Monte-Carlo Simulation-Based Financial Computing on the Maxwell FPGA Parallel Machine

Xiang Tian and Khaled Benkrid

Abstract Efficient computational solutions for scientific and engineering problems are a priority for many governments around the world, as they can offer major economic comparative advantages. Financial computing problems are a prime example of such problems where even the slightest improvements in execution times and latency can generate large amounts of extra profits. However, financial computing has not benefited relatively greatly from early developments in high performance computing, as the latter aimed mainly at engineering and weapon design applications. Besides, financial experts were initially focusing on developing mathematical models and computer simulations in order to comprehend the behavior of financial markets and develop risk-management tools. As this effort progressed, the complexity of financial computing applications grew up rapidly. Hence, high performance computing turned out to be very important in the field of finance.

Many financial models do not have a practical closed-form solution in which case numerical methods are the only alternative. Monte-Carlo simulation is one of the most commonly used numerical methods, in financial modeling and scientific computing in general, with huge computation benefits in solving problems where closed-form solutions are impossible to derive. As the Monte-Carlo method relies on the average result of thousands of independent stochastic paths, massive parallelism can be harnessed to accelerate the computation. For this, high performance computers, increasingly with off-the-shelf accelerator hardware, are being proposed as an economic high performance implementation platform for Monte-Carlo-based simulations. Field programmable gate arrays (FPGAs) in particular have been recently proposed as a high performance and relatively low power acceleration platform for such applications.

X. Tian (✉) • K. Benkrid
The University of Edinburgh, Institute of Integrated Systems, King's Buildings, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK
e-mail: X.Tian@ed.ac.uk; k.benkrid@gmail.com

W. Vanderbauwhede and K. Benkrid (eds.), *High-Performance Computing Using FPGAs*,
DOI 10.1007/978-1-4614-1791-0_2, © Springer Science+Business Media, LLC 2013

In light of the above, the project presented in this chapter develops novel FPGA hardware architectures for Monte-Carlo simulations of different types of financial option pricing models, namely European, Asian, and American options, the stochastic volatility model (GARCH model), and Quasi-Monte Carlo simulation. These architectures have been implemented on an FPGA-based supercomputer, called Maxwell, developed at the University of Edinburgh, which is one of the few openly available FPGA parallel machines in the world. Maxwell is a 32-CPU cluster augmented with 64 Virtex-4 Xilinx FPGAs connected in a 2D torus. Our hardware implementations all show significant computing efficiency compared to traditional software-based implementations, which in turn shows that reconfigurable computing technology can be an efficacious and efficient platform for high performance computing applications, particularly financial computing.

1 Introduction

High performance computing (HPC) is a discipline concerned with the development and use of supercomputers or computer clusters, with applications in a variety of fields including bioinformatics, energy, climate modeling, and computational applications in engineering, of which typical computational demands exceed the TeraFlop/sec.[1] Supercomputers' development has been through several stages during the past decades starting with vector computers, and then symmetric multiprocessors (or SMPs[2]), to massively parallel processors (MPPs) which mostly use off-the-shelf commodity microprocessors nowadays [1]. Supercomputers' performance requirements, however, are increasing at a rate that exceeds the rate of chip-level improvements [2]. In the early days of the technology, mostly engineering and weapons' design applications benefited from the developments in high performance computing. In financial computing, for instance, financial experts were mostly focused on mathematical models and computer simulations in order to understand financial markets and develop risk-management tools. Generally, these models are stochastic process models. As the complexity of these models increased rapidly, personal computers were no longer able to perform the required computations in reasonable times; hence the adoption of high performance computing platforms became the mainstream. In 1999, for instance, a survey of high performance computing in finance and computer-aided design of financial products introduced the development of financial models and supercomputer-based high performance financial computing to a wider community [3].

[1] TeraFlop/sec is an acronym meaning 10^{12} floating point operations per second.

[2] An SMP is a computer system that has two or more processors connected in the same cabinet, managed by one operating system, sharing the same memory, and having equal access to input/output devices.

One widely used computational technique in financial computing is Monte-Carlo simulation. The latter is a numerical computational algorithm which is often used in simulating physical and mathematical systems. It relies on repeated random sampling to compute their result. This method is often used when it is impossible or impractical to get an analytical solution, or closed-form result, to system equations. The Monte-Carlo method is particularly important in physical chemistry, computational physics, and related applied fields. These are characterized by systems with a large number of coupled degrees of freedom, such as liquids, disordered materials, strongly coupled solids, and cellular structures. Monte-Carlo simulations are also used to forecast a wide range of events and scenarios, such as the weather, sales, and consumer demands. In financial computing, the Monte-Carlo technique is used to simulate the various sources of uncertainty that affect the value of the underlying instrument, portfolio, or investment in question. Many financial computing applications have no closed form solutions, as they depend on three or more stochastic variables. Here, Monte-Carlo simulation tends to be numerically more efficient than other procedures [4]. This is because the computational time of Monte-Carlo simulations increases approximately linearly with the number of variables, whereas in most other methods, computational time increases exponentially with the number of variables. One of the important characteristics of Monte-Carlo simulation is parallelism as multiple independent paths need to be computed. This makes it attractive to parallel implementation using multi-threading and/or multi-processing.

When evaluating a high performance computing platform, we have to consider several aspects. The cost of cluster computers and supercomputers can be prohibitive. Area and power consumption can also be a major problem with these computing platforms. For these reasons, various acceleration technologies are being considered. Field programmable gate arrays (FPGAs), for instance, offer the high performance of a dedicated hardware solution of a particular algorithm, with a fraction of the area and power consumption of equivalent microprocessor-based solutions. Moreover, the continuous developments in transistor integration levels mean that it is now possible to implement a considerable number of floating-point arithmetic units on modern FPGAs. If this trend is to continue, FPGA use is set to conquer new application domains, including financial computing.

The work presented in this chapter is mainly targeted on an FPGA parallel machine, called Maxwell. Maxwell was one of the first publicly accessible FPGA parallel machines and was built in Edinburgh, Scotland, by the FPGA High Performance Computing Alliance (FHPCA). Established in 2004, the FHPCA's aim was to explore the computing capability of a heterogeneous high performance computing platform, which combines general purpose processors (GPPs) and Xilinx FPGAs. Led by Edinburgh Parallel Computing Centre (EPCC) at The University of Edinburgh, the FHPCA was funded by Scottish Enterprise and built on the skills of Nallatech Ltd., Alpha Data Ltd., Xilinx Development Corporation, Algotronix and iSLI. The idea developed as a result of increasing FPGA hardware complexity, which makes it possible to execute relatively sophisticated general purpose numerical computing applications on modern FPGAs at a relatively low power

budget. The work presented in this project focuses on financial computing with the aim of implementing a number of financial computing algorithms on Maxwell and evaluating the latter through a comprehensive strategy, whereby computation speed is not the only concern, but also accuracy, cost, and energy consumption. Indeed, the decision making procedure in a financial market always requires real-time processing of huge amounts of real time data. If the simulation results, which may need hours to get, can be achieved in few minutes, the benefit would definitely be remarkable. However, despite the importance of computation speed, we do not want to lose any considerable accuracy during the computation as an error of 0.001 in simulation, for instance, could bring losses in the thousands of pounds in trading if we are dealing with million-pound assets. Moreover, power efficiency is a very important issue nowadays. For instance, the cost of electricity consumed by modern supercomputers could be in the millions of pounds annually. Here lies the advantage of FPGAs as they achieve high speed performance not through high clock frequencies but mainly through massive data and instruction parallelism and deep pipelining. Indeed, typical clock frequencies of GPPs or graphic processing units (GPUs) are in the GHz range. However, FPGA chips are often clocked at few hundred MHzs, hence leading to considerable energy savings.

In light of the above, the aim of the work presented in this chapter is to rigorously assess the efficiency and efficacy of FPGAs in financial computing applications. This is done through the development of novel FPGA hardware architectures for a number of financial computing applications. Comparative evaluation against GPP and GPU technologies are made using the following criteria:

- *Speed performance*: a very important aim of our work is to maximize the computation speed of financial computing applications. This will need careful hardware design and optimization.
- *Arithmetic accuracy*: the accuracy requirement, which is critical in financial computing, needs to be guaranteed.
- *Power consumption*: since financial computing applications are often deployed in massively parallel computers, power consumption is a very important measure.
- *Cost of purchase and development*: the cost of the hardware and development effort also needs to be considered. Indeed, speed performance can always be increased arbitrarily if budgets were unlimited.
- *Productivity*: this is concerned with the return over time. Indeed, in business, time to market makes the difference between success and failure.

The remainder of this chapter is organized as follows. First, a brief overview of the state-of-the-art of high-performance financial computing is given. Then, an overview of the architecture and programming environment of the Maxwell Parallel FPGA machine is presented. After that, four case-study implementations of financial models on Maxwell are detailed, before an evaluation of the resulting implementations is presented. The chapter concludes with a general evaluation of FPGA-based high performance reconfigurable computing.

2 Brief Overview of the State-Of-The-Art of High Performance Financial Computing

In the last decade, researchers have started to use acceleration technology, e.g., in the form of FPGAs and GPUs in financial computing. In [5] for instance, an FPGA-based Monte-Carlo simulation core used for computing the BGM (Brace, Gatarek and Musiela) interest rate model for pricing derivatives was presented. The BGM interest rate model is commonly used to simulate the fluctuation of interest rates over time, something which has an influence on nearly all economic activity. Results show that ∼25x speed-up can be obtained by using an FPGA, compared to an equivalent Pentium IV 1.5GHz-based software implementation. Other hardware architectures for Monte-Carlo-based financial simulations were published in [6]. In the latter, five different Monte-Carlo option pricing simulation algorithms were explored, including log-normal price movements, correlated asset value-at-risk calculation, and price movements under the GARCH model. Using a Xilinx Virtex-4 XC4VSX55 device, implementation results show that FPGA implementations run on-average 80x faster than equivalent software ones (running on a 2.66GHz PC). A comparison of different FPGA implementations of the European option[3] benchmark against other implementations using GPUs, Cell BE, and a traditional software implementation was presented in [7]. In this work, the FPGA implementation was produced using "HyperStreams," which is a high level abstraction for designing arithmetic pipelines built on the Handel-C programming language. An acceleration of 146x compared to a reference software implantation can be obtained using FPGAs. The implementation mapped onto a Xilinx XC5VSX50T chip is over 64 and 71 times faster than corresponding software running on a 3.4 GHz Intel Xeon processor, for one-factor and the multi-factor models, respectively.

The combination of cluster technology and reconfigurable hardware acceleration is a relatively new development in high performance computing, which promises to combine the relatively high performance and low power consumption of reconfigurable hardware with established design flows and consequent knowledge base in traditional microprocessor-based high performance computing. A simple Asian option[4] pricing core was designed as a demonstration application on Maxwell. The implementation results of this demonstrator application are shown in Fig. 1. Here, AlphaData (AD) and Nallatech (NT) refer to the two FPGA companies that donated the FPGA accelerator nodes on the Maxwell machine, 32 each.

The results show that the AlphaData nodes lead to ∼320-time speed-up compared to an equivalent software implementation, whereas the Nallatech nodes lead to a 109-time speed-up. The discrepancy is due to the design language/flow used for each node type: VHDL for AlphaData and a proprietary C-based hardware language, called DIME-C, for Nallatech.

[3] A European option gives its holder the right to buy (a call option) or sell (a put option) an underlying asset at a particular fixed price (called Strike price) on a certain maturity date.

[4] Asian options are a special type of options where the strike price is the average price of the underlying asset over a period of time and not a fixed strike price as in European options.

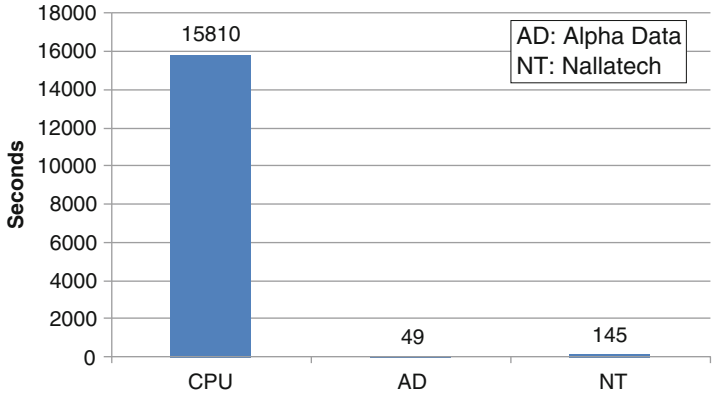


Fig. 1 Single node execution time of Asian option pricing simulation in a Maxwell-based demonstrator

Another important type of financial options is American options[5]. However there are relatively few corresponding FPGA-based implementations reported in the literature. One of these, the basic binomial-tree pricing model of American options, was implemented in [8]. The implementation of this model on a Virtex 4 FPGA achieved 250x speed-up compared to a 2.2GHz Core2 Duo CPU implementation. Another accelerated implementation used the LSMC algorithm [9] and implemented it on a 32 processor IBM BlueGene/P system to achieve 18x speed-up over a single processor implementation.

The Quasi-Monte Carlo method for financial option pricing has been researched for more than a decade. Nevertheless, high performance FPGA implementations of it have been rare in the literature. One such hardware implementation was reported in [10]. In it, a Quasi-Monte Carlo technique was applied to solve a 3-D IC partial inductance extraction problem. The number of dimensions of this design was reported to be 6. A Quasi-Monte Carlo simulator FPGA implementation was also reported in [11] where speed-ups in excess of 50x over a 3GHz multi-core processor were achieved.

As GPUs became more widely used for high performance computing, comparisons between GPUs, FPGAs, and other computing platforms became plentiful. For instance, a speed-up figure of 20x on FPGA compared to a CPU implementation for a European option pricing application was reported in [12], whereas an equivalent GPU implementation achieved a 2 order of magnitude speed-up. In another paper [7], the European option pricing application achieved 146x speed-ups on the FPGA compared to CPU.

As Monte-Carlo simulation relies on a stochastic procedure, random number generation is a key part of it. Software implementations of random number generators cannot meet the requirement of hardware Monte-Carlo simulation cores,

[5]Unlike European options, American options can be exercised at any date up to the maturity date.

and thus hardware random number generation is needed. There are many methods used for hardware random number generation including the Box–Muller method [13], Wallace method [14], and other methods [15, 16].

3 Overview of the Maxwell FPGA Parallel Machine

Maxwell was developed by the FPGA High Performance Computing Alliance in Scotland to demonstrate the feasibility of running computationally demanding applications efficiently on an array of FPGAs. In this section, we will introduce both the hardware architecture and the design flow on Maxwell.

3.1 Hardware Architectures

Maxwell comprises 32 blades housed in IBM Blade Centre. Each blade comprises one 2.8 GHz Xeon with 1 GB memory and 2 Xilinx Virtex4 FPGAs each on a PCI-X sub-assembly developed by Alpha Data or Nallatech. Each FPGA board has either 512 MB or 1 GB of off-chip memory. Whilst the Xeon CPUs and FPGAs on a particular blade can communicate with each other over the PCI bus (typical transfer bandwidths of 600 Mbytes/s), the principal communication infrastructure comprises a fast Ethernet network with a high-performance switch linking the Xeons together and RocketIO linking the FPGAs. Each FPGA has 4 RocketIO links enabling the 64 FPGAs to be connected together in an 8×8 toroidal mesh as shown in Fig. 2. The RocketIOs have a bandwidth of 2.5 Gbits/s per link [17].

Logically, Maxwell can be regarded as a collection of nodes where a node is defined as a software process running on a host machine, plus some FPGA acceleration hardware.

Figure 3 shows the structure of Maxwell, the interconnection, and the detailed architecture of a single FPGA node.

As we can see in the figure, there are three kinds of interconnection on Maxwell: FPGA-FPGA interconnection, which is formed by Rocket IO, the CPU-CPU interconnection, which is Ethernet-based, and the FPGA-CPU connection, which is the PCI-X interface.

There are totally 64 FPGAs on Maxwell as shown in Fig. 3: half of them are Nallatech's off-the-shelf H101-PCIXM with Xilinx XCV4100LX FPGA devices on them. The other 32 nodes are Alpha Data ADM-XRC-4FX cards using Xilinx XCV4FX100 FPGAs.

3.2 Design Flow on Maxwell

The design flow on Maxwell can be divided into four main steps:

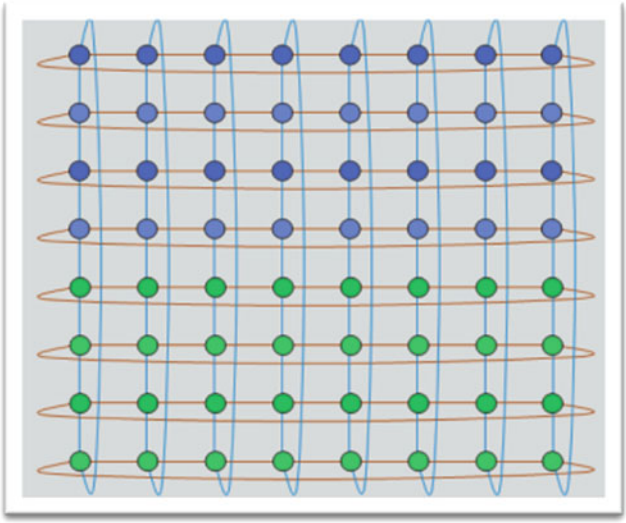


Fig. 2 FPGA links on Maxwell supercomputer

- Hardware design, including HDL coding, simulation, synthesis, and generating bitstream. This stage of work is the main part of the whole design flow.
- Software interface: This step mainly deals with the communication between the FPGA and CPU. On an Alpha Data board, for example, we use ADM-XRC-4FX Co-Processor Development Kit (CPDK) as shown in Fig. 4 to control all registers used to control the behavior of FPGAs.

At higher software level, an API will be used to deal with the standardization of high-level configuration. This tool is called Parallel Toolkit, developed by FHPCA and EPCC [18]. The aim is to configure the FPGA chip with target bitstream, as well as clock setting. The design flow also consists of:

- Message Passing Interface (MPI) [19] coding: communication between nodes is performed using MPI.
- Sun Grid Engine (SGE) job scheduler [20] scripts: allow for orderly safe job submission to the Maxwell machine.

4 Case Studies in High Performance Reconfigurable Computing

4.1 Hardware Random Number Generators

Monte-Carlo methods rely on random samples. Indeed, one random sample is needed for a single step of a Monte-Carlo simulation. For example, suppose

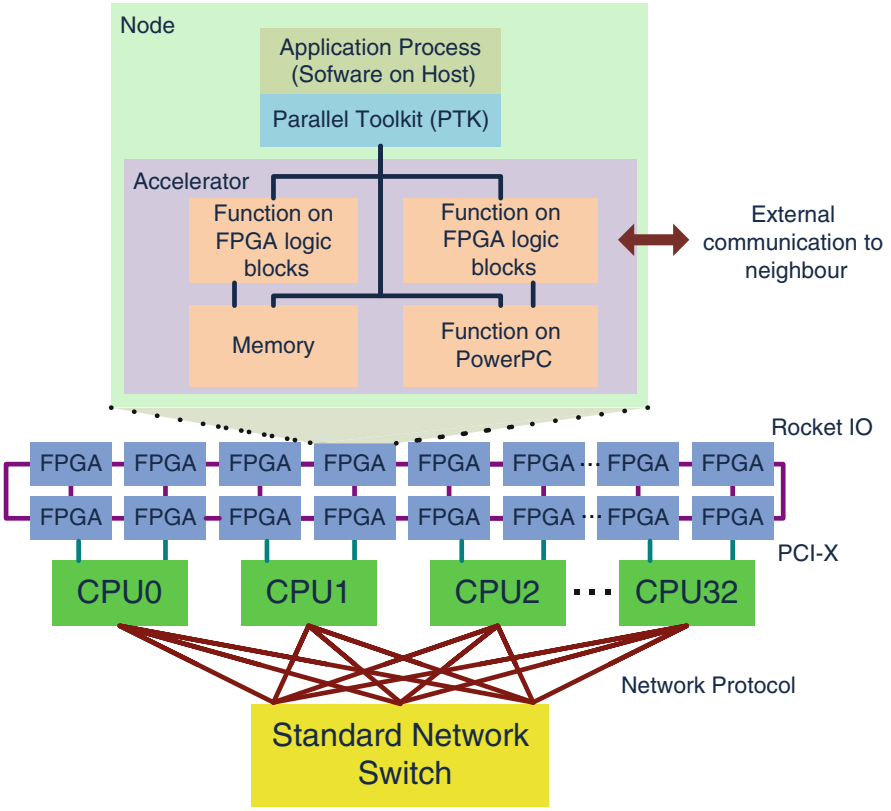


Fig. 3 Architecture of the Maxwell FPGA parallel machine

that a European option has a life length of 1 year, which is discretized to 100 time steps, and 10^6 paths are generated in the Monte-Carlo simulation, therefore, $100 \times 10^6 = 10^8$ random variables are needed for the simulation. Several considerations arise when constructing a random number generator [21]:

- Period length: any pseudo-random number generator will eventually repeat itself. Generally, we want generators with very longer periods.
- Reproducibility: it is often important to be able to re-run a simulation using exactly the same random samples as the previous simulation.
- Speed: as mentioned above, millions or even billions of samples are needed for a single simulation. It is very important to keep a very high throughput of random samples to feed a Monte-Carlo simulation engine.
- Portability: an algorithm for generating random numbers should produce the same sequence of values on all computing platforms.
- Randomness: this is the most important consideration. Theoretical properties and statistical tests could be used to evaluate the quality of the random samples.

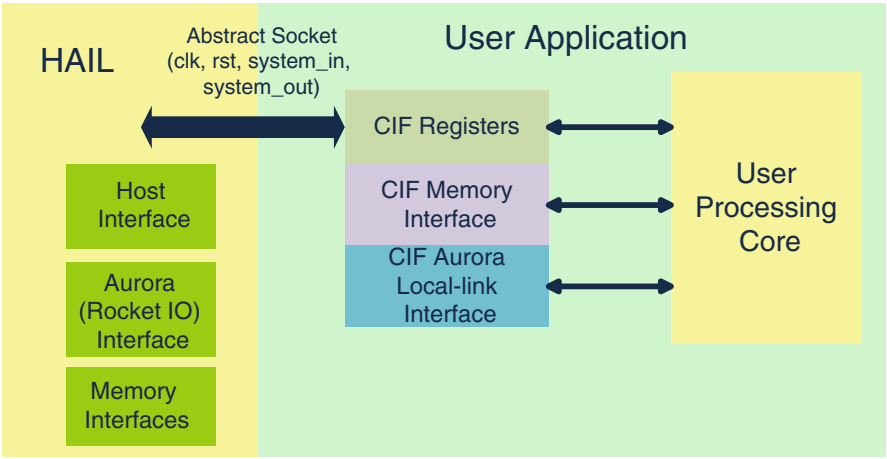


Fig. 4 Structure of CPDK application

The random samples in Brownian motion (often used to model financial option drifts) follow a Normal distribution, or say Gaussian distribution. A generally used method is to produce a set of uniform random samples (over the interval of (0, 1)), and then convert them to Gaussian random numbers. In the following two subsections, we will introduce methods for generating uniform and Gaussian random numbers, respectively.

4.1.1 Uniform Random Number Generator

Uniform random numbers are sampled from a distribution which has the following probability density function:

$$\begin{aligned} p(x) &= 1 \text{ if } 0 < x < 1 \\ &= 0 \text{ otherwise} \end{aligned} \tag{1}$$

It is a convenient distribution as there are many simple methods to transform uniform samples into samples from other distributions. In this section, three different categories of uniform random number generation methods will be introduced, namely: Linear Feedback Shift Registers (LFSR), Mersenne Twister, and Sobol.

LFSR

A generator introduced in [22] is based on a sequence of 0's and 1's generated by a recurrence of the form:

$$b_i = (a_p b_{i-p} + a_{p-1} b_{i-p+1} + \cdots + a_1 b_{i-1}) \bmod 2 \tag{2}$$

where all variables take on values of either 0 or 1. The bs are interpreted as bits and will be formed into binary representations of integers. Because the modulus is a prime, the generator can be related to a polynomial:

$$f(z) = z^p - (a_1 z^{p-1} + \cdots + a_{p-1} z + a_p) \tag{3}$$

over the Galois field GF(2) defined over the integers 0 and 1 with the addition and multiplication being defined in the usual way followed by a reduction modulo 2. An important result from the theory developed for such polynomials is that, as long as the initial vector of b's is not all 0's, the period of the recurrence in (2) is $2^p - 1$ if and only if the polynomial (3) is irreducible over GF(2).

For computational efficiency, most of the a's in Eq. (2) should be zero. The recurrence in Eq. (2) often has the form:

$$b_i = (b_{i-p} + b_{i-p+q}) \bmod 2 \tag{4}$$

Addition of 0's and 1's modulo 2 is the binary exclusive-or operation (represented as $\oplus$), and the recurrence can be written as:

$$b_i = (b_{i-p} \oplus b_{i-p+q}) \tag{5}$$

The recurrence can be performed in a feedback shift register, which is a vector of bits that is shifted, say, to the left, one bit at a time, and the bit shifted out is combined with other bits in the register to form the rightmost bit.

The uniform random number generator used in this design is called Tausworthe URNG [22], which is described by the pseudo-code shown in Fig. 5. Although traditional LFSRs are often sufficient as a uniform random number generator (URNG), Tausworthe URNGs are fast and occupy less area. Furthermore, they provide superior randomness when evaluated using the Diehard random number test suite.

Mersenne Twister

A very popular uniform random number generator, called the Mersenne Twister [23], is based on a recurrence [24] that has approximately 100 terms in the characteristic polynomial of a matrix A. Mersenne Twister has a period of $2^{19937} - 1$ and 623-variate uniformity. The Mersenne Twister algorithm generates a sequence of word vectors, which are considered to be uniform pseudo-random integers between 0 and $2^w - 1$. Dividing by $2^w - 1$, each word vector can be a real number in [0, 1].

A Mersenne prime is a number with the restriction of $2^{nw-r} - 1$. For a word x with w bit width, it is expressed as the recurrence relation:

$$x_{k+n} = x_{k+m} \oplus (x_k^u | x_{k+1}^l) A \; k = 0, 1, \ldots \tag{6}$$

```
unsigned int s0, s1, s2, b;

unsigned int taus()
{
    b = (((s0 << 13) ^ s0) >> 19);
    s0 = (((s0 & 0xFFFFFFFE) << 12) ^ b);
    b = (((s1 << 2) ^ s1) >> 25);
    s1 = (((s1 & 0xFFFFFFF8) << 4) ^ b);
    b = (((s2 << 3) ^ s2) >> 11);
    s2 = (((s2 & 0xFFFFFFF0) << 17) ^ b);
    return s0 ^ s1 ^ s2
}
```

Fig. 5 Tausworthe URNG algorithm

With | as the bitwise or and $\oplus$ as the bitwise exclusive or (XOR), x_u, x_l being x with upper and lower bitmasks applied. We choose a form of the matrix A so that multiplication by A is very fast:

$$A = R = \begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2}, \ldots a_0) \end{pmatrix} \tag{7}$$

with I_{n-1} as the $(n-1) \times (n-1)$ identity matrix (and in contrast to normal matrix multiplication, bitwise XOR replaces addition). The rational normal form has the benefit that it can be efficiently expressed as:

$$xA = \begin{cases} shiftright(x) & \text{if } x_0 = 0 \\ shiftright(x) \oplus a & \text{if } x_0 = 1 \end{cases} \tag{8}$$

where

$$x = (x_k^u | x_{k+1}^l) \quad k = 0, 1, \ldots \tag{9}$$

where $a = (a_{w-1},\ a_{w-2}, \ldots, a_0)$, $x = (x_{w-1},\ x_{w-2}, \ldots x_0)$.

Each generated word is multiplied by a suitable $w \times w$ invertible matrix T from the right to improve k-distribution to v-bit accuracy. For the tempering matrix $x \rightarrow z = xT$, we chose the following successive transformations:

$$y = x \oplus (x >> u) \tag{10}$$

$$y = x \oplus (y << s) \& b \tag{11}$$

$$y = x \oplus (y << t) \& c \tag{12}$$

$$y = x \oplus (x >> l) \tag{13}$$

In order to improve lower bit equi-distribution, we add the first and last transforms.

The coefficients for MT19937 (the commonly used variant of Mersenne Twister, which produces a sequence of 32-bit integers, and has a period of $2^{19937} - 1$) are:

$$\begin{aligned}(w,n,m,r) &= (32, 624, 397, 31)\\ a &= 9908\text{B}0\text{DF}_{16}\\ u &= 11\\ (s,b) &= (7, 9\text{D}2\text{C}5680_{16})\\ (t,c) &= (15, \text{EFC}60000_{16})\\ l &= 18\end{aligned}$$

Figure 6 gives the pseudo code of MT19937. Mersenne Twister implementations cannot be parallelized across parallel computing cores simply through changing the initial seed for each core as this does not provide uncorrelated sequences on each generator sharing identical parameters. To solve this problem and enable Mersenne Twister parallel implementations, the authors of MT19937 developed a library for the dynamic creation of Mersenne Twister parameters. This library receives user's specification such as word length, period, size of working area, and a process ID, so that ID number is encoded in the characteristic polynomial of Mersenne Twister.

The process of generating Mersenne Twister numbers can be separated into the following 4 steps:

- Generating the tempering matrix for each computing core based on the given word length, size of working area, and process ID.
- Initializing the generator based on the given seed number.
- Generating the untempered numbers.
- Tempering.

After an initial feasibility analysis, we noticed that the first two steps consume the most of the hardware resources, but the least computing time: both of these two steps only run once at the beginning of the generation, and this time does not increase with the length of the random sequence. Moreover, the following two steps only require shift and XOR operations which consume relatively few logic resources on FPGA. The output of the first step is the 12 parameters needed by steps 3 and 4, and the output of the second step is 624 initialized numbers. However, since we target a parallel FPGA machine assuming the same bitstream on each FPGA node, each FPGA needs different initial numbers. There are two methods to achieve this: generate the initial numbers on each FPGA in the first stage using a different

```
// Create a length 624 array to store the state of the generator
int[0..623] MT
int index = 0

// Initialize the generator from a seed
function initializeGenerator(int seed) {
    MT[0] := seed
    for i from 1 to 623 { // loop over each other element
        MT[i] := last 32 bits of(1812433253 * (MT[i-1] xor (right shift by 30 bits(MT[i-1])))
+ i) // 0x6c078965
    }
}
 /* Extract a tempered pseudorandom number based on the index-th value, calling
generateNumbers() every 624 numbers */
function extractNumber() {
    if index == 0 {
        generateNumbers()
    }

    int y := MT[index]
    y := y xor (right shift by 11 bits(y))
    y := y xor (left shift by 7 bits(y) and (2636928640))
// 0x9d2c5680
    y := y xor (left shift by 15 bits(y) and (4022730752)) // 0xefc60000
    y := y xor (right shift by 18 bits(y))


    index := (index + 1) mod 624

    return y

}
 // Generate an array of 624 untempered numbers
```

Fig. 6 Pseudo-code of MT19937

seed; or we can generate the initial numbers on CPU, and then transfer them to the FPGAs' BlockRAM. The latter needs 624 data to be transferred through the CPU-FPGA communication or even more if we instantiate more than one simulation core on one FPGA, which will consume considerable communication time. Hence, we chose to generate the initial numbers on FPGA at the expense of a slight resource

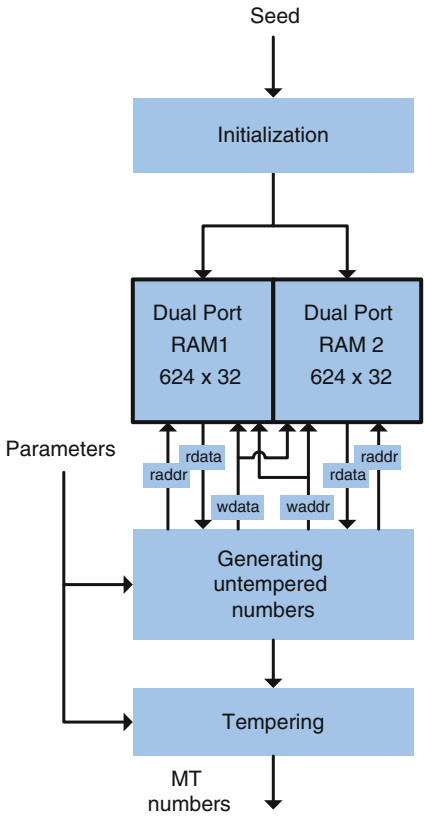


Fig. 7 Mersenne twister random number generator

overhead (mainly one multiplier for each core). The architecture of a Mersenne Twister random number generator core is hence given in Fig. 7.

Based on the Eqs. (8) and (9) we have to read two numbers from the BlockRAM at step 2 each clock cycle. To pipeline the random number generator, two block RAMs are needed: one is for x_l and the other is for x_u. Note that we can pre-compute the parameters and hardwire them to each Mersenne Twister core.

Sobol Random Number Generator

The theory of Sobol numbers starts with modular integer arithmetic. Two integers i and j are called congruent with respect to the modulus m, i.e.

$$i \stackrel{\Delta}{=} j \bmod m \tag{14}$$

if and only if the difference $i - j$ is divisible by m. For m being prime, the combination of addition and multiplication modulo m, plus a neutral element with

respect to both, is also called a finite commutative ring which is isomorphic to a Galois Field with m elements, $GF[m]$. A polynomial $P(z)$ of degree g,

$$P(z) = \sum_{j=0}^{g} a_k z^{g-j}, \tag{15}$$

is considered to be an element of the ring $GF[m,\ z]$ of polynomials over the finite field $GF[m]$ if we assume all of the coefficients a_k to be $\in GF[m]$. A polynomial $P(z)$ of positive degree is considered to be irreducible modulo m if there are no other two polynomial $Q(z)$ and $R(z)$ which are not constant or equal to $P(z)$ itself such that:

$$P(z) \stackrel{\Delta}{=} Q(z)R(z) \bmod m, \tag{16}$$

An irreducible polynomial modulo m in $GF[m,\ z]$ is the equivalent to a prime number in the set of integers. The order of a polynomial $P(z)$ modulo m is given by the smallest positive integer q for which $P(z)$ divides $z_q - 1$, i.e.

$$q = \inf_{q'}\{q' | z^{q'} - 1 \stackrel{\Delta}{=} P(z)R(z) \bmod m\} \tag{17}$$

for some non-constant polynomial.

To construct a Sobol sequence, we initially construct a vector of numbers, known as direction numbers, of word length w which will serve as a base for the calculation of the Sobol numbers. We need a direction number for each digit, in base 2, of the numbers that will be used in the sequence. In our case, we used 24-bit fixed number representation for our implementation (i.e., $w = 24$). The dimension will be indexed by $k = 1, 2, \ldots D$. The construction of direction numbers is sketched below.

Given a series of integers $a_1,\ a_2,\ \ldots a_{d-1}$ that are zero or one, the primitive polynomial modulo 2 of degree d is defined as:

$$P = x^d + a_1 x^{d-1} + a_2 x^{d-2} + \cdots + a_{d-1}x + 1 \tag{18}$$

For each dimension k, a particular primitive polynomial is chosen and a series of integers, $m_{ki},\ d_k < i < w$, is generated, starting from the following recursion with d_k terms, where d_k is the degree of the polynomial associated with the kth dimension:

$$\begin{aligned} m_{ki} = {} & 2a_{k,1}m_{k,i-1} \oplus 2^2 a_{k,2}m_{k,i-2} \oplus \cdots \oplus 2^{d_k-1}a_{k,d_k-1}m_{k,i-d_k+1} \\ & \oplus (2^{d_k}m_{k,i-d_k} \oplus m_{k,i-d_k}) \end{aligned} \tag{19}$$

for $k = 1,\ 2, \ldots D$, where $\oplus$ represents the bit to bit sum, XOR (exclusive OR), applied on the base 2 representation of the integer m_{ki}. It is necessary to supply the d_k initial values of m_{ki} in each dimension. We used the simple method described in [25] i.e. use a separate pseudo-random number generator to draw uniform variates from

(0, 1) and initialize as follows: draw u_{kl} from a separate uniform random number generator such that:

$$w_{kl} = \text{int}[u_{kl} \times 2^l] \tag{20}$$

is odd, and set:

$$m_{kl} = w_{kl} \times 2^{b-l} \quad for\ l = 1, \cdots d_k \tag{21}$$

The following step is to construct the Sobol numbers using the direction numbers. The nth Sobol number of kth dimension can be obtained from the $(n-1)$th using the following equation:

$$y_{nk} = \sum_{j=1}^{d \oplus 2} m_{kj} 1 \ \{j\text{th bit(counting from the right) of in teger is set}\} \tag{22}$$

However, if we realize it using Gray code instead of using the binary representation of the sequence counter n directly, (18) can be re-written as:

$$y_{nk} = y_{(n-1)k} \oplus m_{kj} \ \{j\text{th bit is the rightmost zero of } n-1\} \tag{23}$$

where m_{kj} is the direction number associated with the rightmost zero in the binary representation of $n-1$. Hence, the nth Sobol number for kth dimension can be obtained from the $(n-1)$th using just one direction number.

The architecture of our Sobol sequence generator is shown in Fig. 8. The black block represents a flip-flop. The architecture is a straightforward implementation of the algorithm above. As the generation of each Sobol number needs the number produced in the last cycle, we have to store the last Sobol number according to Eq. (23). Another point that needs to be clarified here is that since the XOR operation is applied between the direction number and the last Sobol number, we have to store the previously generated Sobol numbers for each dimension. In the case of a 100-day simulation, we have to keep 100 Sobol numbers (in a dual port RAM as shown in Fig. 8). The random number generator picks the Sobol number of the last path of the next day and pushes the current Sobol number in the RAM in each clock cycle.

After building the low-discrepancy numbers, we have to separate them into several sub-streams to implement the distributed Quasi-Monte Carlo simulation. There are two possible approaches of partitioning [26]:

Blocking: disjoint contiguous blocks of overall length l of the original sequence are generated by separate processing elements (PEs). This is achieved by simply using a different starting point on each PE (e.g. $\text{PE}_i,\ i = 0, \ldots, p-1$, generates the vectors $x_{il}, x_{il+1}, x_{il+2}, \ldots, x_{il+l-1}$).

Leaping: interleaved streams of the original sequence are generated by the PEs. Each PE skips those consumed by other PEs (*leap-frogging*) (e.g., PE_i, $i = 0, \ldots, p-1$, generates the vectors $x_i, x_{i+p}, x_{i+2p}, x_{i+3p}, \ldots$).

We adopt the *blocking* method to perform the partitioning in our implementations. In it, the separation of a Sobol stream can be realized by using a different initial number in each core as depicted in Fig. 9. Apart from the separation in a

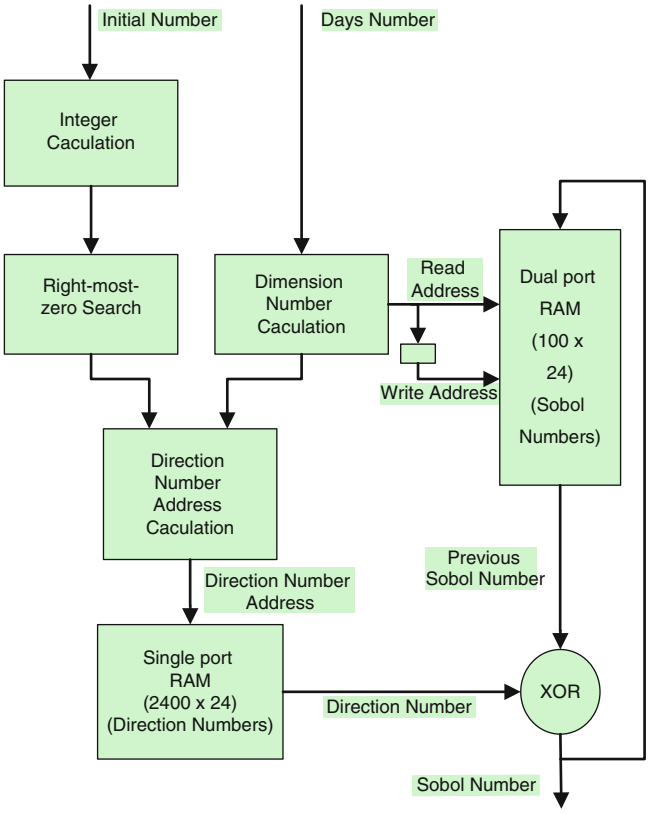


Fig. 8 Architecture of our Sobol sequence generator

single FPGA, or say, node, we have to separate the sequence twice, in the case of a multi-FPGA implementation. In that case, two levels of partitioning are necessary.

The architecture is different from the work mentioned in [10]. We parallelize the Sobol RNG by separating the whole sequence to several sub-sections. This is done by choosing a different initial number for each RNG, which can be seen in Fig. 9. This can be seen as blocking method.

Note finally that since Sobol sequence numbers are always integer numbers, the same module can be used for any subsequent processing arithmetic type, e.g., fixed or floating point.

4.1.2 Gaussian Random Number Generator

As the random variables required in Monte-Carlo simulations follow the Normal distribution, the uniform variables need to be converted to Normal random variables.

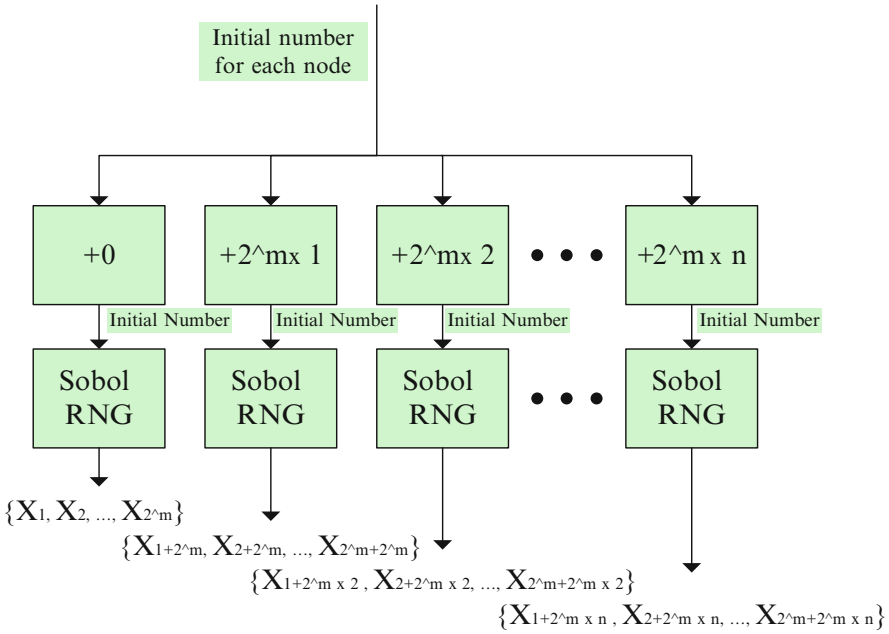


Fig. 9 Parallelism of random number generation

In this section, two methods of conversion will be introduced, namely the inverse cumulative distribution function method (ICDF) and the transformation method.

ICDF

For a random variable X, the cumulative distribution function (CDF) is the function P_X defined by:

$$P_X(x) = \Pr(X \leq x) \tag{24}$$

where $\Pr(A)$ represents the probability of the event A. Two important properties are: the CDF is non-decreasing, and it is continuous from the right. Particularly, the CDF and ICDF of Normal distribution are:

$$F(y) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{y} \mathrm{e}^{(-(t-1)^2/2\sigma^2)} \mathrm{d}t \tag{25}$$

$$f(x) = \left| F^{-1}\left(\frac{x}{2}|\mu, \sigma\right)\right| \tag{26}$$

Figures 10 and 11 show the CDF and ICDF of the standard Normal distribution (with mean 0 and variance 1), respectively. As we can see in Fig. 11, if a uniform variable,

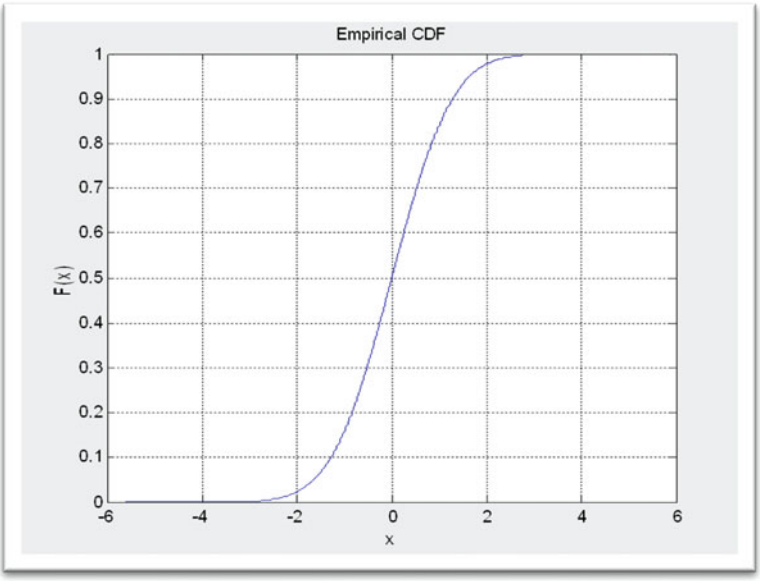


Fig. 10 CDF of standard normal distribution

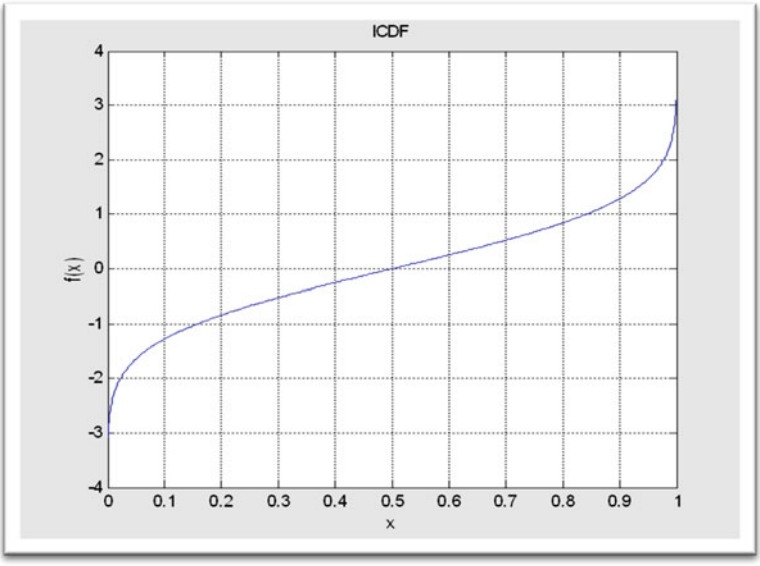


Fig. 11 ICDF of standard normal distribution

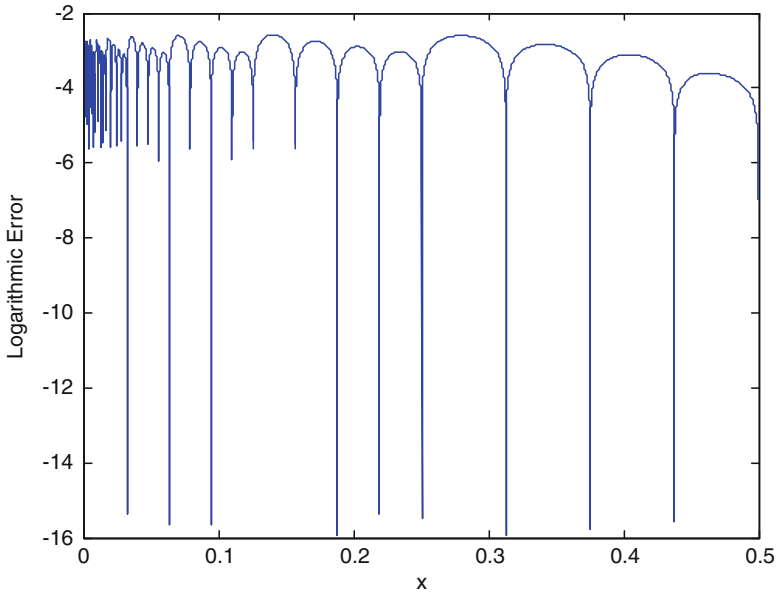


Fig. 12 Logarithmic error of ICDF

which is in the interval of 0 and 1, is the input of the ICDF, the output will follow the Normal distribution. As there is no closed-form for the ICDF, approximation must be adopted in the calculation. Piecewise polynomial approximation is used in this work.

We use a one degree piecewise linear approximation with 80 subsections used between 0 and 0.5 (since the ICDF is an odd function if shifted to the left by 0.5, we only need to calculate its values from 0 to 0.5). The coefficients of the function are pre-computed and stored in a single port RAM. Figure 12 gives the logarithmic error (The logarithmic error is defined as: $LogErr = log_{10}|LinearApproximationResult - GoldenReferenceValue|$) between our ICDF hardware core, using 26-bit fixed point arithmetic, and our golden reference from Matlab's ICDF function. The worst error is around $10^{-2.5}$.

We use two goodness-of-fit tests to check the normality of the Gaussian noise: the chi-square (χ^2) test and the Kolmogorov–Smirnov (K–S) test.

- Chi-Square test:

The Chi-Square test quantizes the x axis into k bins, and then calculates the actual number of samples appearing in each bin. Next, we compare this number with the number of samples which should appear in each bin based on a specific distribution and get a single number. This number can represent the overall quality metric. For example, if n is the number of observations, p_i is the probability that each

observation falls into category I, and Y_i is the number of observations that actually do fall into category i. The Chi-Square statistic is given by

$$\chi^2 = \sum_{i=1}^{k} \frac{(Y_i - np_i)^2}{np_i}$$

- K–S test:

The K–S test tries to determine if two datasets differ significantly. It quantifies the distance between the empirical distribution function of the samples and the cumulative distribution function of the reference distribution. The empirical distribution function F_n for n independent and identically distributed random variables X_i is defined as:

$$F_n(x) = \frac{1}{n} \sum_{i=1}^{n} I_{X_i \leq x}$$

where $I_{Xi \leq x}$ is the indicator function, equal to 1 if $X_i \leq x$ and equal to 0 otherwise. The K–S statistic is then given by:

$$D_n = \sup_x |F_n(x) - F(x)|$$

where sup x is the supremum of the set of distances.

Both of the tests will give p-values for the outputs. The general convention is to reject the null hypothesis—that the samples are normally distributed if the p-value is less than 0.05. Results show that the p-value of samples from our design is more than 0.05.

Transformation

A very crude transformation method to construct normally distributed samples is to add up 12 uniform variates, and subtract 6. This method is the Central Limit Theorem applied to a sample of size 12. It is a poor approximation and always slower than other methods. Another way to transform the uniform variables is the Box–Muller transformation [27]. If u and v are independent standard uniform variables in (0, 1), a pair of independent Normal variables x and y can be generated using

$$\begin{aligned} x &= \sqrt{-1 \ln u} \sin(2\pi v) \\ y &= \sqrt{-1 \ln u} \cos(2\pi v) \end{aligned} \tag{27}$$

One problem of Box–Muller transformation is that it cannot be used for converting the low-discrepancy numbers such as Sobol numbers.

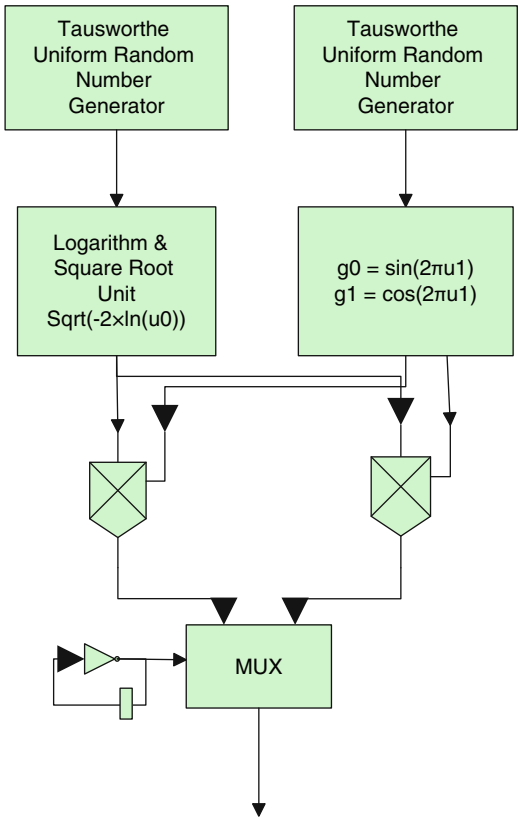


Fig. 13 Gaussian noise generator architecture

The Box–Muller method is conceptually straightforward. Given two independent realizations (u_1 and u_2) of a uniform random variable over the interval [0, 1), and a set of intermediate functions f, g_1 and g_2 so that:

$$f(u_1) = \sqrt{-2 \times \ln(u_1)} \tag{28}$$

$$g_1(u_2) = \sin(2\pi u_2) \tag{29}$$

$$g_2(u_2) = \cos(2\pi u_2) \tag{30}$$

$$x_1 = f(u_1)g_1(u_2) \tag{31}$$

$$x_2 = f(u_1)g_2(u_2) \tag{32}$$

Then providing two samples of a Gaussian distribution $N(0, 1)$, x_1 and x_2.

Based on this algorithm, the corresponding hardware architecture is given in Fig. 13.

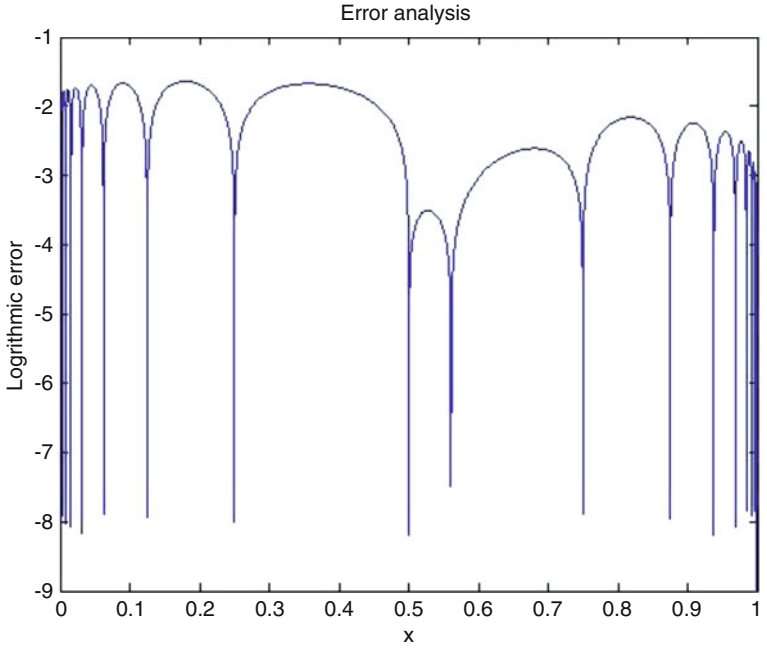


Fig. 14 Logarithmic error of $f(u_1) = \sqrt{-2 \times \ln(x)}$

Logarithm and trigonometric functions are computed using the piecewise linear approximate method [28, 29]. The logarithm errors of both functions are shown in Figs. 14 and 15. We generated 100,000 samples, and the PDF is shown in Fig. 16.

4.2 Financial Computing Models and Their Implementations on Maxwell

4.2.1 European Option Pricing

The Monte-Carlo method for European options pricing is based the Black–Scholes model of option price evolution:

$$S_{\Delta t} = S_0 \left(1 + \left(\left(\mu - \frac{\sigma^2}{2}\right)\delta t + \sigma\varepsilon\sqrt{\delta t}\right)\right) \tag{33}$$

where $S_{\Delta t}$ and S_0 are the stock prices at times Δt and zero, respectively, μ is the expected rate of return of the stock, σ is the volatility of the stock price, and ε is random variable with mean 0 and variance 1.

The simulation process can be described as the following algorithm:

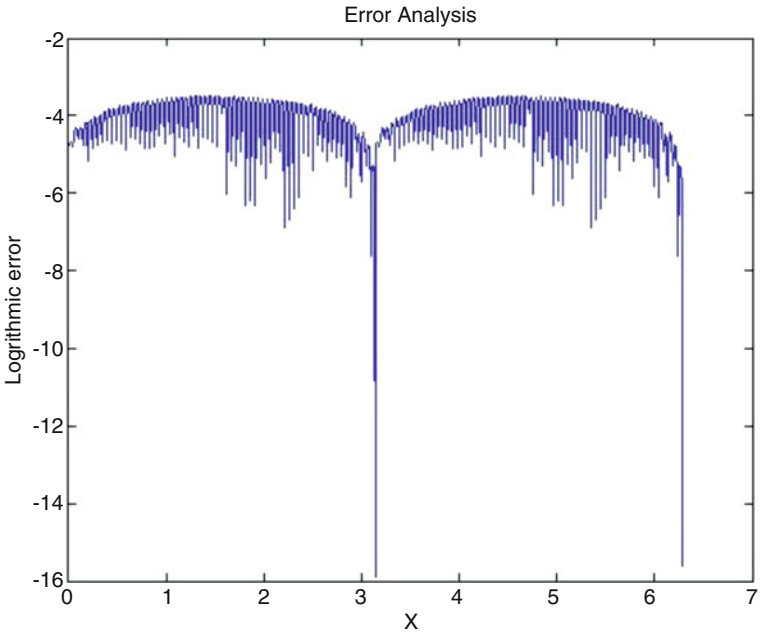


Fig. 15 Logarithmic error of $g_1(x) = \sin(2\pi x)$

For any $n \geq 1$, the estimator $\hat{C}_n$ of the option price is unbiased, in the sense that its expectation is the target quantity:

$$E[\hat{C}_n] = C \equiv E[\mathrm{e}^{-rT}(S(T) - K)^+]$$

The estimator is strongly consistent, meaning that as $n \to \infty$,

$$\hat{C}_n \to C \text{ with probability } 1$$

The algorithm is described by the pseudo-code in Fig. 17.

Note that the simple European option model does not need small time steps to build the paths. Single big time step can be used for generating the stock price path. However, as the Monte-Carlo simulation module is the basic part of all the other option pricing engines, in which small time steps must be used, the implementation of European option pricing model will use the small time step.

The European option pricing engine comprises an LSFR uniform random generator, a Box-Muller Gaussian random generator and the Monte-Carlo simulation core with implements Eq. (33). The architecture can be seen in Fig. 18.

The computing core was captured in Verilog and synthesized using Xilinx ISE 9.2i. We increased the number of computing cores until resources run out. Our user application processor implements 20 Monte-Carlo cores occupying 36144 slices on

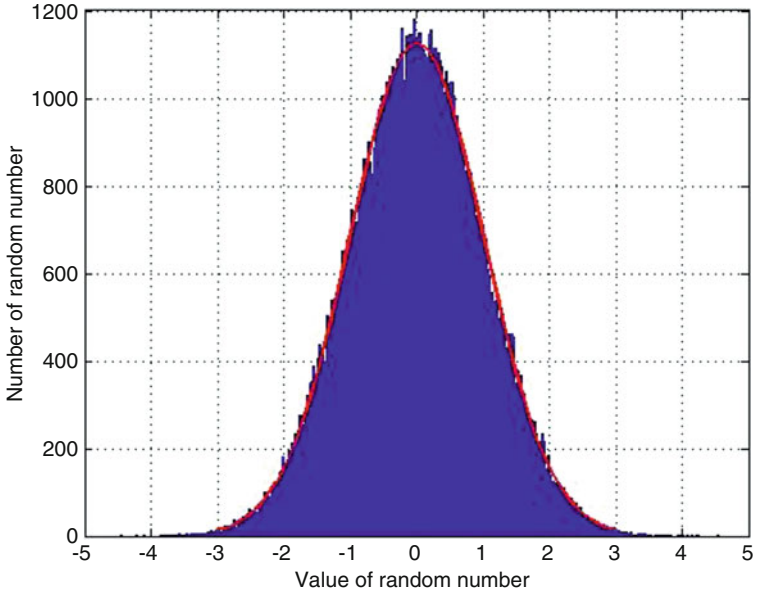


Fig. 16 PDF of the generated random variables

for i = 1 to n
Generate m random samples
for t = 1 to m

$$S_{t+1} = S_t(1+((\mu - \frac{\sigma^2}{2})\delta t + \sigma\varepsilon\sqrt{\delta t}))$$

$$C_i = e^{-rT}(S(T) - K)^+$$

$$\hat{C}_n = \text{mean}(C_i)$$

Fig. 17 Algorithm for path generation of European option prices

an XC4VFX100-10ff1517 FPGA, which has 42176 slices in all; all 160 DSP48s units are utilized. We set the clock frequency on Maxwell's FPGA node to 75 MHz.

We implemented our hardware Monte-Carlo simulation solution for European option pricing on Maxwell. We also ran an equivalent C++ based software solution on Maxwell and run it on the 2.8 GHz Xeon processors (each with 1 GB of memory). The execution time of both FPGA and CPU implementations is shown in Fig. 19.

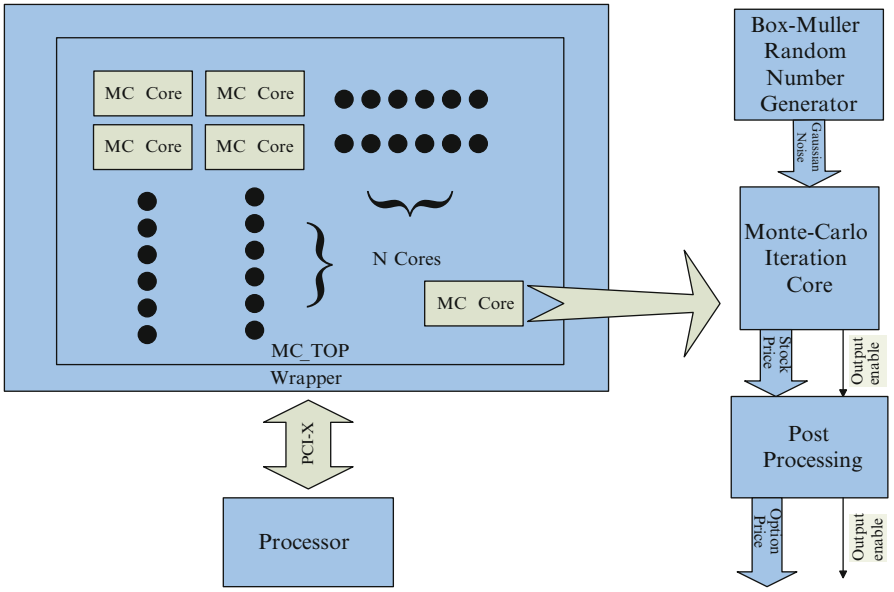


Fig. 18 Generic architecture of Monte-Carlo simulation engine4

From Fig. 19, we can see that the computing time decreases linearly as the number of nodes increases. The reason is that communication time is very limited during Monte-Carlo simulation: only broadcasting parameters at the beginning of simulation and gathering results at the end of simulation. As we pipelined the design and set clock frequency to 75 MHz, our Monte-Carlo computing core finishes one iteration in 13.3 ns. As the Input/Output and communication overheads are limited, we estimate the overall computing time to be:

$$\begin{aligned} ComputingTime = ClockPeriod \times (NumOfPaths \times NumOfDays) \\ -(NumOfCores \times NumOfNodes) \end{aligned} \tag{34}$$

Take the result of a 32 nodes experiment as an example: the clock period is 13.3 ns; the number of paths is $2^{17} \times 10 = 1.31 \times 10^6$; the number of days is 100; the number of cores per FPGA is 20, and the number of nodes is 32. This gives us

$$ComputingTime = 13.3 \times (1.31 \times 10^6 \times 100) - (20 \times 32) \approx 2.731 \times 10^6 \text{ns} \tag{35}$$

Overall, the FPGA implementations are 750x faster than the equivalent software implementations. It is worth mentioning, however, that our software implementation was not optimized on the Xeon processors.

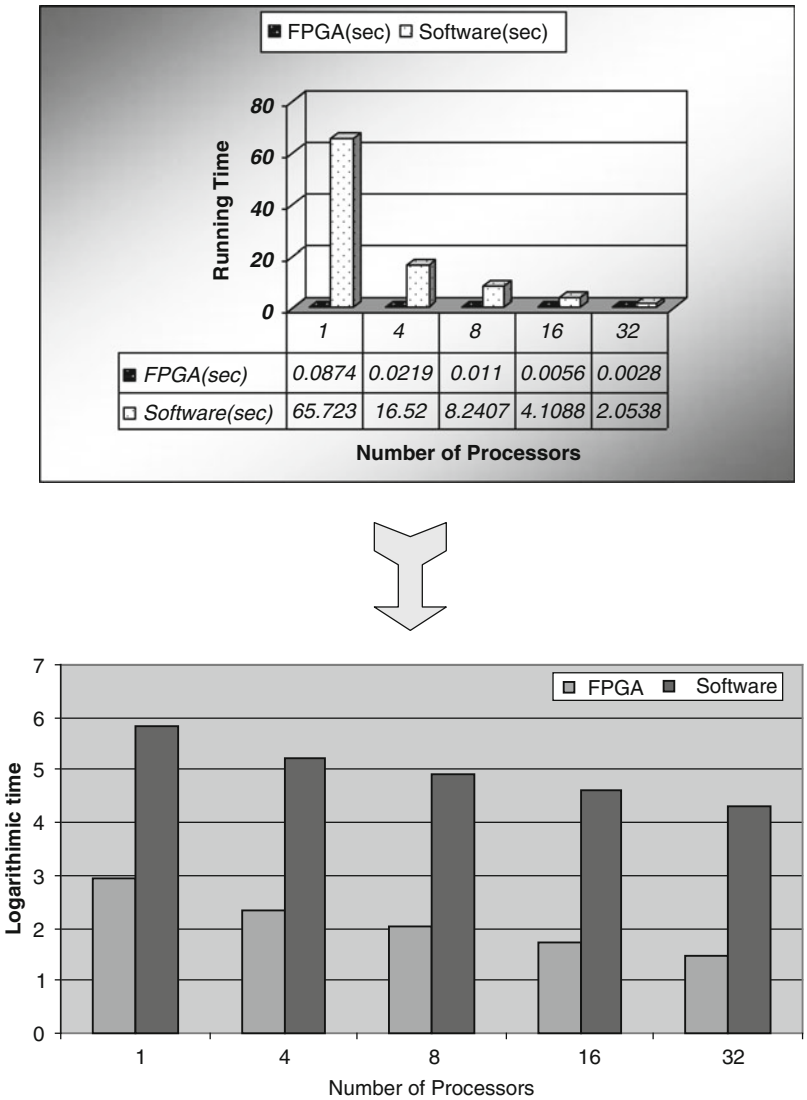


Fig. 19 Running time of C++ & FPGA implementation

4.2.2 Asian Option Pricing

In Asian options, the payoff is determined by the average underlying price over some pre-set period of time. Hence, an Asian option can be calculated as:

$$C_{\text{AsianCall}} = \max(0, S_{\text{avg}} - K)$$
$$C_{\text{AsianPut}} = \max(0, K - S_{\text{avg}})$$

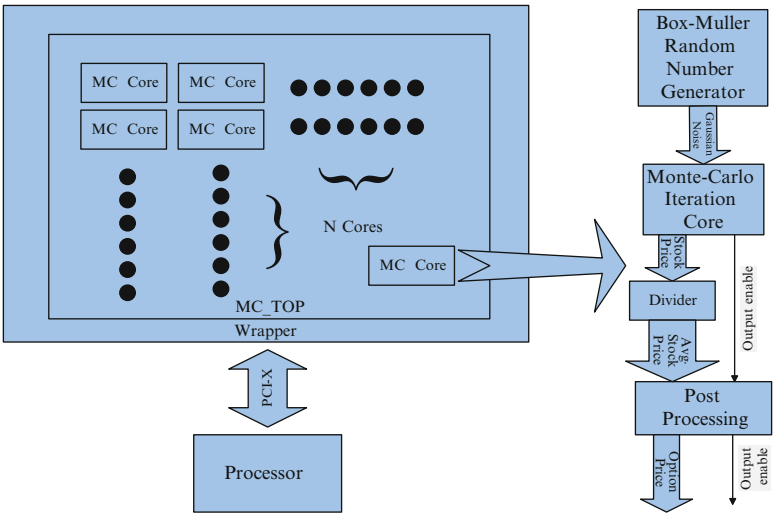


Fig. 20 Architecture of Asian option simulation engine

The average of S can be obtained in many ways. In the continuous case, this is calculated through an integral:

$$S_{\text{ave}} = \frac{1}{T}\int_0^T S(t)\mathrm{d}t$$

or in the discrete version:

$$S_{\text{ave}} = \frac{1}{N}\sum_{i=1}^{N} S(t_i)$$

There is also a type of Asian options with geometric average:

$$S_{ave} = \exp\left(\frac{1}{T}\int_0^T \ln(S(t))\mathrm{d}t\right)$$

From the characteristic of the Asian option, we can see that the Asian option has a lower volatility than the European option. Hence, it is less risky and cheaper than the European option. The Asian option is arguably more appropriate than regular options for meeting some of the needs of corporate treasurers.

In our work, we mainly dealt with the discrete version of the average. From the hardware point of view, the modification implies adding an accumulator (which is within the MC core) and divider after the Monte-Carlo simulation core as shown in Fig. 20.

With the extra divider, the Asian option hardware simulation engine was of 600x faster than the equivalent software implementation. This implementation is

configured as the same wordlength, clock frequency, and the number of computation kernels as the European option pricing model.

4.2.3 The GARCH Model

One assumption in the Black–Scholes model that is not always true in practice is the assumption that volatility is constant. Indeed, practitioners often find it necessary to change the volatility parameter when using the Black–Scholes model to value options. In the case where the stock price and volatility are correlated, there is no simple solution to the model equations and Monte-Carlo-based simulations often become necessary.

One technique for modeling volatility that has become popular is Generalized Autoregressive Conditional Heteroskedasticity—GARCH model [30]. The most commonly used GARCH model is GARCH (1, 1) where the volatility is given by the following equation:

$$\sigma_i^2 = \sigma_0 + \alpha\sigma_{i-1}^2 + \beta\sigma_{i-1}^2\lambda^2 \tag{36}$$

Here α, and β are constants which can be estimated from historical data using maximum likelihood methods. σ_0 is the volatility of the stock price at time 0, σ_i and σ_{i-1} are the volatilities at time $i\Delta t$ and $(i-1)\Delta t$. λ is a random variable with a normal (Gaussian) distribution with a mean of zero and a standard deviation of 1.0. Notice that the random variable in the GARCH model is different from the one used in the describing the evolution of stock prices. The two random variables represent two independent stochastic processes.

For options that last less than 1 year, the pricing impact of a stochastic volatility is fairly small in absolute terms. It becomes progressively larger as the life of the option increases.

The GARCH model has only one extra module compared to the European pricing engine implementation, namely a stochastic volatility model implementation. The architecture is depicted in Fig. 21.

We captured our hardware architecture using Verilog-HDL and synthesized it using Xilinx ISE 9.2i. We could fit 11 Monte-Carlo cores on one single FPGA chip. These occupied 39,466 slices on an XCV4FX100-10ff1517 FPGA, which has 42,176 slices overall (the word length is configured as 26 bits). Besides, all 160 DSP48s units were utilized. The peak clock frequency of the core is 53 MHz. We set the clock frequency on the Maxwell's FPGA nodes to 50 MHz.

Figure 22 gives the execution time of the GARCH option pricing model on the Maxwell machine using an increasing number of nodes. This is shown for our FPGA implementation as well as for an equivalent software implementation running on the 2.8 GHz Xeon processors. In both cases, the execution time decreases linearly as the number of nodes increases. This is because inter-communication time is negligible compared to the computing time. Indeed, the only instances where communication between the host software and the Monte-Carlo cores (running on

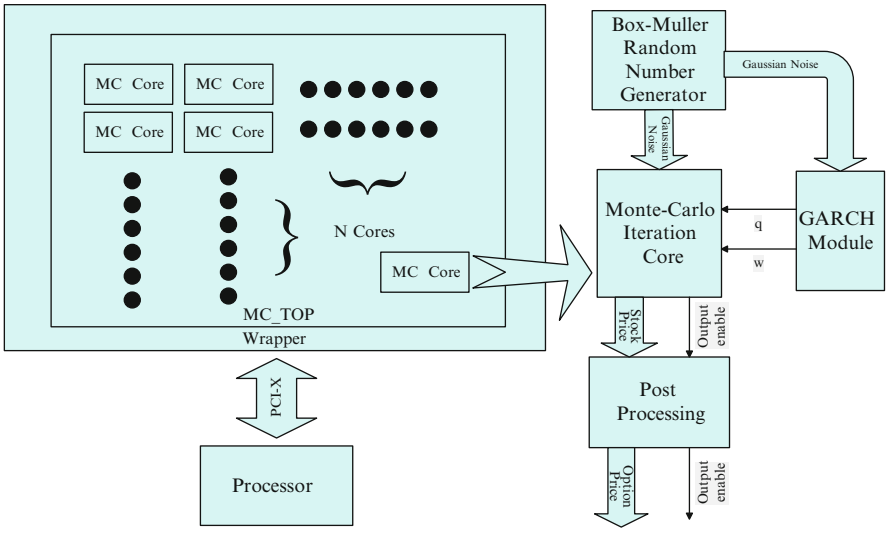


Fig. 21 Generic architecture of GARCH model simulation engine

FPGA or on the Xeon processors) is needed is when parameters are broadcasted to the cores at the beginning of the execution, and when results are gathered from the cores at the end of the simulation. Compared to software, our FPGA implementation results in a 340x speed-up. It is worth mentioning that this speed-up figure is independent of the number of nodes (FPGA/CPU) used.

The reason behind the high speed-up figure of the FPGA implementation, despite the huge difference in clock frequency (50 MHz for the FPGAs compared to 2.8 GHz for the Xeon's) is due to the high level of process parallelism (11 cores running in parallel on each FPGA device) as well as the high degree of pipelining used within each core.

4.2.4 American Option Pricing

As mentioned in Sect. 2, American options are call or put options that can be exercised at any time up to the expiration date. After generating the paths of the stock price, using the same approach as the simulation of European options, we need to go backward to find the best day to exercise the option. There are two different situations in the decision-making procedure: at the final exercise date, the optimal exercise strategy for an American option is to exercise the option if it is in the money; however, prior to the final date, the optimal strategy is to compare the immediate exercise value with the expected cash flows from continuing, and then exercise if immediate exercise is more valuable [31]. Thus, we can see that the strategy to optimally exercise an American option is to identify the conditional

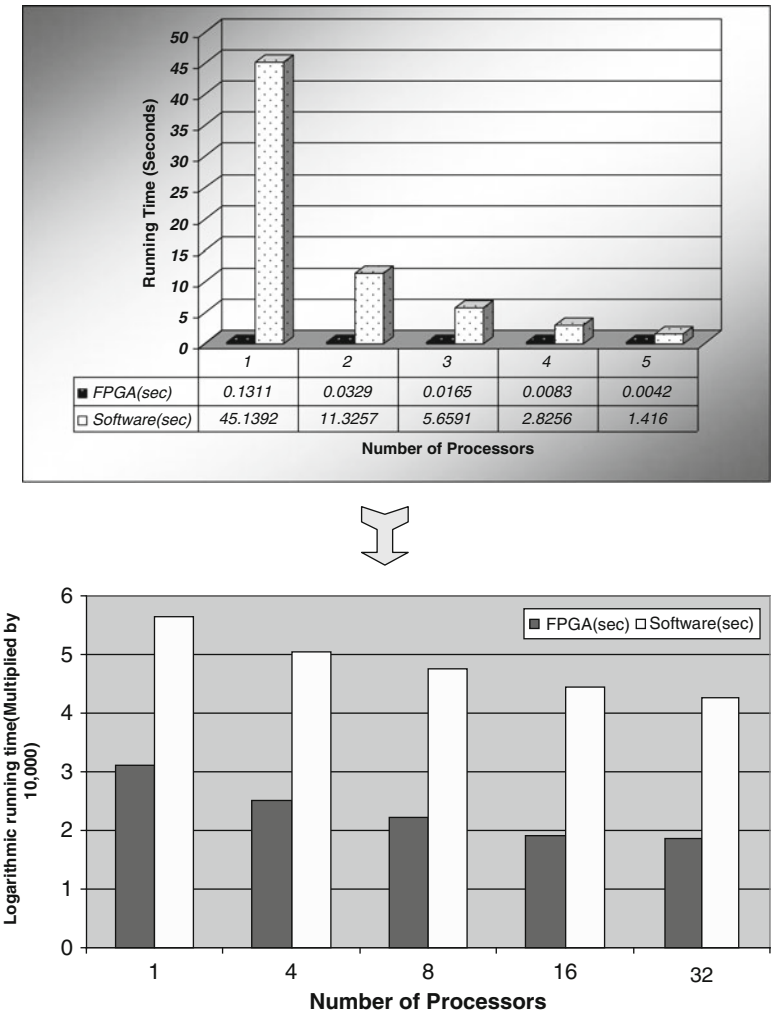


Fig. 22 Execution time of the GARCH option pricing model

expected value of continuation. We use the cross-sectional information in the simulated paths to identify the conditional expectation function. This is done by regressing the subsequent realized cash flows from continuation on a set of basis functions of the values of the relevant state variables. The fitted value of this regression is an efficient unbiased estimate of the conditional expectation function and allows us to accurately estimate the optimal stopping rule for the option.

Here we use a simple example to depict the least-squares regression. A more detailed description can be found in [31]. Consider an American put option on a share of non-dividend-paying stock. The put option is exercisable at a strike price of

Table 1 Stock price paths

Path	$t = 0$	$t = 1$	$t = 2$	$t = 3$
1	1.00	1.09	1.08	1.34
2	1.00	1.16	1.26	1.54
3	1.00	1.22	1.07	1.03
4	1.00	0.93	0.97	0.91
5	1.00	1.11	1.56	1.52
6	1.00	0.76	0.77	0.90
7	1.00	0.92	0.84	1.01
8	1.00	0.88	1.22	1.34

Table 2 Cash-flow matrix at time 3

Path	t=1	t=2	t=3
1	–	–	0
2	–	–	0
3	–	–	0.07
4	–	–	0.18
5	–	–	0
6	–	–	0.20
7	–	–	0.09
8	–	–	0

Table 3 Regression at time 2

Path	Y	X
1	0.00×0.94176	1.08
2	–	–
3	0.07×0.94176	1.07
4	0.18×0.94176	0.97
5	–	–
6	0.20×0.94176	0.77
7	0.09×0.94176	0.84
8	–	–

1.10 at times 1, 2, and 3, where time 3 is the final expiration date. The riskless rate is 6%. We use eight simulation paths for the price of the stock, which are shown in Table 1.

First, considering the situation of not exercising the option before the final expiration date at time 3, the cash flows realized by the option holder from following the optimal strategy at time 3 are given in Table 2.

If the put is in the money at time 2, the option holder must then decide whether to exercise the option immediately or continue the option's life until the final expiration date at time 3. From the stock-price matrix, there are five paths for which the option is the money at time 2. Let X denote the stock prices at time 2 for these five paths and Y denote the corresponding discounted cash flows received at time 3 if the put is not exercised at time 2, the regression at time 2 is shown in Table 3.

To estimate the expected cash flow from continuing the option's life conditional on the stock price at time 2, we regress Y on a constant, X, and X^2. This specification is one of the simplest possible; more general specifications are given

Table 4 Optimal early exercise decision at time 2

Path	Exercise	Continuation
1	0.02	0.0369
2	–	–
3	0.03	0.0461
4	0.13	0.1176
5	–	–
6	0.33	0.1520
7	0.26	0.1565
8	–	–

Table 5 Cash-flow matrix at time 2

Path	$t = 1$	$t = 2$	$t = 3$
1	–	0	0
2	–	0	0
3	–	0	0.07
4	–	0.13	0
5	–	0	0
6	–	0.33	0
7	–	0.26	0
8	–	–	0

Table 6 Regression at Time 1

Path	X	Y
1	1.09	0.00×0.94176
2	–	–
3	–	–
4	0.93	0.13×0.94176
5	–	–
6	0.76	0.33×0.94176
7	0.92	0.26×0.94176
8	0.88	0.00×0.94176

in [31]. Although we Although we only use this specification in the hardware implementation, a more general implementation can easily be adopted. The resulting conditional expectation function is $\mathrm{E}[Y|X] = -1.070 + 2.983X - 1.813X^2$.

With this conditional expectation function, we now compare the value of immediate exercise at time 2, given the first column in Table 3, with the value from continuation, given in the second column in Table 4. This leads to the following matrix in Table 5, which shows the cash flows received by the option holder conditional on not exercising prior to time 2.

Proceeding recursively, we next examine whether the option should be exercised at time 1. Again we choose the paths where the option is in the money. Let X denote the stock prices at time 1 for these paths and Y denote the corresponding discounted cash flows received at time 2 if the put is not exercised at time 1. The regression data at time 1 is shown in Table 6.

The conditional expectation function at time 1 is estimated by again regressing Y on a constant, X, and X^2. Then we use the estimated conditional expectation

Table 7 Optimal early exercise decision at Time 1

Path	Exercise	Continuation
1	0.01	0.0139
2	–	–
3	–	–
4	0.17	0.1092
5	–	–
6	0.34	0.2866
7	0.18	0.1175
8	0.22	0.1533

Table 8 Option cash-flow matrix

Path	t=1	t=2	t=3
1	.00	.00	.00
2	.00	.00	.00
3	.00	.00	.07
4	.17	.00	.00
5	.00	.00	.00
6	.34	.00	.00
7	.18	.00	.00
8	.22	.00	.00

generated by the regression to calculate the estimated continuation values, as shown in Table 7.

After deciding the exercise strategy at times 1, 2, and 3, we can get the final option cash flow matrix as shown in Table 8. Then, the option can be valued by discounting each cash flow in the option cash flow matrix back to time zero, and averaging over all paths. This procedure results in a value of 0.1144 for the American put.

As the main operations in this step are matrix multiplication and inversion, the size of the matrix is very important to the overall architecture. If the number of simulation paths is 4,096, and the number of time steps is 100; the size of matrix X will be $4,096 \times 3$, and Y will be $4,096 \times 1$[6]. Figure 23 depicts the overall architecture of the regression step.

Our FPGA implementation targeted a Xilinx XC4VFX100-10 FPGA chip on an Alpha Data ADM-XRC-4FX card, which contains 42,176 slices, 160 DSP48s, and 376 BlockRAM units. We captured our hardware architectures in generic Verilog and synthesized them using Xilinx ISE 9.2i. To achieve the precision requirement of 10^{-4}, we needed 26 bits fixed point arithmetic for the Monte-Carlo simulation and 32 bits fixed point arithmetic for the regression part. The resource consumption breakdown is shown in Table 9. Moreover, 16 off-chip memory banks (4 physical

[6]Actually, the LSMC algorithm uses only the paths which are in the money. Hence, the number of row of X should be less than 4096. However, the memory size on FPGA is fixed, so we exclude the paths which are not in the money when doing the matrix multiplication. So the number of row of X can still be seemed as 4096.

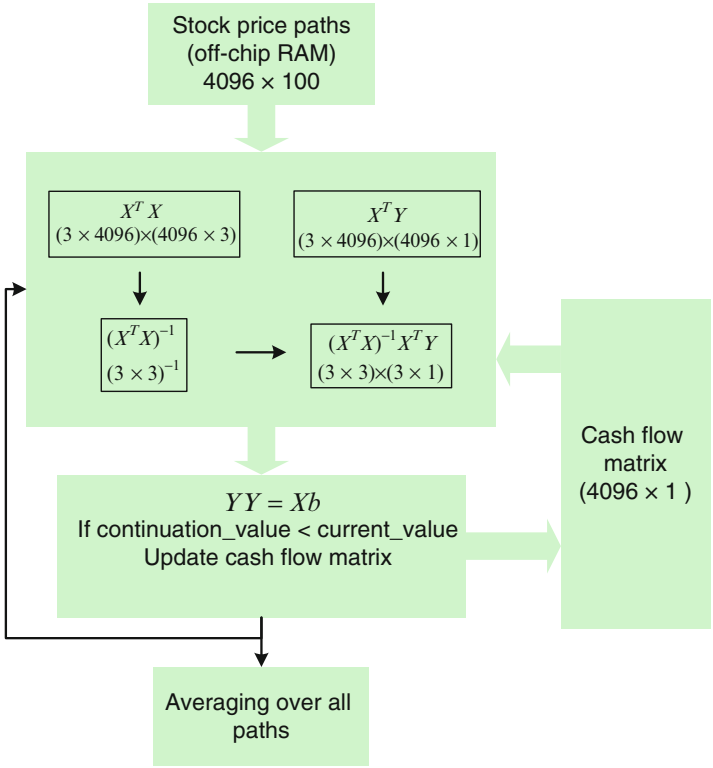


Fig. 23 Architecture of linear-squares regression

Table 9 Resource consumption breakdown

	Slices	FFs	LUTs	RAM	DSP48s
MC	2,655	2,996	3,656	24	44
Regression	37,667	18,385	66,384	53	116
Overall	40,322	21,381	70,040	77	160

banks are used, each bank has 52 bits word-length and we double the clock frequency to the memory) were used for storing the stock price paths, with each memory bank consisting of $512 \times 100 \times 26$ bits. The peak frequency achieved was 76 MHz, and the FPGA card was clocked at 75 MHz.

To compare the hardware implementation with an equivalent software implementation, we also wrote a C++ program for our Least-Squares Monte-Carlo engine and executed it on a 2.8 GHz Xeon processor-based machine with 1Gbyte memory. We used a fully optimized library, namely Intel Math Kernel Library (MKL), to generate the Sobol sequence and convert it to Gaussian noise using the ICDF method (also provided by the MKL). Single precision is used in the software implementation. The wall clock time of both FPGA and CPU-based implementations is shown in Table 10.

Table 10 Calculation time on FPGA and CPU (ms)

	FPGA	CPU
MC	0.683	16.778
Regression	1.368	24.608
Overall	2.051	41.386

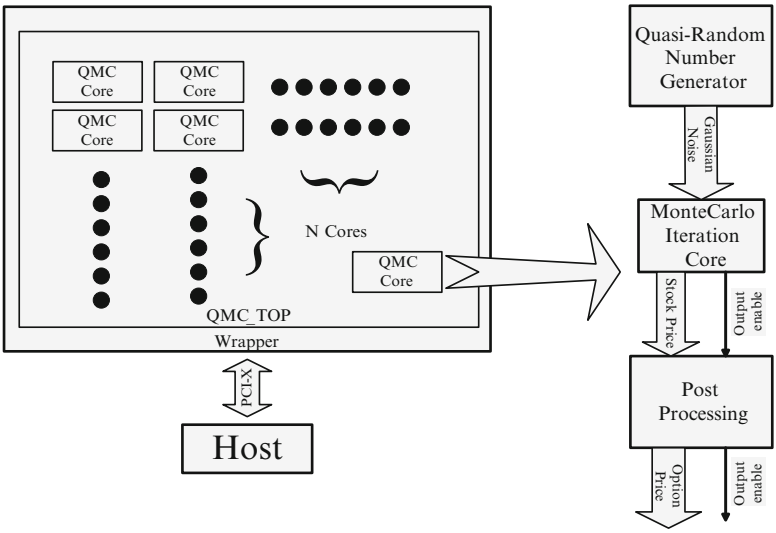


Fig. 24 Generic architecture of Quasi-Monte Carlo simulation engine

From Table 10, we can see that a 25x and 18x speed-up can be achieved in the Monte-Carlo simulation and regression steps, respectively, for an overall speed-up of the American option pricing calculation of 20x.

4.2.5 Quasi-Monte Carlo Simulation

The path generation part of the Quasi-Monte Carlo simulation core is the same as the one in the Monte-Carlo simulation core. However, the principle of the Quasi-Monte Carlo simulation is mainly based on the random number generation mechanism. Figure 24 gives the generic architecture of the Quasi-Monte Carlo simulation engine.

Hence, we plug a different random number generator (in Fig. 25) to the Monte Carlo simulation core. Since all of the modules in Fig. 24 have been described in the previous sections, we here only present the implementation results of the Quasi-Monte Carlo simulation core and the comparison with other implementations.

Our FPGA implementation targeted the XC4VFX100-10 FPGA chips on the Maxwell machine. We captured our hardware architectures in generic Verilog and synthesized them using Xilinx ISE 9.2i. We experimented with bit both fixed and floating point arithmetic as shown in Table 11, where the resources consumed by each module in a single Quasi-Monte Carlo computing core (excluding the PCI interface module) are given.

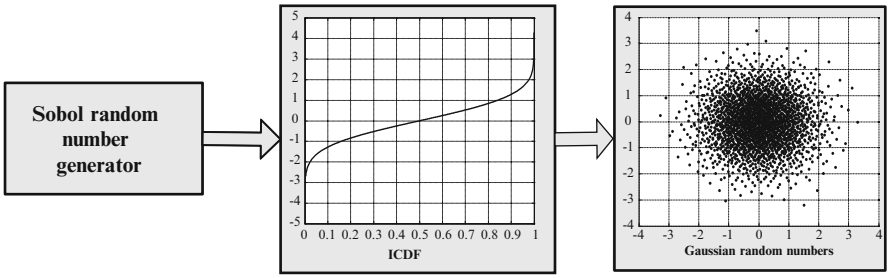


Fig. 25 Gaussian random number generator

Table 11 Resource consumption breakdown

	Fixed point						Floating point	
Arithmetic	26bits		29bits		32bits		Single precision	
Module name	RNG	MC	RNG	MC	RNG	MC	RNG	MC
Slices	944	614	945	675	951	777	951	2,790
	1,558		1,620		1,728		3,741	
DSP48s	1	8	1	9	1	10	1	13
	9		10		11		14	
RAM16s	6	0	6	0	6	0	6	0
	6		6		6		6	
LUTs	1,268	425	1,268	519	1,268	597	1,268	3,037
	1,693		1,787		1,865		4,305	
FFs	1,375	884	1,375	1087	1,375	1,273	1,375	4,392
	2,259		2,462		2,648		5,767	
Freq' (MHz)	211		211		194		180	
Precision	10-E4		10-E5		10-E6		10-E6	
Max No. of Cores	27		26		24		11	

Four RAM16s are used for storing 2,400 direction numbers and one is used for storing Sobol numbers from previous paths (in the Sobol number unit). We note that the precision of 32 bits fixed point implementation is the same as the single precision floating point one. This is largely due to the range of the input data. Another issue is that since we wanted to optimize the single precision floating point implementation, we used the most optimized pipeline stage for the floating point units, which results in a peak clock frequency of 180 MHz. The critical path for all the four implementation is the multiplier. The costs of addition, multiplication, and accumulation are 3, 3, and 1, respectively. In addition, the floating point arithmetic units are generated by Xilinx Core Generator.

Note that as we scaled the parameters of ICDF module, the range of Gaussian numbers in the 32 bits fixed-point implementation is the same as the single precision one. Hence we use the 32 bits fixed-point arithmetic for the RNG module in single precision implementation and converse the numbers to single precision before inputting to the MC module.

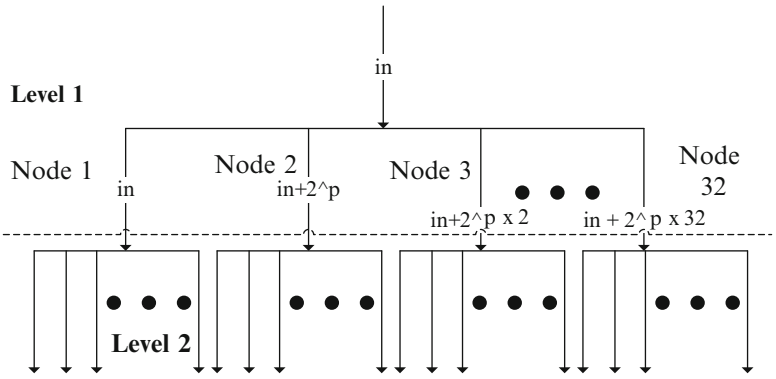


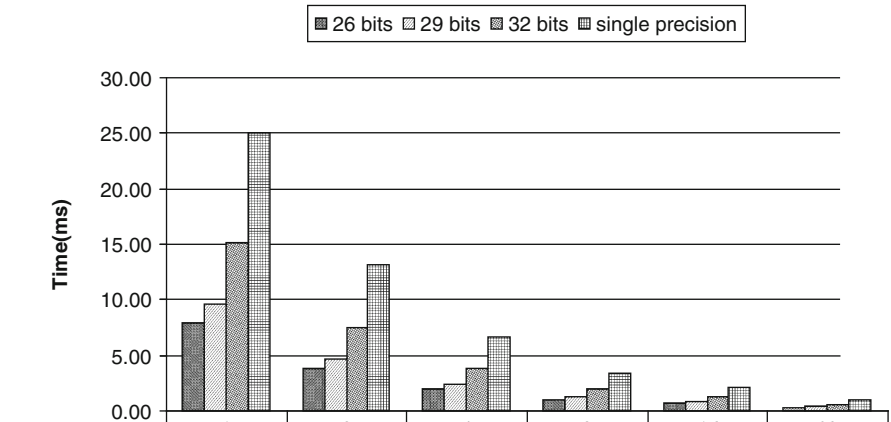
Fig. 26 Parallelism of Sobol sequence

Since we targeted a multi-FPGA platform, we use Message Passing Interface (MPI) [19] for process intercommunication (this is used for all the implementations in this chapter). Thirty-two FPGA nodes were used on the Maxwell machine, with each node loaded with the same bitstream and initial option price parameters, but with different initial numbers for the Sobol random number generator. Indeed, as explained above, we separate the Sobol sequence at two levels in order to allow for parallelism (see Fig. 26). As the initial number for each FPGA node is different, we calculate the initial number in one run of a recurrence and send it to the relative node using MPI. In each node, we calculate the initial number for each core using FPGA resources.

The pipelined design means that there is no need to store the random numbers as one Sobol number and one Gaussian variable are provided by the random number generator each cycle. Memory access time is hence reduced, and an option price estimate is generated every clock cycle in each core after the pipeline fills.

Figure 27 shows the execution time of our Quasi-Monte Carlo simulation engine on the Maxwell machine with different numbers of FPGA processing nodes and different arithmetic types, measured by the wall time function in MPI. As can be clearly seen, the execution time reduces linearly with the number of FPGA nodes used, which is to be expected since inter-process communication is negligible.

The second Quasi-Monte Carlo simulation engine was targeted at an NVIDIA 8800GTX GPU. This device has a core clock frequency of 575 MHz and a shader clock frequency of 1,350 MHz. The memory size is 768 MB, with 900 MHz clock frequency. We installed the newest Compute Unified Device Architecture (CUDA) 2.1 [32] development environment on a MacPro workstation with a 64-bit Linux system, and we implemented the exact same algorithm as in the FPGA implementation on the GPU. In the GPU, parallelism is mainly obtained through multi-threading. The thread hierarchy is as follows: the threads can be identified using a one-dimensional, two-dimensional, or three-dimensional index, forming a



	1	2	4	8	16	32
26 bits	7.87	3.87	1.98	1.03	0.64	0.31
29 bits	9.56	4.66	2.41	1.25	0.78	0.37
32 bits	15.15	7.48	3.81	1.98	1.24	0.59
single precision	25.00	13.12	6.67	3.45	2.17	1.03

Number of FPGAs

Fig. 27 Running time of Quasi-Monte Carlo simulation engine on different number of FPGA processing nodes

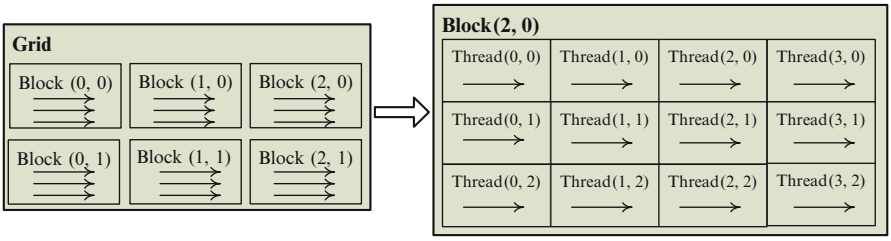


Fig. 28 Grid of thread blocks

one-dimensional, two-dimensional, or three-dimensional thread block, as illustrated in Fig. 28.

CUDA threads may access data from multiple memory spaces during their execution. Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory. We used multiple threads per option to keep the GPU hardware efficiently occupied. We also used multiple thread blocks per option, in which case we have to get partial sums from each thread blocks, which in turn means that data transaction from shared memory to global memory is needed. Hence, we use a second kernel which uses a parallel reduction operation to compute the sums. A parallel reduction is a tree-based summation of

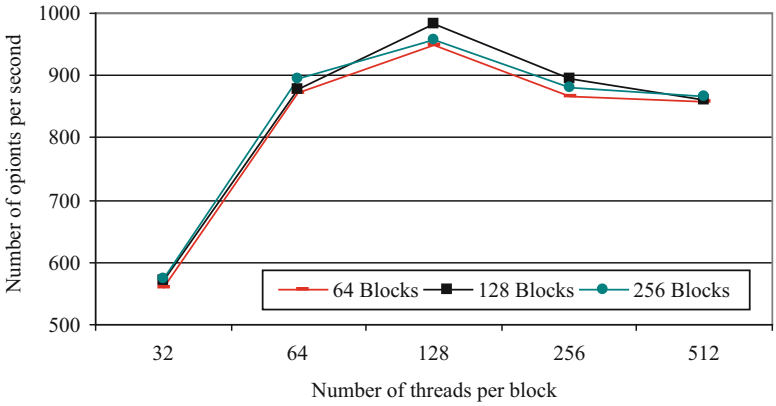


Fig. 29 Performance of our Quasi-Monte Carlo GPU implementation using different numbers of threads per block

values which takes $\log(n)$ parallel steps to sum n values. Parallel reduction is an efficient way to combine values on a data-parallel processor like a GPU; the larger the number of paths the better as this helps in hiding the latency of reading input values randomly.

Given a total number of threads per grid, the number of threads per block, or equivalently the number of blocks, should be chosen to maximize the utilization of the available computing resources. With a high enough number of blocks, the number of threads per block should be chosen as a multiple of 64, as the compiler and thread scheduler schedule the instructions as optimally as possible to avoid register memory bank conflicts, and the best results are achieved when the number of threads per block is a multiple of 64. After several experiments of multiple options pricing on our NVIDIA 8800GTX device, we found that using 128 thread blocks gives the best performance. Moreover, the performance is best when the number of threads per block is 128. Figure 29 shows the performance of our Quasi-Monte Carlo financial option pricing implementation when using different number of blocks and threads per block on the GPU.

We also wrote a C++ program of our Quasi-Monte Carlo simulation engine on the Maxwell machine and executed it on the Xeon processors. The most time consuming module of Quasi-Monte Carlo simulation core is the random number generator. Hence, we used a fully optimized library, namely Intel Math Kernel Library (MKL) [33, 34], to generate the Sobol sequence and transfer it to Gaussian noise using ICDF method (also provided by the MKL).

Figure 30 shows the execution time of our Quasi-Monte Carlo simulation engine on the Maxwell machine with different numbers of Xeon processors. As in the case of FPGAs, here also, the execution time scales linearly with the number of processors.

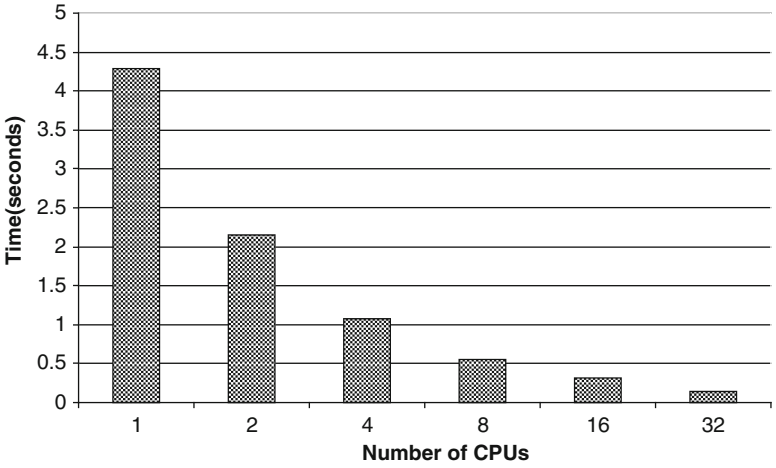


Fig. 30 Running time of our Quasi-Monte Carlo simulation engine on different number of Xeon processors

Table 12 Speed-ups of different platforms

	FPGA			GPU	
CPU	Fixed-26bits	Fixed-29bits	Fixed-32bits	Single Precision	Double Precision
1x	544x	448x	282x	162x	50x

To compare our three different implementation platforms, we ran our Quasi-Monte Carlo simulation engine to price a single option, using 524,288 simulation paths, on FPGA, GPU and GPP. This number of path is chosen as the precision can reach 10^{-4}. Moreover, it is a power of 2, which can benefit from the characteristic of Sobol numbers. We allot the same number of paths to each Quasi-Monte Carlo core on FPGA, Xeon CPU, or every thread and threads block on GPU. Although the optimized numbers of threads and threads blocks come from multiple options pricing, we still use the same parameters.

Table 12 shows comparative performance results between the FPGA, GPU and Xeon processor implementations, normalized to the Xeon CPU result. Here, the FPGA and Xeon implementations are both for a single node experiment, as we only use a workstation with a single GPU, and not a cluster of GPUs.

We can see significant speed-ups from both GPU and FPGA implementations. This is due to the high level of parallelism inherent in the Quasi-Monte Carlo simulation algorithm. Moreover, there is very limited conditional branching in the program, which is beneficial to both FPGAs and GPUs, especially the latter.

Apart from speed, we should also consider other factors when evaluating high performance computing implementations e.g. hardware and software cost, development time, power consumption, maintenance costs, technology maturity.

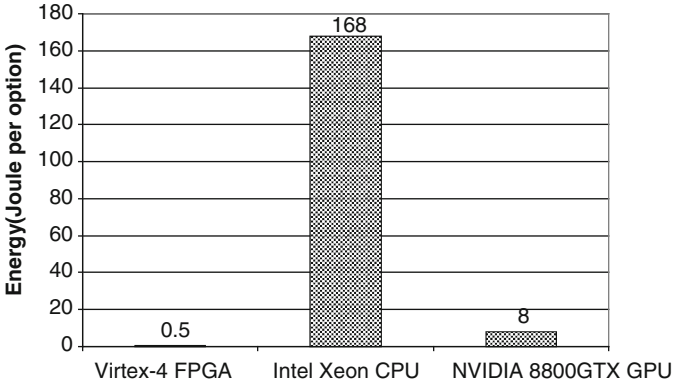


Fig. 31 Energy consumption in Joule

In the following, we will address one of these factors in our implementations, namely: power consumption.

We physically measured the power consumption of our FPGA, CPU and GPU implementations using a power meter, and deduced the energy consumption based on the execution times. Figure 31 gives the corresponding energy consumption results for each implementation. We can clearly see that FPGA offers the most energy efficient implementation, followed by the GPU, and then the CPU. Here FPGAs are 336x more energy efficient than CPUs, and 16x more energy efficient than GPUs.

5 Evaluation of Reconfigurable Hardware in High Performance Financial Computing

The evaluation of computational solutions in the literature is often focused on computation speed and accuracy. Nevertheless, when evaluating a high performance computing platform, several other metrics need to be considered. Those metrics include: the cost of equipment, development time, and power consumption. In fact, these metrics can all be valued in terms of only one metric: *Money*. Hence, to conclude this work in general, we consider the following aspects:

Equipment expense
Development expense
Energy expense

In this section, we perform a comprehensive comparison of the implementation of a Quasi-Monte Carlo simulation engine using three different computing platforms: FPGA, GPU, and GPP.

Table 13 Experimental parameters and results

	FPGA	GPU	GPP
Equipment cost	$10,000	$1,350	$1,000
Development time (days)	60	3	1
Development cost	$9600	$480	$160
Execution time	0.00787s	0.0858s	4.291s
Speed-up	545x	50x	1x
Dynamic power consumption	20W	95W	40W
Total power consumption	150W	225W	170W
Energy consumption	1.1805J	19.305J	729.47J
Annual energy cost	$197	$296	$223
Number of paths	524,288	524,288	524,288
Paths/second	66,618,551	6,110,583	122,183

5.1 Evaluation of FPGA-Based Monte Carlo Simulation Engine

In this comparison, we used the following device technologies: Xilinx Virtex 4 VFX100 FPGA, NVIDIA 8800GTX GPU, and Intel Xeon CPU 2.8GHz. Table 13 presents the experimental parameters and results. The equipment cost including both the expense for the host and the accelerator boards. The development cost is calculated using the following parameters: eight working hours per day and $20 per hour payment. The dynamic power consumption is the power measured at runtime, deducting the idle power. When calculating the energy consumed, we use the total power consumption. Annual energy costs are based on electricity price of $0.15 per kWh.

Several figures are plotted below to show the evaluation of the three implementations. The main metric used here is the number of paths per second, when normalized using development time, power consumption, and dollar expense. First, Fig. 32 shows the number of paths per second per development day using different platforms.

Results show that the GPU implementation gives the best result, followed by the FPGA implementation, and then the CPU one. Despite FPGA's high speed performance, its hardware description programming model is the most time consuming compared to GPU and CPU programming (which is essentially software programming), resulting in a lower performance per design effort compared to GPUs.

Considering power consumption, Fig. 33 presents the number of paths per second per Watt for each of the three implementations.

As we can see from Fig. 33, the normalized performance per Watt of the FPGA implementation outperforms the GPU and CPU implementations, respectively. Moreover, we note that although the power consumption of the GPU implementation is more than the CPU implementation, the former still beats the latter in terms of energy efficiency.

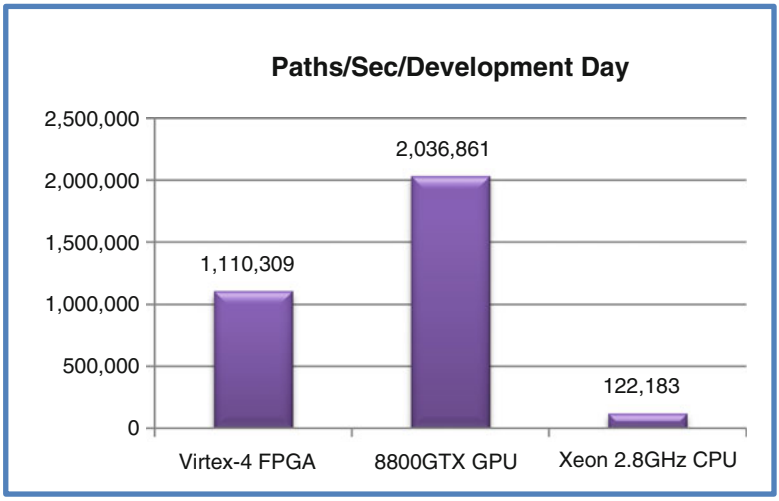


Fig. 32 Number of Paths/Sec/Development day

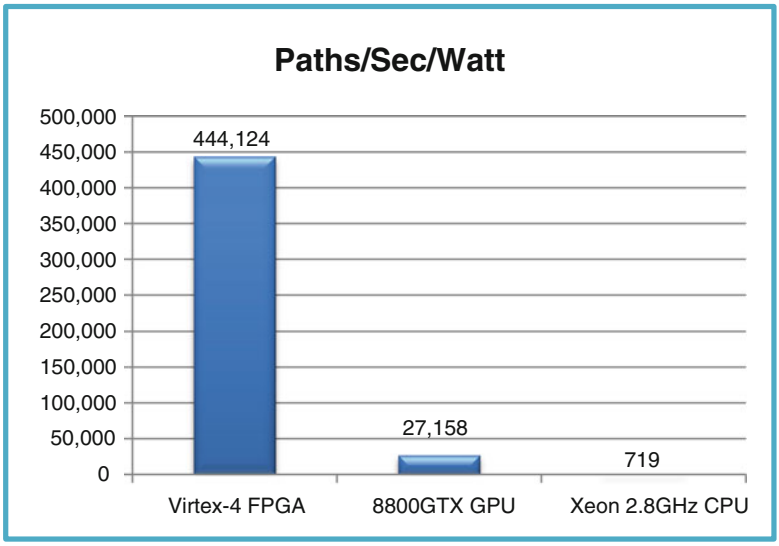


Fig. 33 Number of Paths/Sec/Watt

Considering the cost, Fig. 34 presents the performance normalized per total cost (purchase and development cost). As we can see, the normalized performance per cost of the FPGA and GPU implementations are very close. However, as in the above two figures, the normalized performance per cost of the CPU implementation is still much lower than the other two.

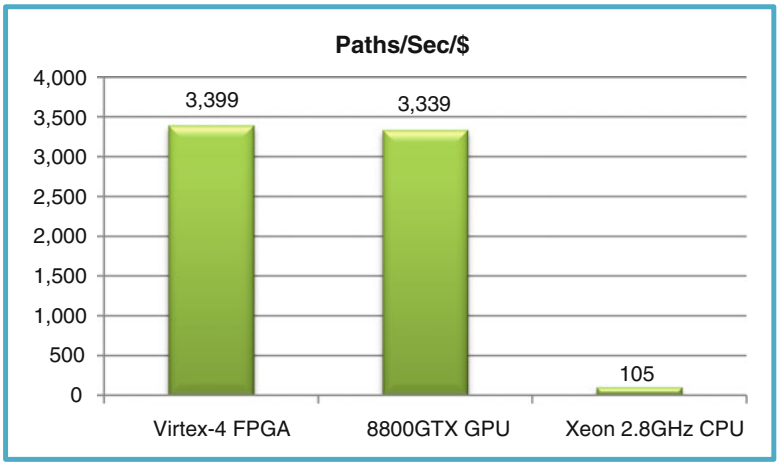


Fig. 34 Number of Paths/Sec/$ (purchase and development cost)

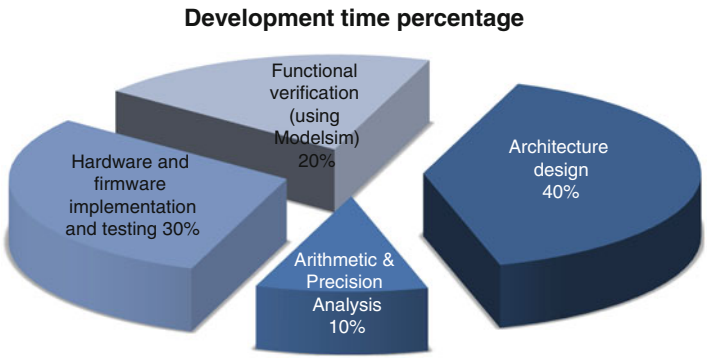


Fig. 35 Development time percentage on FPGAs

From the above results, we note that the main advantage of FPGAs resides in their energy efficiency. However, the development effort is a major drawback which impedes FPGAs' economic advantage. To have a deeper understanding of this issue, Fig. 35 presents a division of the whole FPGA solution's development time into its various steps.

From Fig. 35, we can see that the first three steps (namely arithmetic and precision analysis, architecture design, and verification) take 70% of the whole development time. This can be reduced in the future with high level design tools. Moreover, the hardware implementation and testing takes 30% of the whole development time. This could be reduced through the development of standard FPGA hardware boards with standard application programming interfaces (APIs).

6 Conclusion

Based on the work presented in this chapter, we can conclude that reconfigurable technology in the form of FPGAs has significant advantages compared to other technologies in high performance financial computing as it offers orders of magnitude speed-up compared to general purpose processors. Overall, FPGAs' main advantage lies in their high performance per watt, or energy efficiency. However, FPGAs' lack of high level programming tools and standard hardware and APIs is impeding the economic advantage of this technology, especially in comparison with GPU technology. Higher level programming tools and standard hardware and API platforms are necessary for further penetration of FPGA technology into high performance computing.

References

1. J. Dongarra, Trends in high performance computing: a historical overview and examination of future developments. IEEE Circ. Dev. Mag. 22, 22–27 (2006)
2. P. Marsh, High performance horizons. Comput. Contr. Eng. J. 42–48 (2004)
3. S.A. Zenios, High-performance computing in finance: the last 10 years and the next. Parallel Comput. 25, 2149–2175 (1999)
4. J.C. Hull, *Option, Futures, and Other Derivatives*, 4th edn. (Prentice Hall, Upper Saddle River, 2000)
5. G.L. Zhang, et al., Reconfigurable acceleration for Monte Carlo based financial simulation, in *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology*, 2005, pp. 215–222
6. D. B. Thomas, et al., Hardware architectures for Monte-Carlo based financial simulations, in *FPT 2006. IEEE International Conference on Field Programmable Technology*, 2006, pp. 377–380
7. G.W. Morris, M. Aubury, Design space exploration of the european option benchmark using hyperstreams, in *FPL 2007. International Conference on Field Programmable Logic and Applications*, 2007, pp. 5–10
8. Q. Jin, et al., Exploring reconfigurable architectures for binomial-tree pricing models. Lect. Note Comput. Sci. 245–255 (2008)
9. A.R. Choudhury, et al., Optimizations in financial engineering: The Least-Squares Monte Carlo method of Longstaff and Schwartz, in *IPDPS 2008. IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–11
10. I.L. Dalal, et al., Low discrepancy sequences for Monte Carlo simulations on reconfigurable platforms, in *International Conference on Application-Specific Systems, Architectures and Processors*, 2008, pp. 108–113
11. N.A. Woods, T. VanCourt, FPGA acceleration of quasi-Monte Carlo in finance, in *FPL 2008. International Conference on Field Programmable Logic and Applications*, 2008, pp. 335–340
12. J.H.C. Yeung, et al., Map-reduce as a programming model for custom computing machines, in *FCCM '08. 16th International Symposium on Field-Programmable Custom Computing Machines*, 2008, pp. 149–159
13. D.-U. Lee, et al., A hardware Gaussian noise generator using the Box-Muller method and its error analysis. IEEE Trans. Comput. **55**, 659–671 (2006)
14. D.-U. Lee, et al., A hardware Gaussian noise generator using the Wallace method, in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2005, pp. 911–920

15. D.-U. Lee, et al., A hardware Gaussian noise generator for channel code evaluation, in *FCCM 2003. 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003, pp. 69–78
16. D.-U. Lee, et al., A Gaussian noise generator for hardware-based simulations. IEEE Trans. Comput. 1523–1534 (2004)
17. R. Baxter, et al., Maxwell - a 64 FPGA Supercomputer, in *AHS 2007. Second NASA/ESA Conference on Adaptive Hardware and Systems*, 2007, pp. 287–294
18. R. Baxter, et al., The FPGA high-performance computing alliance parallel toolkit, in *AHS 2007. Second NASA/ESA Conference on Adaptive Hardware and Systems*, Edinburgh, 2007
19. M. Snir, S. Otto, *MPI-The Complete Reference*. (MIT Press, Cambridge, MA, 1998)
20. SUN. *Sun ONE Grid Engine Administration and User's Guide* [Online]. Available: http://www.sun.com
21. J.E. Gentle, *Random Number Generation and Monte Carlo Methods*, 2nd edn. (Springer, New York, 2003)
22. R.C. Tausworthe, Random numbers generated by linear recurrence modulo two. Math. Comput. **19**, 201–209 (1965)
23. M. Matsumoto, T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simulat. **8**, 3–30 (1998)
24. P. Bratley, et al., Implementation and tests of low-discrepancy sequences. ACM Trans. Model. Comput. Simulat. **2**, 195–213 (1992)
25. P. Jäckel, *Monte Carlo Methods in Finance* (Wiley, New York, 2002)
26. K. Entacher, et al., Defects in parallel Monte Carlo and quasi-Monte Carlo in tegration using the leap-frog technique. Int. J. Parallel Emergent Distributed **18**, 13–26 (2003)
27. G.E.P. Box, M.E. Muller, A note on the generation of random normal deviates. Ann. Math. Stat. **29**, 610–611 (1958)
28. O. Mencer, et al., Parameterized function evaluation for FPGAs, in *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, 2001, pp. 544–554
29. O. Mencer, W. Luk, Parameterized high throughput function evaluation for FPGAs. J. VLSI Signal Process. **36**, 17–25 (2004)
30. J.-C. Duan, The garch option pricing model. Math. Finance **5**, 13–32 (1995)
31. F.A. Longstaff, E.S. Schwartz, Valuing American options by simulation: a simple least-squares approach. Rev. Financ. Stud. **14**, 113–147 (2001)
32. Nvidia, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.0 edn. (NVIDIA Corporation, 2008)
33. Intel. Intel® Math Kernel Library Reference Manual [Online]
34. Intel. Intel® Math Kernel Library Vector Statistical Library Notes [Online]. Available: http://developer.intel.com/

http://www.springer.com/978-1-4614-1790-3

High-Performance Computing Using FPGAs
Vanderbauwhede, W.; Benkrid, K. (Eds.)
2013, XI, 803 p. 420 illus., 232 illus. in color., Hardcover
ISBN: 978-1-4614-1790-3