A Dataflow Overlay for Monte Carlo Multi-Asset Option Pricing on AMD Versal AI Engines

Mohamed Bouaziz
KAUST
Thuwal, Saudi Arabia
mohamed.bouaziz@kaust.edu.sa

Michael Samet
RWTH Aachen University
Aachen, Germany
samet@uq.rwth-aachen.de

Suhaib A. Fahmy
KAUST
Thuwal, Saudi Arabia
suhaib.fahmy@kaust.edu.sa

***Abstract*—Monte Carlo simulations are widely used for financial applications, particularly in option pricing. Multi-asset option pricing has not been as widely investigated as single-asset problems and presents specific challenges. This paper presents an efficient way to design a dataflow overlay for multi-asset option pricing on AMD Versal AI Engines by leveraging their SIMD capabilities and the high-performance interconnect network linking them. The proposed dataflow overlay achieves a highly scalable parallelized implementation that can utilize nearly the totality of the available AI Engines on the AMD Versal VCK5000 accelerator. We also show that it achieves up to 25.7× speedup over a traditional FPGA programmable logic implementation using the Vitis Quantitive Finance Library, a specialized AMD library for quantitative finance on FPGAs, and up to 13.41× speedup over a highly parallelized CPU implementation over 128 threads. We show that although an Nvidia RTX A6000 GPU implementation has 0.73× the execution time of our design, our design achieves 1.82× the energy efficiency.**

***Index Terms*—Domain-Specific Accelerator, AMD Versal ACAP, Option Pricing, Monte Carlo Simulation, High-Performance Quantitative Finance.**

I. Introduction

Quantitative finance is critical in applied scientific computing and industry [1] and requires fast and accurate numerical methods. Various acceleration means, including FPGAs, multicore CPUs, and GPUs, have been employed to allow the acceleration of quantitative finance applications [2]. In particular, option pricing requires computationally intensive operations. Options are financial derivatives that give their owner the right, but not the obligation, to buy or sell an underlying asset (or multiple assets) at a predetermined strike price on or before a specific maturity date. Determining the fair value of an option is a core task in quantitative finance [3].

In the financial literature, the time evolution of asset prices is modelled using stochastic differential equations (SDEs) [4]. For risk management purposes, the parameters of these SDEs must be calibrated to market data, possibly multiple times a day [5]. During this calibration, practitioners determine the parameters that consistently reflect the observed prices on the market for different values of maturity dates and strike prices. Hence, a large number of options have to be evaluated for different parameters (strike prices and maturity dates), which can be computationally prohibitive if the price evaluation is not sufficiently fast [6]. This raises the need to meet industrial demands by leveraging reduced latency accelerators [7], [8].

Multi-asset options have become instrumental in risk mitigation in fixed-income, foreign exchange, and energy markets [9]. They present a cheaper alternative for investors to hedge against the combined associated risk of multiple assets. However, due to their complex valuation, as they mostly do not admit closed-form formulas, analytical approximations have been explored [10], [11]. The downside of such approximations is that they are typically specific to a particular model and payoff, i.e. the profit of the option. In contrast, numerical approximations are more applicable for various payoffs and models. Given the existing infrastructures and literature in hardware acceleration for numerical applications, the acceleration of the numerical methods for option pricing can be well supported and achievable.

There are four main approaches in the option pricing literature [12], partial differential equation (PDE) approach [13], transform methods [14], Monte Carlo (MC) simulation methods [15], and the more recent deep learning approaches [16]. The PDE approach relies on formulating the pricing problem as a PDE [17] and appropriately discretising and approximating it [18]. The shortcoming of this approach is that its computational cost grows exponentially with the number of underlying assets. Hence, the problem becomes intractable for multi-asset options, typically with more than three assets [13], a phenomenon known as the curse of dimensionality [19]. Transform methods address the problem by mapping it from the direct space of prices to an image space via an integral transform (e.g. Fourier [20]). This has been achieved through various techniques, such as sparse grid quadrature [21], adaptive sparse grid quadrature [22], and quasi-Monte Carlo [23]. These methods have been shown to alleviate the curse of dimensionality [24], [25]. However, empirical evidence shows that while these methods reduce the impact of dimensionality and outperform traditional MC methods, they tend to become slow with larger problem sizes [21], [22].

MC methods compute the option price by approximating the expected value of its payoff via the simulation of M independent paths of the stochastic price process and averaging their outcome to deduce the result [15]. Nevertheless, the convergence rate is considered to be rather slow [26], which motivated the development of several variance reduction techniques, such as control variates, importance sampling [26] or the multilevel MC (MLMC) method [27]. On the other hand, option pricing using the MC method has been subject to acceleration using CPUs [28], GPUs [29], and FPGAs [30], as it is inherently a parallel process of simulating multiple independent paths. The independence of path simulations allows the parallel scaling of the processing and allows it to scale spatially, given its reduced data communication characteristic.

Recent literature has increasingly focused on using deep neural networks (DNNs) for option pricing and calibration to overcome the curse of dimensionality [16], [31]. A significant advantage of this approach is that after a computationally intensive offline training phase—which often involves MC simulations—option prices can be computed very rapidly during the online forward pass. However, a notable drawback is the complexity involved in controlling the error of DNNs. In contrast, the MC method allows for cheap computation of a probabilistic error estimate [26]. As a result, the MC simulations are not only still required to generate the training data but are also still employed to benchmark their performance and verify their accuracy [16], [31]. This makes the need for acceleration of the MC for option pricing inevitable.

For the asset dynamics, we consider the multivariate geometric Brownian motion (GBM), which is the multivariate extension of the Nobel prize-winning Black-Scholes-Merton model [32]. Over the past two decades, more sophisticated models such as jump-diffusion models [33], stochastic volatility models [34], and non-Markovian rough volatility models [35] have been developed. Despite these advancements, geometric Brownian motion (GBM) model remains the most commonly used model for benchmarking the performance of pricing methods in the financial literature [9], [10], [16], [17], [22], [31], [36]. Hence the importance of building a framework for acceleration of said model.

Emerging reconfigurable dataflow architectures, such as Groq [37], SambaNova [38], and AMD Versal AIE [39], present an exciting platform for accelerating a variety of compute-intensive applications. By combining massive parallelism with programmable functional units connected via a general interconnect network, these architectures address von Neumann data movement challenges, enabling enhanced parallelism and unrolling of dataflow computations.

In this work, we propose a massively parallel dataflow design for MC multi-asset option pricing. We implement the dataflow as an overlay tailored specifically to the AMD Versal AI Engines (AIEs). As illustrated in Fig. 1, the AMD Versal architecture comprises an array of configurable AIE cores running computation and communicating through configurable high-performance stream interconnects. This high degree of reconfigurability allows the implementation of efficient dataflow designs. The AIE cores are highly optimized processors for parallel and vectorized operations, allowing the native support of processing in parallel MC independent path simulations for multi-asset options. The closeness and connectivity of the AIE cores through the stream interconnects allow offloading composing kernels that enable the pricing model calculation on different AIEs for load balancing. Besides, the spatial scaling of the AIE processors over an array of $400 = 8 \times 50$ cores can

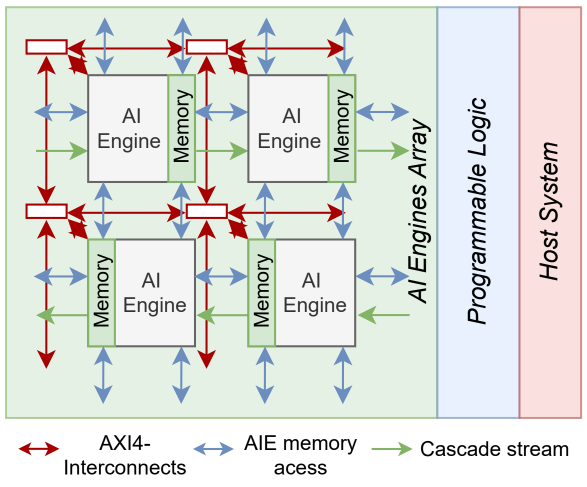

Fig. 1: AMD Versal System-on-Chip architecture.

be leveraged to tile multiple acceleration units, given that the simulations of the independent paths are completely decoupled. These factors enable the design of a massively parallel dataflow overlay. The AIE array interfaces with Programmable logic (PL) (i.e. traditional FPGA fabric) that can be used to communicate with the host system and implement further custom logic.

In this work, we propose a bottom-up dataflow overlay design targeting the acceleration of the MC multi-asset option pricing using the AMD Versal AIEs following these steps:

- We analyze the computational requirements of the GBM model in generating normal random variables, simulating path price, and calculating the resulting payoff and demonstrate how to efficiently map these operations to the SIMD operations supported by the AIEs.
- We pipeline the pricing steps over multiple AIEs operating concurrently while communicating via the high-performance AXI4-Interconnect network.
- We show how to scale the design spatially using multiple compute units over nearly the totality of the AIE array, how to hierarchically broadcast the problem parameters across the cores, and how to stream the data in and out through the PL.
- We propose an automated framework to run simulations with different parameters, i.e. number of assets, independent paths, and time steps.
- We compare the performance of the proposed design to a parallel dataflow implementation on the PL implemented using the Vitis Quantitative Finance Library [30] (a specialized library for quantitative finance applications for AMD FPGAs), a similar highly parallel multi-threaded CPU implementation, and a highly parallel GPU implementation. We showcase significant speedup against the FPGA and CPU, and competitive performance and improved energy efficiency against the GPU.
- We open-source this work to the broader community on this GitHub repository [40].

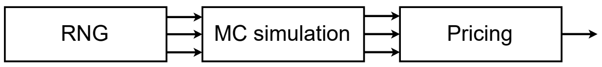

Fig. 2: Generic MC option pricer flow.

II. Multi-Asset Option Pricing Flow

A financial option gives its owner the right (but no obligation) to exchange, by buying or selling, an (or multiple) underlying asset(s) on or before a limit date T, called the maturity date, for a predefined price K, called the strike price. For an option on an asset with stochastic price process $S(t)$ to be profitable, its strike price should be lower than the actual price on the maturity date $S(T)$ in case of buying, and higher in case of selling. The payoff, that is the profit an option can earn, given these parameters, is given as in Eq. 1.

$$\text{Payoff} = \max\{0, S(T) - K\} \tag{1}$$

Determining an option's strike price at T involves predicting the underlying asset's price $S(t)$ in the future. Partial differential equations are used to model price evolution in time, such as in the GBM model. In GBM, the price of the underlying asset $S(t)$ is assumed to follow a geometric Brownian motion characterized by a constant drift r and volatility σ. Mathematically, this is described by the following stochastic differential equation:

$$dS(t) = r\,S(t)\,dt + \sigma\,S(t)\,dW(t),$$

where $W(t)$ denotes a Brownian motion, i.e. a stochastic process that follows the normal distribution of mean 0 and variance t. The solution to this differential equation is:

$$S(T) = S(0)\exp\left(\left[r - \frac{1}{2}\sigma^2\right]T + \sigma W(T)\right),$$

where $S(0)$ is the known initial stock price. Since $W(T)$ is a Brownian motion, and therefore normally distributed, the stock price $S(T)$ can be expressed in terms of a normal random variable Z as follows:

$$S(T) = S(0)\exp\left(\left[r - \frac{1}{2}\sigma^2\right]T + \sigma\sqrt{T}Z\right).$$

For the multi-asset, the GBM model generalizes to:

$$dS_i(t) = r_i\,S_i(t)\,dt + \sigma_i\,S_i(t)\,dW_i(t),$$

where $\{S_i\}_{i=1}^d$ is the set of the prices of the assets, which leads to the multi-dimensional formulation for the multi-option problem with independent $W_i(t)$ given in Eq. 2 where $(L_{i,j})_{1\leq i,j\leq d}$ represent a matrix that correlates the independent Brownian motions.

$$S_i(T) = S_i(0)\exp\left(\left[r - \frac{1}{2}\sigma_i^2\right]T + \sigma_i\sqrt{T}\sum_{j=1}^{d} L_{i,j}Z_j\right) \tag{2}$$

MC simulation is commonly used to model price evolution over time. The simulation engine, illustrated in Fig. 2, is called the option pricer. To simulate prices $\{S_i\}_{i=1}^d$ using the GBM model, univariate independent normal random variables $\{Z_i\}_{i=1}^d$ are generated and transformed into multivariate random variables $\left\{\sum_{j=1}^d L_{i,j}Z_j\right\}_{i=1}^d$.

The SIMD capabilities of AIE cores enable vectorized, parallel generation of independent normal random variables $\{Z_i\}_{i=1}^d$ and their correlation into multivariate random variables $\left\{\sum_{j=1}^d L_{i,j}Z_j\right\}_{i=1}^d$, as detailed in Sections III-A1 and III-A2. The MC simulation engine then independently calculates the asset price for each path at maturity T, $S(T)$. Finally, the pricing stage averages these prices to determine the option's payoff, as given in Eq. 1.

In terms of parameters, the accuracy of MC simulations depends primarily on the number of simulated paths M and time steps N per path. Increasing M reduces statistical error, improving the accuracy. Since all M paths are independent, they can leverage vectorized operations and spatial parallelism across a large number of cores. For each path, the option's lifetime is discretized into N time steps. More time steps enhances simulation granularity and accuracy. As each path's simulation is assigned to an AIE in our case, the N timesteps are pipelined across its pipeline stages. The problem parameters (M, N, K, T, d, $L_{i,j}$, $S_i(0)$, r, σ_i) are broadcast to the compute units via low-latency stream interconnect.

Finally, option pricing involves reducing the vector of asset prices from each MC simulation path into a single value before averaging across all paths to calculate the payoff. The reduction method—average, minimum, or maximum—depends on the option's payoff. This work uses the minimum reduction scheme defined in Eq. 3. The provided analysis and parallelization techniques apply equally to other reduction methods (maximum or average) that AIE operations support.

$$\text{Payoff} = \max\{0, min_{1\leq i\leq d}(S_i(T)) - K\} \tag{3}$$

The dataflow design is general, supporting a wide set of parameters, including multiple reduction schemes with a variable number of assets. The application of the required changes is automated, as discussed in Section III-C.

III. Multi-Asset Option Pricing Dataflow Design

This section details the multi-asset option pricing dataflow overlay design tailored specifically for the Versal AIEs. Section III-A details the architecture of a single Compute Unit (CU) that implements the flow presented in Section II and how each stage is mapped to the AIE resources. Subsequently, Section III-B demonstrates how to scale the design to multiple CUs that exploit near the totality of the AIE array. Finally, Section III-C shows how to exploit the PL fabric for streaming parameters into the CUs using the broadcasting feature of the network of interconnects of the AIE array, receiving results, and how the tiling of the simulation paths over the CUs with a variable number of assets is automated on the host.

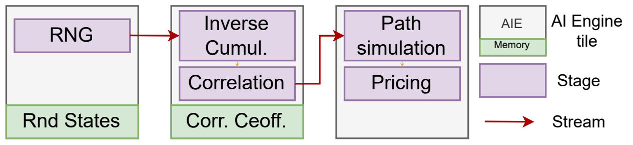

Fig. 3: Single Compute Unit architecture mapping to the AIEs.

A. Single CU Design

As discussed in Section II, the option pricing flow involves an MC simulation of evolving asset prices over time. The simulation runs on multiple independent paths starting from random states, outputting simulated prices that are reduced to one price per path. These reduced prices are averaged to determine the option's payoff. As noted in Section I, quantitative finance typically uses up to three assets [13] due to the compute-intensive nature of the problem. This accelerator supports up to eight assets, enabling experimentation with higher dimensions given the achieved speedups.

Fig. 3 shows a single CU accelerator design for multi-asset option pricing using the GBM model (Eq. 2). The MC simulation requires generating a correlated multivariate random distribution, as discussed in Section II. One AIE generates a vector of random numbers from a uniform distribution using the Fast Mersenne Twister method [41], adapted for AIEs, as discussed in Section III-A1. These numbers are normalized to $[0, 1)$ and streamed to an adjacent AIE, which transforms them into univariate normal distributions using the Inverse Cumulative Gaussian method, as discussed in Section III-A2. The same AIE then correlates the multivariate vector using the correlation coefficients $L_{i,j,\, 1\leq i,j\leq d}$. Finally, the correlated vector is streamed to another AIE, where the prices $S_i(T)_{,\, 1\leq i \leq d}$ are simulated on multiple paths using the GBM model, and the final price is determined, as discussed in Section III-A3. Note that the stages in Fig. 3 mapped to the same AIE tile are fused within the same kernel and occupy the totality of the core's runtime.

As shown in Fig. 3, data exchanges between AIEs are enabled through the stream interconnect in the AIE Array. The choice of stream processing follows the feed-forward nature of the simulation. The stream interconnects ensure 32-bit/cycle communication and are connected to AIEs through FIFOs to have 128-bit 4-word/4-cycle or 32-bit 1-word/cycle access [42]. These streams are accessed using API calls that intrinsically chunk data to align with these access patterns. Additionally, operations within each AIE are vectorized into SIMD instructions, requiring a balance between production and consumption rates to ensure that the vector size consumed by one AIE matches the vector size produced by the preceding. As shown in Fig. 1, the stream AXI4-interconnect network supports bi-directional streaming through available switch boxes and automatically manages back pressure between tiles [42]. This enables flexible placement of compute kernels, facilitating the scaling of the accelerator design to multiple CUs, as discussed in Section III-B.

1) Porting SIMD Mersenne Twister to AIEs:

The Mersenne Twister (MT) [41] is a pseudorandom number generator following the uniform distribution. It is used in the Vitis Quantitative Finance Library [30] and QuantLib [43] for option pricing. It is based on a linear feedback shift register (LFSR) structure, which operates on an internal state array, $W[0..Q-1]$, consisting of Q binary words of fixed word length, as is shown in the recursion in Eq. 4.

$$\begin{aligned} &W[0] \leftarrow W[1], W[1] \leftarrow W[2], \dots, W[Q-2] \leftarrow W[Q-1] \\ &W[N-1] \leftarrow g(W[0], \dots, W[Q-1]) \end{aligned} \tag{4}$$

The MT method is defined by the recursion given in Eq. 5 where $(W[0]|W[1])$ represents the concatenation of the most significant $32-r$ bits (MSBs) of $W[0]$ and the least significant r bits (LSBs) of $W[0]$. The matrix A is a 32×32-matrix, chosen for efficient bitwise multiplication $W \cdot A$, and R is an integer such that $1 < R < Q$. The $\oplus$ operation represents bitwise exclusive-or.

$$g(W[0], \dots, W[N-1]) = (W[0]|W[1]) \cdot A \oplus W[R] \tag{5}$$

This design utilizes the SIMD Fast Mersenne Twister (SFMT) [41] for the uniform random number generation as depicted in Algorithm 1. The SFMT is a vectorized version of the MT over 128 bits, meaning that $W[0..Q-1]$ are 128 bits regarded as 4×32-bit elements. This allows using the AIEs optimized for efficient SIMD operations following the methodology presented in [44].

$$\begin{aligned} g(W[0], \dots, W[Q-1]) = W[0] \cdot A \oplus W[R] \cdot B \\ \oplus W[Q-1] \cdot C \oplus W[Q-2] \cdot D \end{aligned} \tag{6}$$

Typically in SFMT, Q is 156 and the recursions given in Eq. 6 are as follows:

- $W \cdot A = (W << 8) \oplus W$: shifts the 128-bit W 8 bits to the left and applies XOR to itself.
- $W \cdot B = (W \overset{32}{>>} 11) \& (BFFFFFF6\ BFFAFFFF\ DDFECB7F DFFFFFEF)$: shifts every 32-bit chunk of the W 11 bits to the right and applies a bitwise AND operation to a mask. The mask has been randomly chosen in [41], with a $7/8$ probability of choosing 1 for denser feedback.
- $W \cdot C = (W >> 8)$: shifts the 128-bit W 8 bits to the right.
- $W \cdot D = (W \overset{32}{<<} 18)$: shifts every 32-bit chunk of the W 18 bits to the left.

Porting these operations to the AIE using the AIE-API [45] presents two challenges. First, the AIE-API does not support shifting operations on 128-bit. Using the AIE-API, shifts are performed element-wise on the 4×32-bit vectors. This complicates the operations $W \cdot A$ and $W \cdot C$. Second, while many data types are supported, there is no direct support

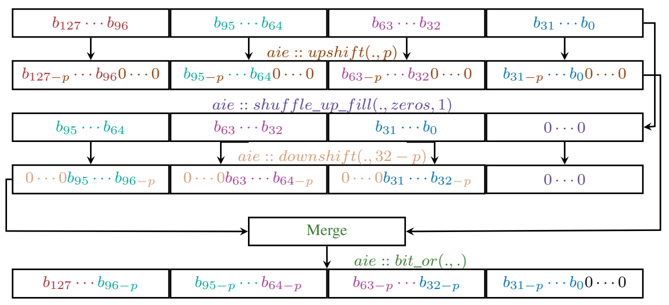
$b_{127}\cdots b_{96}$
$b_{95}\cdots b_{64}$
$b_{63}\cdots b_{32}$
$b_{31}\cdots b_{0}$
$aie :: upshift(.,p)$
$b_{127-p}\cdots b_{96}0\cdots 0$
$b_{95-p}\cdots b_{64}0\cdots 0$
$b_{63-p}\cdots b_{32}0\cdots 0$
$b_{31-p}\cdots b_{0}0\cdots 0$
$aie :: shuffle\_up\_fill(., zeros, 1)$
$b_{95}\cdots b_{64}$
$b_{63}\cdots b_{32}$
$b_{31}\cdots b_{0}$
$0\cdots 0$
$aie :: downshift(., 32-p)$
$0\cdots 0b_{95}\cdots b_{96-p}$
$0\cdots 0b_{63}\cdots b_{64-p}$
$0\cdots 0b_{31}\cdots b_{32-p}$
$0\cdots 0$
Merge
$aie :: bit\_or(.,.)$
$b_{127}\cdots b_{96-p}$
$b_{95-p}\cdots b_{64-p}$
$b_{63-p}\cdots b_{32-p}$
$b_{31-p}\cdots b_{0}0\cdots 0$

Fig. 4: 128-bit vector shift left operation.

for unsigned integers, which are crucial to cast output values into positive floating-point random numbers in $[0, 1)$.

Four operations are required to shift the 128-bit binary word stored in a 4×32-bit vector using the AIE-API, as illustrated in Fig. 4. The elements are shifted to the left n bits using `aie::upshift`. This zeros the n LSBs, and the goal is to replace them with the initial n MSBs of the adjacent elements to the right. For that, the elements of the vector are moved leftwise one element, and the right-most element is filled with 0 using `aie::shuffle_up_fill`. This step allows the extraction of the n MSBs of the adjacent elements to the right. In order to align them with the n zeros in LSBs of each element, a shift to the right of each element by $32 - n$ is performed using `aie::downshift`. Finally, since the required values are aligned with the zeros of the LSBs, a bitwise OR operation is performed, resulting in the desired 128-bit left shift operation needed for the $W \cdot A$ operation. For the 128-bit right shift, the same steps apply oppositely.

As shown in Fig. 3, the initial states are stored and updated in the AIE core internal memory whenever a new random number is needed. The states are initialized by random seeds as described in Section III-C.

Finally, as shown in Algorithm 1, the 4×32-bit vector generated by SFMT undergoes a bitwise AND operation with the mask `0x7fffffff` to nullify the MSB. This ensures the values are non-negative, as the AIE-API does not support the unsigned integer data type (L6). The resulting vector is transformed into four single-precision floating-point numbers in $[0, 1)$ using `aie::to_float` by specifying 31 (MSB) as the position of the input decimal point (L7). Since the SFMT generates 128-bit random numbers by looping throughout the state array of Q elements, the generation process is done in blocks of Q elements to generate the $d \times M \times N\times$ 32-bit random numbers required for the simulation (L2). This also ensures that the timestep dependencies of this design can be kept within the AIE and, therefore, do not require updating the whole AIE graph for synchronization.

2) Multivariate Normal Distribution: Generating multivariate normal distribution from a uniform distribution requires two steps: using the Inverse Cumulative Normal Distribution (ICND) to generate univariates and multiplying them by the correlation coefficients $L_{i,j\,1\leq i,j\leq d}$ as discussed in Section II.

The rational fraction approximation P, given in Eq. 7 (see coefficient in Table I), from [29] (adapted from [46]), approximates the ICND. Unlike the Vitis Quantitative Finance Library, which relies on the method in [47] that prevents vectorization due to varying computations across vector elements, this approximation applies the same operation to all elements, making it suitable for the SIMD capabilities of the AIEs.

Algorithm 1 SFMT Random Number Generation.

```
1:  Input: d, N, M
2:  Initialize: block_bound = $\lceil \frac{\frac{d}{128/32} \times M \times N}{Q} \rceil$
3:  for block = 0 to block_bound do
4:      for $i = 0$ to $Q$ (or fewer for last block) do
5:          rand_vec = perform_SFMT()
6:          rand_vec = aie::bit_and(rand_vec, 0x7fffffff)
7:          rand_vec = aie::to_float(rand_vec, 31)
8:          aie::writeincr(rand_vec)            ▷ Output 4
    single-precision floats in $[0, 1)$
9:      end for
10: end for
```

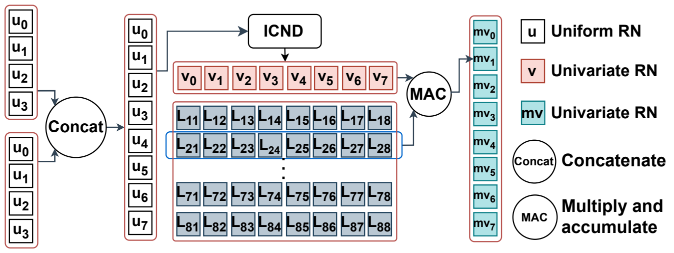

Fig. 5: Generation of correlated normal random numbers.

$$P(R(X)) = \frac{(((((A_1R^2 + A_2)R^2 + A_3)R^2 + A_4)R^2 + A_5)R^2 + A_6)R}{((((B_1R^2 + B_2)R^2 + B_3)R^2 + B_4)R^2 + B_5)R^2 + 1}$$
$$\text{, where } R(X) = X - 0.5 \tag{7}$$

For a multivariate of 8 elements (recall that 8 is the maximum number of assets supported), the flow involves consuming and concatenating two incoming 4 × single-precision (32-bit) floating-point streams from the SFMT core, applying the ICND approximation to generate the univariates $\{Z_j\}_{j=1}^{d}$, and performing the multiply-and-accumulate operation to calculate the multivariate components $\{\sum_{j=1}^{d} L_{i,j} Z_j\}_{i=1}^{d}$, as shown in Fig. 5. The multivariate components are then streamed to the path simulation and pricing stage.

3) Path generation and pricing: Recall the price evolution of the GBM model for an asset S_i from Eq. 2

$$S_i(T) = S_i(0) \exp\left(\left[r - \frac{1}{2}\sigma_i^2\right] T + \sigma_i\sqrt{T}\sum_{j=1}^{d} L_{i,j} Z_j\right)$$

For the path simulation, the only term that depends on the correlated multivariate and evolves over time is $\sigma_i\sqrt{T}\sum_{j=1}^{d} L_{i,j} Z_j$. Hence, the simulation calculates $\left[r - \frac{1}{2}\sigma_i^2\right]$ once and keeps it constant while the evolving term $\sigma_i\sqrt{T}\sum_{j=1}^{d} L_{i,j} Z_j$ is computed for each path. The simulation for multiple assets is vectorized using SIMD operations.

TABLE I: Coefficients of fraction P in Eq. 7.

Coeff.	Value	Coeff.	Value
A_1	-39.696830286653757	B_1	-54.476098798224058
A_2	220.94609842452050	B_2	161.58583685804089
A_3	-275.92851044696869	B_3	-155.69897985988661
A_4	138.35775186726900	B_4	66.801311887719720
A_5	-30.664798066147160	B_5	-13.280681552885721
A_6	2.5066282774592392		

The exponential operation is required to calculate $S_i(T)$, but since the AIEs do not natively support exponentiation [45], a sequence of operations is used to emulate it. To avoid overhead, the exponential operation is delayed until the pricing phase where it is applied only to some values of S_i. This is because, as discussed in Section II, multi-asset pricing reduces the final simulated asset prices to one price, typically the mean, minimum, or maximum. This work uses the minimum price per path, meaning that $S(T) = min(S_i)$, and not all values of S_i are needed for the pricing phase.

Recall that the payoff per path from Eq. 1

$$\text{Payoff} = \max\{0, S(T) - K\}$$

This is equivalent to the payoff being zero if and only if $S(T) < K$, which is in turn equivalent to Eq. 8.

$$log(S(T)) = log(min(S_i)) < log(K) \\ \text{, since } log \text{ is a monotonically increasing function} \tag{8}$$

Therefore, the payoff is calculated only if the condition in Eq. 8 is satisfied. Since log is a monotonically increasing function, then:

$$log(min(S_i)) = min(log(S_i))$$

where

$$log(S_i) = log(S_i(0)) + \left[r - \frac{1}{2}\sigma_i^2\right] T + \sigma_i\sqrt{T}\sum_{j=1}^{d} L_{i,j} Z_j$$

, which can be propagated up to the condition given in Eq. 8. Therefore, the exponential calculation is only applied to finalize the calculation of the value of S_i that corresponds to the minimum price when the condition in Eq. 8 is satisfied. This reduces the number of required exponential calculations by a factor equal to at least the number of assets d (since not all paths satisfy the condition).

Finally, the calculation of $log(S_i(0))$ and $log(K)$ are offloaded to the host, as they are only required once.

The same applies to the maximum reduction scheme, and the preceding optimization can be performed similarly. The average reduction scheme, however, needs all values; therefore, the optimization does not apply in that case.

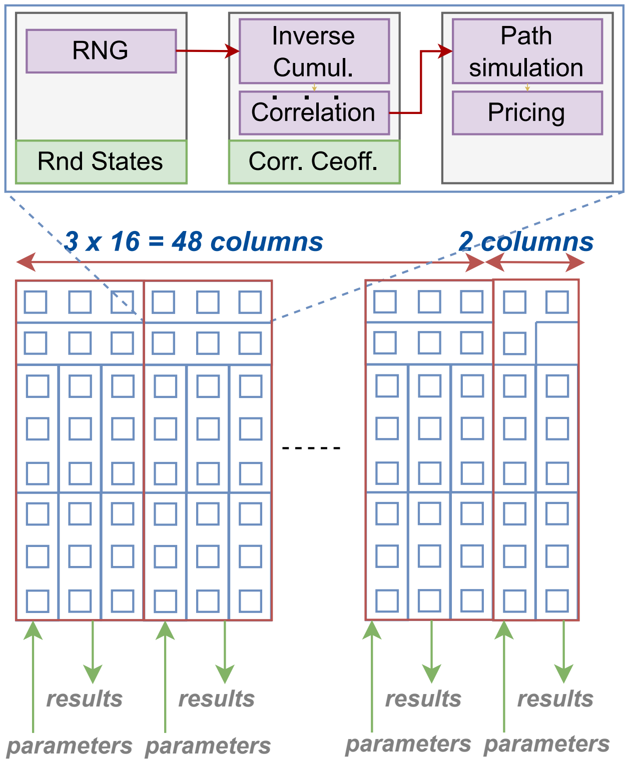

Fig. 6: Dataflow overlay arrangement.

B. Mapping CUs to the AIE Array

The MC simulation is highly parallel (Section II), and path simulation can be parallelized across multiple CUs. Scaling the dataflow design from one CU (3 AIEs) to multiple CUs increases the throughput.

The AIE array comprises 400 AIEs in an 8×50 grid, interconnected by AXI4-interconnects. The dataflow design scales by overlaying 133 CUs across 399 AIEs, with 8 CUs occupying 3 columns and 5 CUs occupying 2 columns for maximum utilization. As shown in [48] and [49], broadcasting efficiently uses the AXI4 network. Despite shared problem parameters (M, N, K, T, d, $L_{i,j}$, $S_i(0)$, r, σ_i), each CU cluster receives a stream containing the parameters separately and broadcasts it over its CUs. This ensures that the parameters streamed from the PL enter from a nearby PL interface tile, reducing propagation delays and routing complexity.

C. Interfacing with PL and automated tiling on the host

The MC multi-option pricing dataflow design, running on 133 CUs (Section III-B), interfaces with the PL and host as shown in Fig. 7. The host configures the communication and computation kernels. As the PL is primarily used for data movement and caching in the AIE literature [48], [50]–[53], it is mainly used in this design to stream the parameters and the results. This is because offloading additional computation to the PL would increase traffic at the PL-AIE interface, degrading the overall performance. Besides, it limits the portability of the designs to AIE architectures without a PL (e.g., Ryzen-NPU). In this overlay design, the PL streams SFMT random number seeds and problem parameters into the AIE array. The seed generation represents the state initialisation discussed in Section III-A1 and is performed on the PL to overlap state generation with communication to the AIE buffers. Additionally, the AIE-PL interface connection can be configured as a 32-bit or 128-bit stream. In this design, AIE-PL is 32 bits wide, given the small amount of data movement between AIE and PL. This is because 128-bit AIE-PL streams consume 4×32-bit interconnects, overutilizing the available interconnect for a small data transfer.

Eight seed generators on the PL are used in this design. Each generator generates the seeds of a particular CU in each cluster. This separation of the generators is used to keep the PL kernels simple for faster implementation. The eight generators, indexed by c, each receive a set of values $\{seeds_c = host\_seed_i^{(c)}, i \in [1,16]\}$ from the host and stream a $4 \times N$ 32-bit sequence $\left(u_n^{(i,c)}\right)$ defined in Eq. 9.

$$\left(u_n^{(i,c)}\right)_{1,\ldots,4\times N} : u_n^{(i,c)} = host\_seed_i^{(c)} + n \times i \qquad \forall i \in [1,16], \forall c \in [1,8] \tag{9}$$

Notice that every generator generates seeds for $16 \times 8 = 128$ CUs.

Five smaller seed generators are used for the distinct 5-CU cluster. The remaining five generators operate using $\left(u_n^{(i,c)}\right)$ similarly except that $i = 1,$ and $c \in [1,5]$.

The rest of the parameters, i.e. the number of paths, time steps, and assets, are similarly streamed through the PL to the CUs, except that they require no additional computation. The outputs of the CUs are streamed out through the PL, and their mean is calculated on the host to determine the final result.

The host automates the launching for different values of simulated paths M and assets d. As the paths are independent, they are equally distributed across the CUs, with each CU assigned $\frac{M}{\#compute\_units}$ path. This distribution is arbitrary, as the paths are independent, and their evolution depends on the multivariate calculated at runtime within the same CU.

The number of assets is another parameter for automation. Recall from Section I that three assets are typically considered large [13]. This design supports up to 8 assets priced in parallel using SIMD operations of the AIEs. To maintain the SIMD structure, unused asset dimensions must not contribute to the final payoff. Recall the payoff from Eq. 1 Payoff $=$ $\max\{0, S(T) - K\}$ where $S(T) = min(S_i(T))$. Unused asset dimension must not affect the value of $S(T) = min(S_i(T))$. With the minimum reduction scheme, large enough values of $S_i(T), \forall i \in UnDim$, where $UnDim$ is the set of the unused dimensions, ensure this. By looking at Eq. 2,

$$S_i(T) = S_i(0) \exp\left(\left[r - \frac{1}{2}\sigma_i^2\right] T + \sigma_i \sqrt{T} \sum_{j=1}^{d} L_{i,j} Z_j\right)$$

, having a large enough $S_i(0)$ eliminates the effect of $S_i(T)$ in the reduction. However, this dimension could still impact the other dimensions when calculating multivariate values $\sum_{j=1}^{d} \sigma_{i,j} Z_j$. To eliminate this, the following constraint is employed: $\sigma_i = 0, \forall i \in UnDim$, which nullifies $\sigma_i \sqrt{T} \sum_{j=1}^{d} L_{i,j} Z_j$. When the maximum reduction scheme is employed, the same method is preserved while choosing small enough $S_i(0), \forall i \in UnDim$ values. For the average reduction scheme, it is sufficient to nullify $S_i(0), \forall i \in UnDim$ values. These constraints are automatically tuned on the host, and the final parameter values are sent to the CUs.

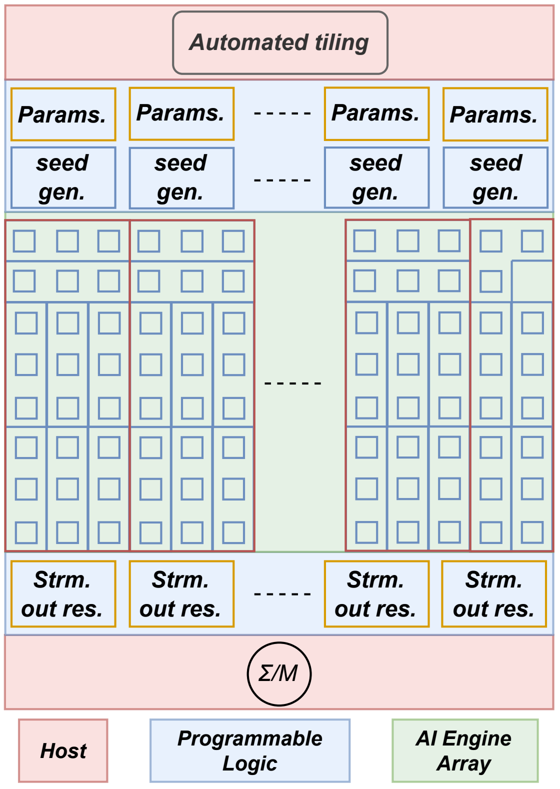

Fig. 7: Tasks allocation on the Versal SoC.

TABLE II: Hardware accelerator configuration.

Component	Description
Board Type	AMD Versal ACAP VCK5000
Device	XCVC1902-VSVD1760-2MP-E-S
AI Engine	400× 1st gen. AIE
Vitis Toolkit	Vitis 2022.1

IV. Experimental Results

In this section, the performance of the MC multi-asset option pricing dataflow design on the AIEs is compared against a traditional parallel dataflow FPGA implementation, multi-threaded parallel implementation on CPU, and GPU implementation. The details of the AMD Versal platform used to implement the AIE dataflow and the PL dataflow are given in Table II. The system details used for the implementations on the CPU and GPU are given in Table III.

The Vitis Quantitative Finance Library [30] is used to build the parallel dataflow FPGA implementation on the PL as

TABLE III: System configuration.

Component	Description
CPU	AMD EPYC 7763 @2.45 GHz with 64-Core-128-Thread
Memory	8× 64 GB RDIMM DDR4 @3.2 GHz
GPU	Nvidia Ampere RTX A6000
OS	Ubuntu Server 20.04
Compiler	GNU C++ Compiler with OpenMP 4.5

TABLE IV: Used components from the Vitis Quant. Fin. Lib.

Component	Layer	Description
MT19937	L1	MT random number generator
inverseCumulativeNormalPPND7	L1	Inverse Cumulative transform using [47]
MT19937IcnRng	L1	Univariate random number generator
MultiVariateNormalRng	L1	Multivariate random number generator
mcSimulation	L1	Monte-Carlo Framework implementation

detailed in Section IV-A.

Besides, OpenMP is used for building a multi-threaded parallel implementation on the CPU, as detailed in Section IV-B. Performance in speedup and differences in scaling to a higher degree of parallelism are given.

Finally, the relative speedups of the AIEs over the PL, CPU, and GPU are compared in Section IV-C.

A. Composing a Multi-Asset Option Pricer on Programmable Logic Using Vitis Quantitative Finance Library

The Vitis Quantitative Finance Library [30] is a High Level Synthesis (HLS) library designed to build complete FPGA accelerators for quantitative finance applications in C++. It is an open-source library that delivers components structured at three levels. The foundational level (L1) offers essential modules and functions for statistical calculations, numerical methods, and linear algebra that enable implementation of models. This includes modules such as random number generators, MC simulations, singular value decomposition, and matrix solvers. The middle level (L2) provides pre-defined hardware kernels for feature pricing engines provided as kernels for evaluating different financial derivatives, such as equity, interest rate, foreign exchange, and credit products. The top-level (L3) comprises software APIs that interact with hardware overlays.

To implement the multi-option pricing dataflow for the GBM model, the components detailed in Table IV and Table V are employed. The components given in Table IV are imported from the library as they are, while those in Table V are adapted to build the missing components for the GBM model.

The components `MT19937`, `inverseCumulativeNormalPPND7`, `MT19937IcnRng`, and `MultiVariateNormalRng` are used to generate a correlated multivariate random number generator by generating uniform random numbers, transforming them into normal random numbers, building independent univariates, and transforming them into correlated multivariates respectively. `MultiAssetHestonPathGenerator` allows multi-asset Heston path simulation. Its structure is adapted to design a similar component called `MultiAssetGBMPathGenerator` that enables path simulation

TABLE V: Adapted components from Vitis Quant. Fin. Lib.

Component/New Component	Layer	Description
MultiAssetHestonPathGenerator → MultiAssetGBMPathGenerator	L1	Multi-asset Heston path simulation engine
MultiAssetPathPricer → MultiAssetPathPricer_redEXP	L1	Pricing engine
MCEuropeanEngine → MCMultiAssetEuropeanGBMEngine	L2	European option pricing engine

TABLE VI: Resource utilization on FPGA PL.

	1× CU dataflow	12 × CUs dataflow	Scaling Factor
Slices	21K (18.68%)	108.8K (96.80%)	5.18×
FF	103.4K (5.75%)	528.8K (29.38%)	5.11×
LUT	90.6K (10.07%)	643.9K (71.56%)	7.11×
BRAMs	46.5 (4.81%)	91.5 (9.46%)	1.96×
DSPs	82 (4.17%)	729 (37.04%)	8.9×
Freq (MHz)	300	211.4	0.7×

for the GBM model. `MultiAssetPathPricer` allows the pricing of simulated paths. It is modified to a simpler component `MultiAssetPathPricer_redEXP` that calculates the payoff of the GBM model, while the reduction scheme is pulled upstream to the `MultiAssetGBMPathGenerator`. This allows the use of `mcSimulation` that composes all these components to build the desired CU. Finally, `MCEuropeanEngine` is a component that enables the European option pricing by initializing all the components and launching them. It is modified to a new component `MCMultiAssetEuropeanGBMEngine` to initialize and operate the designed CU for the GBM model.

`mcSimulation` is designed to implement a dataflow of parallel CUs. Table VI shows the resource utilization of the two implementations. The first instantiates one CU, while the second instantiates twelve parallel CUs. Although all the scaling factors are below 12, the second implementation is empirically the maximum achievable level of parallelism that could be implemented on the PL due to high logic utilization.

Contrary to the AIEs, that maintain a constant frequency of 1 GHz, the elevated congestion when parallelizing the PL design results in a reduction in frequency by 30%, as shown in Table VI. Despite this reduction, the parallelized dataflow maintains a linear scaling with the number of simulated paths and is, on average, 6.59× faster than the single-CU dataflow accelerator, as illustrated in Fig. 8. Therefore, it is used for comparison for the rest of the experimental results.

B. Parallel CPU-based Multi-Asset Option Pricer

The HLS design in C++ used to design the dataflow on the PL using the Vitis Quantitative Finance Library, is based on QuantLib for CPUs, detailed in Section IV-A. We refactored this to implement a multi-threaded design for CPU using OpenMP in C++.

Parallel CUs are now each assigned to a parallel thread, and the instantiation and initialization of composing components (as shown in Fig. 2) is pushed downstream to `mcSimulation`. This ensures that by giving each thread a private copy of the required parameters, there are no competing threads

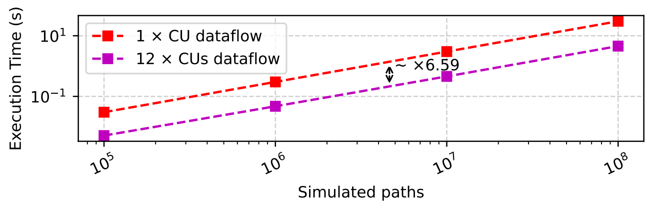

Fig. 8: Execution time on PL implementation.

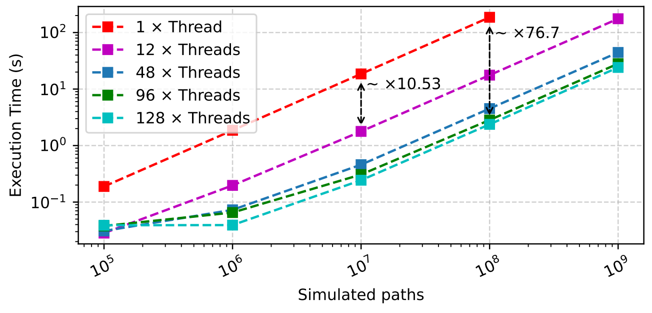

Fig. 9: Execution time on multi-threaded CPU.

TABLE VII: Performance comparison to existing works.

	Platform	**Volume (Asset × Path × Step)**	**Exec. time**	**Rate**
Tian et al. (2013) [54]	Virtex-4 XV4FX100	$1 \times 1 \times 1$	13.3 ns	$75.18 \cdot 10^6$
Valera et al. (2015) [55]	Zynq ZC702	$2 \times 10^3 \times 365$	16.94 ms	$43.09 \cdot 10^7$
This work	**AMD Versal AIEs**	$\mathbf{8 \times 10^8 \times 1}$	**177.07 ms**	$45.17 \cdot 10^8$

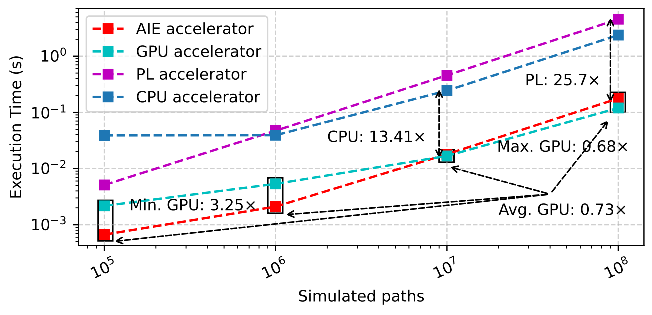

Fig. 10: Execution time of 1 timesteps over multiple paths.

reading from the same memory locations. The private copy of parameters is ensured using the directive `firstprivate` in the OpenMP pragma that defines the parallel region. Besides, using the OpenMP directive `reduction`, the sum of payoffs is reduced to one sum at the end of the threads execution.

The multi-threaded design runs over up to 128 parallel threads, which is the physical limit of the CPU (Table III). As shown in Fig. 9, the speedup is, on average, 10.53× from a single thread design to a 12-thread design, which is a higher speedup compared to the 1 × CU dataflow to 12 × CUs FPGA implementation in Section IV-A. This is because CPU frequency remains constant as the parallelism scales.

Finally, the speedup for the 128-thread CPU implementation is, on average, 76.7× over the single-threaded version. Scaling with the number of simulated paths maintains a steady linear increase starting from 10^6 paths. Therefore, this is used for comparison for the rest of the experimental results.

C. Performance Evaluation

The MC simulation process independently launches multiple simulation paths, each over multiple time steps. This represents linear complexity over the number of simulated paths and over the number of time steps.

Fig. 10 and Fig. 11 demonstrate that our dataflow design, built on the AIEs, scales linearly with the number of simulated paths and time steps, respectively. Besides, they demonstrate that our design is 12.9× and 25.7× faster than the dataflow design on the PL built using the Vitis Quantitative Finance Library when running on different numbers of simulated paths and time steps, respectively. Moreover, our design is 10.66× and 13.41× faster than the parallel multi-threaded CPU implementation over the same parameters.

We adapted a GPU implementation from [29] and [56] for multi-asset option pricing using the GBM model and optimized it by storing parameters in read-only constant memory for efficient broadcasting to all threads. Our design is up to 3.25× faster for simulations with fewer than 10^7 paths. The GPU requires 0.73× on average to 0.68× the time of our dataflow design for the same paths and 0.45× for the same timesteps, making it faster. However, power measurements using NVIDIA Nsight Systems and post-routing reports of the AIEs show our design is 1.82× more energy-efficient than the GPU, as summarized in Table VIII.

Comparing speedup across designs with differing parallelism levels provides insights into theoretical limits. This is computed using

$$max\_speedup_{ij} = speedup \times \frac{para\_cu(design_i)}{para\_cu(design_j)}$$

, where $para\_cu$ represents the number of parallel compute units: $para\_cu(AI\ Engines) = 133$, $para\_cu(CPU) = 128$, $para\_cu(PL) = 12$.

Using this model, the PL design speedups of 12.9× and 25.7× adjust to 1.16× and 2.32×. This shows that with a larger PL fabric matching the number of CUs we implement on the AIEs (assuming no further frequency drops), our dataflow design remains 1.16× to 2.32× faster. For the multi-threaded CPU, speedups of 10.66× and 13.41× adjust to 10.26× and 12.9×. This highlights our design's efficiency against larger PL resources or more CPU cores. As the GPU does not implement parallel CUs as is the case with the other designs due to the SIMT execution model, the model changes to $\frac{para\_cores(design_i)}{para\_cores(design_j)}$, where $para\_cores(GPU) = 10752$, and $para\_cores(AI\ Engines) = 3192$. 10752 is the number of CUDA cores while 3192 is the number of utilized AIE cores multiplied by the number of SIMD lanes per core. This adjusts

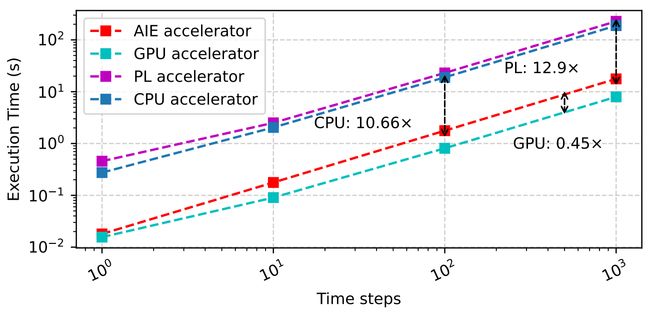

Fig. 11: Execution time of 10^8 paths over multiple time steps.

TABLE VIII: Energy efficiency comparison.

	GPU	**AIEs**
Power consumption (W)	279.1	103.2
Average energy consumption (J)	33.40	18.31
Energy efficiency	1×	1.82×

the GPU speedups of 0.45× and 0.68× to 1.52× and 2.29×, showing a dataflow design on the AIEs with equivalent parallel capacity would outperform the GPU. Note that with the same size AIE array, finer pipelining of the computation could further benefit performance. However, this would break the modular design shape shown in Fig. 6, which is what enables filling the grid of 50×8 AIE cores.

Finally, existing works on PL accelerators [54], [55] targeted rather simpler models mainly by pricing multiple single-asset options. Single-asset option pricing involves one option per asset, whereas multi-asset pricing uses multiple assets to price one option. The difference in their pricing is the correlation of the multivariate, which only applies to the multi-asset case. Table VII compares these simpler cases with our work using the metric *Volume* = *Asset* × *Path* × *Step*, which characterizes the simulation workload for a given execution time. The *Rate*, defined as $\frac{Volume}{Execution\ time}$, represents the simulation workload processed per second. Despite their simpler tasks, our dataflow design outperforms these works on the single-asset problem by 10.48× to 60.08×.

V. Related Work

Several works have focused on accelerating option pricing using MC simulations on CPUs and GPUs. [57] proposed a parallel MC option pricing approach for European and American options, leveraging heterogeneous many-core architectures with CPUs and GPUs. Their model achieved parallelism by distributing workloads across distributed computing infrastructures using both multi-core CPUs and many-core accelerators. Besides, [56] demonstrates CUDA-based acceleration for exotic single-asset option pricing on Nvidia GPUs, while [29] compares MC option pricing on GPUs using manually optimized CUDA [58] and OpenCL [59] kernels against HMPP [60] and OpenACC [61].

On FPGAs, MC simulation has been subject to many studies [62]–[65] and used in many applications [66]–[80] including financial applications [81]–[87]. For option pricing, the acceleration of Quasi-MC option pricing on an Altera Startix III for a limited number of time steps and paths for a single stock option was explored in [88]. Parallel acceleration on multiple Xilinx Virtex-4 FPGAs using MC and Quasi-MC methods for a single stock option for American, European, and Asian options was explored in [54], [89], [90]. An optimization framework and domain-specific language, Contessa, were proposed in [91]–[93] for automatic generation and optimization of MC simulators on FPGAs for financial applications. An implementation of MC integration using stratified sampling for an Asian option on a single stock using a Virtex-6 FPGA was proposed in [94]. The approach involved dataflow and pipelined processing, achieving good precision and handling a high number of time steps. All these works do not handle the multi-asset option pricing case; rather, they only target single-asset options. In addition, they tailor their optimizations to smaller FPGAs than the AMD Versal board. Besides, as they do not tackle the multi-asset case, these works lack the automation to handle different numbers of correlated assets. With regard to High-Level Synthesis (HLS), [7] demonstrated the suitability of employing HLS for accelerating the MC option pricing by applying a series of optimizations such as task-level parallelism loop pipelining. The Vitis Quantitative Finance Library [30] is an example of a well-established modular HLS framework for quantitative finance applications. While it is inspired by the QuantLib library for CPUs [43], it leverages specific optimizations for AMD FPGAs.

VI. Conclusion

In this work, we presented a highly parallel implementation of a Monte Carlo option pricing dataflow overlay tailored specifically for AMD Versal AIEs, specifically targeting multi-asset option pricing. We leveraged the specialized vectorized operation capabilities of the AIE cores, split the various stages of the option pricing application, and arranged these spatially. The proposed design showcases the scalability of parallel compute units, providing an optimized framework for accelerating Monte Carlo multi-asset option pricing utilizing 399 of the 400 available AIE cores. We demonstrated that our dataflow design achieves 25.7× speedup over a parallel dataflow design in programmable logic using the AMD Vitis Quantitive Finance Library and a 13.41× speedup over a highly parallelized CPU implementation using 128 threads. We also demonstrated that although the GPU requires 0.73× the execution time of our design, our design achieves 1.82× better energy efficiency.

In future work, we aim to extend this implementation to more quantitative finance kernels to develop a specialized library on AMD Versal AIEs, similar to the Vitis Quantitative Finance Library for FPGAs. We also plan to template the design to accommodate non finance kernels, to generalize to a framework of MC simulation on AMD Versal AIEs.

References

[1] M. Haugh and A. Lo, "Computational challenges in portfolio management," *Computing in Science & Engineering*, vol. 3, no. 3, pp. 54–59, 2001.

[2] A. Sedighi and D. Jacobson, "Computational challenges and opportunities in financial services," in *International Conference on Smart Computing and Communication (ICSCC)*, 2019.

[3] J. C. Hull and S. Basu, *Options, futures, and other derivatives*. Pearson Education India, 2016.

[4] B. Oksendal, *Stochastic differential equations: an introduction with applications*. Springer Science & Business Media, 2013.

[5] S. Liu, A. Borovykh, L. A. Grzelak, and C. W. Oosterlee, "A neural network-based framework for financial model calibration," *Journal of Mathematics in Industry*, vol. 9, no. 9, pp. 9:1–9:28, 2019.

[6] J. Rosen, C. Kahl, R. Goyder, and M. Gibbs, "Computationally expensive problems in investment banking," in *High-Performance Computing in Finance*. Chapman and Hall/CRC, 2018, pp. 3–24.

[7] G. Inggs, S. Fleming, D. B. Thomas, and W. Luk, "Is high level synthesis ready for business? an option pricing case study," *FPGA Based Accelerators for Financial Applications*, pp. 97–115, 2015.

[8] D. McLean, "Challenges in scenario generation: Modeling market and non-market risks in insurance," in *High-Performance Computing in Finance*. Chapman and Hall/CRC, 2018, pp. 115–171.

[9] R. Carmona and V. Durrleman, "Pricing and hedging spread options," *Siam Review*, vol. 45, no. 4, pp. 627–685, 2003.

[10] S. Borovkova, F. J. Permana, and H. v. Weide, "A closed form approach to the valuation and hedging of basket and spread options," *Journal of Derivatives*, vol. 14, no. 4, pp. 4:1–4:18, 2007.

[11] R. Caldana, G. Fusai, A. Gnoatto, and M. Grasselli, "General closed-form basket option pricing bounds," *Quantitative Finance*, vol. 16, no. 4, pp. 535–554, 2016.

[12] A. Hirsa, *Computational methods in finance*. CRC Press Boca Raton, FL, 2013.

[13] D. J. Duffy, *Finite difference methods in financial engineering: a partial differential equation approach*. John Wiley & Sons, 2013.

[14] D. Duffie, J. Pan, and K. Singleton, "Transform analysis and asset pricing for affine jump-diffusions," *Econometrica*, vol. 68, no. 6, pp. 1343–1376, 2000.

[15] P. Boyle, M. Broadie, and P. Glasserman, "Monte Carlo methods for security pricing," *Journal of economic dynamics and control*, vol. 21, no. 8-9, pp. 1267–1321, 1997.

[16] D. Elbrächter, P. Grohs, A. Jentzen, and C. Schwab, "DNN expression rate analysis of high-dimensional PDEs: application to option pricing," *Constructive Approximation*, vol. 55, no. 1, pp. 3–71, 2022.

[17] C. Reisinger and G. Wittum, "Efficient hierarchical approximation of high-dimensional option pricing problems," *SIAM Journal on Scientific Computing*, vol. 29, no. 1, pp. 440–458, 2007.

[18] A. Quarteroni and S. Quarteroni, *Numerical models for differential problems*. Springer, 2009.

[19] R. E. Bellman, "Dynamic programming," *science*, vol. 153, no. 3731, pp. 34–37, 1966.

[20] E. Eberlein, K. Glau, and A. Papapantoleon, "Analysis of Fourier transform valuation formulas and applications," *Applied Mathematical Finance*, vol. 17, no. 3, pp. 211–240, 2010.

[21] C. Leentvaar and C. W. Oosterlee, "Multi-asset option pricing using a parallel Fourier-based technique," *Journal of Computational Finance*, vol. 12, no. 1, pp. 1–26, 2008.

[22] C. Bayer, C. Ben Hammouda, A. Papapantoleon, M. Samet, and R. Tempone, "Optimal damping with a hierarchical adaptive quadrature for efficient Fourier pricing of multi-asset options in Lévy models," *Journal of Computational Finance*, 2023.

[23] C. Bayer, C. B. Hammouda, A. Papapantoleon, M. Samet, and R. Tempone, "Quasi-Monte Carlo with domain transformation for efficient Fourier pricing of multi-asset options," *arXiv preprint arXiv:2403.02832*, 2024.

[24] F. Y. Kuo and I. Sloan, "Lifting the curse of dimensionality," *Notices of the AMS*, vol. 52, no. 11, pp. 1320–1328, 2005.

[25] M. Holtz, *Sparse grid quadrature in high dimensions with applications in finance and insurance*. Springer Science & Business Media, 2010.

[26] P. Glasserman, *Monte Carlo methods in financial engineering*. Springer, 2004.

[27] M. B. Giles, "Multilevel Monte Carlo path simulation," *Operations research*, no. 3, pp. 607–617, 2008.

[28] K. Spanderen, "Beyond simple Monte-Carlo: Parallel computing with QuantLib," https://www.quantlib.org/slides/qlws13/spanderen.pdf, 2013.

[29] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos, "Accelerating financial applications on the gpu," in *Workshop on General Purpose Processor Using Graphics Processing Units*, 2013.

[30] "Vitis quantitative finance library," https://docs.amd.com/r/en-US/Vitis_-Libraries/quantitative_finance/index.html, 2022.

[31] K. Glau and L. Wunderlich, "The deep parametric PDE method and applications to option pricing," *Applied Mathematics and Computation*, vol. 29, no. 1, pp. 440–458, 2022.

[32] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of political economy*, vol. 81, no. 3, pp. 637–854, 1973.

[33] P. Tankov, *Financial modelling with jump processes*. Chapman and Hall/CRC, 2003.

[34] L. Bergomi, *Stochastic volatility modeling*. CRC press, 2015.

[35] C. Bayer, P. Friz, and J. Gatheral, "Pricing under rough volatility," *Quantitative Finance*, vol. 16, no. 6, pp. 887–904, 2016.

[36] C. Bayer, M. Eigel, L. Sallandt, and P. Trunschke, "Pricing high-dimensional Bermudan options with hierarchical tensor formats," *SIAM Journal on Financial Mathematics*, vol. 14, no. 2, pp. 383–406, 2023.

[37] D. Abts, J. Kim, G. Kimmell, M. Boyd, K. Kang, S. Parmar, A. Ling, A. Bitar, I. Ahmed, and J. Ross, "The groq software-defined scale-out tensor streaming multiprocessor: From chips-to-systems architectural overview," in *IEEE Hot Chips Symposium (HCS)*, 2022.

[38] M. Emani, V. Vishwanath, C. Adams, M. E. Papka, R. Stevens, L. Florescu, S. Jairath, W. Liu, T. Nama, and A. Sujeeth, "Accelerating scientific applications with SambaNova reconfigurable dataflow architecture," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 114–119, 2021.

[39] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versaltm architecture," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[40] GitHub Repoistory, 2025. [Online]. Available: https://github.com/accl-kaust/mc-option-pricing-aie

[41] M. Saito and M. Matsumoto, "SIMD-oriented fast Mersenne twister: a 128-bit pseudorandom number generator," in *Monte Carlo and Quasi-Monte Carlo Methods*. Springer, 2006, pp. 607–622.

[42] "Versal Adaptive SoC AI Engine architecture manual (am009)," https://docs.amd.com/r/en-US/am009-versal-ai-engine, 2023.

[43] N. Firth, "Why use QuantLib," vol. 34, 2004.

[44] M. Bouaziz and S. A. Fahmy, "PRNGine: Massively parallel pseudo-random number generation and probability distribution approximations on AMD AI Engines," in *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2025.

[45] "AI Engine kernel and graph programming guide (ug1079)," https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding, 2022.

[46] P. J. Acklam, "An algorithm for computing the inverse normal cumulative distribution function," *University of Oslo, Statistics Division*, vol. 37, no. 3, pp. 477–484, 2000.

[47] M. J. Wichura, "Algorithm AS 241: The percentage points of the normal distribution," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 37, no. 3, pp. 477–484, 1988.

[48] J. Zhuang, J. Lau, H. Ye, Z. Yang, Y. Du, J. Lo, K. Denolf, S. Neuendorffer, A. Jones, J. Hu, D. Chen, J. Cong, and P. Zhou, "CHARM: Composing heterogeneous accelerators for matrix multiply on versal ACAP architecture," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2023.

[49] G. Singh, A. Khodamoradi, K. Denolf, J. Lo, J. Gomez-Luna, J. Melber, A. Bisca, H. Corporaal, and O. Mutlu, "Sparta: Spatial acceleration for efficient and scalable horizontal diffusion weather stencil computation," in *International Conference on Supercomputing (ICS)*, 2023.

[50] J. Zhuang, J. Lau, H. Ye, Z. Yang, S. Ji, J. Lo, K. Denolf, S. Neuendorffer, A. Jones, J. Hu, Y. Shi, D. Chen, J. Cong, and P. Zhou, "CHARM 2.0: Composing heterogeneous accelerators for deep learning on Versal ACAP architecture," *ACM Transactions on Reconfigurable Technology and Systems*, 2024.

[51] Z. Yang, J. Zhuang, J. Yin, C. Yu, A. K. Jones, and P. Zhou, "AIM: Accelerating arbitrary-precision integer multiplication on heterogeneous reconfigurable computing platform Versal ACAP," in *International Conference on Computer Aided Design (ICCAD)*, 2023.

[52] C. Zhang, T. Geng, A. Guo, J. Tian, M. Herbordt, A. Li, and D. Tao, " H-GCN: A graph convolutional network accelerator on Versal ACAP architecture," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[53] J. Zhuang, Z. Yang, and P. Zhou, "High performance, low power matrix multiply design on ACAP: from architecture, design challenges and DSE perspectives," in *Design Automation Conference (DAC)*, 2023.

[54] X. Tian and K. Benkrid, "Monte-Carlo simulation-based financial computing on the Maxwell FPGA parallel machine," *High-performance computing using FPGAs*, pp. 33–80, 2013.

[55] J. A. Varela, C. Brugger, S. Tang, N. Wehn, and R. Korn, "Pricing high-dimensional American options on hybrid CPU/FPGA systems," *FPGA Based Accelerators for Financial Applications*, pp. 143–166, 2015.

[56] Y. Dong, "Accelerating python for exotic option pricing," https://developer.nvidia.com/blog/accelerating-python-for-exotic-option-pricing/, 2020.

[57] S. Zhang, Z. Wang, Y. Peng, B. Schmidt, and W. Liu, "Mapping of option pricing algorithms onto heterogeneous many-core architectures," *The Journal of Supercomputing*, vol. 73, pp. 3715–3737, 2017.

[58] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *International symposium on Memory management (ISMM)*, 2007.

[59] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous computing with openCL: revised openCL 1.2*. Newnes, 2012.

[60] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A hybrid multi-core parallel programming environment," in *Workshop on general purpose processing on graphics processing units*. Citeseer, 2007.

[61] R. Farber, *Parallel programming with OpenACC*. Newnes, 2016.

[62] Y. Meng, R. Kannan, and V. K. Prasanna, "A framework for Monte-Carlo tree search on CPU-FPGA heterogeneous platform via on-chip dynamic tree management," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2023.

[63] ——, "Accelerating Monte-Carlo tree search on CPU-FPGA heterogeneous platform," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[64] T. Wang, W. Chang, A. Srivastava, R. Kannan, and V. Prasanna, "Monte Carlo tree search for task mapping onto heterogeneous platforms," in *International Conference on High Performance Computing, Data, and Analytics, (HiPC)*, 2021.

[65] E. Qasemi, A. Samadi, M. H. Shadmehr, B. Azizian, S. Mozaffari, A. Shirian, and B. Alizadeh, "Highly scalable, shared-memory, Monte-Carlo tree search based blokus duo solver on FPGA," in *International Conference on Field-Programmable Technology (FPT)*, 2014.

[66] M. W. Hassan, H. Ltaief, and S. A. Fahmy, "High throughput massive MIMO signal decoding using multi-level tree search on FPGAs," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2024.

[67] Y. Wang and P. Li, "Algorithm and hardware co-design for FPGA acceleration of hamiltonian Monte Carlo based no-u-turn sampler," in *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021.

[68] F. Ortega-Zamorano, M. A. Montemurro, S. A. Cannas, J. M. Jerez, and L. Franco, "FPGA hardware acceleration of Monte Carlo simulations for the ising model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2618–2627, 2016.

[69] S. Hung, M. Tsai, B. Huang, and C. Tu, "A platform-oblivious approach for heterogeneous computing: A case study with Monte Carlo-based simulation for medical applications," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[70] X. Tian and C. Bouganis, "A run-time adaptive FPGA architecture for Monte Carlo simulations," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[71] P. Kinsman and N. Nicolici, "NoC-based FPGA acceleration for Monte Carlo simulations with applications to SPECT imaging," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 524–535, 2013.

[72] Y. Lin, F. Wang, X. Zheng, H. Gao, and L. Zhang, "Monte Carlo simulation of the Ising model on FPGA," *Journal of computational Physics*, vol. 237, pp. 224–234, 2013.

[73] G. C. T. Chow, A. H. T. Tse, Q. Jin, W. Luk, P. H. W. Leong, and D. B. Thomas, "A mixed precision Monte Carlo methodology for reconfigurable accelerator systems," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2012.

[74] J. H. C. Yeung, E. F. Y. Young, and P. H. W. Leong, "A Monte-Carlo floating-point unit for self-validating arithmetic," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2011.

[75] H. Yuasa, H. Tsutsui, H. Ochi, and T. Sato, "A fully pipelined implementation of Monte Carlo based SSTA on FPGAs," in *International Symposium on Quality Electronic Design (ISQED)*, 2011.

[76] J. Cong, K. Gururaj, W. Jiang, B. Liu, K. Minkovich, B. Yuan, and Y. Zou, "Accelerating Monte Carlo based SSTA using FPGA," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2010.

[77] M. Smerdis, P. Malakonakis, and A. Dollas, "CarlOthello : An FPGA-based Monte Carlo Othello player," in *International Conference on Field-Programmable Technology (FPT)*, 2010.

[78] J. Luu, K. Redmond, W. Lo, P. Chow, L. Lilge, and J. Rose, "FPGA-based Monte Carlo computation of light absorption for photodynamic cancer therapy," in *International Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2009.

[79] A. Gothandaraman, G. D. Peterson, G. L. Warren, R. J. Hinde, and R. J. Harrison, "FPGA acceleration of a quantum Monte Carlo application," *Parallel Computing*, vol. 34, no. 4–5, pp. 278–291, 2008.

[80] A. Ejlali and S. G. Miremadi, "FPGA-based Monte Carlo simulation for fault tree analysis," *Microelectron. Reliab.*, vol. 44, no. 6, pp. 1017–1028, 2004.

[81] X. Tian, K. Benkrid, and X. Gu, "High performance Monte-Carlo based option pricing on FPGAs," *Engineering Letters*, vol. 16, no. 3, pp. 434–442, 2008.

[82] D. Sanchez-Roman, V. Moreno, S. López-Buedo, G. Sutter, I. González, F. J. Gomez-Arribas, and J. Aracil, "FPGA acceleration using high-level languages of a Monte-Carlo method for pricing complex options," *Journal of Systems Architecture*, vol. 59, no. 3, pp. 135–143, 2013.

[83] C. de Schryver, P. Torruella, and N. Wehn, "A multi-level Monte Carlo FPGA accelerator for option pricing in the Heston model," in *Design, Automation and Test in Europe, (DATE)*, 2013.

[84] C. de Schryver, I. Shcherbakov, F. Kienle, N. Wehn, H. Marxen, A. Kostiuk, and R. Korn, "An energy efficient FPGA accelerator for Monte Carlo option pricing with the heston model," in *International Conference on Reconfigurable Computing and FPGAs, (ReConFig)*, 2011.

[85] A. Kaganov, P. Chow, and A. Lakhany, "FPGA acceleration of Monte-Carlo based credit derivative pricing," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2008.

[86] X. Tian and K. Benkrid, "Massively parallelized quasi-Monte Carlo financial simulation on a FPGA supercomputer," in *International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA@SC)*, 2008.

[87] D. Diamantopoulos, R. Polig, B. Ringlein, M. Purandare, B. Weiss, C. Hagleitner, M. A. Lantz, and F. Abel, "Acceleration-as-a-μservice: A cloud-native Monte-Carlo option pricing engine on CPUs, GPUs and disaggregated FPGAs," in *International Conference on Cloud Computing (CLOUD)*, 2021.

[88] N. A. Woods and T. VanCourt, "FPGA acceleration of quasi-Monte Carlo in finance," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2008.

[89] X. Tian and K. Benkrid, "Design and implementation of a high performance financial Monte-Carlo simulation engine on an FPGA supercomputer," in *International Conference on Field-Programmable Technology (FPT)*, 2008.

[90] ——, "High-performance quasi-Monte Carlo financial simulation: FPGA vs. GPP vs. GPU," *Transactions on Reconfigurable Technology and Systems*, vol. 3, no. 4, pp. 4:1–4:22, 2010.

[91] D. B. Thomas, J. A. Bower, and W. Luk, "Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations," in *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2007.

[92] D. B. Thomas and W. Luk, "A domain specific language for reconfigurable path-based Monte Carlo simulations," in *International Conference on Field-Programmable Technology (FPT)*, 2007.

[93] D. B. Thomas, "Acceleration of financial Monte-Carlo simulations using FPGAs," in *Workshop on High Performance Computational Finance*, 2010.

[94] M. De Jong, V.-M. Sima, K. Bertels, and D. Thomas, "FPGA-accelerated Monte-Carlo integration using stratified sampling and brownian bridges," in *International Conference on Field-Programmable Technology (FPT)*, 2014.