Multi-level Customisation Framework for Curve Based Monte Carlo Financial Simulations

Qiwei Jin[1], Diwei Dong[2], Anson H.T. Tse[1], Gary C.T. Chow[1],
David B. Thomas[3], Wayne Luk[1], and Stephen Weston[4]

[1] Department of Computing, Imperial College London
[2] Department of Mathematics, Imperial College London
[3] Department of Electrical and Electronic Engineering, Imperial College London
[4] Credit Quantitative Research, J.P. Morgan, London

Abstract. One of the main challenges when accelerating financial applications using reconfigurable hardware is the management of design complexity. This paper proposes a multi-level customisation framework for automatic generation of complex yet highly efficient curve based financial Monte Carlo simulators on reconfigurable hardware. By identifying multiple levels of functional specialisations and the optimal data format for the Monte Carlo simulation, we allow different levels of programmability in our framework to retain good performance and support multiple applications. Designs targeting a Virtex-6 SX475T FPGA generated by our framework are about 40 times faster than single-core software implementations on an i7-870 quad-core CPU at 2.93 GHz; they are over 10 times faster and 20 times more energy efficient than 4-core implementations on the same i7-870 quad-core CPU, and are over three times more energy efficient and 36% faster than a highly optimised implementation on an NVIDIA Tesla C2070 GPU at 1.15 GHz. In addition, our framework is platform independent and can be extended to support CPU and GPU applications.

1 Introduction

Numerical methods such as Monte Carlo simulations play an important role in the finance industry, as complex mathematical models without closed form solutions are created to accommodate the growing complexity of financial products. Interest rate modelling is one of the most important fields in mathematical finance research to price fixed income products. In the past two decades this field has evolved from modelling a single instantaneous interest rate [8] to modelling the dynamics of an entire forward rate curve [7]. Modelling each curve has a complexity of $\mathcal{O}(n^2)$ in the number of time-steps, compared to the conventional single-point modelling (such as stock option payoff evaluation) which has a complexity of $\mathcal{O}(n)$. In a large financial institution where overnight sensitivity tests and risk management are vital and required by regulators, curve based Monte Carlo modelling consumes over 30% of the total computational capacity on the corporate compute grid. With computational requirements doubling every year, hardware accelerators such as FPGAs and GPUs are increasingly being used to offload computationally demanding tasks from CPUs, in order to improve performance while reducing power consumption and data center space.

This paper proposes a customisable Monte Carlo framework for the automated generation of highly efficient curve based payoff evaluation accelerator, based on the Heath-Jarrow-Morton (HJM) mathematical framework. The main contributions are:

- a flexible Monte Carlo framework with multiple levels of functional specialisations which can be used to generate FPGA solutions for different applications without using a soft processor. The framework is designed to be platform independent and easily extendible to support CPU and GPU implementations (Section 3);
- a domain specific language to enable automatic generation of application-specific components and to support architecture specialisation to a particular application (Section 4);

– a process to identify the optimal floating point data format on target reconfigurable hardware for our architecture (Section 4);
– an evaluation of the proposed framework by comparing processing speed and energy efficiency to general purpose processors and graphics processing units over three case studies (Section 5).

2 Background

FPGAs are increasingly being used for acceleration of Monte Carlo models used in financial simulations. For instance, a platform independent domain specific language has been invented to produce optimised pipelined designs with thread level parallelism for Monte Carlo simulations from a high level abstraction [15]; an FPGA-based stream accelerator with higher performance than GPUs and Cell processors has been proposed for evaluating European options [13]; an architecture with a pipelined datapath and an on-chip instruction processor has been reported for speeding up the Brace, Gatarek and Musiela (BGM) interest rate model for derivatives evaluation [20]; an American option valuator using least-squares Monte Carlo method has been implemented [17]; a control variate Monte Carlo design for Asian options is presented [18], and a successful FPGA project in industry has been reported for collateralised default obligation (CDO) pricing [19]. However, most of the existing work seeks optimisations and generalisations for single-point simulations, while the more complex implementations usually involve a less efficient FPGA-based softcore to handle general control functions [20]. Moreover, a highly optimised complex hardware design is usually less flexible, hence problematic when changes occur frequently. The appropriate balance of performance and programmability of designs remains a challenging problem. We use the Heath-Jarrow-Morton mathematical framework to illustrate our approach to design space exploration for Monte Carlo designs with complex control.

Algorithm 1 HJM Monte Carlo Algorithm: a Single Path

Input: $f(0,T)$ = initial forward curve, σ = volatility model
Output: $f(t,T)$ = forward surface

1: **for** t=0 to t_{max} **do**
2: **for** T'=0 to T'_{max} **do**
3: **Calculate Drift**: obtain $\sigma(t,T)$ and get $\mu(t-\delta t, t+T')$ using Equation 2
4: **Update forward Surface**: get $f(t,t+T')$ using Equation 1
5: **Price Derivative State 1**: Use $f(t,t+T')$ to price the target derivative
6: **end for**
7: **Price Derivative State 2**: Use result from State 1 to price the target derivative
8: **end for**

The Heath-Jarrow-Morton (HJM) Framework [7] is a general framework for modelling instantaneous forward interest rate curve. It differs from short rate models in the way that it models the full dynamics of the entire forward interest rate curve, as opposed to a single point on the curve which is the short rate $r(t)$. The equation of the framework is shown in Equation 1:

$$df(t,T) = \mu(t,T)dt + \sigma(t,T)^T dW(t) \tag{1}$$

$$\mu(t,T) = \sigma(t,T)^T \int_t^T \sigma(t,u)du \tag{2}$$

where $f(t,T)$ is the instantaneous forward rate at time T as seen from time t and $0 \leq t \leq T$; $\sigma(t,T)$ is the forward volatility column vector of size d, where d is the number of factors in the framework;

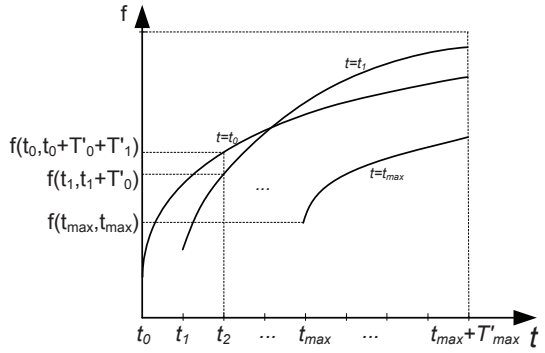

Fig. 1. The evolution of the forward rate curve from $t = 0$ to $t = t_{max}$ in one Monte Carlo path, giving $t_i = T_i = t_i + T'_{i-1}$

$W(t)$ is a random variable under standard normal distribution. For convenience we call t the time and T' the time offset from time t, with $T = t + T'$, therefore $f(t, T) \equiv f(t, t + T')$.

It can be seen that along each Monte Carlo path, a surface constructed by $f(t, T)$ is generated. Figure 1 shows the evolution of the forward rate curve over time for an arbitrary path of a Monte Carlo simulation. A general Monte Carlo algorithm for the HJM model is shown in Algorithm 1.

From line 5 of Algorithm 1, the forward curve $f(t, T)$ is used as a basic building block to evaluate interest rate products. The HJM framework is flexible in two ways: (a) the user can choose which volatility model to use (line 3 in Algorithm 1) and (b) the payoff evaluation function differs for different financial products (lines 5 and 7 in Algorithm 1). Based on different assumptions and applications, different volatility functions $\sigma(t, T)$ can be chosen. Table 3 shows some parameter settings for Equation 4 under different volatility models. Forward curves generated by the HJM framework are used to value different financial products. Table 4 shows a non-exhaustive list of valuation functions for different interest rate products.

Table 1. Parameters in our framework

Model Parameters	
d	number of factors in the framework
t	the variable that tracks time in the model
T	another time in the future, given current time is t
T'	time offset from t to T, $T = t + T'$
$f(t, T)$	the instantaneous forward rate at time T, as seen from time t
$r(t)$	short rate at time t
$\sigma(t, T)$	forward volatility, a d-dimensional column vector
$W(t)$	d-dimensional standard random process

Statistical Test Parameters	
$\overline{X_1}$	mean of the reduced precision result
$\overline{X_2}$	mean of the "true" result
σ_1	standard deviation of the reduced precision result
σ_2	standard deviation of the "true" result
n_1, n_2	number of sampling in the simulation to get the reduced precision result and the "true" result
t	the t-statistic to test whether the population means are different
$d.f.$	degrees of freedom in significance testing
wE, wF	number of exponent bits and mantissa bits in a floating point number

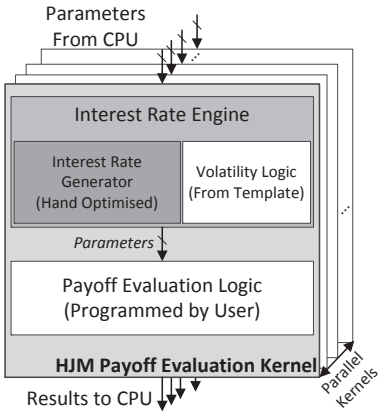

Fig. 2. Proposed framework for the HJM model, dark gray indicates stable part in the kernel, while white indicates flexible parts.

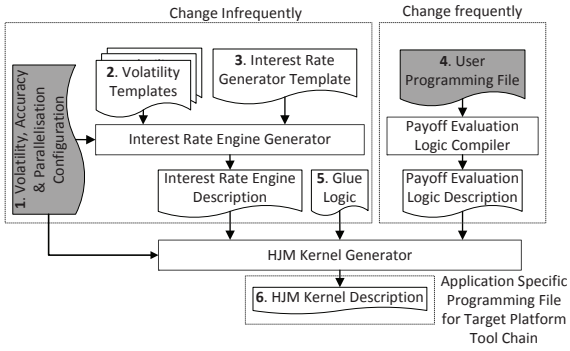

Fig. 3. Proposed workflow for our Monte Carlo framework. Refer to Table 2 for details about the input and output files (numbered).

3 Multi-level Customisation Framework

Figure 2 shows our proposed framework for the HJM model, which is independent of the choice of volatility structure and interest rate product. We define our framework based on a procedure for developing evaluators of financial product payoff, in which three levels of functional specialisations can be identified:

- **heavily specialised modules** do not change with applications and are platform dependent;
- **mediumly specialised modules** change occasionally with applications and can be platform dependent;
- **lightly specialised modules** are application dependent but platform independent.

The procedure has two phases: in the **model developing phase** platform experts develop heavily specialised modules and define templates for mediumly specialised modules and optimise them for potential target platforms; in the **payoff evaluator developing phase** users choose a mediumly specialised module as a base component and a target platform, on which platform dependent financial product payoff evaluators are generated using a platform independent domain specific language and a special compiler. Since the user only programmes the platform independent lightly-specialised module in the framework, we have a clear separation of tasks in the financial product payoff evaluator development procedure. In addition, since we expect the model developing phase to be an one-off effort and the payoff evaluator developing phase to be a continuous effort thereafter, we have created an acceleration procedure in which platform dependent expertise is not required from application developers (users).

The main part of the framework is the HJM payoff evaluation kernel, which consists of three components: volatility logic (mediumly specialised), interest rate generator (heavily specialised) and payoff evaluation logic (lightly specialised). Volatility logic, corresponding to line 3 of Algorithm 1, is flexible in the model developing phase and it is stable in the payoff evaluator developing phase. Pre-defined templates are used to allow limited flexibility in volatility logic; this part is developed by collaboration between platform experts and users. Platform experts need to understand the user's requirements which are usually platform dependent. The interest rate generator, corresponding to line 4 of Algorithm 1, is stable by nature and can be developed by platform experts who also define the interface between the module and the payoff evaluation logic. On the other hand, the payoff

Table 2. Explanation of input and output files in our framework as illustrated in Figure 3.

Idx[1]	Created By	Phase[2]	Specialisation[3]	P.D. [4]	Purpose
1	User	M	low	No	For non-expert users to target his/her design to a particular platform
2	User & Expert	M	Medium	Maybe	Optimised volatility model design
3	Expert	M	High	Yes	Optimised interest rate generator design
4	User	P	low	No	Payoff evaluation logic design without the need of knowing the underlying platform
5	Expert	M	high	Yes	Platform dependent glue logic, e.g. moving data around, etc.
6	Framework	P	high	Yes	Platform dependent programming file, e.g. VHDL, etc.

[1] These indices identity the six types of files in Figure 3
[2] The phase that creates the file. M stands for the model developing phase and P stands for the payoff evaluator developing phase
[3] Level of functional specialisation
[4] Whether the file is platform dependent

Table 3. Volatility structures used in the HJM framework

Volatility Structure	$\sigma(t,T)$	$\mu(t,T)$
Constant[1]	α	$\frac{1}{2}\alpha^2[T^2-(T-t)^2]$
Exponential[1]	$\alpha e^{-\beta(T-t)}$	$\frac{\alpha^2}{\beta}(e^{-2\beta(T-t)}-e^{-\beta(T-t)})$
Stochastic[2]	$\widetilde{\sigma}(t,T)f(t,T)$	$\sigma(t,T)\int_t^T \sigma(t,u)du$

[1] α and β are calibrated model constants
[2] $\widetilde{\sigma}$ is a stochastic volatility process

evaluation logic is prone to change and we expect many instances of payoff evaluation logic to be created in the payoff evaluator developing phase over a long time. A platform independent domain specific language is used to allow non-experts to use the accelerated framework easily.

Figure 3 shows our proposed workflow. The user begins the payoff evaluator developing phase by defining the choice of underlying platform, volatility model, accuracy requirement and parallelisation requirement in a configuration file. The choice of underlying platform defines whether the user wants the application to run on FPGA, CPU, GPU etc.; volatility model defines the interest rate engine; accuracy requirement defines word length of the datapath, and parallelisation determines the number of parallel datapaths in the system. Based on the configuration file, the interest rate engine generator combines an appropriate interest rate generator template and a volatility module template to produce an interest rate engine description. The engine description is a programming file that describes the target platform. The user writes domain specific programmes to utilise the interest rates generated by the engine and then builds appropriate designs to evaluate interest rate derivatives. The platform independent user programming file is compiled to a payoff evaluation logic description programming file for the target platform, by the payoff evaluation logic compiler. The HJM kernel generator combines the interest rate engine description and payoff evaluation logic description to produce a complete programming file, which is then used as input to the target platform tool chain to generate executables.

Table 4. Example interest rate products

Target Instrument	Payoff Evaluation Function
Bond	$B(t,T) = \exp\left(-\int_t^T f(t,u)du\right)$
Bond Option[1]	$(B(t,T) - K)^+$
CMS[2]	$Y(t,T) = \frac{1-B(t,T)}{\sum_{a=t}^{T} B(t,a)}$
Swaption	$(Y(t,T) - K)^+ \sum_{a=t}^{T} B(t,a)$
CMS S.O.[3]	$(Y(t,T_1) - Y(t,T_2) - K)^+$

[1] $(x)^+ \equiv max(0,x)$
[2] Constant Maturity Swap
[3] CMS Spread Option

4 Application Specialisation Flow

In this section we discuss the specialisation process in our framework. We propose a "C" style control-based domain specific programming environment to demonstrate the programmability of our framework. The programming environment has the following assumptions:

- The programming environment provides a set of environment parameters P generated by its underlying framework at each iteration (one clock cycle for FPGAs, one for-loop iteration for CPUs, etc.), which can be utilised by the developer (user). This means that data are provided in a temporal manner as opposed to the conventional spatial manner. Instead of requesting a piece of desired data, the developer waits until the data are provided by the programming environment.
- Operator latency is implicit; results appear to be produced instantaneously. This allows the developer to use the framework without expertise in the target platform.
- The user can create model input variables, intermediate variables, accumulation logic, control logic and nothing else. The set of input variables V, once declared, is treated as a data environment, in other words $P' = P \cup V$.
- The user specifies outputs from intermediate variables, and output conditions.

The language is natively supported by CPU and GPU implementations and can be used to generate control and datapaths in hardware. We give a simplified definition of the grammar in Listing 1.1 in order to provide an overview of our domain specific language, it is not intended to be rigorous or comprehensive.

Listing 1.1. Simplified grammar for the domain specific language used in our framework. Note that the grammar is subject to extension and it is intended to show the capability of the language, hence it may neither be rigorous nor be comprehensive.

```
<configuration> ::= <statement>+
<statement> ::= <calc_statement>|<io_statement>
<calc_statement> ::= <ident> = <expression>
       | if ( <expression> ) {  <calc_statement>+ }
              [else { <calc_statement>+ } ]?
<io_statement> ::= input (  <ident> )
       | output ( <expression> )
<expression> ::= <literal>
       |<unary_op> <expression>
       |<expression> <binary_op> <expression>
       |<func> ( <expression> [, <expression>]* )
<literal> ::= <env_literal> | <user_literal>
```

We now demonstrate the application specialisation process for reconfigurable hardware. To begin with, we map our language to the hardware architecture. We assume $P = \{f_{i,j}, discount_i, dt, i, j\}$,

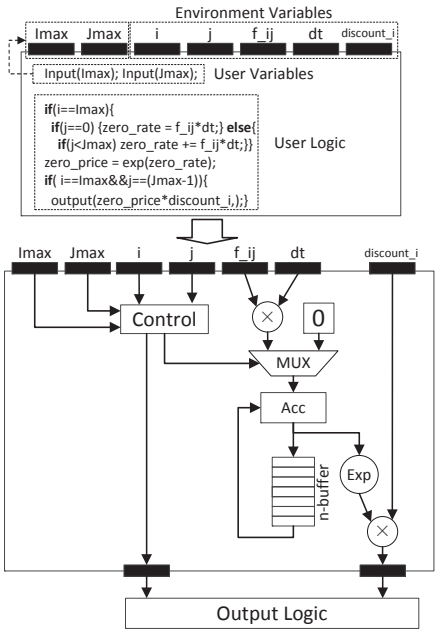

Fig. 4. Domain specific language Example 1.

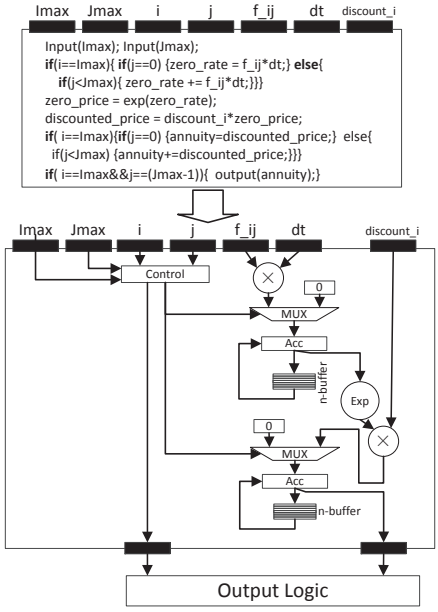

Fig. 5. Domain specific language Example 2, extending Example 1.

$V = \{Imax, Jmax\}$ and $P' = P \cup V$, as shown in Figure 4, where i is the index for time step t, the incrementation of i depends on index j which is the index for time offset T'; $f_{i,j}$ is the discretised instantaneous forward rate at time T_j , as seen from time t_i; dt is the time difference between T_j and T_{j-1} or t_i and t_{i-1}; and $discount_i$ is the discount factor to present time. The top box in Figure 4 shows the programming environment in which one P' is provided in each clock cycle. The user relies on the programming environment to provide a correct set of parameters, and assumes that variable j counts inside each variable i. The bottom box in Figure 4 shows a fully pipelined design generated from the description in the top box. The n-buffer under the accumulator is to hide pipeline latency of the accumulator by keeping a history of n previous value. This effectively increases total pipeline latency of the datapath by n, however it allows full pipelining in the datapath without using low performance un-pipelined accumulator. Latency balancing buffers are omitted in the figures for simplicity. Figure 5 shows a possible extension based on Example 1, in which annuity is calculated based on prices of zero bonds maturing over a time period.

Reconfigurable hardware supports customised word length of its datapath in order to optimise hardware utilisation based on an accuracy requirement. Previous research focuses on fine-grained bitwidth optimisation, such as simulation [10], interval arithmetic [4], backward propagation analysis [6], affine arithmetic [12] and polynomial algebra [2]. However most of them are not straight forward for complex Monte Carlo problems where multiple levels of combinational logic consisting of floating point datapath and accumulators are combined with complex control-flow and feedback paths. We propose a purely statistical method to determine the optimal data format for reconfigurable hardware.

The accuracy of a financial instrument payoff calculated by a numerical method is usually affected by two main factors:

– Discretisation error: the error caused by transforming the model from a continuous mathematical space to a discretised computational space. In our case when the Monte Carlo method is used, the discretisation error comes from insufficient sampling of the underlying random source.

– Finite precision error: the error caused by using number representations of insufficient accuracy. The error can be amplified or diminished by numerical operators in the datapath. In Monte Carlo simulations we expect the finite precision error to be a normally distributed random factor, according to the law of large numbers and the central limit theorem.

We therefore define the measure of accuracy to be the observed error due to the combination of discretisation error and finite precision error. Since Monte Carlo is a statistical method relying on the law of large numbers, we use Welch's t-test to assess the statistical significance of the error in the result [11]. The test determines whether there is any statistical evidence suggesting the Monte Carlo result is different from the "true result", which is the result calculated by a high precision datapath, e.g. double precision. We therefore set the null hypothesis to be that the Monte Carlo result and the true result are equal, assuming we know the true result and its standard deviation beforehand. We use Equation 3 to calculate the t-value and Equation 4 to calculate the degree of freedom ($d.f.$). These two values can then be used to obtain the p-value via the Students-t CDF. The definition of the variables are shown in Table 2. The experiment will run the Monte Carlo simulation using a customised floating point data format with wE bits of exponent and wF bits of mantissa. For the sake of simplicity from now on we define $wE = 8$ for all floating point number formats. We use the custom data format to build the datapath for a given Monte Carlo payoff evaluation simulation and run different experiments with n_1 starting from a smaller number and incrementing towards infinity. During the experiments we monitor the p-value, and once the p-value falls below a pre-selected statistically significant threshold (e.g. p=0.05 for 5% significance) the simulator is considered to have failed. If the test does not fail on the custom data format, we can conclude that the result from the custom datapath is not statistically different from the data format used to obtain the "true result".

$$t = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\sigma_1^2/n_1 + \sigma_2^2/n_2}} \tag{3}$$

$$d.f. = \frac{\left(\sigma_1^2/n_1 + \sigma_2^2/n_2\right)^2}{\left(\sigma_1^2/n_1\right)^2/(n_1 - 1) + \left(\sigma_2^2/n_2\right)^2/(n_2 - 1)} \tag{4}$$

5 Result

In this section we discuss the applications of our framework over three case studies: bond option, swaption, and CMS spread option (CMS S.O.). The details of these options are listed in Table 4.

We use the MaxWorkstation reconfigurable accelerating system from Maxeler Technologies to evaluate our framework. It has one MAX3 card with a Xilinx Virtex-6 SX475T FPGA. The card is connected to an Intel i7-870 CPU through a PCI express link with a measured bandwidth of 2 GB/s. The general purpose processor (GPP) in our comparison is a 4-core Intel i7-870 CPU running at 2.93 GHz.

We use the Intel Compiler (ICC) and the Intel Math Kernel Library for our software implementations. The SFMT random number generator and the Box-Muller transformation provided by Intel Vector Statistical Library (VSL) are used for random number generation. We have optimised the software implementations to the best of our knowledge, to ensure the comparisons are fair and accurate.

The FPGA implementations are generated based on: user programming files compiled automatically by our payoff evaluation logic compiler, and configuration files, volatility templates, glue logic and interest rate generator template written and optimised by hand. The files are assembled into a final design following the proposed workflow manually, while a fully automated system is under development. We use the MaxCompiler as our high level synthesis tool and our payoff evaluation logic compiler generates intermediate descriptions compatible with the MaxCompiler based on our domain specific language. We use one CPU core to drive the FPGA in our case studies. The payoff

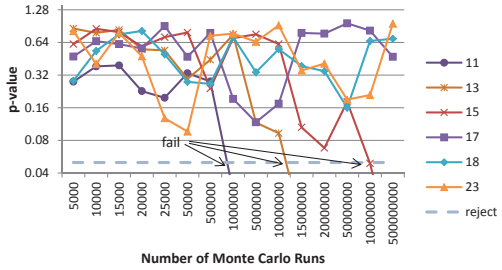

Fig. 6. p-value in log scale for Bond Option.

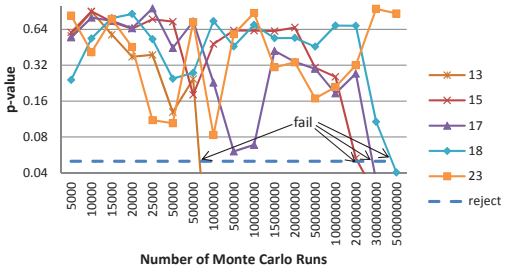

Fig. 7. p-value in log scale for Swaption.

evaluation logic compiler is generated by ANTLR parser generator [1]. The hand-optimised interest rate generator consists of a LUT Optimised uniform random number generator [14], a wrapper to transform uniform random numbers to standard normal random numbers [16] and a floating point exponential operator [3]; other components are generated by Xilinx CoreGen.

We use Welch's t-test described in Section 4 to determine the optimal floating point number format to use on FPGA. The test is designed so that we have a set of applications A, let A_i ($i \in \mathbb{N}$) indicate the ith application in set A, and $A_{i,wF}$ indicate a variation of application A_i using a specific floating point data format with wF mantissa bits. We define a test result to be a tuple $(A_{i,wF}, n)$, which is obtained from a Monte Carlo simulation of n paths using application $A_{i,wF}$. The test result is then compared to a reference result obtained from $(A_{i,53}, 10^9)$, which is a double precision variation of A_i running one billion Monte Carlo paths. We assume that all reduced precision data formats under consideration have 8 exponent bits ($wE = 8$). We use Equation 3 to calculate the t-value and Equation 4 to calculate the degrees of freedom, from which the p-value can be obtained from standard t-distribution tables. Setting the null hypothesis to state that the result obtained from a reduced precision datapath is the same as the result obtained from a double precision datapath, if p-value is smaller than or equal to the significance level ($p = 0.05$), the reduced precision result is then rejected, since there is enough information to tell that the reduced precision result and the double precision result are from two different distributions. Otherwise the observation is consistent with the null hypothesis and we consider the reduced precision result to be not statistically different from the double precision result.

In our experiment we have $A = \{$ Bond, Bond Option, CMS, Swaption, CMS Spread Option$\}$, $8 \leq wF \leq 23$ and $5000 \leq n \leq 5 \times 10^8$. We use the MPFR library [5] to build reduced precision datapaths in order to carry out the experiments in a scalable way; however, because the calculations mirror those performed in hardware, the results also apply to the FPGA data-path. Due to space limitation we only discuss two representative cases: Bond Option and Swaption. Figure 6 shows different p-values of the bond option application with different number of bits for mantissa over various number of Monte Carlo runs. The data formats with $wF \leq 10$ are ignored as they fail the t-test when $n = 5000$. The 11-bit mantissa version fails when $n = 10^6$, which means there is no statistically significant evidence to tell the reduced precision result from the double precision result when $n < 10^6$. Therefore if the user only intends to run Monte Carlo simulations of $n < 10^6$ paths, he/she can get an answer not statistically significantly different from the double precision result by a reduced precision datapath. On the other hand, with a 17-bit mantissa we do not fail until 5×10^8 samples, which means the 17-bit version can produce a good enough result for $n < 5 \times 10^8$. Other variations all fail t-tests before $n = 5 \times 10^8$.

Figure 6 shows p-values for Swaption application, which is an extreme case where the 17-bit mantissa version fails t-test when $n = 3 \times 10^8$; it means that we need to increase the mantissa bits if n exceeds 3×10^8. However, this is the only application where the 18-bit mantissa version fails t-test. On the other hand, we don't see any 23-bit mantissa version fails any t-test during our experiments.

In finance industry Monte Carlo simulations usually use $n = 5 \times 10^4$, which is well below the 3×10^8 threshold, so floating point numbers with 17 to 23 bit mantissas should be good enough to produce reliable results.

Table 5 shows a resource consumption comparison between double precision and reduced precision implementations. It can be seen that when double precision is used in the datapath, all implementations are bounded by Block RAM resource, since the designs require significant amounts of FIFO buffer to pipeline accumulators and to retime pipelines with long delay. We can only fit 2-3 double precision cores on the FPGA and utilise around 20% of the logical hardware resource. On the other hand, if we use reduced precision data format ($wE = 8, wF = 17$), we find Block RAM resource usage reduced by 15 to 20 times, and the design is now bounded by logic resources instead. This means that we can utilise more area on the FPGA to do computation and expect a higher throughput. The top clock frequencies also increase by about 1.4 times accordingly.

Table 5. Resource Comparison: $wE = 8, wF = 53$ (double precision) and $wE = 8, wF = 17$ (reduced precision) on a Virtex-6 SX475T FPGA

	Bond Option		Swaption		CMS S.O.		Device
Num. Mantissa Bits	53	17	53	17	53	17	-
LUT (%)	6.2	**3.26**	7.95	**3.77**	11.64	**5.1**	297600
FF (%)	4.04	2.18	5.43	2.5	7.94	3.33	595200
BRAM (%)	**28.76**	1.88	**29.04**	1.88	**41.82**	2.02	1064
DSP (%)	6.55	1.39	7.04	1.49	8.09	1.74	2016
Clock Freq. (MHz)	195	270	185	265	170	230	-
Normalised Area	15x	1x	15x	1x	20x	1x	-
Normalised Freq.	1x	1.4x	1x	1.4x	1x	1.35x	-

Table 6 contrasts GPP and GPU implementations with the FPGA implementations generated by our framework. Each FPGA implementation uses about 80% of the total logic resource available on the FPGA to avoid congestion in the place and route phase. The FPGA implementations are based on a reduced precision data format ($wE = 8, wF = 17$). The testing cases are Monte Carlo simulations of 100 million paths for GPP and FPGA, and an 89.6 million paths Monte Carlo simulation for GPU. The GPU testing case is designed to fit the GPU parallelism granularity to ensure fair comparison. It can be seen that the FPGA implementations are about 40 times faster than software implementations utilising one of the four CPU cores. They are about 10 times faster than the corresponding software implementations utilising all four CPU cores. It is not surprising to see that the FPGA CMS spread option implementation is the slowest, since it requires complex logic to sample two different sections on the forward curve, which implies larger kernels, slower clock frequency and fewer parallel kernels in the FPGA. The software implementation suffers less from the increase of complexity since the two samplings are independent of each other and the instructions can be efficiently pipelined.

We use an Ethernet-connected power measuring socket from Oslon electronics to measure average power consumption of the system, with a measuring resolution of 1 sample per second. As shown in Table 6, it is not surprising to see that FPGA implementations are generally about 20 times more energy efficient than software implementations, given that all power readings include idle power consumption of the system.

We now discuss our Graphics Processing Unit (GPU) benchmark to compare the FPGA implementations generated by our framework. The GPU is an NVIDIA Tesla C2070 device with 448 cores running at 1.15 GHz and has a peak double precision performance of 515 GFlops. The benchmark implementation on bond option is based on the standard parallel random number generator provided by CURAND Library and nvcc compiler with maximum optimisation flags turned on. The implementation is hand optimised so that access to off chip memory only occurs at the beginning

and at the end of the kernel launch. We use warp level control to ensure the kernel only accesses on chip cache during the execution without any bank conflict or branch divergence. As shown in Table 6, the GPU implementation is about two times less energy efficient and the corresponding FPGA implementation is about 36% faster. Given the fact that both devices are using the 40nm technology, it can be seen that the FPGA implementations are gaining speed advantage and energy effciency from customisable data format, fully pipelined datapath and lower clock frequency.

It is difficult to make precise qualitative comparison between our approach and the traditional hand written approach, in terms of development time and quality of code. However, when compared with simple hand-written designs using a high-level programming language [9], which requires the user to write hundreds of lines of code, our automated approach requires less than ten lines of code (Figure 4).

Table 6. Comparison of MC simulations using double precision GPP(SW), reduced precision FPGA and single precision GPU

	Bond Option			Swaption		CMS S.O.[6]	
Device	SW[1]	FPGA	GPU[5]	SW	FPGA	SW	FPGA
Clock Freq. (MHz)	2930	160	1150	2930	150	2930	150
Num. of Cores	4	26	448	4	19	4	16
Num.Evaluations (B)[4]	177	177	154	177	177	177	177
Exe. Time (Seconds)	476	50.3	50.5	738	69.4	822	73.75
Power Consumption (W)[3]	183	87	240	184	87	184	85
Energy Efficiency[2]	2	40.5	12.7	1.3	29.3	1.3	28.2
Speed-up vs Single-Core[7]	4x	**44.8x**	32.8x	4x	**42.4x**	4x	**39.2x**
Speed-up vs Quad-Core[8]	1x	**11.2x**	8.2x	1x	**10.6x**	1x	**9.8x**
Normalised Energy	19.9x	1x	3.2x	22.5x	1x	24.1x	1x

[1] The software utilises all 4 physical cores by process level parallelism
[2] Measured in number of evaluations/second/Joule
[3] The idle power consumption of the system is 80W
[4] Number of point evaluations in the simulation, measured in billions of $f(t, T)$ calculation
[5] The benchmark GPU is an NVIDIA Tesla C2070 device
[6] CMS Spread Option
[7] Speedup against one core of a quad-core CPU
[8] Speedup against all four cores of a quad-core CPU

6 Conclusion

This paper proposes an application independent Monte Carlo framework for interest rate derivatives payoff evaluations based on the HJM model. By identifying three levels of functional specialisations in the model, we allow a hand optimised component, a templated component and a programmable component in our framework, to retain good performance and to support multiple applications. The framework is designed to be platform independent and easily extendible to support CPU and GPU implementations. To specialise our framework to a particular application, we propose a domain specific language for the programmable component. We also propose a process for the FPGA platform to identify the optimal floating point data representation to ensure maximum utilisation of hardware resource. We have shown that, by adopting optimal number representation in the datapath, we can reduce the memory resource usage by 15 to 20 times, allowing better utilisation of logic resource.

The designs generated by our framework for a Xilinx Virtex-6 SX475T FPGA are generally about 40 times faster than a single-core implementation on a i7-870 quad-core CPU at 2.93 GHz, are over 10 times faster and 20 times more energy efficient than 4-core implementations on the same i7-870 quad-core CPU, and are three times more energy efficient and 36% faster than an NVIDIA Tesla C2070 GPU at 1.15 GHz.

Current and future work includes the following. First, explore effective automation of the proposed workflow while allowing user guidance where essential. Second, extend our framework to cover other numerical methods and other applications. Third, explore how advanced techniques, such as run-time reconfiguration, can further improve performance and energy efficiency.

Acknowledgment

The research leading to these results has received funding from UK EPSRC, Maxeler Technologies, J.P. Morgan and European Union Seventh Framework Programme under grant agreement number 248976 and 257906.

References

1. ANTLR: http://www.antlr.org/
2. Boland, D., Constantinides, G.: Automated precision analysis: A polynomial algebraic approach. In: IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM) (2010)
3. de Dinechin, F., Pasca, B.: Floating-point exponential functions for DSP-enabled FPGAs. In: Proc. Int. Conf. on Field-Programmable Technology (2010)
4. Fang, C.F., Rutenbar, R.A., Chen, T.: Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs. In: Proc. Int. Conf. on Computer-aided design (2003)
5. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: Mpfr: A multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. 33 (2007)
6. Gaffar, A.A., Mencer, O., Luk, W., Cheung, P.Y.K.: Unifying bit-width optimisation for fixed-point and floating-point designs. In: IEEE Symp. on Field-Programmable Custom Computing Machines (2004)
7. Heath, D., Jarrow, R., Morton, A.: Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. Econometrica 60(1), 77–105 (January 1992)
8. Ho, T.S.Y., Lee, S.b.: Term structure movements and pricing interest rate contingent claims. Journal of Finance 41(5), 1011–29 (December 1986)
9. Jin, Q., Thomas, D.B., Luk, W., Cope, B.: Exploring reconfigurable architectures for tree-based option pricing models. ACM Trans. Reconfigurable Technol. Syst. 2, 21:1–21:17 (September 2009)
10. Kum, K.I., Sung, W.: Combined word-length optimization and high-level synthesis of digital signal processing systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20, 921 –930 (2001)
11. Larsen, R.J., Marx, M.L.: An Introduction to Mathematical Statistics and Its Applications. Pearson Education (2011)
12. Lee, D.U., Gaffar, A., Cheung, R., Mencer, O., Luk, W., Constantinides, G.: Accuracy-guaranteed bit-width optimization. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 25, 1990 –2000 (2006)
13. Morris, G., Aubury, M.: Design space exploration of the European option benchmark using Hyperstreams. In: Proc. Int. Conf. on Field Programmable Logic and Applications. pp. 5–10 (2007)
14. Thomas, D.B., Luk, W.: High quality uniform random number generation using LUT optimised state-transition matrices. J. VLSI Signal Process. Syst. 47, 77–92 (2007)
15. Thomas, D., Bower, J., Luk, W.: Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations. In: Proc. Int. Conf. on Application-Specific Systems, Architectures and Processors. pp. 685–689 (2007)
16. Thomas, D., Luk, W.: Non-uniform random number generation through piecewise linear approximations. In: Proc. Int. Conf. on Field Programmable Logic and Applications (2006)
17. Tian, X., Benkrid., K.: American option pricing on reconfigurable hardware using least-squares monte carlo method. In: Proc. Int. Conf. on Field-Programmable Technology. pp. 263 –270 (2009)
18. Tse, A.H., Thomas, D.B., Tsoi, K., Luk, W.: Reconfigurable control variate Monte-Carlo designs for pricing exotic options. In: Proc. Int. Conf. on Field Programmable Logic and Applications. pp. 364–367 (2010)
19. Weston, S., Spooner, J., Racanire, S., Mencer, O.: Rapid computation of value and risk for derivatives portfolios. Concurrency and Computation: Practice and Experience (to appear)
20. Zhang, G., Leong, P., Ho, C., Tsoi, K., Cheung, C., Lee, D.U., Cheung, R., Luk, W.: Reconfigurable acceleration for Monte-Carlo based financial simulation. In: Proc. IEEE Int. Conf. on Field-Programmable Technology. pp. 215–224 (2005)