AUTOMATIC FRAMEWORK TO GENERATE RECONFIGURABLE ACCELERATORS FOR OPTION PRICING APPLICATIONS

Nam Khanh Pham[1,2], Khin Mi Mi Aung[2], Akash Kumar[3]
[1]*Department of Electrical and Computer Engineering, National University of Singapore, Singapore*
[2]*Data Storage Institute, A*STAR, Singapore*
[3] *TU Dresden, Center for Advancing Electronics Dresden (cfaed), Germany*
phamnamkhanh@u.nus.edu
Mi_Mi_AUNG@dsi.a-star.edu.sg
akash.kumar@tu-dresden.de

***Abstract*—Option Pricing is a fundamental application in most financial institutions dealing with derivative market. It frequently requires huge computational effort and low latency demand. Therefore, a number of different Option Pricing implementations have been developed on FPGA-based platform. However, none of the existing works cover more than one models or different types of options, which yields problem of productively implementing several hardware accelerators for different models. To fill in the gap, we propose a design flow for generating efficient hardware accelerators for option pricing applications with different models and option types. The framework boosts the designers productivity and enables quick prototyping on FPGA platform by providing general template architecture for option pricing applications. The architecture comes along with a prebuilt design library, which covers a wide range of popular financial models. Experimental results for four models show that the accelerators generated from our design flow outperform their counterpart software implementation with two order of magnitude speedup. While comparing with existing hardware designs for the same models, our framework can produce the accelerators that overcome most of manual designed engines.**

I. Introduction

Option trading is the fundamental operation of every financial institution; therefore, evaluating the option accurately and fairly is essential for the business of these financial company and their customers [11]. However, the appearance of highly complex options, which involve multiple underlying assets or contains complicated contractual features, makes the pricing problem a very challenge task. Besides, new underlying models for describing the stock prices have been added more and more features to better reflect the real behavior of the market. All of these complexities make the option pricing problems computationally intensive. The low latency demand is another important factor that drives financial firms accelerate their pricing process. In the era of automated trading, the prices of the financial products change within a small fraction of second. In this competitive environment, faster and more accurate pricing can bring a huge advantage and profit for financial firms. Therefore, high performance computing needs to be exploited to support real time pricing [25].

Amongst other options for high performance computing in finance like GPU or CPU cluster, FPGA has proven itself as a promising candidate for option pricing application due to the following features. First of all, FPGA accelerators can exploit different levels of parallelism inherent in the pricing methods: from instruction level, thread level to data level with pipeline parallelism [17]. Moreover, they can provide the same performance with much less power, size and cost in comparison with commodity CPU [17]. Last but not least, the potential on low latency network interface for updating market information and trading decision brings the unique advantage for FPGA implementation.

Because of its high potential and advantages, FPGA has been intensively studied over the last few years to accelerate the option pricing problem. Although a number of hardware (HW) implementations for option pricing have been reported with huge speedup and lots of energy saving [20], the productivity is still a challenge when it comes to FPGA design as compared to GPU and CPU [14]. Most existing works proposed the hardware designs for one particular pricing model only: either Black Scholes [19–21] or Heston [5, 6, 8, 9]. Because the computational features/ characteristics of pricing models are different in nature, when users need to switch between different models, the designers have to restart the design process from beginning, which consumes lots of time and effort. To address this problem, we introduce a generic framework that can help designers to generate efficient and high quality hardware accelerators, which can facilitate different pricing models as well as various types of options. Following are the main **Contributions** of our framework:

- A generic design flow that can automatically optimize and generate the hardware accelerators from a high level description of option pricing application;
- A template of modular and parameterizable hardware architecture that covers all different computational features of various pricing models;
- A library for the hardware implementation of most popular pricing models: Black Scholes; Merton; Heston and Bates;

978-1-5090-3707-0/16/$31.00

- A heuristic to find optimization parameters for above-mentioned hardware designs;

The rest of the paper is organized as follows. In Section 2, an overview of option pricing application and underlying models are provided. Section 3 highlights existing works related to hardware implementation of the option pricing. The proposed design flow and the heuristic for optimizing design parameters are detailed in Section 4. The generic hardware architecture is described in Section 5. Section 6 discusses the experimental setup and the results. Finally, summary and future plans are concluded in Section 7.

II. Background

A. Option overview

Option is a contract between two parties that provide the buyer the right to execute a transaction on one or several underlying assets (stock, currency, index or debt) with a *strike price K* at a future moment T (*maturity* or *expiry*) under specific conditions. An option is a Call Option if the buyer has the right to buy in the future; whereas an Option with the right to sell is called Put Option. The profit from Option at maturity is defined by the *payoff*, which depends on the exercise condition, the Strike price K and the underlying asset price at maturity T. Our framework can be applied to a wide range of European options, which are executed only at Expiry. Based on the definition in [11], the following Option Type are implemented in our framework:

- Vanilla Option: the most popular and traditional type of option, the payoff condition depends on the strike price K, and the stock price at Expiry;
- Asian Option: the Strike price is defined as the arithmetic average of the underlying asset price during the contract period;
- Barrier Option: the payoff condition is defined by whether its underlying assets price achieves specific values.
- Binary Option: Binary options are options with discontinuous payoffs. A simple example of a binary option is a cash-or-nothing call. This pays off nothing if the asset price ends up below the strike price at time T and pays a fixed amount, Q, if it ends up above the strike price.
- Lookback Option: The payoffs from lookback options depend on the maximum or minimum asset price reached during the life of the option.

B. Option pricing problem

Options are one of the most widely traded products in the market [11]. Therefore, financial institutions need to define the fair option price to avoid the arbitration opportunity from their competitors. The typical option pricing procedure is illustrated in Figure 1.

First of all, the evolvement of the underlying asset price needs to be described through a model. Based on the model

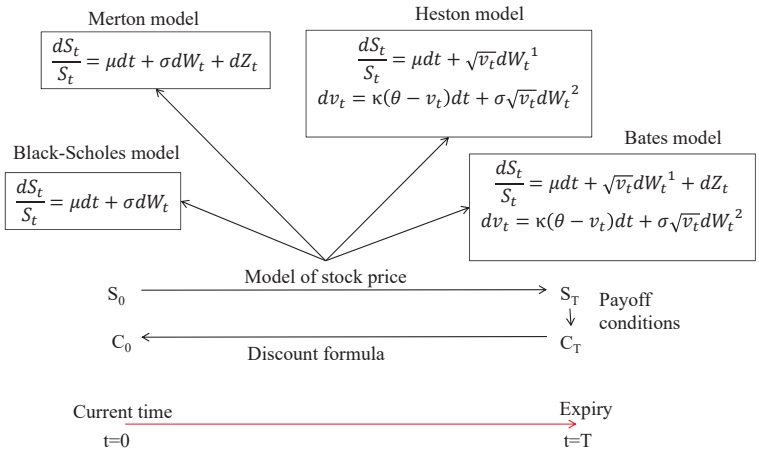

Figure 1. Option pricing procedure

and current assets price S_0, the price at maturity S_T can be calculated. After that, the option price at maturity C_T is computed using $payoff$ function. Finally, the future option price C_T is discounted to get the current option price C_0. From the pricing process, we can recognize the ultimate importance of the underlying asset price model. The most popular and widely used models are Black-Scholes model, Merton model, Heston model and Bates model [11].

1) Black-Scholes model: Black-Scholes (BS) model has been introduced in 1973 and widely adopted by financial institution for option pricing problem [11]. In this model, Black and Scholes use perfect market hypothesis to assume that all the known information of market has been included in the prices of traded asset. Therefore, the asset price follows a Brownian motion with constant drift μ and volatility σ:

$$\frac{dS_t}{S_t} = \mu dt + \sigma dW_t \tag{1}$$

where W_t is the Wiener random process, which is in the order of $N(0,T)$, a Gaussian distribution with T as standard deviation.

$$dW_t = W_t - W_0 \approx N(0,T) \tag{2}$$

2) Heston model: One disadvantage of the BS model is the assumption that the volatility is always constant. This assumption does not reflect the properties of real market and may introduce discrepancy between computed option price and real price. Therefore, Steven L. Heston introduced the stochastic volatility to improve the Black-Scholes model. This improvement described the stock price behavior much more accurately and the model is widely accepted in financial community [11]. The Heston model has two stochastic differential equations which describe the randomness in both the asset price S and volatility V:

$$\frac{dS_t}{S_t} = \mu dt + \sqrt{v_t} dW_t^1 \tag{3}$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} dW_t^2 \tag{4}$$

Here, W^1 and W^2 are two Brownian motions with the correlation ρ that model the randomness of the market. t is the time, and the other parameters further specify the specific behavior of the financial market.

3) Merton model: Another way to bring the BS model closer to the real market is to include a Jump component to the diffusion process of the stock price. That was proposed by Merton [13], and the formula is described in Equation 5.

$$\frac{dS_t}{S_t} = \mu dt + \sigma dW_t + dZ_t \tag{5}$$

where dZ_t describes the Jump component and usually follows a Poison process.

4) Bates model: Bates introduced a new model which contains both the features of stochastic volatility and jump process in 1996 [3]. The asset price is then described in Equation 6 and 7.

$$\frac{dS_t}{S_t} = \mu dt + \sqrt{v_t} dW_t^1 + dZ_t \tag{6}$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} dW_t^2 \tag{7}$$

C. Pricing methods

From these models we can have several choices to define the dependency of the Option price on the Stock price S, the volatility V and the time T. The way that we describe this relationship also defines the numerical method used to price the option. First of all, the stochastic differential equations (SDE) are the most straight forward way to describe the abovementioned dependencies. For numerical computation, there are several options to derive the option price, namely: Monte-Carlo (MC) Simulation, Finite Different methods, Binomial tree and Quadrature Method. Among them, MC is the most widely used methods because of its simplicity and generality: it can be applied to almost all option types as well as underlying stock models [10]. Therefore, our framework focuses on the hardware accelerators for the Monte-Carlo method in option pricing applications.

III. Related Work

Because of its potential and relevance, FPGA is extensively studied for improving the performance of option pricing applications.

A. Black Scholes Model

The earliest works on FPGA-based accelerators for option pricing were dedicated to the traditional Black Scholes models. Being one of the active academic group focused on this problem, Xiang Tian *et al.* developed their first option pricing accelerator on Maxwell platform, which achieves 340 times speedup over 2.66 GHz Intel Pentium IV software implementation [19]. Their later design on Quasi Monte Carlo pricing engine has 544 times speedup over Intel Xeon CPU and more than 10 times over NVIDIA 8800GTX GPU [20]. Another research group which is very active in this area is Custom Computing group from Imperial College London. Tse *et al.* introduced the FPGA implementation for exotic option that outperforms the CPU 313 times and 2.2 times faster than Tesla C1060 GPU implementation [23]. A mixed-precision approach has been proposed by Gary *et al.* for accelerating MC simulation in [7]; as a result, pricing the Asian option can be shortened by 44 times using FPGA Virtex-6.

B. Heston Model

Heston model is the second most popular pricing model that has been adopted in the hardware community. The pioneer FPGA design by Schryver *et al.* [8] can achieve 35% speed of the Tesla C2050 GPU implementation with only 40% energy consumption. The same authors have reported a better design for Multi Level Monte Carlo (MLMC) method with Barrier Option in [9]. Extending previous works, Brugger et.al. proposed a framework for mixed-precision design for accelerating MLMC pricing method in [6] and a hardware implementation on Zynq platform that is 12.5 times faster and 153 times more power efficient than previous design [5].

Our work is different from above-mentioned designs in the fact that our framework can be used to generate the pricing engines for multiple different pricing models and option types, while previous works introduced the architectures applicable for one particular pricing model only.

C. Framework

The closest work to our contribution is the one proposed by Thomas *et al.* in [18]. The authors have developed a methodology to automatically generate reconfigurable hardware for Monte Carlo simulation in financial applications. The hardware accelerators from the design flow can achieve an average of 87 times speedup compared with software implementation on 2.66GHz Pentium IV. Our framework fundamentally differentiates from previous one in 2 ways. First, we focus closely on the option pricing applications; hence, our proposed generic hardware architecture are carefully customized to the computational characteristics of the pricing problem. This customization brings advantages in both the development time as well as the performance of the generated accelerators. Second, we integrate an optimization process to derive the most efficient design parameters for the hardware accelerators.

IV. Design flow and optimization framework

In this section, we propose a generic design flow that can generate efficient accelerators for different types of options and all the models mentioned-above. Besides, a framework that can be used to optimize the engine parameters according to different design constraints is also developed to shorten the development process. Our proposed design is illustrated in Figure 2. The input of the design flow is a specification of an option pricing query which includes all the information needed for generating the HW accelerator and obtaining the

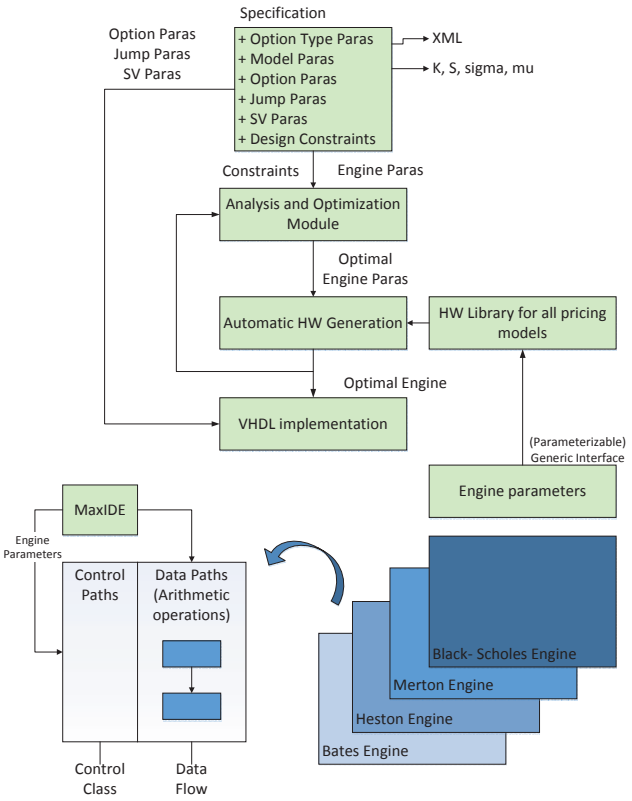

Figure 2. Generic framework

option price. The specification is written in XML file and contains following details about the pricing request:

- Option Type Parameters: Vanilla, Asian, Barrier, Binary or Lookback;
- Model Parameters: Black Scholes, Merton, Heston or Bates;
- Option Parameters (general variables for all types of option): Strike price K, current stock value S_0, volatility σ, mean of expected return μ, number of time steps, number of paths, maturity T;
- Jump Parameters: jump rate λ, jump mean α, jump deviation β ;
- Stochastic Volatility (SV) Parameters: reversion rate κ, Variance of volatility ξ, long term variance θ, correlation coefficient ρ ;
- User constraints: timing constraint, precision constraint or HW resource constraint.

The first main block of the framework is the Analysis and Optimization Module (AOM). Basically, this module considers the Option Type and Option Model, analyzes the Design Constraints and generates the optimal Engine Parameters for the next block. The essence of AOM is an optimization algorithm which is a heuristic search algorithm that intelligently traverses the design space to find the most efficient design point. The design space is given by all the available set of Engine Parameters such as: number of bits for exponential (e) and mantissa (m) of floating point representation; number of hardware instances $pipes$ and the frequency of the Hardware F. The procedure is described in **Algorithm** 1. The input of the Algorithm is the design constraints given by the designer in the Specification of pricing request. The constraints might contain requirement of the error ϵ, the available hardware resource HW, and the timing condition, which is translated to throughput requirement tp. Taking into account the input, the heuristic produces the most efficient engine E_{opt} configuration, which is represented as a tuple of number representation (m, e), number of HW instances $pipes$, and the operating frequency F. Using profiling method, the Algorithm first defines the number representation $(m_0 = 41, e_0 = 7)$, which satisfied the error requirement $\epsilon = 5\%$. Then, it synthesizes the smallest configuration $E_0 = \{(m_0, e_0), pipes = 1, F = 100\}$ to obtain the HW usage of 1 instance with standard frequency $F = 100MHz$. The flag of successfully finding the optimal configuration is set to False in line 3. The maximal frequency that makes the minimal configuration with 1 HW instance satisfy the throughput requirement is assigned in line 4. The upper bound of the number of HW instance implemented is set in line 5. The next FOR loop iterates through possible number of HW instances (Line 6), computes the minimal operating frequency (Line 7), and checks if the minimum configuration for that number of HW instance is feasible to implement on FPGA (line 8). If the minimal configuration is not feasible, the loop reduces the number of HW instances (Line 21). Otherwise, it indicates that an available configuration found (Line 11) and continues searching for the maximal frequency using binary search (the While loop). At the current state, the heuristic search is quite simple and require synthesis tool to check the feasibility of the design. In future, a hardware utilization model along with a performance model can be integrated for faster evaluation and facilitate more advance optimization algorithms.

The output of the AOM is the most efficient value of Engine Parameters which are passed to the Automatic HW Generation (AHG) block. This block uses these Engine Parameters to configure the available HW modules in the HW library, combine these modules into a complete engine and generate the VHDL file for the engine.

To better understand the mechanism of the framework and the functionality of each module , an overview of the general architecture of the pricing engines is provided in the next Section. As presented in Figure 2, the HW library contains the architecture of 4 pricing engines associated with abovementioned 4 pricing models. Each pricing engine has two main parts: control path and data path. The data paths mainly contain arithmetic operations and will dominate the HW consumption as well as the latency of each engine. The control paths are used to manage the data transfer and synchronization between data path modules. Both the

Algorithm 1 Finding the most efficient Engine Parameters

Input: Design constraint: HW, tp, ϵ
Output: Most efficient Engine Parameters : $E = \{(m, e), pipes, F\}$
Profiling to get the number representation (m_0, e_0) satisfied the accuracy ϵ
Synthesize the minimal HW configuration with: $E_0 = \{(m_0, e_0), pipes = 1, F = 100\}$ to get the results HW_0
$available = FALSE$
$F_{max} = tp$
$max\_pipes = HW/HW_0$
for $i = max\_pipes$ to 1 **do**
    $F_{min} = tp/i$
    if synthesize($E_i = \{(m_0, e_0), pipes = i, F = F_{min}\}$) = feasible **then**
        $pipes_{opt} = i$
        $F_{opt} = F_{min}$
        $available = TRUE$
        while $(F_{max} > F_{min} + 1)$ **do**
            if synthesize($E_i = \{(m_0, e_0), pipes = i, F = F_{max}\}$) = feasible **then**
                $F_{opt} = F_{max}$
                Break
            else
                $F_{max} = (F_{max} + F_{min})/2$
            end if
        end while
    else
        $Next$
    end if
end for
if available=TRUE **then**
    Return $E = \{(m, e), pipes_{opt}, F_{opt}\}$
else
    Print("there is no configuration satisfied the constraint")
end if

control paths and data paths are designed with a focus on flexibility and modularization, that means these modules contain some features that can be parameterized with the Engine Parameters. In the data paths, these features will be the constant and the user defined type, while in the control path the I/O enable signals and the routing signals between blocks are customizable.

V. Pricing Engine Architecture

This section provides the implementation details and architecture of the generic pricing engine mentioned above. First we review the procedure of MC simulation applied to pricing problem with different models; then design of the HW accelerator for MC pricing method is developed with parameterizable functionality in mind.

A. MC method overview

MC method is a numerical method that is widely used to simulate stochastic processes. The essence of the method is based on the procedure of sampling underlying random variables, then computing the outcome of the process and setting the average of all simulated outcomes as the required value. The MC simulation method is intensively used for option pricing problems because it is robust and stable; it can be used to derive options without closed form formula and the complexity of the method does not increase exponentially with the dimension of underlying assets. Therefore, this method is a promising candidate for high performance computing accelerator. Moreover, the independence between simulated paths makes it more attractive for parallel computing systems like FPGAs and GPUs.

The procedure of MC method can be described as follows: firstly, the pricing period is discretized into small time steps δt; then, the continuous stochastics differential equations (SDE) of the pricing model is translated into discrete version to describe the change of the asset price and volatility in one time step. After that, all the random movements of price and volatility within period $[0, T]$ are accumulated to get the asset price for one simulation path. The option price for each path at time T is computed using payoff function. Then, the expected option price at time T is defined as the average of all simulated option prices. Finally, the current price for the option is discounted from its price in time T.

B. HW design of MC engine

The HW architecture of the MC engine is presented in Figure 3 and closely follows the procedure described in the previous Section. The blocks inside the dotted rectangle Point are responsible for computing the movement of stock price for each time step, while the Payoff Core is only executed at the last time step to compute the final price of a Path (outer dotted rectangle). In other words, the Point Rectangle is the inner most loop iterating through all the time steps, while the Path Rectangle is the outer loop iterating through all the simulated paths. The Coeff. Precomputation block is implemented in the highest hierarchal level and executed only once at the beginning of pricing procedure. As suggested by its name, this block precomputes all the constant parameters during the pricing process, so that they are not redundantly recomputed in the inside process. As can be seen in Figure 3, there are two types of input parameters: the Model Parameters and Option Parameters (red color) are configuration inputs, which are used by the HW Generator Block to define appropriate architecture that needs to be loaded. The Model Parameters determine which HW modules from library need to be configured in the Coeff. Precomputation block, Variance Core and Price Core. Based on the type of underlying models, they also decide whether or not to include the Possion Generator and Jump Generator in the design, and how many Gaussian Number Generators (GNG) need to be implemented. Having smaller impact on the design, the Option Type Parameters decide the configuration need to be loaded in the Existence Core and Payoff Core. In contrast to Model Parameters and Option Parameters, the rest of the inputs (Jump Paras, SV Paras, Option Paras) are presented in black color and affect only the execution phase when a particular pricing engine is already loaded into the FPGA. These parameters are fed as input data to relevant modules of the configured pricing engine to produce appropriate output results during execution.

As can be observed from Figure 3, the overall architecture of the pricing engine is developed in a highly modular fashion so that it can accommodate various types of option and underlying models. The functionality of each block is described in detail as follows:

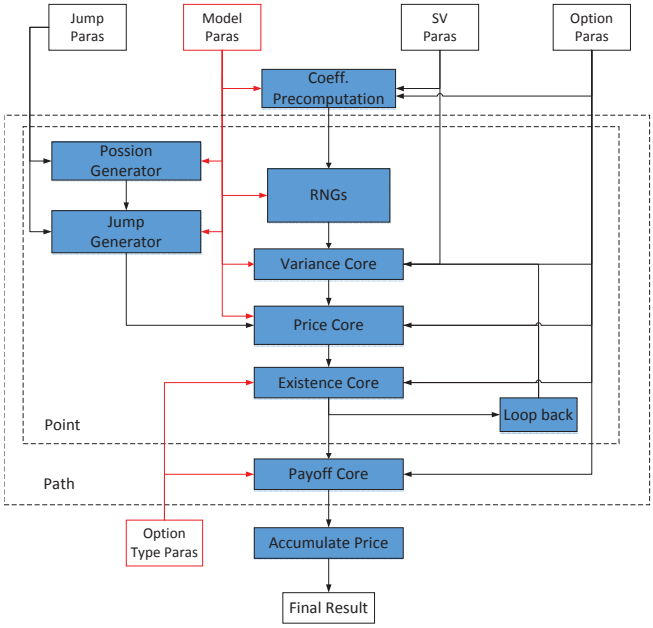

Figure 3. Generic architecture of pricing engines

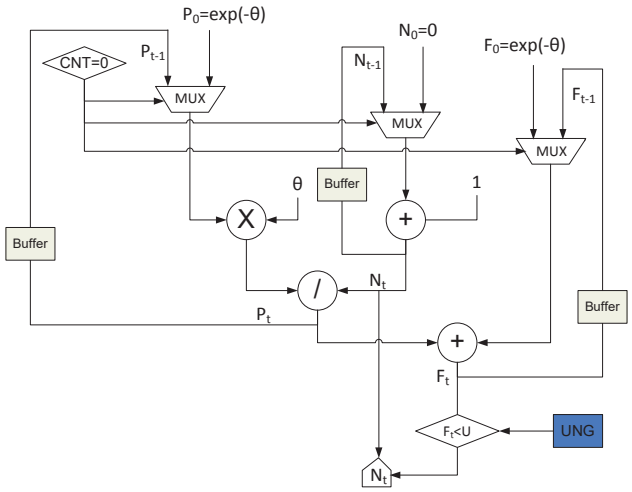

Figure 4. Poisson Generator block

The Poisson Generator block generates a series of numbers following Poisson distribution for the next block; the mean θ of the Poisson distribution are set from Jump Paras input. The HW implementation of the Poisson Generator is presented in Figure 4, following the algorithm in described in Figure 3.9 of [10].

Taking into account the Jump Paras and Possion number N_t from previous block, the Jump Generator computes the sudden change in stock price and passes the result to the Price Core Module. These 2 blocks are available in the pricing engine of Merton and Bates models only.

The GNGs block is used to generate a series of numbers following Gaussian distribution. Firstly, the uniform random

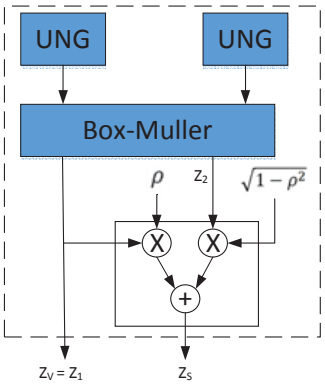

Figure 5. GNG block for SV models

number are generated using Min-Twister methods (block UNG). Then they are converted to Gaussian random number using Box-Muller method. This block is designed with the algorithms given in [12] and [26] .The configuration factor of this block, which is controlled by Model Paras, is the number of GNG instances. For models with constant Volatility (Black Scholes and Merton), there is only 1 instance implemented, while SV models (Heston, Bates) require two instances per time step. Moreover, the Gaussian numbers generated for SV models are correlated to each others with correlation coefficient ρ. The functional scheme of GNGs block for SV models are presented in Figure 5.

The Variance Core is included in pricing engine by Models Paras when working with Stochastic Volatility models (Heston, Bates). Its function is to compute the volatility of the next time step. The discretization formula of this block is given in Equation 8 [16] and the optimized version with precomputed parameters is given in Equation 9.

$$V(t+dt) = V(t) + \kappa(\theta - V^*(t))dt + \sigma\sqrt{V^*(t)}\sqrt{dt}Z_v \quad (8)$$

$$V(t+dt) = V(t) + kpd - V^*(t) * kd + sd * \sqrt{V^*(t)}Z_v \quad (9)$$

where $kpd = \kappa * \theta * dt$; $kd = \kappa * dt$; and $sd = \sigma * \sqrt{dt}$ do not change over iterations and are precomputed in the Coeff. Precomputation block.

The Price Core computes the movement of the price for the next time step and has two different implementation versions for SV models (Heston, Bates) and non-SV models (BS and Merton). Moreover, the discretization formula of this block is also differentiated by the Jump component. Therefore, the discretization formulas of Black Scholes and Heston model are given as in Equation 10, 11 [16]. While considering the Jump component, the coefficient r is adjusted as in Equation 12 [4] and the discretization scheme for Merton and Bates are described in Equation 13, 14. The decision on configuring the appropriate version for this block again depends on the Model Paras input.

$$S(t+dt) = S(t) * exp((r - 0.5 * \sigma^2)dt + \sigma\sqrt{dt}Z_s) \quad (10)$$

Table I
IMPLEMENTATION FOR DIFFERENT OPTION TYPES

Option Type	Carry Value	Carry Core	Payoff Core
Vanilla	NA	NA	$C = max(S_T - K; 0)$
Asian	Sum of previous prices	$S_{sum} = S_{sum} + S_t$	$C = max(\frac{S_{sum}}{n} - K; 0)$
Barrier	Existence	$E = E\&(S_t < H)$	$C = (E)?max(S_T - K; 0) : 0$
Binary	NA	NA	$C = (S_T > K)?Q : 0$
Lookback	Min of previous prices	$S_{min} = min(S_{min}, S_t)$	$C = max(S_{min} - K; 0)$

$$S(t+dt) = S(t) * exp((r - 0.5 * V(t))dt + \sqrt{dtV(t)}Z_s) \quad (11)$$

$$r_{adj} = r - \lambda * (exp(a + 0.5 * b^2) - 1) \quad (12)$$

$$S(t+dt) = S(t) * exp((r_{adj} - 0.5 * \sigma^2)dt + \sigma\sqrt{dt}Z_s + J) \quad (13)$$

$$S(t+dt) = S(t) * exp((r_{adj} - 0.5 * V(t))dt + \sqrt{dtV(t)}Z_s + J) \quad (14)$$

The Carry Core, which computes the additional value needed to be carried during the pricing path to define special execution condition. The Carried Value and their implementation for specific type of options such as Asian Option, Barrier Option and Lookback Option are presented in Table I.

The Payoff Core is computed only once per simulated path at maturity T and its configuration depends on the value of Option Type Parameters. The implementations for different option types are given in Table I. For the sake of brevity, only the formula for Call options are presented. Finally, the price of all the simulated paths are accumulated to form the final accumulated price.

As can be seen from the implementation of Variance Core, Price Core and Existence Core modules, the result values of these blocks are dependent on the values from previous time steps which are stored and transfered to them by the Loop back Block.

VI. EXPERIMENTAL RESULTS

A series of experiments are conducted to evaluate the efficiency and performance of the hardware accelerators generated from our framework. In our implementation, the option pricing request is described in XML format, the proposed design flow and optimization framework in Section IV are developed using Python and Maxeler IDE [15]. The generic Pricing Engine in Section V is developed by Maxj data flow language, the Java High Level Synthesis language from Maxeler [15]. All 4 pricing engines are developed with C-slow optimization techniques [24]. The experiment results are obtained by implementing and running the engines on Maxeler Workstation model MAX3424A [15], which features with a Xilinx Virtex-6 SX475T FPGA device and Intel Core i7 870 2.93 GHz with 16GB RAM.

A. Comparison with SW implementations

In the first experiment, we compare the throughput of our FPGA accelerators with the software implementation for CPU. The competitor in this experiment is the online option pricing service Premia provided by INRIA (the French national institute for research in computer science and control) [2]. For all the models, we choose to price European Vanilla Option with 1 million paths and 100 time steps. The throughput of both implementations and the speedup of our pricing engines over the CPU implementation are reported in Figure 6. As can be observed from the Figure, all the pricing engines from our framework achieve two orders of magnitudes higher throughput over the SW implementations. The speedup is more significant for complex models since each simulated path of these models requires much more computation effort and execution time for SW implementation. On the other hand, for highly pipelined hardware accelerators with one output per clock cycle, the complex data path of these models do not significantly affect the throughput of the engine.

B. Comparison with other HW accelerators

To further examine the performance of our HW accelerators, we compare the throughput and speedup of our pricing engine with other available engines in literature. Since there is no work covering all the pricing models as ours, we have considered different competitors: for the Black Scholes engine the most recent work is reported by [22], while the most efficient manual design is proposed in [23]; for Heston pricing engine, [8] is the work using the same Monte Carlo method; for Bates models, the only available

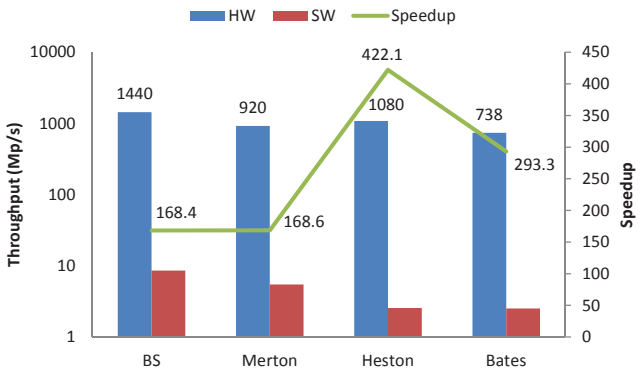

Figure 6. Comparison with other SW implementations

Table II
COMPARISON WITH OTHER HW IMPLEMENTATIONS

Work	FPGA	Frequency	Throughput			
			BS	Merton	Heston	Bates
[23]	Virtex 5	200	3200	NA	NA	NA
[22]	Virtex 5	80	640	NA	NA	NA
[8]	Virtex 5	100	NA	NA	142.7	NA
[5]	Zynq	100	NA	NA	459	NA
[1]	Virtex 6	175	NA	NA	NA	700
Ours	Virtex 6	115-140	1680	920	1080	738

work reported is from Maxeler [1]. Table II summarizes the comparison results. Since the results from above-mentioned works were reported with different pricing set-up (option type, number of simulated paths, number of time steps), we used throughput (number of computed time steps per second) as the performance metrics to put all the results in the same perspective. As can be seen from the table, our engine for Black Scholes model has around 2.25 times better performance over the design in [22] and achieves about 45% performance of the highly customized design in [23]. For the Heston models, the accelerators from our framework have clear advantage over previous works. Part of the reasons for this improvement comes from the technology of the devices, but the main explanation comes from our highly pipelined architecture. For the Bates model, we use the same technology as Maxeler implementation but can achieve around 5% improvement in throughput by using more efficient discretization scheme and simpler methods of generating volatility.

Although the main advantage of our proposed framework is the productivity and reduction on development time, it is hard to quantify and compare with previous work in this aspect. From authors' experience, following the framework and modular architecture, a designer with little knowledge about option pricing applications can develop a new engine for new models or new option types in less than a week.

VII. CONCLUSION

In this work, a framework for generating option pricing hardware accelerators has been proposed. The framework is combined with a highly modular architecture design that can cover four popular pricing models and numerous type of options. Moreover, a heuristic for finding local optimal parameter set for the pricing engines is developed to further improve the performance of generated accelerators. As a result, the engines developed from our framework can achieve a speedup of 2 orders of magnitude compared to SW implementations and overcome most of other reported accelerators. In the future, we plan to develop performance and hardware resource models for each of the pricing engines to reduce the time for estimating performance and hardware usage so that we can apply more sophisticated optimization techniques for design space exploration. Another direction that we would like to consider is to develop the framework on Vivado HLS platform to bring more flexibility and further performance improvement to the architecture.

ACKNOWLEDGMENT

This work is supported in part by the German Research Foundation (DFG) within the Cluster of Excellence Center for Advancing Electronics Dresden (cfaed).

REFERENCES

[1] Maxeler App Galery, 2015. http://appgallery.maxeler.com/.
[2] PREMIA - A platform for pricing financial derivatives, 2015. www.rocq.inria.fr/mathfi/Premia/index.html.
[3] D. S. Bates. Jumps and stochastic volatility: Exchange rate processes implicit in deutsche mark options. *Review of financial studies*, 9(1):69–107, 1996.
[4] M. Briani. *Numerical methods for option pricing in jump-diffusion markets*. PhD thesis, PhD thesis, Università degli Studi di Roma La Sapienza, 2003.
[5] C. Brugger et al. Hyper: A runtime reconfigurable architecture for monte carlo option pricing in the heston model. In *FPL*, pages 1–8. IEEE, 2014.
[6] C. Brugger et al. Mixed precision multilevel monte carlo on hybrid computing systems. In *Computational Intelligence for Financial Engineering & Economics (CIFEr), 2104 IEEE Conference on*, pages 215–222. IEEE, 2014.
[7] G. C. T. Chow et al. A mixed precision monte carlo methodology for reconfigurable accelerator systems. In *FPGA*, pages 57–66. ACM, 2012.
[8] C. de Schryver et al. An energy efficient fpga accelerator for monte carlo option pricing with the heston model. In *ReConFig*, pages 468–474. IEEE, 2011.
[9] C. de Schryver et al. A multi-level monte carlo fpga accelerator for option pricing in the heston model. In *DATE*, pages 248–253. EDA Consortium, 2013.
[10] P. Glasserman. *Monte Carlo methods in financial engineering*, volume 53. Springer Science & Business Media, 2003.
[11] J. C. Hull. *Options, futures, and other derivatives*. Pearson Education India, 2006.
[12] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.
[13] R. C. Merton. Option pricing when underlying stock returns are discontinuous. *Journal of financial economics*, 3(1):125–144, 1976.
[14] B. Nelson. Fpga design productivity–a discussion of the state of the art and a research agenda. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 1–1. Springer, 2009.
[15] O. Pell and V. Averbukh. Maximum performance computing with dataflow engines. *Computing in Science & Engineering*, 14(4):98–103, 2012.
[16] F. D. Rouah. Euler and milstein discretization. *Documento de trabajo, Sapient Global Markets, Estados Unidos. Recuperado de www. frouah. com*, 2011.
[17] D. B. Thomas. Acceleration of financial monte-carlo simulations using fpgas. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–6. IEEE, 2010.
[18] D. B. Thomas, J. A. Bower, and W. Luk. Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations. In *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pages 168–173. IEEE, 2007.
[19] X. Tian and K. Benkrid. Design and implementation of a high performance financial Monte-Carlo simulation engine on an FPGA supercomputer. In *FPT*, pages 81–88. IEEE, 2008.
[20] X. Tian and K. Benkrid. High-performance quasi-monte carlo financial simulation: Fpga vs. gpp vs. gpu. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(4):26, 2010.
[21] X. Tian and C. Bouganis. A run-time adaptive fpga architecture for monte carlo simulations. In *FPL*, pages 116–122. IEEE, 2011.
[22] J. K. Toft and A. Nannarelli. Energy efficient fpga based hardware accelerators for financial applications. In *NORCHIP, 2014*, pages 1–6. IEEE, 2014.
[23] A. H. Tse, D. B. Thomas, K. H. Tsoi, and W. Luk. Efficient reconfigurable design for pricing asian options. *ACM SIGARCH Computer Architecture News*, 38(4):14–20, 2011.
[24] N. Weaver et al. Post-placement c-slow retiming for the xilinx virtex fpga. In *FPGA*, pages 185–194. ACM, 2003.
[25] C. Wynnyk and M. Magdon-Ismail. Pricing the american option using reconfigurable hardware. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 2, pages 532–536. IEEE, 2009.
[26] G. Zhang et al. Reconfigurable acceleration for monte carlo based financial simulation. In *FPT*, pages 215–222. IEEE, 2005.