FPGA acceleration using High-Level Languages of a Monte-Carlo method for pricing complex options

Diego Sanchez-Roman, Victor Moreno*, Sergio Lopez-Buedo, Gustavo Sutter, Ivan Gonzalez, Francisco J. Gomez-Arribas, Javier Aracil

Escuela Politécnica Superior, Universidad Autónoma de Madrid
c/Francisco Tomás y Valiente, 11 28049 Madrid, Spain

Abstract

In this paper we present an FPGA implementation of a Monte-Carlo method for pricing Asian Options using Impulse C and floating-point arithmetic. In an Altera Stratix-V FPGA, a 149x speedup factor was obtained against an OpenMP-based solution in a 4-core Intel Core i7 processor. This speedup is comparable to that reported in the literature using a classic HDL-based methodology, but the development time is significantly reduced. Additionally, the use of a HLL-based methodology allowed us to implement a high-quality gaussian random number generator, which produces more precise results than those obtained with the simple generators usually present in HDL-based designs.

Keywords: High Level Language Synthesis, Field Programmable Gate Arrays, financial data processing, parallel machines, floating-point arithmetic.

1. Introduction

Options are derivative financial instruments which consist on a contract that gives the right to its holder to buy or sell some amount of the underlying

*Principal corresponding author

Email addresses: d.sanchez@uam.es (Diego Sanchez-Roman), victor.moreno@uam.es (Victor Moreno), sergio.lopez-buedo@uam.es (Sergio Lopez-Buedo), gustavo.sutter@uam.es (Gustavo Sutter), ivan.gonzalez@uam.es (Ivan Gonzalez), francisco.gomez@uam.es (Francisco J. Gomez-Arribas), javier.aracil@uam.es (Javier Aracil)

asset at a future date, for a pre-specified price. Options do not imply any obligation to buy or sell, they just grant the right to execute a transaction at an agreed price. Although contracts similar to options have existed since ancient times, the first time that contemporary financial options were traded in an orderly fashion was in 1973, with the opening of the CBOE in Chicago. Since then, option trading has grown tremendously, and nowadays options are traded on a wide range of assets: stocks, indexes, currencies, commodities, etc. An option giving the right to buy a certain asset is named a "call", while an option giving the right to sell a certain asset is named a "put".

Options are useful for hedging the risk. For example, if you know that you are going to buy oil in the future, you can buy call options for protecting yourself against a sudden rise in its price. If the price of oil does not rise, you simply let the option expire, but if there is an unexpected increase in its price, you can exercise the option to buy at a lower price. Alternatively, options can also be used for speculative purposes. If you suspect that there is going to be market crash soon, you can buy put options for a given equity. If the market crash finally occurs, you can buy that equity and then exercise the option to sell at a higher price. The buyer of the option pays an amount named premium to the originator of the option. Depending on the premium paid, it will be profitable or not to exercise the option at its expiration date.

Option valuation is a complex problem. There are many pricing models, which usually depend on the current price of the asset being negotiated, the price at which it will be traded (strike price), the cost of holding the asset (e.g. interests), the time to the expiration of the option, and volatility of the asset price during the life of the option. Depending on the style of the option, pricing may require complex numerical models. For example, in the case of Asian option pricing, an asset price is calculated in terms of the mean value during its lifetime. For American options, the asset price is calculated at its exercise point which, in contrast with the European option style, can be different to the option's expiration time. This way, Asian options are cheaper (as their potential benefits are lower) and less sensitive to market peaks and manipulation. However, the mean value feature of Asian option pricing has the problem of not having a closed-form solution for the pricing equation. In such cases, some approximation form [1] or simulation method must be applied, typically Monte-Carlo simulations. This papers focuses on the usage of Monte-Carlo simulations due to their ubiquity in the High-Performance Computing world. These Monte-Carlo simulations are based on the computation of random experiments following a determined probabilistic

distribution. However, the low convergence rate of such methods creates the need of running huge number of path simulations in order to converge to the proper value. This is translated into simulations taking enormous amounts of time.

In those cases were pricing is based on complex models, there is a competitive advantage for those traders which can compute the models faster. The sooner that you can realize that the price of an option is favorable, the less possibilities exist that other trader buys the option and profits from it. However, not only the speed in getting the estimated price is relevant, but the accuracy of the estimation is also of capital relevance to decide if the price of an option is favorable. The ubiquity of high-performance computing systems has made difficult to stand out in the financial market: Nowadays anyone can use a powerful computer in order to calculate an asset price before his competitors. It is therefore necessary to find out new technologies which allow traders to significantly accelerate simulations in order to provide a relevant advantage to the competition.

FPGAs have already been proved as good candidates for algorithm acceleration since they allow engineers to design custom architectures optimized to solve a specific problem by taking advantage of much higher degrees of parallelism than CPUs. One of the main drawbacks in hardware development is the use of hardware description languages (HDLs), whose utilization entails high development times and painful verification processes. In order to ease this process, in the recent years we have seen the advent of high level languages (HLL) compilers capable of generating HDL code from high level languages such as C or derivates. In this work, we use the Impulse C tool to generate a hardware description of a highly accurate Monte-Carlo method for pricing Asian Options and compare it against a multicore solution based on the OpenMP programming framework. It is shown that a HLL-based methodology is not only capable of providing similar accelerations to those of a HDL-based design, but also with a better accuracy, since more precise algorithms can be implemented.

The remainder of the paper is organized as follows. In section 2 an overview of the Asian option model is presented, as well as the mathematical mechanisms used in this paper to simulate this model. A keynote on the High-Level Language used during the implementation phase (Impulse C) is pictured in section 3. The implementation procedure is depicted along section 4 and its results are presented at section 5, compared to a multicore CPU solution. Finally, some conclusions are drawn in section 6.

2. Asian options and Monte-Carlo simulations

2.1. Asian options

As it has been set along the introduction of this paper, options are contracts between two parts regarding some asset or stock price. The price in the contract is known as the exercise price or strike price (E), and the date in the contract is known as the expiration date or maturity (T). When acquiring the option, the "owner to be" must pay to the option issuer the fixed price for the option to be created. The calculation of the price usually depends on the desired interest rate r and the expiration time of the option as shown in Eq.(1).

$$option_{value} = e^{-rT} option_{payoff} \tag{1}$$

Once the option has been created, the option issuer must accept the decision of the option owner regarding the execution or not of the call/put operation. Under those terms the acquiring price of an option must be carefully calculated, therefore both the accuracy and the latency of this calculation become of critical relevance.

Regarding the time when the option can be exercised, we can distinguish between American or European options. The former ones allow the holder to exercise the option at any time up to the expiration date. However, European options can only be exercised at maturity. Traditional Asian options behave like the European ones regarding the moment they can be exercised at.

The *payoff* of an option is the quantity used for the option exercise price calculation process. The payoff of a call option is then the difference between the price of the underlying asset at the exercise time ($Price(T)$) and the exercise price for both American and European options. If this difference is lower than zero then the payoff will be zero because the option will never be exercised (see Eq.(2)). Similarly, the payoff of a put American or European option is the difference between the exercise price and the asset price at the exercise time unless this difference is lower than zero (see Eq.(3)).

$$P_{call} = max(Price(T) - E, 0) \tag{2}$$

$$P_{put} = max(E - Price(T), 0) \tag{3}$$

Asian options come to remove the risk of market manipulation, since the strike price is computed as the mean price of the underlying asset over the

life of the option [2] as shown in Eq.(4) and Eq.(5).

$$P_{call} = max(\frac{1}{n}\sum_{i=1}^{n} Price(t_i) - E, 0) \tag{4}$$

$$P_{put} = max(E - \frac{1}{n}\sum_{i=1}^{n} Price(t_i), 0) \tag{5}$$

Modifications of the basic Asian option model have been formulated, such as the American Asian option model [3]. This model allows the option to be exercised any time during its lifetime, keeping the payoff calculation using the mean price of the asset. However, this model is not widely extended, so we focus our work in the traditional Asian option model.

In year 1973 Fischer Black and Myron Scholes articulated a mathematical model capable of capturing the behaviour of a stock price movement [4]. This model, known as the Black Schole's model, says that the price of a stock over time is governed by Eq.(6):

$$S_{i+1} = S_i e^{((r-\frac{v^2}{2})\Delta t + v\sqrt{\Delta t}W)} \tag{6}$$

were $S_i = S(t_i)$ is the price of the asset at time step t_i, r is the interest rate, v is the volatility of the underlying stock price, Δt is the time between two time steps and W is a Gaussian random number (GRN).

2.2. Monte-Carlo simulations

Monte-Carlo computations need to evaluate a large number of independent paths in order to converge to the solution. In our case this is translated into the need of a huge amount of Gaussian independent and identically distributed samples. When working with stock prices the quality of the Gaussian samples used becomes a critical matter: the very tiniest fraction of money is relevant. The more pseudo-random the Gaussian random number generator is, the more biased the simulation results will be, which could lead to relevant economic losses [5].

Most common methods for non-uniform RN generation are based on a first stage in which $\mathcal{U}[0, 1]$ samples are generated [6]. Let u_i be a sample

of the $\mathcal{U}[0,1]$ distribution, X the target distribution and $ICDF$ its Inverse Cumulative Distribution Function. The inversion method states that

$$z_i = ICDF(u_i)$$

is a sample of X. Even direct methods which claim to directly generate non-uniform RN such as [7],[8] use a uniform RN generator in their core.

Along the rest of this section, the methodology used to generate high-quality GRN will be described, separated in two steps: generating uniform samples and using these uniform samples to generate Gaussian ones. At the end of the section our approach will be placed in contrast with the previously developed works on this matter.

2.2.1. Mersenne Twister

As it has already been stated, a high quality uniform pseudo-random generator is required. There are several key properties to measure the goodness of such generator [9, 10]:

- its period p. This is a determining parameter not only in order to avoid the repetition of pseudo-random sequences, but also in order to be able to run larger simulations. As explained in [10], different authors sustain that the number of values to be used in an experiment using a pseudo-random generator with period p must be much lower than p itself: some say that $p/100$ is enough, but others defend that even lower amounts of numbers must be used, such as $\frac{\sqrt{p}}{200}$ or even $\sqrt[3]{p}$.
- its k-dimensional equidistributed property[1]

However, when working with hardware devices, there is another key factor that must be taken into account when using a RNG:

- the amount of area (in terms of memory and other silicon resources) required by the RNG. There are several RNG with have a very large

[1] A pseudorandom sequence x_i of w-bit integers of period P is said to be k-distributed to v-bit (v *lew*) accuracy if for each subsequence

$$(trunc_v(x_i), trunc_v(x_{i+1}), \cdots, trunc_v(x_{i+k-1}))\ \ 0 \leq i \leq P$$

each of the 2^{kv} possible combinations of bits occurs the same number of times in a period, except for the all-zero combination that occurs once less often.

period due to the use of a bigger amount of memory keeping the RNG state information.

Table 1 shows some of the features for several RNG that have been evaluated during the development of this work. This table shows as well the results obtained when applying a suite of difficult-to-pass uniformity tests [11] to such generators. Those tests has been run with more than $5.6\dot{1}0^9$ values generated for each RNG. Mersenne Twister was chosen amongst the considered RNGs due to their good results and its period-memory tradeoff.

Table 1: Properties of different RNGs

RNG	memory (32b words)	$log_2(Period)$	% of tests passed	Refs.
taus88	3	88	96.49	[12]
mcw256	256	8222	97.37	[10]
lr	345	> 345	97.37	[13]
mt19937	624	19937	99.12	[9]
sobol (40-dim)	1280	32	98.25	[14, 15, 16]
cmwc4096	4096	131086	97.37	[10]
superKiss	41790	1337279	99.12	[10]

The Mersenne Twister (MT) is a high quality uniform random number generator with a period of $2^{19937} - 1$ and a 623-dimensional equidistributed property. The original C source code can be found in [9] and its CPU execution is highly efficient, both in terms of memory and performance. A matrix of 624 32-bit integers is initialized in a first stage. After that, one value of the matrix is returned with each call to `genrand()` through some bit manipulation process known as *tempering*. Once all the values in the matrix have been used, it is reconstructed by shuffling and masking its bits.

Although the Mersenne-Twister code behaves properly on CPU computing, the lack of constant throughput (because of shuffling and masking the generation matrix) of this approach turns it into a quite pipeline-unfriendly code. It is important to remark that we have decided to use such a pipeline-unfriendly approach for generating random uniform samples in contrast with other HW-friendlier existing approaches [17]. We have made this decision because accuracy has been considered a much more critical issue rather than

throughput for our Asian option simulation solution. We consider that our approach presents a more appropriate accuracy-throughput tradeoff than other existing solutions such as [18], [19] or [20]. Moreover, previous works such as [21] shows that despite its pipeline-unfriendliness, MT can be highly benefited using FPGA technology.

2.2.2. Box-Muller transform

For our Asian option pricing problem we are interested in generating Gaussian samples, whose CDF is given by Eq.(7).

$$CDF(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt \tag{7}$$

As there is no closed form for its $ICDF$ function, the generic inverse method can not be applied. Some approaches have been developed in previous works such as [19] or [20] in which a approximation to the $ICDF$ function is used. A tradeoff must be met between the accuracy requirements and the complex expressions that need to be evaluated in order to meet these accuracy constraints. These expressions require a substantial amount of floating-point operations so their implementation would severely restrict the number of Monte-Carlo cores that could be mapped inside a FPGA.

Fortunately, there is another method for generating Gaussian samples from uniform samples, which is the Box-Muller transformation [22]. It requires less floating-point arithmetic, and its good quality has already been proved [23].

The Box-Muller transformation produces two independent Gaussian samples from two independent uniform ones. Let u_1 and u_2 be two independent samples of the $\mathcal{U}[0,1]$ distribution, then

$$z_0 = \sqrt{-2\ln u_1}\cos(2\pi u_2)$$
$$z_1 = \sqrt{-2\ln u_1}\sin(2\pi u_2)$$

are two independent Gaussian samples $\sim \mathcal{N}(0,1)$.

The implementation of this transform with Impulse C is trivial in a pipelined process, generating one sample per clock cycle. The only issue is the area required by the transcendental and trigonometric floating-point operators from the Altera Megafunctions [24], which limits the number of Monte-Carlo cores that can be implemented in the FPGA. In order to slightly

reduce this area utilization, the identity $sin^2\alpha + cos^2\alpha = 1$ is utilized, saving one trigonometric evaluation at the cost of adding one square root and one multiplication.

2.3. Related work

As for other financial problems, there has been a lot of effort made in the resolution of Asian option pricing. Other approaches rather than Monte-Carlo methods have been researched [25], but the parallelizability of Monte-Carlo simulation has made this approach to be much widely extended in the High-Performance Computing world.

However, the use of Monte-Carlo simulation methods for this calculation makes this task a heavy computational one, so many high performance computing approaches has been developed to solve this problem. Some of those solutions are based on message passing programming [26] [27] for high performance clusters. Nevertheless, the most notable efforts have been made in the field of reconfigurable/reprogrammable coprocessor computing, both based on FPGA [18] [19] [28] and GP-GPU [19] [29] technologies.

All those works have focused their efforts in creating a high-performance simulation solution, giving priority to the throughput of the system rather than to its accuracy. In this work, we have tried to create an as accurate as possible solution for the Asian option pricing problem, keeping in mind the relevance of the timing behaviour of the system.

Table 2 summarizes the performance results shown in the mentioned related works, as well as the technology approach used for each solution and their performance reference.

3. High Level Languages

The use of High Level Languages has a set of advantages against the use of low level programming languages. The most indisputable of those advantages is the decrease of the development time. Thanks to that, a fast evaluation of the feasibility of a project via prototyping becomes a reality. As previous works has shown [30], [31], [32] this effect becomes more evident when it comes to programming FPGA devices in contrast with hardware description languages (VHDL, Verilog).

Of course, HLL-based FPGA design leads to less area-efficient design in contrast with HDL-based design. A tradeoff must be met regarding the development time and the area efficiency constraints for each design.

Table 2: Related work technology and performance features

Ref.	Year	Technology	RNG		Speedup	
			Uniform	Gaussian		
[19]	2011	Xilinx Virtex-5	lr	$ICDF$ piecewise interpolation	313.00	vs Intel Xeon 2.5 GHz
		Nvidia Tesla C1060	mt19937	Box-Muller	141.00	vs Intel Xeon 2.5 GHz
[26]	2011	2x Intel dual-core 1.6 GHz	lr	Box-Muller	2.13	vs Intel dual-core 1.6 GHz
[29]	2010	Nvidia Quadro FX4600	sobol	$ICDF$ approximation	150.00	vs Quad-Core Xeon
[28]	2008	64x Xilinx Virtex-4	taus88	Box-Muller	340.00	vs Intel Xeon 2.8 GHz

3.1. Impulse C

The development of this work has been made using the Impulse C programming framework [33]. The choice of this language has been made because of its outstanding usability compared to other high-level approaches for FPGA programming, as stated in [30]. Moreover Impulse C potential has been shown in previous works such as [34], [35] and [36].

Impulse C extends standard ANSI-C using C-compatible predefined library functions in support of a communicating process parallel programming model. This programming model is conceptually similar to a dataflow or a communicating sequential process programming model in the sense that it simplifies the expression of highly parallel algorithms through the use of well-defined data communication, message passing, and synchronization mechanisms. The programming model supports a wide range of applications and parallel process topologies.

In Impulse C, the programming model emphasizes the use of buffered data streams as the primary method of communication between independently synchronized processes, which are implemented as persistent (rather than being repetitively called) C subroutines. This buffering of data, which is implemented using FIFOs that are specified and configured by the application programmer, makes it possible to write parallel applications at a higher level of abstraction, without the clock cycle-by-cycle synchronization that would otherwise be required .

Programming with Impulse C processes is conceptually similar to programming with threads. As with thread programming, each Impulse C process has its own control flow, it is independently synchronized and it has access to its own local memory resources (which will vary depending on the target platform). For this reason it is relatively easy to convert applications written in threaded C (for example, using the Posix thread library [37]) to Impulse C. Thus, the coarse parallelism in Impulse C is simply achieved by coding multiple concurrent processes. Synchronization among these concurrent computational kernels is easily achieved by the stream dataflow [38].

Finer levels of parallelism can be exploited inside each process. Instruction level parallelism is automatically generated by the scheduler. Impulse C automatically generates and analyses the instruction dependence graph so that independent instructions are scheduled in parallel. In the case of loops, further level of parallelism can be achieved by means of precompiler directives or pragmas. They are loop unrolling and loop pipelining, which

are specified by placing `#pragma CO UNROLL` and `#pragma CO PIPELINE` just after the header of a loop.

An additional feature of the Impulse C framework is the ability to easily modify the IP cores to be instantiated whenever a *hard-function* or an arithmetic operator appears in the C source code. This is made by editing a set of XML files attached to the target platform. The Impulse C vendor refers to this feature as Platform Support Package (PSP). The ease-of-use of the PSP turns the Impulse C framework into an extremely interesting option when working with custom cores, so anyone can create their own optimized HDL-based core and use C code to instantiate it along the design. The use of PSPs strengthens as well the portability of a project between different platforms.

4. Implementation

4.1. Option price calculation

The Monte-Carlo algorithm for pricing Asian Options is depicted in Listing 1. For each simulation path, stock prices are updated by using Equation (6) and accumulated in order to compute the mean value until the maturity time T. The payoff is evaluated as the profit involved in the transaction. Again, values are accumulated in order to obtain the mean value. Finally, the price is adjusted by the fixed interest rate. The architecture of the solution is shown in Fig. 1 where each of the functional modules are identified, as well as the FIFO channels that Impulse C instantiates for inter-module communication.

In order to accelerate this algorithm with Impulse C, loop pipelining must be used. This is trivially achieved by simply inserting `#pragma CO PIPELINE` after the head of the body. However, Impulse C does not allow pipeline nesting and the two loops should be flattened into a single loop executing $nSim * nSteps$ iterations. However, there is loop dependence between iterations which prevents a pipeline with rate 1, i.e, a pipeline computing iterations every clock cycle. To avoid the loop dependence, we must take advantage of the independence of the stock prices among different simulations, so that `PARALLEL_PATHS` simulations are concurrently computed.

To this end we split the problem in two different Impulse C processes. The first one computes the stock prices and performs the arithmetic mean. It also computes the payoff, which is written to the second process. Listing 2 depicts the Impulse C pseudo-code working at rate 1. There are two different

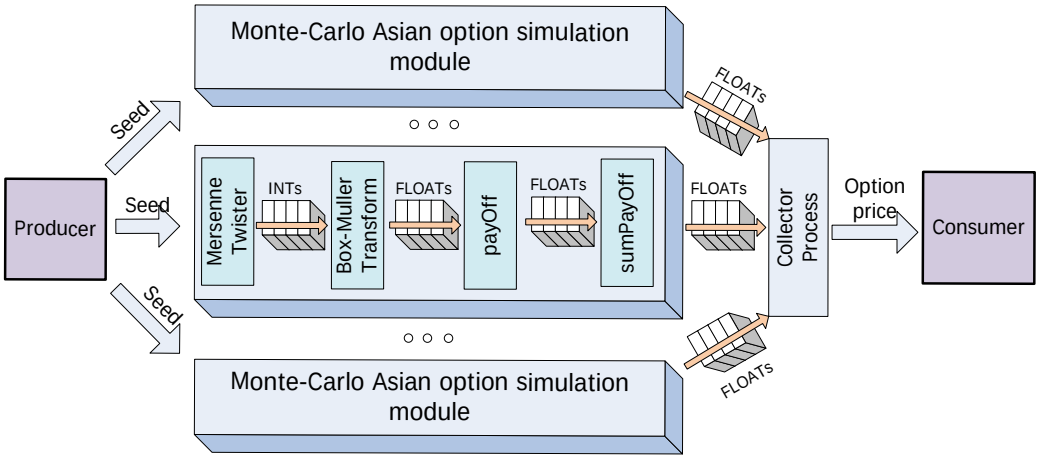

Figure 1: Architecture of the FPGA-based solution

arrays of `PARALLEL_PATHS` elements which store the stock and the addition until the i-th step, respectively. We use the `CO NONRECURSIVE` pragma to tell Impulse C that there are no data dependences regarding the array access, so that waits are not inserted. However, we must set `PARALLEL_PATHS` big enough so that the previous assertion is actually true. The optimum value for `PARALLEL_PATHS` would be the latency of the critical code in between the read from and the write to the memories. As `PARALLEL_PATHS` is going to be small and it does not affect performance, it is easier to simply use the total routine latency (85) as the lower bound, since it is automatically reported by Impulse C. Therefore, with the rate 1 pipeline, all the payoffs are computed after $nSteps * nSims$ plus the pipeline latency, which is despicable.

Each of these computed payoffs must be added, that is the function of the second Impulse C processes. A naive implementation would be the one in Listing 3. Here again there is the problem of loop dependence and the pipeline can only perform at a rate equal to the latency of the addition operator. A simple trick can be used to avoid this inconvenient. The accumulation is performed in different positions of an array with a size `BUFF_SIZE` greater or equal to the addition latency, as shown in Listing 4. This way, the additions can be performed with rate 1 plus a negligible penalization to add the final values stored in the array. As before, we must only ensure that `BUFF_SIZE` is bigger than the pipeline latency so that it is true that there is no dependence between all the stages in the pipeline.

Therefore the extra time needed for adding the $nSims$ values would be

Listing 1: Monte-Carlo algorithm for pricing Asian Options

```
1  payOffSum = 0;
2  for (i = 0; i < nSims; i++){
3          priceSum = S_0;
4          for (j = 0; j < nSteps; j++){
5                  W = getGaussian();
6                  S = S*exp((r - v^2/2)Δt + v*W*sqrt(ΔT));
7                  priceSum += S;
8          }
9          payOff = max(0, priceSum /(nSteps+1) - K);
10         payOffSum += payOff;
11 }
12 price = exp(-r*T)*payOffSum/nSims;
```

approximately of $nSims + l_a + \texttt{BUFF\_SIZE} \cdot l_a \approx nSims$ clock cycles, where l_a is the addition latency. However, the overall computation time of our Monte-Carlo core is essentially of $nSteps * nSims$ cycles, since there is overlapping between the payoff computation and the sum reduction.

4.2. Random number generation

The traditional Mersenne-Twister CPU code is not HW-friendly because it lacks a constant throughput due to the shuffling and masking stages made once all of the values in the matrix have been used. To overcome this limitation we have rewritten the algorithm so that once the value of the matrix is used, it is replaced by the new one. Moreover, the MT algorithm reads two data and writes one for each random number generation. Since the matrix is mapped into a BlockRAM, whose simultaneous access is limited to a maximum of two ports, we need two different matrices to achieve a one sample per clock cycle throughput. After initialization, the matrix A takes the read-only role. Its values are tempered and returned as the uniform samples, one each clock cycle. At the same time, the matrix B is filled with a new Mersenne Matrix. Once all the values in the matrix A have been used, the roles are interchanged so that B becomes read-only.

A Mersenne Twister core with a throughput of 623 samples per clock cycle has also been developed by implementing the Mersenne matrix with registers. This is somehow a tedious task when working with Impulse C,

Listing 2: Impulse C pseudo-code for the payoff computation in a pipeline with rate 1

```
1  for (i = 0; i < nSteps*nSims; i++){
2          #pragma CO PIPELINE
3          #pragma CO NONRECURSIVE lastStockPrice
4          #pragma CO NONRECURSIVE lastStockSum
5
6          co_stream_read(sGauss, &W, sizeof(float));
7
8          if (initStage(i)){
9                  S = S_0;
10                 priceSum = S_0;
11         }else{ //Acumulation stage
12                 S = lastStockPrice[i % PARALLEL_PATHS];
13                 priceSum = lastStockSum[i %
                      PARALLEL_PATHS];
14         }
15         S = updateStock(W, S);
16         priceSum += S;
17         lastStockSum[i % PARALLEL_PATHS] = priceSum;
18         lastStockPrice[i % PARALLEL_PATHS] = S;
19
20         if (outputStage(i)){
21                 payOff = computePayOff(priceSum, K);
22                 co_stream_write(sPayOff,&payOff,
                              sizeof(float));
23         }
24 }
```

Listing 3: Impulse C pseudo-code for the payoff accumulation in a pipeline with a rate equal to the latency of the addition operation (7 cycles for the lowest latency Altera Megafunction)

```
1 payOffSum = 0;
2 for (i = 0; i < nSims; i++){
3         #pragma CO PIPELINE
4
5         co_stream_read(sPayoffIn, &payOff,
                  sizeof(float));
6         payOffSum += payOff;
7
8 }
```

as there is no way to tell the tool that we want a matrix inferred this way. Instead, we declared each element of the matrix as a different 32-bit unsigned integer variable and we generated the Impulse C code with the help of a python script. However, the area required for this implementation was not worth the effort and it is preferable to instantiate several BlockRAM based Mersenne cores if more throughput is required.

Once one have been able to generate independent uniform random number using our Mersenne-Twister implementation, applying the Box-Muller transform to obtain Gaussian samples is a completely straightforward procedure using Impulse C.

Note that in order to get unbiased simulation results, some care must be taken with regard to the choice of the random seeds that will feed the different Monte-Carlo simulation cores so the pseudo-random number sequences generated by each simulation path are independent. Consequently, the seeding must be made taking into account the principles detailed in [39, 40]. However, this seeding process must only be done once for each simulation, so it can be done in software and the different seeds loaded into the FPGA before starting the execution.

4.3. Floating-point arithmetic cores

Hardware acceleration requires high throughput, and pipelining is a must. However, most pipelined intellectual property (IP) cores focus on working at high clock frequencies, what in the end increases register utilization too much,

Listing 4: Impulse C pseudo-code for the payoff accumulation in a pipeline with rate 1

```
1  payOffSum = 0;
2  for (i = 0; i < nSims; i++){
3          #pragma CO PIPELINE
4          #pragma CO NONRECURSIVE sumsBuffer
5
6          co_stream_read(sPayoffIn, &payOff,
                          sizeof(float));
7
8          if (initStage(i))
9                  payOffSum = 0;
10         }else{
11                 payOffSum = sumsBuffer[i % BUFF_SIZE];
12         }
13
14         payOffSum += payOff;
15         sumsBuffer[i % BUFF_SIZE] = payOffSum;
16 }
17
18 //Final addition
19 payOffSum = 0;
20 for (i = 0; i < BUFF_SIZE; i++){
21         payOffSum += sumsBuffer[i];
22 }
```

not only because of the internal registers of each core, but also because the registers needed to synchronize the signals in the pipeline.

Additionally, the high clock frequencies that these individual cores achieve are rarely reached in a whole design, due to I/O restrictions or routing restrictions. Therefore, the use of low-latency operators either reduces the utilization of FPGA resources by allowing the implementation of more complex algorithms or improves performance by allowing further parallelism by kernel replication.

Our arithmetic addition/subtraction and multiplication cores are fully compliant with the IEEE 754-2008 32-bit (single-precision) standard except for signaling NaNs and subnormal numbers. The former are only useful when collaborating with a microprocessor that handles exceptions, which is not typically the case in reconfigurable solutions. The latter are rarely implemented in FPGAs since the area and complexity involved in their implementation are not justified. By simply adding one more bit to the exponent, the numeric range outperforms by far that achieved by subnormals. Our single-precision arithmetic cores have been developed with the purpose of working at a moderate frequency of $\sim 100MHz$ at the lowest latency possible. Their area/timing characteristics are depicted in Table 3 in contrast with the standard Altera floating point cores [24].

Table 3: Area and performance for Stratix 5SGSMD8N3F45C2

Operation	Core	Latency	ALUTs	Registers	DSPs	Mem bits	F_{max} (Mhz)
Add/Sub	Altera	7	581	373	0	0	284.5
	Own	2	526	105	0	0	167.3
Mul	Altera	5	150	190	1	0	308.2
	Own	1	154	147	1	0	287.8
Div	Altera	6	189	245	5	4608	195.2
		14	200	702	5	4742	285.7
	Own	7	1377	540	0	0	136.2
Sqrt	Altera	16	436	538	0	121	233.0
	Own	7	435	298	0	0	132.4
Available			532720	1065440	1755	35942400	

The development of those floating-point cores has been completely made using HDL-based design. However, the use of the Impulse C framework

allows to instantiate our cores whenever a floating-point arithmetic operation appears in the C source code. To that purpose, only some modifications must me made in the PSP files (which follow a XML syntax) corresponding to the target platform.

5. Results

The area utilization of the implementation discussed for a single computational kernel is shown in Table 4. These results are obtained with Quartus II 12.2 after Place and Route. The floating-point operators have been generated with the Megawizard Plug-in Manager from Altera. These operators are seamlessly integrated within Impulse C by editing some XML files. As it can be seen, the area bottleneck is found in the Box-Muller component, because of the area used by the trigonometric and logarithmic operators [24]. In the Stratix V 5SGSMD8, a total number of 40 computational Monte-Carlo kernels are instantiated. The outputs of these individual components are averaged by a collector Impulse C process which then performs the interest rate update in the price of the underlying asset.

Table 4: Area utilization for Stratix 5SGSMD8N3F45C2

Core	ALUTs	Registers	DSPs	Mem bits	F_{max} (Mhz)
Mersenne	744	867	2	40666	269.69
Box-Muller	8803	8155	22	1698	172.63
PayOffs	1242	1375	11	2268	206.48
SumPayOffs	1242	510	0	2498	156.48
1 Monte-Carlo core	9855	9409	29	41227	154.9
	1.9%	0.9%	1.5%	0.1%	
40 Monte-Carlo cores	396729	378509	1166	1682900	142.3
	75.6%	36%	59.4%	3.2%	
Available	532720	1065440	1755	35942400	

Performance results are depicted in Table 5 compared to several CPU implementations (for fairness reasons, the CPU version uses single-point floating operations). The CPU code is executed on Ubuntu 10.10 64 bit and the source is compiled with `gcc 4.4.5`, which allows the use of the OpenMP multicore programming framework [41], using the optimization flags `-O3 and`

`-ffast-math`. By using the OpenMP framework, we allow the program to run through several computational cores just by placing the `OMP PARALLEL FOR` pragma before the outer loop of the Monte-Carlo simulation code. The CPU hardware consists of an Intel i7 860 CPU which runs at 2.8 GHz and it has 8 MB of cache.

For all the different versions the number of time steps, $nSteps$, is set to 3650 and the number of independent path simulations, $nSims$, is 10^7. The FPGA execution time has been obtained by simulation but the performance in a real system will not suffer from communications, as data transfer time will be negligible compared to computation time. The tool reports a maximum frequency of 142.3 MHz, which leads to a total computation time of $6.41s$. Comparing to the multi-core implementation, the reconfigurable implementation runs ~ 149 times faster (and ~ 504 faster than the single-core version).

Table 5: Execution time comparison between CPU and FPGA

	Time (s)	SpeedUp
CPU (1 core)	3232.60	1.00
CPU (2 cores)	1735.83	1.86
CPU (4 cores)	955.63	3.38
FPGA (40 cores)	6.41	504.07

The speedup obtained by our solution is comparable to the ones shown in the related work (see Table 2). A remark must be made again emphasizing the fact than keeping a comparable performance, our solution generates a better-quality amount of gaussian random number, which improves the Monte-Carlo simulation quality.

6. Conclusions

We have implemented a Monte-Carlo method for pricing Asian Options in FPGAs using floating-point arithmetic and Impulse C. The focus of this implementation has been made in the quality of the random number generation, a feature that has been obscured in the previous works in the best interest of performance. In contrast to that, our solution achieves a good accuracy-performance tradeoff.

With some tricks in the Impulse C code, and by exploiting path independence, we achieved pipelines with rate one both in the pseudo-random generator (Mersenne + BoxMuller) and in the Asian option pricing computation. The use of Impulse C language has allowed to obtain a good performance improvement in a drastically shorter period of time than a HDL-based development would have taken. Moreover, the use of the Impulse C framework has allowed to use some custom arithmetic in order to reduce the latency of the global system, as well as the total area of the simulation module.

As a result 40 Monte-Carlo cores have been located inside our Stratix V 5SGSMD8 Altera FPGA, showing a speed-up improvement of ~ 504 times over a single-core CPU implementation, and ~ 149 times over a 4-core one.

References

[1] Path integral approach to Asian options in the Black-Scholes model. Physica A: Statistical Mechanics and its Applications 2010;389(4):780 –8.

[2] Kemna A, Vorst A. A pricing method for options based on average asset values. Journal of Banking & Finance 1990;14(1):113–29.

[3] Wu L, Kwok YK, Yu H. Asian options with the american early exercise feature 1999;2(1):101–11.

[4] Black F, Scholes MS. The pricing of options and corporate liabilities. Journal of Political Economy 1973;81(3):637–54.

[5] Robinson S. General concepts of quality for discrete-event simulation. European Journal of Operational Research 2002;138(1):103–17.

[6] L'Ecuyer P. Random number generation. In: Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice. John Wiley & Sons, Inc.; 2007, p. 93–137.

[7] Wallace CS. Fast pseudorandom generators for normal and exponential variates. ACM Trans Math Softw 1996;22:119–27.

[8] Thomas D, Luk W. Non-uniform random number generation through piecewise linear approximations. In: Field Programmable Logic and Applications, 2006. FPL '06. International Conference on. 2006, p. 1 –6.

[9] Matsumoto M, Nishimura T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans Model Comput Simul 1998;8:3–30.

[10] Jones D. Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications. http://www0.cs.ucl.ac.uk/staff/D.Jones/GoodPracticeRNG.pdf; 2010.

[11] Brown RG. Dieharder: A Random Number Test Suite. http://www.phy.duke.edu/ rgb/General/dieharder.php; 2009-2011.

[12] L'Ecuyer P. Maximally equidistributed combined Tausworthe generators. Math Comput 1996;65(213):203–13.

[13] Matrices and the structure of random number sequences. Linear Algebra and its Applications 1985;67(0):147 –56.

[14] On the distribution of points in a cube and the approximate evaluation of integrals. USSR Computational Mathematics and Mathematical Physics 1967;7(4):86–112.

[15] Bratley P, Fox BL. Algorithm 659: Implementing sobol's quasirandom sequence generator. ACM Trans Math Softw 1988;14(1):88–100.

[16] Intel . Vector Statistical Library Notes for Intel Math Kernel Library 10.3 Update 12. http://software.intel.com/sites/products/documentation/hpc/mkl/vslnotes/8_4_9_SOBOL.htm; ????

[17] Thomas DB, Luk W. High quality uniform random number generation through lut optimised linear recurrences. In: In Proc. International Conference on Field-Programmable Technology. IEEE Computer Society. 2005,.

[18] Tian X, Benkrid K. Design and implementation of a high performance financial monte-carlo simulation engine on an fpga supercomputer. In: ICECE Technology, 2008. FPT 2008. International Conference on. 2008, p. 81 –8.

[19] Tse A, Thomas D, Tsoi K, Luk W. Efficient reconfigurable design for pricing asian options. ACM SIGARCH Computer Architecture News 2011;38(4):14–20.

[20] Echeverria P, Lopez-Vallejo M. Fpga gaussian random number generator based on quintic hermite interpolation inversion. In: Circuits and Systems, 2007. MWSCAS 2007. 50th Midwest Symposium on. 2007, p. 871–4.

[21] Tian X, Benkrid K. Mersenne twister random number generation on FPGA, CPU and GPU. In: 2009 NASA/ESA Conference on Adaptive Hardware and Systems. IEEE; 2009, p. 460–4.

[22] Box G, Muller M. A note on the generation of random normal deviates. The Annals of Mathematical Statistics 1958;29(2):610–1.

[23] Okten G, Goncu A. Generating low-discrepancy sequences from the normal distribution: Box-muller or inverse transform? Mathematical and Computer Modelling 2011;53(5-6):1268 –81.

[24] Altera C. Floating-Point Megafunctions User Guide. `http://www.altera.com/literature/ug/ug_altfp_mfug.pdf`; 2011.

[25] Fu M, Dilip B, Wang T. Pricing continuous asian options: a comparison of monte carlo and laplace transform inversion methods. Journal of Computational Finance 1998;2:49–74.

[26] Seghiouer H, Lidouh A, Nqi FZ. Parallel Monte Carlo method for pricing Asian options using trapezium scheme. Appl Math Sci 2011;5.

[27] Campolieti G, Makarov R. Path integral pricing of asian options on state-dependent volatility models. Quantitative Finance 2008;8(2):147–61.

[28] Tian X, Benkrid K, Gu X. High performance monte-carlo based option pricing on fpgas. Engineering Letters 2008;16(3):434–42.

[29] Joshi MS. Graphical asian options. Wilmott Journal 2010;2(2):97–107.

[30] El-Araby E, Merchant SG, El-Ghazawi T. A framework for evaluating high-level design methodologies for high-performance reconfigurable computers. IEEE Transactions on Parallel and Distributed Systems 2011;22:33–45.

[31] El-Araby E, Nosum P, El-Ghazawi T. Productivity of high-level languages on reconfigurable computers: An hpc perspective. In: Field-Programmable Technology, 2007. ICFPT 2007. International Conference on. 2007, p. 257 –60.

[32] El-Araby E, Taher M, Abouellail M, El-Ghazawi T, Newby G. Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study. In: Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on. 2007, p. 99 –106.

[33] Impulse Acelerated Technologies . Impulse-C. `http://www.impulseaccelerated.com`; 2010.

[34] Sanchez-Roman D, Sutter G, Lopez-Buedo S, Gonzlez I, Gomez-Arribas F, Aracil J. "An Euler solver accelerator in FPGA for computational fluid dynamics applications". VII Southern Programmable Logic Conference, SPL2011 2011;.

[35] Xu J, Subramanian N, Alessio A, Hauck S. Impulse c vs. vhdl for accelerating tomographic reconstruction. In: Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on. 2010, p. 171 –4.

[36] Sanchez-Roman D, Lopez-Buedo S, Sutter G, Gonzalez I, Gomez-Arribas F, Aracil J. FPGA Acceleration of a Monte Carlo method for pricing Asian Options using High Level languages. In: XI Edicion Jornadas de Computacion Reconfigurable y Aplicaciones (JCRA2011). 2011,.

[37] Butenhof DR. Programming with POSIX threads. Addison-Wesley Longman Publishing Co., Inc.; 1997.

[38] Pellerin D, Thibault S. Practical FPGA programming in C. Prentice Hall Press Upper Saddle River, NJ, USA; 2005. ISBN 0131543180.

[39] Marsaglia G. Seeds for random number generators. Commun ACM 2003;46(5):90–3.

[40] Bradley T, du Toit J, Tong R, Giles M, Woodhams P. Parallelization Techniques for Random Number Generators. In: Hwu , editor. GPU

Gems: Emerald Edition. Amsterdam: Morgan Kaufman; 2011, p. 231–46.

[41] Dagum L, Menon R. "OpenMP: an industry standard API for shared-memory programming". IEEE Computational Science and Engineering Jan-Mar 1998;.

Diego Sanchez-Roman is a PhD student at the Autonomous University of Madrid, Spain. He received the degree in computer science and mathematics in 2009. In 2011, he has obtained the master's degree in in Computer Science and Telecommunication Engineering. His research interests include computer architecture and high performance computing.

Victor Moreno is a PhD student at the Universidad Autnoma de Madrid, Spain. He joined the High Performance Computing and Networking group of the same university in year 2008, where he participated in the European Union project OneLab2. He received his degrees in Computer Science and Mathematics both in year 2010. In 2011, while finishing his studies of master's degree in Computer Science and Telecommunication Engineering, he has been awarded with a four-year fellowship by the Ministry of Education of Spain (F.P.U scholarship). His research interest areas include High Performance Heterogenous Computing (with GPUs, FPGAs, etc.), Computer Architecture and performance tuning.

Sergio Lopez-Buedo received in 2003 his Ph.D. in Computer Engineering from Universidad Autonoma de Madrid (Spain), where he currently serves as associate professor in the area of Computer Architecture. He was a visiting researcher at University of British Columbia (2005) and at The George Washington University (2006, 2007), and he has also collaborated in the doctorate program of Universita degli Studi di Trento (2007-2009). FPGA technology is his main research interest, especially high-performance reconfigurable computing and communication applications. Dr. Lopez-Buedo holds more than 50 publications, including journals, conferences and books as editor, and he is also co-founder of Naudit HPCN, a company dedicated to providing high-performance computing and networking solutions.

Gustavo D. Sutter received an MS degree in Computer Science from State University UNCPBA of Tandil (Buenos Aires) Argentina, in 1997, and a PhD degree from the Autonomous University of Madrid, Spain, in 2005. He has been a professor at the UNCPBA Argentina and is currently a professor at Universidad Autonoma de Madrid, Spain. His research interests include ASIC and FPGA design, digital arithmetic, development of embedded systems and High Performance Computing. He is the author of three books and more than fifty international papers and communications.

Ivan Gonzalez received the Computer Engineering degree (Ms.C.) in 2000 and the Ph.D. degree in Computer Engineering in 2006, both from Universidad Autonoma de Madrid (UAM), Spain. From October 2002 to October 2006 he was Teaching Assistant at the Computer Engineering Department of UAM. From November 2006 to January 2008 he was a Postdoctoral Research Scientist at the High Performance Computing Laboratory (HPCL), Electrical & Computer Engineering Department, The George Washington University (Washington DC, USA). He was a faculty member of the NSF Center of High Performance Reconfigurable Computing (CHREC) at The George Washington University. His main research interests are heterogeneous computing (with GPUs, FPGAs, etc.), parallel algorithms and performance tuning. Other interests include FPGA-based reconfigurable computing applications, with a special focus on dynamic partial reconfiguration, embedded systems and robotics.

Francisco J. Gomez-Arribas received the Ph.D. from Universidad Autonoma de Madrid (UAM), Spain, in 1996. From October 1996 until November 2000 he was Assistant Professor at the Computer Engineering Department of the UAM. He is currently Professor of Computer Architecture and Parallel Computing courses at the same university. His research field of interest concern reconfigurable computing applications based in FPGA circuits, with a special focus on the design of multiprocessor systems with reconfigurable architecture. Secondary fields of interest include network computing, cryptographic coprocessors, embedded

system on-a-chip and experimental support of C.S. and E.E. education on Internet.

Javier Aracil received the M.Sc. and Ph.D. degrees (Honors) from Technical University of Madrid in 1993 and 1995, both in Telecommunications Engineering. In 1995 he was awarded with a Fulbright scholarship and was appointed as a Postdoctoral Researcher of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. In 1998 he was a research scholar at the Center for Advanced Telecommunications, Systems and Services of The University of Texas at Dallas. He has been an associate professor for University of Cantabria and Public University of Navarra and he is currently a full professor at Universidad Autonoma de Madrid, Madrid, Spain. His research interest are in optical networks and performance evaluation of communication networks. He has authored more than 100 papers in international conferences and journals.