pp. 65–92, 2003.

[7] P. Wilmott, *Derivatives: The Theory and Practice of Financial Engineering*. Wiley, university ed., 1998.

[8] Z. Fei, Y. Goto, and E. Kita, "Solution of black-scholes equation by using rbf approximation.," in *Proceedings of the International Symposium on Frontiers of Computational Science*, vol. 81, pp. 637–654, 2005.

[9] O. Pironneau and Y. Achdou, *Computational Methods for Option Pricing*. SIAM, illustrated ed., 2005.

[10] M. J. Brennan and E. S. Schwartz, "Finite difference methods and jump processes in the pricing of contingent claims: A synthesis," *The Journal of Financial and Quantitative Analysis*, vol. 13, pp. 461–474, 1978.

[11] W. F. Ames, *Numerical methods for partial differential equations*. Computer science and applied mathematics, Academic Press, 2nd ed., 1977.

[12] L. N. Trefethen and D. Bau, *Numerical linear algebra*. SIAM, 1st ed., 1997.

[13] B. Neta and H.-M. Tai, "Lu factorization on parallel computers," *Computers and Mathematics with Applications*, vol. 11, pp. 573–579, June 1985.

[14] H. S. Stone, *Parallel Tridiagonal Equation Solvers*, vol. 1 of *ACM Transactions on Mathematical Software (TOMS)*. New York, USA: ACM, 1975.

[15] M. I. Bueno and F. M. Dopico, "Stability and sensitivity of tridiagonal lu-factorization without pivoting," *BIT numerical mathematics*, vol. 44, pp. 651–673, Dec. 2004.

[16] W. Gander and G. H. Golub, "Cyclic reduction - history and applications," in *Proceedings of the Workshop on Scientific Computing*, pp. 73–85, Mar. 1997.

[17] M. Elm and J. K. Anlauf, "Pricing of derivatives by fast, hardware based monte carlo simulation.," in *proc. ASAP IEEE International Conference on Application-specific Systems, Architectures and Processors*, (Boston, Massachusetts, USA), July 2006.

[18] D. B. Thomas, J. A. Bower, and W. Luk, "Automatic generation and optimisation of reconfigurable financial monte-carlo simulation," in *Proc. SIAM Conference on Financial Mathematics and Engineering (FM 2006)*, pp. 168 – 173, July 2007.

[19] X. Tian, K. Benkrid, and X. Gu, "High performance monte-carlo based option pricing on fpgas," in *Engineering Letters of International Assotiation of Engineers*, vol. 16, Aug. 2008.

[20] R. Baxter, R. Booth, *et al.*, "Maxwell a 64 fpga supercomputer.," in *Proc. Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, (University of Edinburgh, Scotland, United Kingdom), pp. 287–294, Aug. 2007.

[21] V. Podlozhnyuk and M. Harris, "Monte carlo option pricing," June 2008.

[22] N. Singla, M. Hall, B. Shands, and R. D. Chamberlain, "Financial monte carlo simulation on architecturally diverse systems," in *Proc. Workshop High-Performance Computational Finance*, pp. 168 – 173, Nov. 2008.

[23] Q. Jin, D. B. Thomas, W. Luk, and B. Cope, "Exploring reconfigurable architectures for binomial-tree pricing models," in *Lecture Notes in Computer Science*, pp. 244 – 255, Aug. 2008.

[24] E. Motuk, S. Bilbao, and R. Woods, "Implementation of finite difference schemes for the wave equation on fpga," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, (Philadelphia, USA), pp. 237–240, May 2005.

[25] Q. Jin, D. B. Thomas, and W. Luk, "Exploring reconfigurable architectures for explicit finite difference option pricing models," in *proc. International Conference on Field Programmable Logic and Applications*, pp. 73 – 78, 2009.

[26] W. Kluge, *The organization of reduction, data flow, and control flow systems*. MIT Press, 1992.

[27] J. A. Sharp, *Data Flow Computing: Theory and Practice*. Intellect Books, 1992.

[28] Xilinx, "Floating-point operator v4.0," Apr. 2008.

[29] P. S. Graham and M. Gokhale, *Reconfigurable Computing Accelerating Computation with Field-Programmable Gate Arrays*. Dordrecht, The Netherlands: Springer, first ed., 2005.

[30] Intel, "Intel vtune performance analyzer how-to guide."