PRNGine: Massively Parallel Pseudo-Random Number Generation and Probability Distribution Approximations on AMD AI Engines

Mohamed Bouaziz and Suhaib A. Fahmy
King Abdullah University of Science and Technology (KAUST)
Thuwal, Saudi Arabia
mohamed.bouaziz@kaust.edu.sa

***Abstract*—Generating large volumes of random numbers is essential for high-performance computing applications such as Monte Carlo simulations, machine learning, and dynamic gameplay. Many of these applications require random number generation within a processing pipeline. Coarse-Grained Reconfigurable Architectures (CGRAs) are well-suited for this task, enabling efficient dataflow-based distribution across processing elements. This work explores efficient random number generation on AMD AI Engines (AIEs) through two execution models: a co-processor model and a standalone dataflow accelerator model. Key challenges in porting Pseudo-Random Number Generators (PRNGs) to AIEs, including the lack of support for certain operations, unsigned data types, and efficient vectorization, are identified and overcome. Additionally, the challenges of approximating a normal distribution on AIEs are analyzed and addressed. Optimized implementations of essential PRNG operations are presented, demonstrating linear complexity and enabling scalable random number generation. Performance evaluation provides insights into the suitability of both execution models for various applications.**

***Index Terms*—PRNG, AI Engine, CGRA, Dataflow**

I. Introduction

The use of random numbers is crucial in many applications. They are used to simulate uncertainty, ensure fairness, and introduce stochasitcity in fields such as cryptography in secure key generation [1], Monte Carlo (MC) methods for scientific simulations [2], and machine learning for data shuffling and model initialization [3]. This has led to the development of many Pseudo-Random Number Generators (PRNGs) such as the Mersenne Twister generator [4] and the Marsaglia XORSHIFT generator family [5].

Traditional CPU performance does not scale with each new generation of hardware. On the other hand, the need to generate a massive number of random numbers is notable. In quantitative finance, MC simulations require a massive number of random numbers to simulate price evolution in time for financial stocks and derivatives [6], [7]. MC simulations are also used in computational biology, materials science, and chemistry [8]. Parallel MC simulations require independent streams of random numbers. The generators must provide a large period and be able to split into many non-overlapping sub-sequences. Similarly, procedural content generation in video games and graphics uses random numbers to create large-scale game content such as maps, levels, textures, and game mechanics. It often requires high throughput random numbers to produce varied and dynamic gameplay experiences for video games [9]. This need for massive randomness has led to adapting PRNGs to parallel computation on multi-core CPUs and GPUs and removing the dependence between the random sequences, making them inherently parallel [10].

FPGAs are suited to PRNGs due to their fine-grained architecture, which enables the customization of bitwise operations and the creation of LUT-optimized generators that treat the random state as a vector of bits rather than words [11]. Besides, they can implement many parallel transformations, and the output can be formed from any permutation of the generator's state without correlations between consecutive bits. This allows for highly parallel transformations and high throughput generation, making them suitable for massively parallel applications [12].

Coarse-grained architectures offer acceleration by combining both parallelism and programmability. Ambric AM2045 [13] is an example of a coarse-grained architecture that was used with parallel PRNGs leveraging its array of RISC processors that communicate through small memories [12]. This obsolete architecture suffered from a restricted instruction set and limited memory space, which complicated the adaptation of many PRNGs. Recently, the AMD Versal [14] architectures emerged, comprising an array of configurable RISC processors, named the AI Engines (AIEs), highly programmable and optimized for parallel vectorized operations. AIE cores are connected by a high-performance stream interconnect network. Each AIE is equipped with a 32KB Scratchpad Memory, which is shared with neighboring AIEs in the cardinal directions. The high degree of parallelism and reconfigurability reopens directions for high-throughput random number generation on coarse-grained architectures by many of the existing PRNGs. The use of coarse-grained AIEs allows seamless implementation of efficient dataflow designs contrary to fine-grained FPGAs. However, a number of challenges arise when porting PRNGs to the AIEs.

- **Supported operations:** PRNGs often include operations the AIE do not natively support. For instance, most XOROSHIRO generators [15] rely on an operation that rotates a 64-bit binary word by applying a sequence of

shifting of its Most-Significant Bits (MSBs) and Least-Significant Bits (LSBs). Similarly, SFMT [16] requires right and left shifting operations on 128-bit binary words. The challenge is that shifting operations supported by AIEs cannot be performed natively on more than 32-bit words.

Another challenge is that the PRNGs generate random numbers based on an array of states. Depending on the design of the PRNG, the state array may be accessed through an indexing function that requires costly operations. For instance, SFMT [16] uses four state values by looping through an array of 156 states. One of the four states is indexed M (typically 122) steps ahead of the iterator modulo 156. AIEs do not natively support the modulo operation, and the compiler emulates it through a precompiled series of operations. This incurs substantial computational overhead. Besides, it breaks the execution pipeline to branch to the precompiled modulo function.

- **Data types**: The AIEs have two types of engines: scalar engines and vector engines. The vector engines are typically used to perform computation, and they are optimized to perform element-wise operations on 128-bit, 256-bit, 512-bit, and 1024-bit vectors of 32-bit elements. Besides the lack of operation support for binary words larger than 32 bits, the AIEs do not support unsigned data types, which are fundamental for PRNGs.

  Additionally, on AIEs, the outputs of PRNGs can be cast to floating-point numbers by evaluating the generated random number as a decimal number in fixed-point representation and specifying the position of the decimal point. This poses a challenge as the MSB is considered the sign-bit instead of a constituting part of the random value and can alter the quality of the generator. This is also challenging for probability distribution approximations using the Inverse Cumulative Distribution Functions (ICDFs) that expect numbers in $[0, 1)$ (non-negative).
- **Vectorization**: A subset of PRNGs uses states and recursions to generate random numbers, which create dependencies in the generated values. As AIEs are typically optimized for vectorized operations, picking a suitable vectorized PRNG becomes a challenge. Although this bottleneck is not specific to the AIE architecture, it still presents a design constraint to consider.

  Another challenge is the conditional processing of vector elements, where different vector elements may follow different computation paths. This is the case with the Percentage Points algorithm [17] and Acklam's Approximation [18] used to approximate the ICDF of the Normal distribution where different polynomial expressions are applied to the inputs depending on their values. In traditional Programmable logic (PL) implementations, this challenge can be mitigated by designing predicated datapaths. Similarly, this can be achieved natively on SIMD architectures that support predictions such as ARM SVE [19].

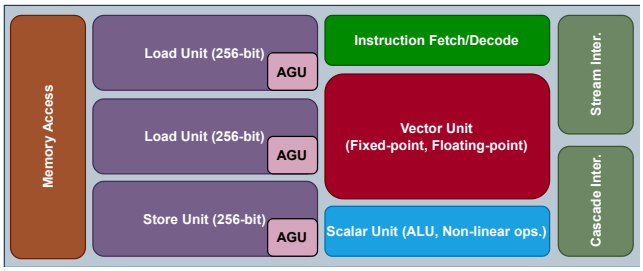


Fig. 1: AI Engine core architecture

This paper addresses the challenges of porting PRNGs to AMD Versal AIEs by analyzing the algorithmic requirements of three PRNGs: XORWOW, XOROSHIRO, and SFMT. We highlight the core functionalities of AIEs and implement the necessary operations using native AIE-API [20] calls. Two execution models are proposed: a co-processor model, where AIEs generate random numbers and send them to the host processor, and a standalone dataflow accelerator model, where random number generation is pipelined with computation within a dataflow overlay on AIEs. We evaluate and compare the performance of these models, demonstrating their scalability. Additionally, we address the challenge of approximating the normal distribution using the ICDF approximation within the dataflow model.

While PRNG performance is often benchmarked on CPUs and GPUs, this work focuses on the design constraints of embedding PRNGs into dataflow architectures implemented on coarse-grained platforms such as AMD AIEs. Unlike traditional Von Neumann architectures, leveraging the AIE cores and their network of interconnections enables spatial acceleration by executing workloads like Monte Carlo simulations in a distributed, pipeline-parallel fashion.

II. Architectural Overview and Algorithmic Requirements

In this section, the AIE core architecture is detailed, highlighting its main scalar and vector datapaths. The three PRNGs are analyzed and discussed in terms of functional requirements in relation to AIE architectural capabilities.

A. AI Engine Core Architecture

AIE cores are optimized for high-performance computing, integrating both single-instruction multiple-data (SIMD) and very-long instruction word (VLIW) capabilities. They support integer, fixed-point, and floating-point precision. The architecture includes various functional units to handle complex computations, such as a 32-bit scalar RISC unit, a vector unit, two load units, one store unit, an instruction fetch and decode unit, and three address generation units (AGUs) supporting multiple addressing modes.

The scalar unit in the AIE features an arithmetic logic unit (ALU) and supports non-linear functions such as square root, sine, cosine, and inverse square root. Its ALU supports 32-bit scalar operations and enables data type conversion between scalar fixed-point and floating-point with a one-cycle latency.

TABLE I: RNG requirements

	Data type	Required OPs	Vectorization
XORWOW [5]	32-bit Unsigned integer		
XOROSHIRO [15]	64-bit Unsigned integer	64-bit ops.	
SFMT [16]	128-bit Unsigned integer	Modulo op. 128-bit ops.	128-bit mask

The vector unit supports both fixed-point and floating-point operations. It is equipped with vectorized multiply-accumulate (MAC) capabilities to manage data permutation from vector registers, perform pre-adds, multiply, and accumulate results. It also supports concurrent operations across multiple vector lanes and accommodates various precisions for complex and real operands. Additionally, a permutation network manages data movement for temporary accumulator registers and storage in vector registers or memory. The single-precision floating-point (SPFP) vector unit shares the permute network of the fixed-point data path, enabling concurrent operations across multiple vector lanes and incorporating eight single-precision MACs per cycle.

For data movement, the AIE's load and store units manage a 5-cycle latency for data memory. Each of these data memory ports can operate in either 256-bit/128-bit vector register mode or 32-bit/16-bit/8-bit scalar register mode. The AIE also features direct stream interfaces with two input and two output streams, configurable to either 32-bit or 128-bit widths, as well as a 384-bit cascade stream for both input and output. These are essential for building pipelined dataflow overlays.

B. Algorithmic Requirements

As shown in Table I, three PRNGs are selected for algorithmic requirements evaluation.

XORWOW is a PRNG from the XORSHIFT family [5], featuring an additional step that transforms the output into a Weyl sequence—a uniform sequence over $[0, m)$ based on an integer k that is relatively prime to m. It is the default PRNG in CUDA [21] and operates on 32-bit unsigned integers. As shown in Algo. 1, XORWOW relies on bitwise operations for number generation and can be vectorized by running parallel PRNGs from different initial states.

XOROSHIRO [15] (xor, rotate, shift, and rotate) improves upon XORSIFT [5] by incorporating rotations alongside shifts, resulting in faster performance and higher-quality output. It has a variant, XOSHIRO (xor, shift, and rotate), but XOROSHIRO remains the more generic design. As shown in Algo. 2, this PRNG operates on 64-bit unsigned integers, which the AIE-API does not natively support. Like XORWOW, it is based on bitwise operations and can be vectorized using parallel PRNGs with different initial states. Besides, it includes a built-in jump function that advances the state by a power of two, improving state initialization in parallel PRNGs.

SFMT [16] (SIMD Fast Mersenne Twister) is a PRNG based on the widely used Mersenne Twister algorithm MT19937 [4].

Algorithm 1 XORWOW Random Number Generation

Input: $s_0, \cdots, s_4$, $\#RNs$
Output: RNs
for $i \leftarrow 0$ to $\#RNs$ **do**
  $cnt \leftarrow cnt + 362437$
  $value \leftarrow s_4$
  $value \leftarrow value \oplus (value >> 2)$
  $value \leftarrow value \oplus (value << 1)$
  $value \leftarrow value \oplus (s_0 \oplus (s_0 << 4))$
  $value \leftarrow value + cnt$ ▷ Weyl sequence
  $RNs[i] \leftarrow value$
  $s_4 \leftarrow s_3, s_3 \leftarrow s_2, \cdots, s_1 \leftarrow s_0$
  $s_0 \leftarrow value$
end for

Algorithm 2 XOROSHIRO Random Number Generation

Input: s_0, s_1, $\#RNs$
Output: RNs
for $i \leftarrow 0$ to $\#RNs$ **do**
  $RNs[i] \leftarrow$ rotate_left$(s_0 + s_1, 17) + s_0$
  $s_1 \leftarrow s_0 \oplus s_1$
  $s_0 \leftarrow$ rotate_left$(s_0, 49) \oplus s_1 \oplus (s_1 << 21)$
  $s_1 \leftarrow$ rotate_left$(s_1, 37)$
end for

It is optimized for high-speed generation of high-quality random numbers, featuring a long period of $2^{19937}-1$. MT19937 ensures well-distributed output while avoiding common PRNG pitfalls such as short periods and poor randomness in lower bits. Due to its speed and statistical robustness, it is a standard choice in scientific computing, simulations, and gaming applications. As shown in Algo. 3, this PRNG runs a mix of 128-bit and vectorized 32-bit operations. For instance, L7 and L9 perform left ($\overset{128}{<<}$) and right ($\overset{128}{>>}$) 128-bit shift operations, respectively. The AIE-API does not natively support these. On the other hand, L8 and L10 perform vectorized right ($\overset{32}{>>}$) and left ($\overset{32}{<<}$) 4 $\times$ 32-bit shift operations, respectively. These are supported natively by the AIE-API using the `aie::downshift` and `aie::upshift` functions.

The selected PRNGs gradually require design constraints for mapping on the AIE. The practical requirements can be evaluated and categorized as follows:

- **Data type:** The data types required for running these PRNGs become increasingly complex, ranging from 32 to 128 bits. While the AIE-API natively supports 32-bit data types, the AIE architecture lacks support for unsigned integers—except for 8-bit ones, which are irrelevant to this work. XORORSHIRO requires 64-bit numbers, which the AIE-API does not support, adding to the complexity. SFMT, which operates on 128-bit numbers, also faces challenges despite native support for 128-bit vectors, as this support is limited to handling four 32-bit vectors rather than true 128-bit numbers.
- **Required operations:** Performing the necessary oper-

Algorithm 3 SFMT Random Number Generation

Macros: For MT19937 based algorithms, $N = 156$ and $M = 122$
Input: $s_0, s_1, \cdots, s_N$, $\#RNs$
Output: RNs
$r_1 \leftarrow s_{N-2}, r_2 \leftarrow s_{N-1}$
for $i \leftarrow 0$ to $\#RNs$ **do**
    $v_0 \leftarrow s_i, v_M \leftarrow s_{(i+M)\ mod\ N}$
    $A \leftarrow (v0 \overset{128}{<<} 8) \oplus v_0$
    $B \leftarrow (vM \overset{32}{>>} 11)\ \&\ vMASK$
    $C \leftarrow r_1 \overset{128}{>>} 8$
    $D \leftarrow r_2 \overset{32}{<<} 18$
    $r_1 \leftarrow r_2$
    $r2 \leftarrow A \oplus B \oplus C \oplus D$
    $RNs[i] \leftarrow r_2$
    $s_{N-1} \leftarrow r_2$
end for

ations for the PRNGs in Table I presents challenges. XOROSHIRO and SFMT rely on 64-bit and 128-bit operations, respectively, while the AIE-API natively supports only 32-bit operations. These operations can be performed either in scalar or in vectorized mode, operating element-wise on vectors of at least $128 = 4 \times 32$ bits. Moreover, SFMT requires the modulo operation to be emulated through a sequence of operations on the AIE core. Execution trace analysis shows that this emulated modulo function is loaded from a precompiled symbol, necessitating a branch to a different memory location during execution. This incurs context-switching overhead.

- **Vectorization:** Generating successive 32-bit random numbers using bitwise operations on a predefined 32-bit mask. SFMT extends this approach by adapting the mask to 128 bits [1] for 128-bit random number generation. While this is not restrictive for the AIE architecture, it is a crucial constraint. PRNGs that rely on masked operations should be scalable to vectorized implementations with masks that align with the target architecture's bit width—128 bits in the case of AIE.

C. Implementing Required Operations

Multiple challenges are discussed in Section II-B. We deduce that to enable running the discussed PRNGs on AIE, we require the following:

- AIE-API vectors must adapted to be used as vectors of $4 \times$ 32-bit, $2 \times$ 64-bit, or $1 \times$ 128-bit binary words for implementing the required data types of XORWOW, XOROSHIRO, and SFMT, respectively.
- Implementation of the unsupported rotation left and left shift operations on the $2 \times$ 64-bit vectors for XOROSHIRO.
- Implementation of the unsupported left and right shift operations on the $1 \times$ 128-bit vector for SFMT.
- Implementation of a cheaper modulo (constant) M operation.

The implementation of these operations on the required data types is given in the following.

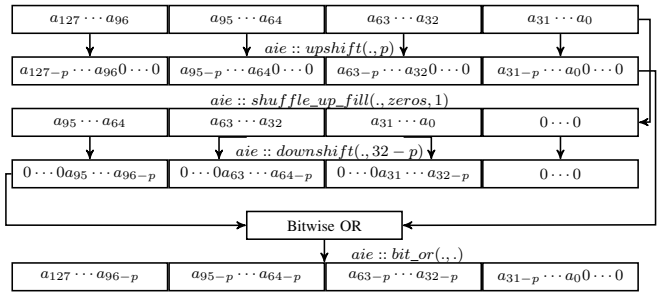
The `shl_128bit` and `shr_128bit` functions, illustrated in Fig. 2a and Fig. 2b, perform left and right shifts, respectively, on a 128-bit vector by a specified number of bits, `p`. These functions operate on four 32-bit vector elements and leverage a sequence of AIE-API operations. For `shl_128bit`, the process begins with `aie::upshift`, which shifts each 32-bit element left by `p` bits. Next, `aie::shuffle_up_fill` shifts the vector elements upward by one position while filling the rightmost position with zero. To handle cross-element bit shifts, `aie::downshift` is applied to shift the intermediate result right by `(32 - p)` bits, ensuring proper alignment. Finally, the `aie::bit_or` operation merges the two shifted segments into a single 128-bit output. The `shr_128bit` function follows a similar procedure but in the opposite direction. It first applies `aie::downshift` to shift each element right by `p` bits. Then, `aie::shuffle_down_fill` moves the vector elements downward, filling the leftmost position with zero. The `aie::upshift` operation then shifts the intermediate result left by `(32 - p)` bits to align the zero-filled positions. Finally, `aie::bit_or` combines the two shifted results into the final 128-bit value.

The `rotl_every64_in128` function, shown in Fig. 3, performs a 64-bit rotate-left operation independently on the two 64-bit halves of a 128-bit vector. For `p` $<$ 32, each half is first left-shifted by `p` bits, then right-shifted by `(32 - p)`, and finally, the lower and higher 32-bit segments are swapped using `swap_high_low_per64bit`. When `p` is 32 or greater, the process is adjusted: each half is right-shifted by `(64 - p)`, then left-shifted by `(p - 32)`, followed by the same 32-bit swap operation. The final result is obtained using `aie::bit_or`, ensuring that both 64-bit halves are correctly rotated within the 128-bit vector.

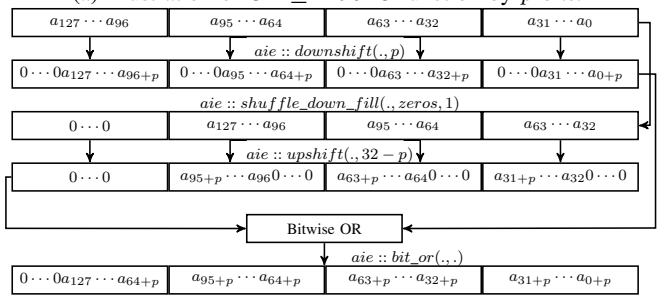
Note that the `swap_high_low_per64bit` function is not an AIE-API native function. It is built upon two other functions that we developed, `swap_low_per64bit_in128` and `swap_high_per64bit_in128`. For each 64-bit half in the 128-bit vector, `swap_low_per64bit_in128` extracts 32 LSBs and places them in the 32 MSBs. Similarly, `swap_high_per64bit_in128` extracts 32 MSBs and places them in the 32 LSBs.

The `shl_every64_in128` function, shown in Fig. 4, performs a left shift operation on the two 64-bit halves of a 128-bit vector. It follows a similar principle as `rotl_every64_in128` but only extracts the 32 LSBs using `swap_low_per64bit_in128` when `p` $<$ 32. For `p` is 32 or greater, only the 32 LSBs are retained, as the 32 MSBs are truncated by shifting. The same logic applies to `shr_every64_in128`, illustrated in Fig. 5, which performs

[1] In Algo. 3, $vMASK$ equals $BFFFFFF6BFFAFFFFDDFECB7FDFFFFFEF$



(a) Illustration of `shl_128bit` function by p bits.



(b) Illustration of `shr_128bit` function by p bits.

Fig. 2: 128-bit shift operations.

a right shift operation on the two 64-bit halves of a 128-bit vector.

In SFMT, the modulo operation is always performed with the constant $M = 156$ (Algo. 3, L6). Instead of using a costly generic modulo operation, the implementation extracts the 8 LSBs using the bitwise AND operation `&0xFF`, ensuring the result stays within [0, 255]. If the extracted value exceeds 156, subtracting 156 brings it within [0, 155], eliminating the need to run the costly modulo and branching to its location in memory, reducing runtime overhead.

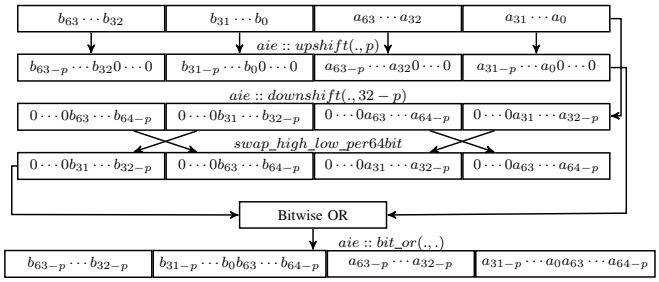
III. Experimental Setup and Performance Evaluation

This section evaluates the performance of the PRNGs listed in Table I using the developed routines from Section II-C. Two execution models, illustrated in Fig. 6, are considered. The first model treats the AIEs as a co-processor (Fig. 6a), where a large number of random numbers are generated and sent to the host processor on demand, similar to a GPU execution model. The second model employs the AIEs as a standalone dataflow accelerator (Fig. 6b), running a PRNG-Compute cluster overlay. Each cluster consists of an AIE generating random numbers that feed another AIE performing subsequent computations.

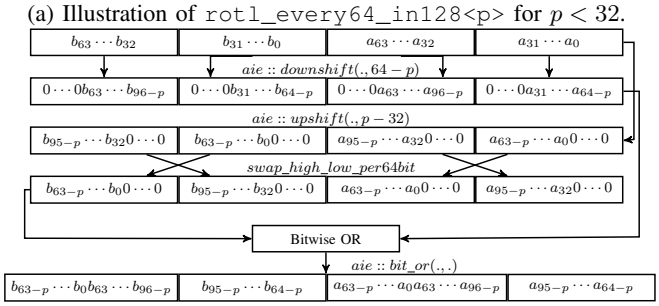
Notably, the second execution model is templated, allowing the AIE responsible for computation to be adapted for any kernel. Both execution models are open-sourced on GitHub [22].

A. Co-Processor Model

As shown in Fig. 6a, the AIEs function as a co-processor, offloading random number generation. Each AIE runs an independent PRNG, with every two vertically adjacent AIEs

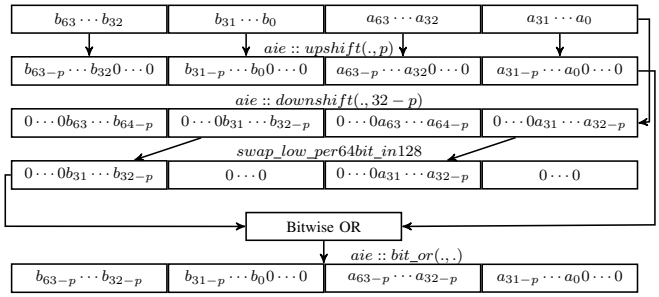


(a) Illustration of `rotl_every64_in128<p>` for $p < 32$.

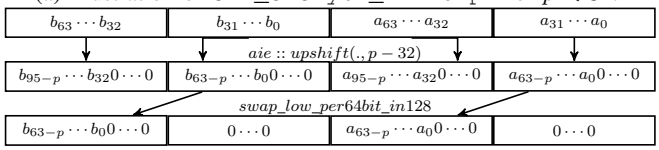


(b) Illustration of `rotl_every64_in128<p>` for $p \geq 32$.

Fig. 3: 2 × 64-bit rotate left operation.



(a) Illustration of `shl_every64_in128<p>` for $p < 32$.

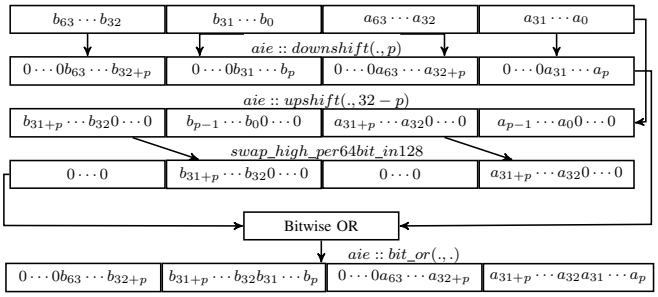


(b) Illustration of `shl_every64_in128<p>` for $p \geq 32$.

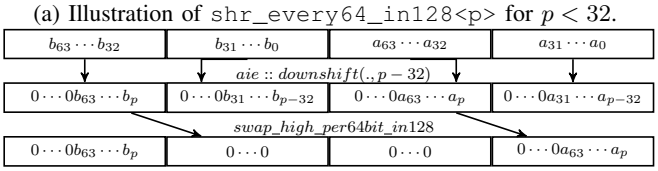
Fig. 4: 2 × 64-bit left shift operation.

in the same column forming a cluster. Each cluster receives two seed packets via a shared 32-bit stream interconnection and outputs random numbers through another 32-bit stream interconnection. Due to hardware constraints, packet streaming is necessary to fully utilize all eight AIEs per column, as each column has only six input and four output stream interconnections [23].

Four `MM2S` (Main-Memory to Stream) and four `S2MM` (Stream to Main-Memory) kernels handle seed input and random number output, respectively, ensuring direct communication with the host. These kernels are implemented on the PL. Since each column consists of four clusters, each input `MM2S` and output `S2MM` kernel is connected to one of the four rows of clusters across all the columns. Note that the



(a) Illustration of `shr_every64_in128<p>` for $p < 32$.



(b) Illustration of `shr_every64_in128<p>` for $p \geq 32$.

Fig. 5: 2×64-bit right shift operation.

`MM2S` and `S2MM` kernels enable parallel communication with the AIE clusters through separate channels. To simplify the hardware design, these channels are grouped into four `MM2S` and four `S2MM` kernels. This grouping does not serialize the communication and ensures that data transfer remains fully parallel. This enables efficient data transfer between the host and AIEs.

Finally, this model employs 320 AIEs (40 columns $\times$ 8 rows), utilizing 80% of the available 400 AIEs (50 $\times$ 8). The model is constrained to a maximum of 40 columns because only 39 out of 50 columns contain PL-interface tiles necessary for AIE-to-PL connectivity. However, these interface tiles provide four incoming and four outgoing 32-bit stream interconnections to adjacent horizontal tiles, allowing the use of the 40th column despite the 39-column limitation.

B. Standalone Dataflow Accelerator

As shown in Fig. 6b, the AIEs operate as a standalone dataflow accelerator. In this execution model, each cluster of two vertically adjacent AIEs runs the random number generation followed by another compute step. Each cluster receives the PRNG seeds on a 32-bit stream interconnection, sends the output random numbers to another compute kernel mapped on the second AIE, and outputs the result of the computation on another 32-bit stream interconnection. The four `MM2S` and four `S2MM` PL kernels to communicate the AIEs to the host, and 320 AIEs (40 columns $\times$ 8 rows) out of 400 AIEs, are used similarly to the Co-processor model in Section III-A.

To evaluate this model, the generation of the Normal distributed numbers is used as the compute stage that follows the PRNG. For this, the incoming numbers from the PRNGs must be converted to single-precision floating-point (SPFP) numbers normalized in the interval $[0, 1)$. The AIE-API function `aie::to_float` is used for this conversion. This function treats the integer input as a 32-bit fixed-point number and converts it to SPFP by specifying the position of the binary point. Since all the numbers in the interval $[0, 1)$ have their binary point at the MSB, the function is configured accordingly to ensure correct normalization.

However, due to the lack of unsigned integer support on the AIE, the PRNG-generated integer may be misinterpreted as negative if its MSB is 1. To convert it to the desired interval correctly, we apply an arithmetic adjustment to ensure equivalence with an unsigned-to-float conversion as follows.

A SPFP number r consists of a 1-bit sign S, an 8-bit exponent E, and a 23-bit mantissa M, and is represented as $r = (-1)^S \times M \times 2^{E-128}$. Here, the mantissa M follows an implicit leading-one convention, i.e. $M = [b_{22} \cdots b_0]$ represents $1.[b_{22} \cdots b_0]$. Therefore, for a fixed-point number to be converted to a SPFP number, it should be *normalized* to $1.[b_{22} \cdots b_0]$. Therefore, if the MSB of the fixed-point number is 1, then the 23 most significant bits (MSBs) correspond to the mantissa, while the remaining $32 - 23 = 9$ bits are truncated during conversion. To ensure proper alignment, we first apply a 9-bit right shift using `aie::downshift`, bringing the mantissa into the LSBs. Then, the `aie::to_float` function is applied while specifying the binary point position at bit 23. This ensures that the MSB of the input fixed-point number is correctly interpreted as part of the number itself rather than as a sign bit. We construct a function `unsigned_to_float` that applies both this method and `aie::to_float` to convert unsigned integer inputs to SPFP numbers correctly. The function evaluates each element of the integer input vector and applies the appropriate conversion based on the sign bit. This selection is efficiently handled using the `aie::select` function, ensuring that each element undergoes the correct conversion while maintaining vectorization efficiency.

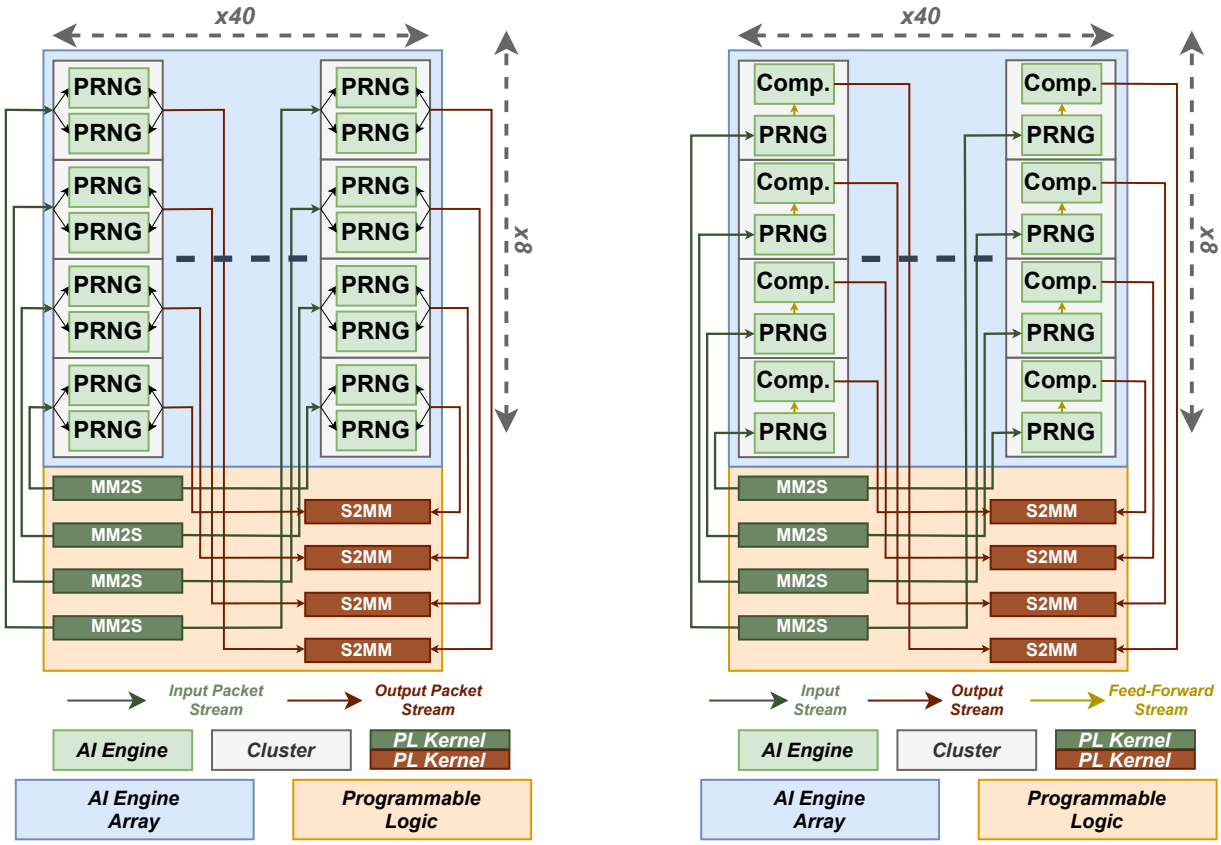
For Normal distributed number generation, Acklam's Approximation [18] is utilized on the converted SPFP numbers in $[0, 1)$. This method is used for Inverse Cumulative Distribution Function (ICDF) approximation through a rational fraction, as given in Eq. 1 (coefficients given in Table III). This method applies different calculations for the input values close to 0 and 1, but this is omitted here for simplicity as was done in [24].

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

C. Performance evaluation

The performance of the two implemented models described in Section III-A and Section III-B are compared in Table II. The details of the AMD Versal platform used for experiments are given in Table IV. For comparison, 160K to 160M random numbers are generated.

In Table II, the execution times for all PRNGs are shown scaling by four orders of magnitude. Linear scaling would result in a $10\times$ increase moving one column to the right. Lower multipliers indicate better scaling. We notice that the scaling is approximately linear. Besides, fixed overheads such as packet switching are amortized over the increasing number



(a) Co-processor model.

(b) Standalone dataflow accelerator model.

Fig. 6: PRNG accelerator architectures.

TABLE II: Execution time (in milliseconds) for the different PRNGs.

PRNG	Model	160K RNs	1.6M (10× ←) RNs	16M (10× ←) RNs	160M (10× ←) RNs	M2 to M1
XORWOW	M1: Co-processor	2.858	17.477 (6.11×)	164.762 (9.42×)	1634.63 (9.92×)	3.15×
	M2: Dataflow	6.315	59.749 (9.46×)	593.921 (9.94×)	5934.01 (9.99×)	
SFMT	M1: Co-processor	4.831	30.494 (6.31×)	285.603 (9.36×)	2835.71 (9.92×)	1.86×
	M2: Dataflow	6.722	60.242 (8.96×)	594.355 (9.86×)	5934.5 (9.98×)	
XOROSHIRO	M1: Co-processor	7.637	70.596 (9.24×)	697.669 (9.88×)	6967.69 (9.98×)	0.96×
	M2: Dataflow	6.961	68.520 (9.84×)	683.628 (9.97×)	6833.75 (9.99×)	

TABLE III: Coefficients of the fraction P in Eq. 1

Coeff.	Value	Coeff.	Value
A_1	-3.9696830286653757e+01	B_1	-5.4476098798224058e+01
A_2	2.2094609842452050e+02	B_2	1.6158583685804089e+02
A_3	-2.7592851044696869e+02	B_3	-1.5569897985988661e+02
A_4	1.3835775186726900e+02	B_4	6.6801311887719720e+01
A_5	-3.0664798066147160e+01	B_5	-1.3280681552885721e+01
A_6	2.5066282774592392e+00		

TABLE IV: Hardware accelerator configuration.

Component	Description
Board	AMD Versal ACAP VCK5000
Device	XCVC1902
AI Engine Array	400× 1st Gen. AIE
Tool Version	Vitis 2022.1

of random numbers generated. For the Co-Processor model with XORWOW and SFMT, we see a better than linear scaling at 1.6M RNs, indicating this amortization.

XORWOW is slightly faster than SFMT, which is faster than XOROSHIRO. In the Co-processor model (M1), XOROSHIRO is consistently slower than XORWOW and SFMT. However, in the Standalone dataflow accelerator model

(M2), XOROSHIRO slightly matches SFMT and XORWOW. This suggests that using XORWOW and SFMT with M1 is more beneficial.

The last column represents the geometric mean of the execution times using the M2 to M1 ratio. We notice that the overhead of the Normal ICDF approximation, which is implemented in M2, is not well hidden within the latency of the fast XORWOW and SFMT PRNGs compared to the slower XOROSHIRO. This suggests that using XOROSHIRO with M2 is more beneficial. This also suggests that when using XORWOW and SFMT, more pipelining of the second computation stage of the Normal ICDF approximation is needed to improve pipeline balance and match the performance of M1.

IV. Discussion and Conclusion

This paper has demonstrated how to port massive random number generation to AMD AI Engines. This was achieved by implementing and optimizing custom functions to enable the implementation of PRNGs. We identified key challenges, including the lack of support for certain operations, unsigned data types, and vectorization constraints and addressed them through sequences of operations supported by the AIE-API. Additionally, we presented the porting of normal distribution approximation. We presented experimental results for two execution models. The first model uses the AIEs to offload the random number generation, while the second is used to enable seamless integration of random number generation within computational pipelines. Performance evaluation confirms that the implemented methods exhibit linear scalability, enabling the generation of large volumes of random numbers while maintaining efficiency. The results also highlight that the co-processor model performs better with fast PRNGs like XORWOW and SFMT, rather than XOROSHIRO.

This work also demonstrates that the AMD AI Engine architecture could benefit from allowing cross-lane operations such as shifts as well as combining registers to form larger vector lanes, as is the case with the AVX [25] extension on x86 processors. It could also benefit from supporting unsigned data types natively. This would reduce the overhead of running sequences of operations, extracting better performance for random number generation.

In future work, we will tackle another class of PRNGs, MWC (Multiple-With-Carry) [5], which are sign-insensitive as they require arithmetic operations such as addition and multiplication. These are challenging to run on the AIE architecture as it does not support unsigned arithmetic, as has been discussed earlier. They are further challenging as the AIE architecture uses special 48-bit registers, called accumulator registers, to store the results of multiplications of 32-bit integers (including the sign extension), making combining lanes even more complicated. We have provided an open-source release of our code [22] for the wider community to build further applications that rely on random number generation on AIEs.

References

[1] R. Rolland, *Randomness in Cryptography*. Springer International Publishing, 2015.

[2] L. Lista, *Random Numbers and Monte Carlo Methods*. Springer International Publishing, 2023.

[3] B. Antunes and D. R. C. Hill, "Random numbers for machine learning: A comparative study of reproducibility and energy consumption," *Journal of Data Science and Intelligent Systems*, 2024.

[4] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3—30, 1998.

[5] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software*, vol. 8, no. 14, 2003.

[6] P. Glasserman, *Monte Carlo Methods in Financial Engineering*. Springer New York, 2003.

[7] M. Bouaziz, M. Samet, and S. A. Fahmy, "A dataflow overlay for Monte Carlo multi-asset option pricing on AMD Versal AI Engines," in *ISC High Performance Research Paper Proceedings*, 2025.

[8] D. P. Kroese, T. Brereton, T. Taimre, and Z. I. Botev, "Why the Monte Carlo method is so important today," *WIREs Computational Statistics*, vol. 6, no. 6, pp. 386–392, 2014.

[9] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games*. Springer International Publishing, 2016.

[10] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: as easy as 1, 2, 3," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[11] D. B. Thomas and W. Luk, "High quality uniform random number generation using LUT optimised state-transition matrices," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 47, no. 1, p. 77–92, 2007.

[12] D. B. Thomas, L. Howes, and W. Luk, "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2009.

[13] I. Ambric, "Am2000 family architecture reference," 2008.

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

[15] D. Blackman and S. Vigna, "Scrambled linear pseudorandom number generators," *ACM Transactions on Mathematical Software*, vol. 47, no. 4, pp. 36:1–36:32, 2021.

[16] M. Saito and M. Matsumoto, "SIMD-oriented fast Mersenne Twister: a 128-bit pseudorandom number generator," in *Monte Carlo and Quasi-Monte Carlo Methods*, 2006.

[17] M. J. Wichura, "Algorithm AS 241: The percentage points of the normal distribution," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 1988.

[18] 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.

[19] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[20] "AI Engine kernel and graph programming guide (UG1079)," https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding, 2022.

[21] "CUDA C++ programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide, 2024.

[22] GitHub Repoistory, 2025. [Online]. Available: https://github.com/accl-kaust/PRNGine

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

[24] 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.

[25] "Intel AVX-512 instructions," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html, 2014.