SciComp

# ADVANCED MONTE CARLO SIMULATIONS ON XILINX FPGA

Whitepaper


## Abstract

FPGA (Field-Programmable Gate Array) is an ideal technology for addressing the massive computational requirements and high operational costs associated with derivatives portfolio/risk management services within financial institutions. This whitepaper analyses the feasibility of the advanced Monte Carlo option pricing model code development for Xilinx FPGA and compares it to the corresponding multi-core CPU and GPU implementations. The comparison is made in terms of development efforts, code performance, and energy efficiency



Mikhail Tushentsov

mtushentsov@scicomp.com

# Table of Contents

# Introduction

Recent rapid growth in demand for risk analytics in the Financial Services Industry has led to the increasing need for implementations with the requirements of high performance and energy efficiency. FPGAs (Field-Programmable Gate Arrays) satisfy these requirements with added versatility, reconfigurability, parallelism, determinism, and low latency. An FPGA is a programmable device with a flexible architecture that has a broad set of hardware components that can be configured and interconnected to perform any given task.

In the past, pricing and risk management groups in financial services organizations simplified models to reduce the computational load and operation costs. However, evolving regulations now specify that the pricing models in risk analytics workloads, such as Comprehensive Capital Analysis and Review (CCAR) and Fundamental Review of the Trading Book (FRTB), should closely emulate and even duplicate sophisticated front office trading models.

Typically, risk management groups run a multitude of risk scenarios that entail thousands of risk factors on huge derivatives portfolios. With millions of risk calculations, financial organizations have invested in massive compute farms to process the required risk analysis, typically overnight. Such hardware infrastructures are expensive, given the high energy consumption and operation and maintenance costs.

FPGA accelerated derivative pricing and risk models provide significant performance acceleration (typically comparable to GPU-enabled models) at a tiny fraction of the energy consumption, resulting in reduced power and cooling costs. Furthermore, FPGA offers deterministic latency, which means no jitter and predictable reaction times, thereby reducing variations in performance.

In the past, application development on FPGA required knowledge of digital design and Hardware Description Languages (HDL), such as Verilog and VHDL. This has been the main obstacle preventing their broad adoption by software programmers in the financial services industry. Now, the introduction of the Xilinx SDAccel software development environment [1] has dramatically changed the FPGA development landscape enabling a CPU/GPU-like development experience in the FPGA realm. Developers now can use a familiar workflow to develop and optimize their applications and take advantage of FPGA platforms with little to no prior FPGA experience.

This whitepaper analyses the feasibility of the advanced Monte Carlo option pricing model code development for Xilinx FPGA. The example model, Equity Linked Structured Note with Heston stochastic volatility, is implemented for Xilinx FPGA, Nvidia GPU, and Intel multi-core CPU. The implementations are compared in terms of development efforts, code performance, and energy efficiency.

# Background

The effort involved in converting serial code to an efficient parallel code for multi-core CPU, GPU, and FPGA is roughly equivalent in terms of the development time. These projects are greatly facilitated by modern development tools, e.g., Intel Parallel Studio for CPU, Nvidia CUDA toolkit, and libraries for GPU and SDAccel for Xilinx FPGA. In the typical development scenario, the original code is split into the serial CPU-only part, called host code, and the kernel code that runs in parallel on multi-core CPU or accelerator hardware. While host code mostly remains unchanged, the kernel code must be modified according to the target platform requirements. Also, the kernel wrapper code must be added to match the hardware interface.

## FPGA development flow – baseline code

The first step is to decide on the kernel language. Possible choices are plain C/C++, or OpenCL C. C++ kernels offer the most flexibility, and they require minimal changes from the original code. The next step is to choose the kernel development flow. The standard approach is the top-down flow where the developer writes an OpenCL kernel wrapper and works only in the SDAccel Integrated Development Environment (IDE). In this case, the Vivado High-Level Synthesis (HLS) tool that translates the kernel code to HDL runs behind the scene. Alternatively, in the case of C/C++ kernel code, there is a bottom-up flow [2] where the developer writes a minimal kernel wrapper and the test harness, called the test bench, and validates the kernel and performs the kernel optimizations from Vivado HLS project. This enables a clear separation of the kernel code and host code/application development processes.

Next, one proceeds with the development of a baseline version of the C/C++ kernel code. While OpenCL kernels are possible, the C/C++ kernel is preferable in the case of serial code porting. The target kernel code is refactored using the HLS kernel code guidelines. For example, dynamic memory allocations must be replaced with parametrized static memory declarations, and all library calls must be replaced with HLS compatible libraries.

After that, code should compile and run in the emulation mode of Vivado HLS, and kernel code must synthesize without errors. At this stage, one can switch to SDAccel from Vivado HLS to develop a baseline version of the complete application. The kernel code interface must be specified using HLS pragma directives according to SDAccel requirements. Then the host code call to the kernel function must be wrapped using the OpenCL. At this point, the code should compile and run in SDAccel software emulation mode without any issues.

## FPGA development flow – optimization

At this stage, the kernel code must be optimized by changing its structure and using the HLS pragma directives. It can be done in both Vivado HLS and SDAccel. Most of the initial optimization efforts are concentrated around the loops - unrolling and pipelining. Analyzing the HLS synthesis reports and following the guidelines, one can optimize the code to be ready for the hardware emulation build and run. The code parameters at this stage should be scaled down to ensure the reasonable hardware emulation execution time. Once the hardware emulation mode builds and runs successfully, the code is ready for the actual hardware build. The hardware emulation run can provide a reasonably good ballpark estimate for the runtime performance and energy efficiency.

In contrast with CPU and GPU, the hardware build for FPGA takes a significant amount of time (hours) due to the circuit's optimal placing and routing. While it can be considered an inconvenience within the standard developing practices, there is typically the need to do it only at the code release point, because the software and hardware emulation usually provide a reasonable estimate of the real hardware code performance. Also, the hardware build time can be hidden in the overnight build/tests within a continuous integration practice.

## Monte Carlo code development

The Monte Carlo code is a massively parallel application. The most straightforward FPGA compatible approach is to divide the paths into several blocks, and each block is processed using a part of the FPGA resources area. These blocks are processed in parallel, and it is equivalent to unrolling a path loop with a small factor, say 8 or 16, so there are enough resources to process each block. Path blocks are then processed in batches, where the batch size depends on the resources available for the block. Within a batch, the processing is pipelined, and the goal is to minimize the initiation interval (II). With a small initiation interval, the loop latency is reduced to a number of cycles close to the number of loop iterations.

The availability of reliable, well-designed core low-level components and building blocks plays a significant role in the financial Monte Carlo code development. The most crucial building block of MC simulation is the random number generator. The parallel code structure and FPGA hardware impose special requirements on possible implementations. They should have the capability to generate independent sequences of random numbers, so there should be a fast-forward (fast jump ahead in the sequence) functionality. While many RNGs have a very long period, there is no guarantee that different initial seeds produce independent (non-overlapping, uncorrelated) sequences of random numbers. Without the fast-forward functionality, RNG initialization can be very time consuming, eliminating the benefits of the parallel implementation. A second requirement is that the RNG state must be small, and the state advance function must be FPGA resource-efficient, i.e., it should have only simple bit-wise operations with minimum multiplication/division operations.

The second essential building block of MC simulation is the generation of uniform and normal distribution variates. Most financial models rely on normal and log-normal distributions. In the case of quasi-random numbers, the inverse cumulative transform method is required to produce normal variates. This can be tricky in the case of single-precision floating-point implementations since the RNG produces 32-bit unsigned integers and one must be careful about the one-to-one mapping of 32-bit integer to 32-bit floating-point variable on (0,1) interval, specifically to avoid any clustering around 0 and 1, which can break the transformation to normal variates.

Using quasi-random sequences requires one further building block to ensure the variance reduction in path-dependent cases, the Brownian Bridge (BB). The critical aspect of the quasi-random generator and BB implementation of FPGA is that the code must be split between host and kernel to minimize FPGA are consumption and maximize performance. Typically, it makes sense to keep all initializations on the host side and then move the pre-calculated data to kernel.

The use of quasi-random generators imposes additional restrictions on the dataflow between code sections. There is a specific order to how random variates are processed. They must be generated for all time steps simultaneously to apply the BB transform, so it is not possible to process them in time step by time step fashion (path parallel).

# Equity Linked Structured Note model

The Equity Linked Structured Note belongs to the equity-linked note class investment products that combine a fixed-income investment with additional potential returns that are tied to the performance of equities. Equity-linked notes are usually structured to return the initial investment with a variable interest portion that depends on the performance of the linked equity. Equity-linked notes provide a way for investors to protect their capital while also getting the potential for an above-average return compared to regular bonds. This code prices an equity-linked structured note in which the note is linked to a basket of indices.

## Model description

If on observation dates, the performances of all indices, relative to a reference, are above the call barrier for that date, the note redeems early at notional plus a bonus coupon. If early redemption does not occur, then at maturity (final coupon date) either: the note redeems at par plus a maturity coupon or, if the performance of at least Knock-In Number of the indices has ever been below the Knock-In Barrier during the tenor, on a continuously observed basis, then the Note redeems at the performance percentage of the worst index. For each index, the boolean Knocked flag is either 0 or 1, depending on whether it has dropped below the Knock-In Barrier prior to Value Date.

The indices follow correlated Heston stochastic volatility processes, and we allow for a term structure of rates:

$$\begin{aligned} dS &= (r-q)Sdt + \sqrt{\mathrm{v}}SdW_S \\ d\mathrm{v} &= \kappa(\theta-\mathrm{v})dt + \sigma\sqrt{\mathrm{v}}dW_{\mathrm{v}}\,, \\ dW_S dW_{\mathrm{v}} &= \rho_{S\mathrm{v}}dt \end{aligned}$$

where $S$ is the index level, $r$ is the forward rate, $q$ is the dividend yield of the index, and $\kappa$, $\theta$, and $\sigma$ are the variance reversion speed, long term variance, and volatility of variance of each index. We allow for a full correlation matrix, comprising three correlation sub-matrices:

$$\rho = \begin{pmatrix} \rho_{SS} & \rho_{S\mathrm{v}} \\ \rho_{S\mathrm{v}} & \rho_{\mathrm{vv}} \end{pmatrix},$$

where $\rho_{SS}$ specifies the correlation of index levels to one another, $\rho_{S\mathrm{v}}$ specifies the correlation of index levels to index instantaneous volatilities, and $\rho_{\mathrm{vv}}$ specifies the correlation of index instantaneous volatilities to one another.

Continuous monitoring of the Knock-In Barrier uses a Brownian Bridge-based correction factor. Since this correction is exact only for uncorrelated assets, we allow for extra number barrier monitoring dates between each pair of Observation Dates. The Brownian bridge approximation converges quite rapidly as the number of extra barrier monitoring dates is increased unless the indices are nearly perfectly correlated. On contracted observation dates, we monitor the indices for knockout (discretely monitored), and if knockout has occurred, make the early payment and set the early redemption flag Redeemed. If redemption has not occurred at maturity, the payout is either the minimum performance if KI >= KINum, or else par plus the maturity coupon.

## Test Details and Code complexity

The following input parameters are used in the test:

- Six (6) assets
- Maturity coupon, bonus coupon
- Continuously monitored knockout barrier
- Four (nsub=4) barrier monitoring dates between the six observation dates
- Maturity: 3 years
- Semi-annual coupons
- Not Cancellable

For the given input parameters, a 2x24x6 = 288-dimensional Sobol sequence is required with Brownian Bridge and correlated normals and an additional 144-dimensional Sobol sequence of uniforms for the monitoring. For example, for 1M (2^20) paths:

- 432M Sobol quasi-random numbers
- 288M inverse cumulative normal transforms
- 12M Brownian Bridge transforms (24 steps each)
- 24M Correlated normal transforms (12x12 lower tridiagonal matrix-vector multiplication)

Once the correlated random variates are produced, the Heston process with continuous monitoring of the Knock-In Barrier is simulated (24 times steps for 6 indices, index level, and volatility simulation).

## Hardware and software setup

The test environment was set up on workstations with 8 core Intel Xeon E5-2686v4 CPU running Ubuntu 16.04LTS OS. The GPU workstation had Nvidia V100 16GB GPU installed, and the code was built using CUDA 10.1 toolkit and GCC 5.4.0 compiler. The FPGA workstation had Xilinx Alveo U200 FPGA and SDAccel 2018.3 development environment. The CPU code was compiled for the multi-core CPU using the Intel compiler from Intel Parallel Studio XE 2019 suite.

## Power estimation

The power estimation was performed using the following power analysis tools: Intel SoC Watch (socwatch) for CPU, NVIDIA System Management Interface (nvidia-smi) for GPU, and Xilinx Vivado suite Power Report module for Xilinx FPGA.

## Cost estimation

The Capital Expenditures (CapEx) cost estimations were done using the last three-month average price from the hardware pricer trackers and the distributors price-lists.

## Results

The FPGA resource utilization for the test case is presented in Table 1.

*Table 1. FPGA resource utilization*

| Name | BRAM_18K | DSP48E | FF | LUT | URAM |
|---|---|---|---|---|---|
| **Utilization (%)** | 19 | 75 | 35 | 57 | 0 |

The performance results for the different hardware platforms are presented in Table 2. The key performance metrics are performance (paths per second) per CapEx (USD) and performance per unit of power (mW). While the performance per CapEx for GPU and FPGA are roughly the same, the FPGA performance per power is about 4.5 times better due to almost order of magnitude less power consumption of FPGA.

*Table 2. Performance for different hardware platforms*

| | Price (3m avg) | Mpath/sec | W | paths/sec/$ | paths/sec/mW |
|---|---|---|---|---|---|
| **CPU** | $2,777.00 | 0.427 | 110 | 154 | 4 |
| **GPU** | $6,420.00 | 3.922 | 203 | 611 | 19 |
| **FPGA** | $4,495.00 | 2.222 | 26 | 494 | 85 |

## Conclusion

Considering that the level of effort to develop and maintain GPU and FPGA codes are typically the same, the FPGA-enabled code demonstrates a reduced Total Cost of Ownership (TCO) with respect to the GPU-enabled codes. It, therefore, confirms the feasibility of the advanced Monte Carlo option pricing model code development for Xilinx FPGA.

# Bibliography

[1] Xilinx, The Xilinx SDAccel Development Environment, Backgrounder.

[2] Xilinx, SDAccel Environment User, UG1023 (v. 2018.3).

## Revision History

The following table shows the revision history for this document:

| Date | Version | Description of revisions |
|---|---|---|
| **08/23/2019** | 1.0.0 | Initial release |
| **09/24/2019** | 1.0.1 | Initial feedback edits |
| **11/20/2019** | 1.0.2 | Second feedback edits |