BSRNG: A High Throughput Parallel BitSliced Approach for Random Number Generators

Saleh Khalaj Monfared
Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
monfared@ipm.ir

Omid Hajihassani
University of Alberta
Edmonton, Canada
hajihass@ualberta.ca

Mohammad Sina Kiarostami
Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran

Soroush Meghdadi Zanjani
Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran

Dara Rahmati
Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran

Saeid Gorgin
Iranian Research Organization for Science and Technology (IROST)
Tehran, Iran

ABSTRACT

In this work, a high throughput method for generating high-quality Pseudo-Random Numbers using the bitslicing technique is proposed. In such a technique, instead of the conventional row-major data representation, column-major data representation is employed, which allows the bitslicing implementation to take full advantage of all the available datapath of the hardware platform. By employing this data representation as building blocks of algorithms, we showcase the capability and scalability of our proposed method in various PRNG methods in the category of block and stream ciphers. The LFSR-based (Linear Feedback Shift Register) nature of the PRNG in our implementation perfectly suits the GPU's many-core structure due to its register oriented architecture. In the proposed SIMD vectorized GPU implementation, each GPU thread can generate several 32 pseudo-random bits in each LFSR clock cycle. We then compare our implementation with some of the most significant PRNGs that display a satisfactory performance throughput and randomness criteria. The proposed implementation successfully passes the NIST test for statistical randomness and bit-wise correlation criteria. For computer-based PRNG and the optical solutions in terms of performance and performance per cost, this technique is efficient while maintaining an acceptable randomness measure. Our highest performance among all of the implemented CPRNGs with the proposed method is achieved by the MICKEY 2.0 algorithm, which shows 40% improvement over state of the art NVIDIA's proprietary high-performance PRNG, cuRAND library, achieving 2.72 Tb/s of throughput on the affordable NVIDIA GTX 2080 Ti.

ICPP Workshops '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8868-9/20/08…$15.00
https://doi.org/10.1145/3409390.3409402

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; • **Computer systems organization** → *Parallel architectures*; • **Security and privacy** → Pseudonymity, anonymity and untraceability.

KEYWORDS

PRNG, Cryptography, High-performance, CUDA, cuRAND, Stream cipher, Bitslicing

ACM Reference Format:
Saleh Khalaj Monfared, Omid Hajihassani, Mohammad Sina Kiarostami, Soroush Meghdadi Zanjani, Dara Rahmati, and Saeid Gorgin. 2020. BSRNG: A High Throughput Parallel BitSliced Approach for Random Number Generators. In *49th International Conference on Parallel Processing - ICPP : Workshops (ICPP Workshops '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3409390.3409402

1 INTRODUCTION

The emergence of cost-effective, high-performance parallel platforms such as Graphical Processing Units (GPU) and their programmability have allowed researchers across various fields of science and engineering to utilize GPUs' specialized processing capability to accelerate their computationally demanding applications. GPUs' processing power has been fully leveraged to implement machine learning algorithms [13, 34], medical image processing [9], and many other applications.

Recently, the high-performance execution on GPU has attracted many researchers' attention to adapt cryptography problems for execution on massively-parallel GPU platforms [2, 40]. One problem which is the particular concern of this paper is the high-throughput generation of sequences of pseudo-random numbers. High-performance random number generation with an acceptable randomness criterion is a vital necessity in many computer science disciplines, including stochastic computing, stochastic simulation, i.e., Monte Carlo simulation [7], and cryptography [1].

The acceptable criteria for quality of randomness vary across different fields, demanding the random number sequence. Perhaps one of the most rigorous fields holding very high standards for the randomness is cryptography [38]. In order to showcase the randomness quality of random number generator algorithms, it

is conventional to employ the criteria used for cryptographic purposes. The underlying pseudo-random number generator process, apart from statistical randomness, must accompany other security assurances that vary based on the intended application. The Cryptographically Secure Pseudo-random Number Generator (CSPRNG) processes work based on increasing the output sequences' entropy, which makes the output sequence indistinguishable from uniformly random bit sequences. Moreover, the unpredictability of the next-bit must be further guaranteed. Here, we intend to apply the bitslicing technique in the software implementation of CSPRNG processes.

In the bitslicing technique, by altering the representation of the input data and computations, we strive first to increase the computation units' utilization, and second, reduce the required operations from costly operations to hardware-friendly basic bit-level operations, such as XOR, AND, and OR operations. With the incorporation of the bitslicing technique in our implementations, we can achieve highly-parallel, vectorized execution in the SIMD manner. Authors in [6] have proposed the successful utilization of the bitslicing technique in implementing the Data Encryption Standard (DES) on a 64-bit processor, where the processor is viewed as a SIMD processing unit. In this implementation of the DES, the 64-bit CPU can be perceived as 64 1-bit CPUs that process 64 chunks of data, simultaneously. In [2], the authors have proposed the high-throughput implementation of the bitslicing DES exhaustive essential search cryptanalysis technique on programmable GPU platforms.

Although the bitslicing technique has been utilized for cryptanalysis and fast implementation of cryptosystems, it has not been used for CPRNGs. A fast, secured, high performance and reliable CPRNG usually fails to satisfy the criteria in some applications due to its complexity compared to regular PRNGs. Here, we would like to improve this drawback for the CPRNGs by incorporating bitslicing technique and altering the cryptographic algorithms where performance is improved, and the security criteria are maintained. A characteristic of the software implementation of LFSR-based PRNGs is the intrinsic need for repetitive, costly bit-level shift and mask operations. By proposing the bitslicing technique and changing the data and computation representation, we have successfully transformed those described above costly bit-level shift and mask operations to more efficient register swapping techniques.

As our main contribution, in this paper, we provide a study on the implementation of PRNGs on GPUs by utilizing the bitslicing technique. In order to showcase the scalability of our proposed method and demonstration of suitability cryptosystem for our purpose, we employ multiple streams and block ciphers, some of which have not been studied before with the bitslicing technique. We demonstrate that under the proper choice of the suitable algorithm and by conveniently applying the required necessary steps with transformations of functional building blocks and data structures, our proposed method exhibits significant performance in terms of raw computational throughput, normalized throughput per computational power and also reliability and quality of the generated random numbers by putting it under the NIST test. On top of it all, we present a PRNG implementation based on Mickey 2 stream cipher that, to the best of our knowledge, outperforms PRNG's available implementations in terms of performance. In our GPU implementation, our version of Mickey 2 outperforms the Nvidia's proprietary cuRAND, random number generator, by 1.9 X on a GTX 980 Ti, despite its complexity in the algorithm itself.

The rest of this paper is organized as follows: In section II, we will introduce a detailed background on the PRNGs, Linear Feedback Shift Registers (LFSR), and some crypto-systems are employed in our proposed method such as Mickey 2.0. Section III discusses the related efforts to PRNG and RNG implementations. Section IV gives our proposed methodology and elaborates on incorporating the bitslicing technique in our implementation, along with examples in different applications. Section V gives the evaluation results achieved from our proposed methodology's performance and correctness on multiple GPUs. Section VI concludes the paper and discusses future works.

2 BACKGROUND

This section will give a background on the random number generation literature, the bitslicing technique, and related concepts such as linear-feedback shift register (LFSR) and the underlying mechanisms of employed algorithms for pseudo-random number generation. Moreover, a brief study will be presented on the stream and block ciphers employed in this work.

2.1 Random Number Generation

Truly random number generator processes are set to be non-deterministic, a condition under which the generated random sequence can not be determined in advance. Random number sequences can be generated from sampling of truly random sequences such as physical random phenomenon, including *Thermal Noise* [16], *Electrical Noise* [8], and *Laser* or *Optical* mechanisms [17]. However, such random number generators that use physical phenomena are costly. Also, the unavailability of the required apparatus limits the scope of the usage for general applications. Although, such random sequences can be stored for later use, which also limits the availability and security in certain applications.

The issues mentioned above of cost and availability lead to the use of digital computers in the generation of random numbers. Pseudo-random number generators are not genuinely random processes that root from the deterministic essence of digital computers but are specifically designed to meet specific randomness criteria in their generated sequences. One of the first PRNG methods that use a random seed and relies on the seed's randomness for the generation of reproducible random sequences in the Middle Square Method (MSM) [44]. PRNGs can generate random sequences with truly random seeds until the seed is repeated, and the sequence repeats in the output. The initial seed size indicates the size of the generated random sequence before the repeat in the generated sequence.

One feature of PRNG processes is that with the same seed, the generated random sequence can be reproduced, which can be exploited in some scenarios such as end-to-end communications. On the other hand, it would also be computationally infeasible to find the random input seed that the PRNG process uses to generate the pseudo-random sequence by exhaustively searching the seed space with a part to find and predict the next-bit of the sequence. To ensure this, the size of the seed must be set to a large number.

2.2 Linear Feedback Shift Registers

Based on the mathematical foundation of cyclic codes over the finite field of $GF(2)$, the Linear Feedback shift registers have been employed both in software and hardware for a wide range of applications, including transmission error checks [19], high-performance counters, and pseudo-random number generators[26]. In LFSR, the feedback tabs that determine the next state of the system, if combined linearly, could directly impact the system's input when the register is shifted at each clock cycle. Figure 1 demonstrates a basic representation of a single n-bit LFSR. The arrangements of the tabs could be represented by a polynomial referred to as the feedback polynomial.

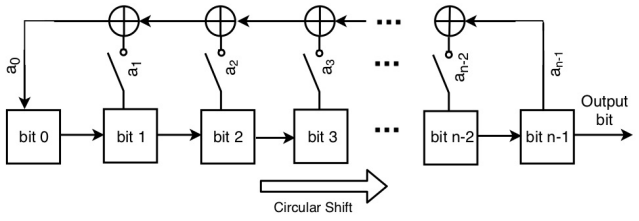

Figure 1: A basic *n*-bit Linear Feedback Shift Register

For a simple n-bit LFSR, the coefficients, operations are defined over $GF(2)$, and the reciprocal characteristic polynomial could be represented as is in Equation 1:

$$p(x) = \sum_{i=0}^{n-1} a_i x^i; a_i, x \in GF(2)$$
$$a_0, a_{n-1} = 1 \tag{1}$$

Also, it is worth noting that in many applications, in order to maximize the LFSR period length ($i.e.\ 2^n - 1$), a primitive polynomial is chosen as the tapping coefficients for the LFSR.

2.3 Stream Ciphers and Block Ciphers

As already mentioned, block and stream ciphers' cryptographic properties are known to be suitable to generate high-quality pseudo-random numbers. Among all of various and different proposed stream and block cipher algorithms, we investigate the two-stream and a block cipher, which are to be exploited by the bitsliced representation. Some of these ciphers are specifically designed for efficient hardware implementation while guaranteeing an acceptable level of security. The ECRYPT Stream Cipher Project (eSTREAM) Profile 2 stream ciphers are particularly suitable for hardware applications with restricted resources such as limited storage, gate count, or power consumption [4]. We recommend using stream ciphers instead of block ciphers for fast and high-performance implementations due to their light-weights' architecture. As shown in the evaluation section, the Mickey 2.0 algorithm shows more promising results than block ciphers such as AES (Advanced Encryption Standard).

2.3.1 MICKEY 2.0 Stream Cipher. MICKEY 2.0 or Mutual Irregular Clocking KEYstream generator is the second generation stream cipher of the MICKEY family, introduced by Babbage and Dodd [5]. Armed with the fact that the MICKEY 2.0 algorithm is inherently light-weight in hardware implementation, the feedback shift register-based architecture can be easily embodied with the proposed bitslicing technique.

The state machine of the algorithm consists of two 100-bit shift registers, one linear and one non-linear, both clocked irregularly under each other's control. It is stated that each key can be used with up to 2^{40} different IVs of the same length, and that 2^{40} keystreams can be generated from each key/IV pair. Figure 2 shows the Galois-based structure of the MICKEY 2.0 algorithm.

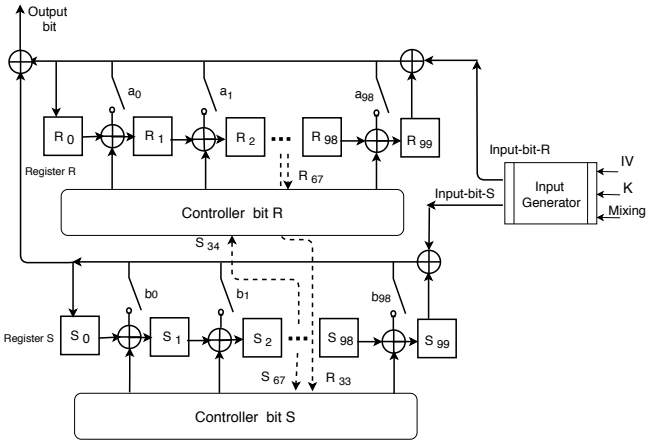

Figure 2: Illustration of Mickey 2.0 Stream Cipher Algorithm

The MICKEY 2.0 designers have also specified a scaled-up version of the cipher called MICKEY-128 2.0, which employs a 128-bit key and an initialization vector of up to 128 bits. It is stated in the specification that the irregular clocking mechanism makes the parallel implementation somehow *not so straightforward.* However, as will be investigated later, our proposed bitsliced algorithm utilizes a fully parallel implementation of MICKEY. Furthermore, it has been noted by Gierlichs et al. [11], those straightforward implementations of the MICKEY ciphers are likely to be susceptible to timing or power analysis attacks. However, the system could be immunized by software techniques like masking, making these attacks significantly ineffective. Furthermore, there have been no known cryptanalytic advances against MICKEY 2.0 or MICKEY-128 2.0 after its publication in eSTREAM.[36]

2.3.2 Advance Encryption Standard. Advanced Encryption Standard, also is known as AES, is the most famous and used block cipher in communication today. After five years of competition and standardization, the National Institute of Standards and Technology (NIST) selected Rijndael block cipher to supersedes Data Encryption Standard (DES) in 2001 as AES [35]. NIST's AES specification introduces three versions of Rijndael cipher with 10, 12, and 14 rounds of ciphering with 128, 192, and 256 bits of keys. The AES algorithm is consists of three major building blocks and is processed in byte granularity in extended Galois Field of $GF(2^8)$. The state matrix of 16 bytes in a 4×4 matrix is constructed; each step is iterated in AES. S-box or the substitute byte is the only non-linear part of the AES, a

simple substitutions table, and responsible for a non-linearity in the cipher. A look-up table usually implements the S-box in memory in software and hardware implementations. However, the S-box could be efficiently implemented by bit-level gates as well. The Mix-Column and Shift-rows boxes are responsible for the diffusion of the data in cipher. The Mix-Column is a Galois-based Matrix Multiplication step, and the Shift-Row is a simple linear byte swapping in the rows of the state matrix. For PRNG purposes, AES is utilized on the CTR (Counter Mode), and a data batch of 128-bits is generated as random blocks while the counter is incremented. The AES-CTR usage for PRNG is shown in Figure 3. A secure and private random

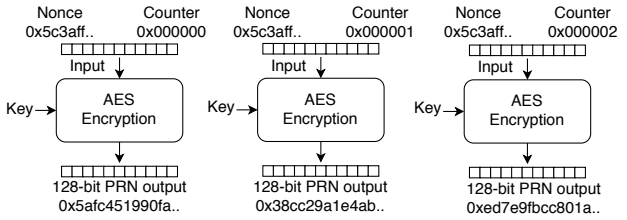

Figure 3: AES-CTR implantation as secure random number generator

number as nonce is generated and concatenated with a counter as the input. By encrypting the input with an arbitrary key each encryption blocks provide a 128-bit random data-chunk. Note that this procedure could be fully employed in parallel since each AES block encryption is independently implemented.

2.3.3 Grain Stream Cipher. The Grain is also a winner of the eSTREAM portfolio for Profile 2, designed explicitly for restricted hardware environments. The Grain developed by Hell et al. [15], is constructed by two 80 bits Linear and Non-Linear Feedback Shift Registers (NFSR) which are shifted together at each clock cycle. The NFSR is controlled by a feedback function of itself and LFSR output. Similarly, the LFSR is also controlled by feedback function. The cipher is normally initialized with an 80-bit key and a 64-bit Initial Vector(IV), which is directly fed into the NFSR and LFSR, respectively at the beginning. The specification recommends 160 clocks of initialization before the keystream generation. The light-weighted architecture of the Grain structure couple with the shift-registers used in this algorithm makes it a great nominee for the bit-sliced implementation. A high-level demonstration of Grain stream Cipher is given in Figure 4.

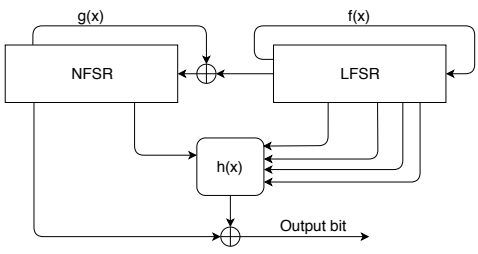

Figure 4: Illustration of Grain Stream Cipher Algorithm

3 RELATED EFFORTS

Random number generation has been a topic of interest for researchers and developers for decades. Numerous theoretical and practical studies have investigated the complexity of generating high-quality random numbers and evaluating them [22, 28]. Using parallel platforms for the acceleration of RNGs has also been a matter of consideration in the literature [27]. Staring around the 2000s, the emergence of general-purpose computing on graphics processing units has opened new horizons for the high-performance generation of random numbers. In 2006, M Sussman et al. published one of the first works utilizing the power of GPGPU on the subject of RNG [39]. In the years to follow, many researchers and developers reported successful implementations of PRNGs with an increase in performance on parallel platforms, achieving remarkable speedups over the CPU platforms of their time and outperforming similar efforts [12]. In the subject of high-performance parallel PRNG, the performance of Nvidia's proprietary PRNG cuRAND library [42] has always been a forceful competition, still, in some cases, researchers have reported their work excelling the performance of the cuRAND of their time [31]. Compared to the most significant recent efforts on pseudo-random number generation on GPU [3, 41], the cuRAND library seems to remain the major dominating player in the field with performances brighter than competitors in terms of both throughput and throughput per processing power of the device underutilization.

Regardless of all the performance advancements in the HPC community, there have always been skepticism and critical opinions regarding arithmetic methods for generating random numbers. Highly favored from physics academic community [24] quote by Von Neumann stating, "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" [44] is famously cited for designating the difference between natures of generated random numbers by physical and computational methods. Although not truly random numbers by definition, computationally generated random numbers have gained serious attention in recent years due to their accessibility, affordability, and employment ability in high-performance platforms thanks to the development in parallel hardware devices. Theoretical and practical methods regarding generation and evaluation of computationally generated random numbers have advanced so far that these arithmetically generated random numbers are securely utilized in sensitive applications like cryptography.

It is worth noting that the quality of random numbers is dependent on its target user and application domain [22]. Trivial computer games and sensitive cryptographic applications have different requirements and criteria for measuring how *good* a random number is. Therefore to ensure that the randomness properties of a sequence are satisfactory for a certain application, the measurement of the quality of the generated sequence is of high significance. Consequently, efforts have been made to develop procedures and tools capable of evaluating the desirable statistical properties of a given random sequence [23]. NIST SP 800-22 [36] is a statistical test suite from the national institute of standards and technology (NIST) designed to probe RNGs for both statistical and cryptographical properties, ensuring the qualification of the passing RNG for its target users in cryptographic applications.

Taking advantage of the GPU platform in the generation of random numbers compared to the physical and the optical methods [17, 25, 45], other hardware platforms [43] such as FPGA [37] and CPU [29] enables the users to strike a balance between the quality of randomness, flexibility, obtainability, affordability, performance, and outstanding performance per cost metrics.

Table 1: Priviously proposed PRNG Implementaions on GPU

Ref	*Year*	*GPU*	*GPU's GFLOPS*	*Method*	*Method's Gbps*	*(Gbps/GFLOPS)*
[20]	2008	8800 GTX	345.6	RapidMind	26	0.0752
[33]	2008	7800 GTX	20.6	CA-PRNG	0.41	0.0199
[21]	2009	T10P	622.1	ParkMiller	35	0.0562
[12]	2010	S1070	2488.3	N/A	4.98	0.0020
[31]	2011	GTX 480	1344.96	xorgensGP	527.5	0.3922
[10]	2013	GTX 480	1344.96	GASPRNG	37.4	0.0278

Unfortunately, current random number generation methods on GPU do not take full advantage of this hardware platform. The latest efforts on CSPRNG on high-end GPUs perform poorly in utilizing GPU's massive parallelization capabilities to reach high-generation performances [3]. Table I represents the claimed performance of some of the related efforts and the processing power of the GPU on which they could reach their peak performance. Also, to establish a fair platform for comparison, each method's peak performance was normalized to the processing power of their employed device and reported as well. The performance achieved by the state of the art RNGs such as Nvidia's proprietary cuRAND library still does not fully exhibit the full potential of modern GPUs. We show this performance can be further enhanced while maintaining reasonable cryptographically properties via applying the bitslicing approach to the implementation of GPU-based CSPRNGs.

4 PROPOSED METHOD

In this section, we propose implementing the bitslicing software technique for high-throughput cryptographically safe pseudo-random generation based on cryptographic algorithms. The column-major bitsliced data representation scheme's approach is first introduced and discussed, and their advantages compared to the na "ive implementation are explained. Afterward, by incorporating the bitslicing technique into the implementation of our work, a high-throughput algorithm for LFSR architecture is presented, and then, as another application, a cyclic redundancy check (CRC) using the proposed architecture is described. The implementation of parallel MICKEY algorithm as an example for bitsliced stream ciphers is presented for Random Bit Generation (RBG). Finally, further GPU optimization techniques used in our implementations are discussed. It is also worthy of mentioning that we have implemented the bitsliced version of three cryptosystems, namely *AES*, *MICKEY 2*, and *Grain* algorithms as CPRNGs to show the extensiveness of the proposed method. However, here only the alteration MICKEY 2 algorithm is described as an example.

4.1 Bitsliced SIMD Vectorization and Data Representation

Bitslicing technique was employed by Biham [6] for the implementation of cryptographic algorithms. At the time, the technique accelerated the previous implementations of the Data Encryption Standard (DES) to accelerate the exhaustive search procedure. As mentioned, by the emergence of high-performance, affordable general-purpose GPUs, the bitslicing technique has been successfully employed by many works such as [32], [14], and [2] as a software solution for high-throughput demanding cryptographic applications on GPUs. This bitslicing-based implementation, leveraged by the column-major data representation, reaches an unprecedented throughput of Terabits per second (Tbps) both in encryption and decryption.

Before getting into the details of the proposed PRNG via MICKEY 2.0, we discuss the data representation scheme employed in our work. The proposed representation scheme fits the GPU's parallel architecture and utilizes the available datapath of the computational units in the deployment hardware.

This proposed data representation scheme, uses column-major data representation, instead of the conventional row-major representation. By the row-major representation, we refer to the representation used to store the data in standard programming practices. In our implementation, we store state bits and other supplementary and temporary registers in the column-major representation. We strive to achieve full SIMD execution of several 32 (in the case of single-precision calculations) bits from different data chunks at each execution clock cycle. In our LFSR implementation, a batch of 32 bits data stored in a single register, represents state bits from 32 uncorrelated different parallel LFSRs having the same bit significance. Hence,

The first step is to alter the representation of the column-major data representation. For a simple LFSR implementation operating in the conventional row-major representation, one or more registers are used to store the LFSR algorithm's state bits. Hence, in order to store the n-bit LFSR states with a primitive polynomial (feedback polynomial) of $p(x) = \sum_{i=0}^{n-1} a_i x^i$, a number of $\lceil \frac{n}{m} \rceil$ registers are needed to store LFSR state bits. For instance, for a simple 20-bit LFSR, assuming single precision operations, a single register of 32-bit width is employed to handle the LFSR state machine's computation.

As investigated in the previous section, the shift operation is necessary for the LFSR architecture, and in the conventional naïve implementation, costly bit-level shift and mask operations are mandatory at every single rotation of the LFSR state machine. This would considerably limit the RNG circuit's overall performance since these bit-access operations should be executed at each rotation. Moreover, in some scenarios, the register utilization in terms of the platform's datapath width cannot be maximized due to the available number of bits in the conventional row-major data representation. The column-major bitsliced data representation compensates for the shortcomings as mentioned above inherent to the common practice na [ıve implementation, but also maximizes the utilization of the processing units in the GPU.

4.2 Bitslicing Approach Application: CRC Example

As indicated, bitslicing technique could be employed in register-based processors in many applications. In this context, as an example, we examine the usage of column data representation in a simple 8-bit Cyclic Redundancy Check (CRC-8) to show the extensibility of the discussed method. CRC is used to check the error in communication channels with a wide range of applications in

Wireless Mobile Networks, Wired Ethernet, and countless other applications. As shown in Figure 5, a simple 8-bit CRC is constructed by shift-registers, and the state value of the CRC is changed by input stream at each cycle. Typically, a CRC is implemented on a single register, and the computation is handled by a simple shift and mask operation within the register. The CRC output of a specific input data is the final state bits stored in the register, which is used to check the original data's correctness.

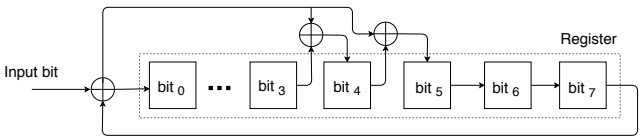

Figure 5: A simple CRC-8 example.(Naïve implementation)

Taking advantage of the bitslice data order, one can implement the CRC-8 as shown in Figure 6. Considering a processor with a 32-bit register, this representation constructs a fully paralleled CRC calculation for 32 different data streams simultaneously without any computational overhead. The shift and mask operation is completely removed and replaced with trivial register reference swapping.

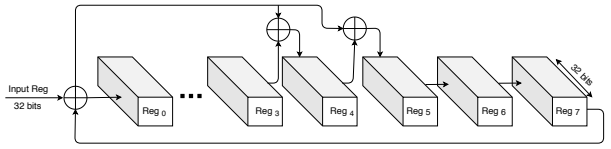

Figure 6: A simple CRC-8 example(bit-sliced implementation)

4.3 Bitsliced LFSR Implementation

Along with the fact that the costly shift and rotation operations can be further reduced to simple register swapping operations in the bitsliced data representation, here, we investigate the underlying architecture of our proposed bitsliced LFSR implementation on GPU which will be employed for the CPRNGs such as MICKEY 2.0 algorithm. Moreover, we indicate our proposed bitslicing technique's properties and advantages accompanied by the column-major data representation. As shown in Figure 6 and explained earlier, the LFSR conventional implementations suffer from a massive bit-level shift and mask operations. For fair comparison and the sake of simplicity, consider the naïve implementation of 32 parallel LFSRs governed by the primitive polynomial $g(x)$ shown in Equation 2. Note that there are at least a k number of feedback paths in the LFSR algorithm.

$$
\begin{aligned}
&g(x) = \bigoplus_{i=0}^{n-1} a_i x^i; a_i, x \in GF(2) \\
&|A| = k \\
&A = \{a_i | a_i \neq 0\}
\end{aligned}
\tag{2}
$$

As illustrated in Figure 7, each of these parallel LFSRs are managed by a single thread which results in the execution of 32 parallel threads. Hence, to generate a total number of M pseudo-random bits, each LFSR module should be shifted for $M/32$ times where many $32 \times k$ bit-level XOR operations are needed.
It is worth noting to know that to use parallel LFSRs in this manner; the shift-registers should be carefully initialized to eliminate any statistical correlation between the LFSR state machines when the output is not mixed (it is highly recommended to use non-linear mixing before generating the bitstream). Moreover, from the cryptanalysis point of view, the secure threshold for the repeat period (not $2^n - 1$ in this case) of the employed parallel system should be estimated.

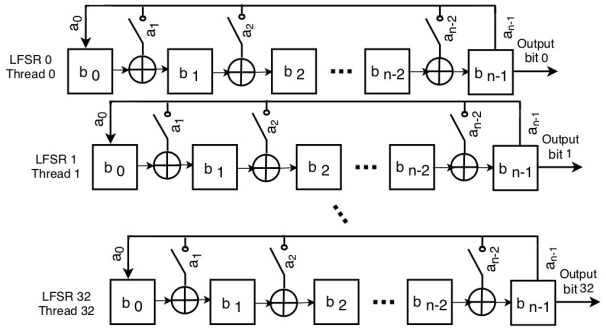

Figure 7: Number of 32 parallel LFSR modules executed in 32 threads

Considering the same scenario for pseudo-random bit generation by using LFSR, in Figure 8, the bitsliced LFSR implementation with column-major representation is demonstrated. Compared to the previous conventional model, to generate M bits in this proposed methodology, the same number of $M/32$ LFSR shift cycles are required. However, this procedure could be executed by a single thread. Also, the $32 \times k$ number of costly bit-wise XOR operations (needed at each cycle) is reduced to k number of full-width XOR operations. These operations maximize the datapath utilization. Moreover, as shown in Figure 8, the costly bit-level shift operations are replaced by cheaper and more trivial register swapping operations, which can be easily done by changing the references of the registers in the software code. Although changing references in code might be a burdensome task, it dramatically reduces the number of needed instructions in the code. Similarly, in this case, the registers should be safely initialized from the perspective of cryptanalysis, and the period of the usage should be considered. Note that to maximize the repeat period of the LFSR algorithm for PRNG, it is recommended to choose an LFSR with a higher n value.

4.4 MICKEY 2.0 Bitsliced Implementation

As explained, MICKEY 2.0 stream cipher is comprised of two 100-bit registers, namely S and R registers. By incorporating the bitslicing technique into the implementation of the MICKEY 2.0 algorithm, instead of two 100-bit registers, the data representation is altered into column-major order, and 200 registers, each containing 32 bits are employed. Note that our implementation utilizes single-precision computation, which occupies 32-bit registers. By doing

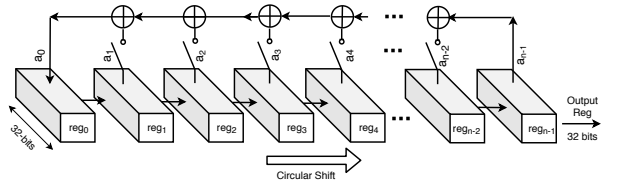

Figure 8: The execution of a 32-bit bitsliced LFSR by a single thread

so, 32 parallel Mickey stream ciphers are executed simultaneously. Figure 9 demonstrates the parallel bitsliced Mickey architecture. $Rreg_i$ and $Sreg_i$ represent the i^{th} bits of the R and S registers in the bitslicing manner, respectively. Each of these registers stores 32 different bits of the same significance for 32 parallel LFSRs modules. Hence, in our implementation, each GPU thread can execute 32 parallel Mickey 2.0 ciphers, and each thread at each clock cycle generates 32 random bits. Also, the XOR operation is executed on two 32-bits registers, and the register-based operations are utilized compared to the naïve implementation.

To securely and properly initialize our bitsliced Mickey algorithm, we employ a non-linear function to expand a carefully selected pre-stored random number set, which generates an 80-bit Initialization Vector (IV) for each MICKEY module (32×80 bits of IV for each thread). It is worth noting that the controller bit functions are designed in the bitsliced representation to calculate all the 32 bits of the controller bits for the feedback procedure. This bitsliced controller is optimized to compute the underlying parameters responsible for the feedback procedure in the algorithm. Moreover, the input handler is also executed in 32-bit width mode with no additional overhead and is optimized in terms of datapath width.

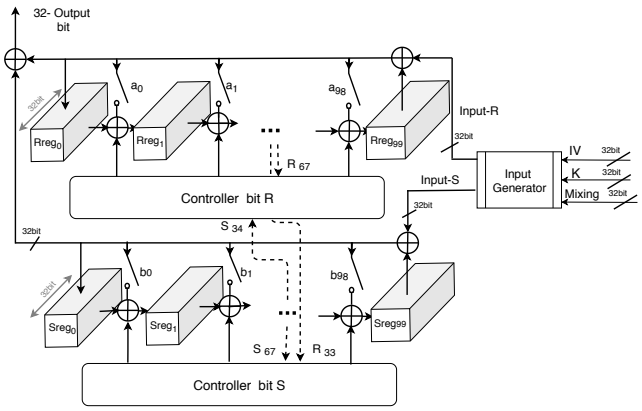

Figure 9: 32-bit Bitsliced implementation of Mickey 2.0 stream cipher algorithm

The real challenge in this technique's employment is the development of bit-level functions in the building blocks of the MICKEY algorithm. Specifically, the S controller and R controller's irregular clocking procedures are not simply described in bit-level granularity. Extending these descriptions in binary requires heavy coding deployment efforts. Some researchers have developed a high-level API to generate similar codes in this context.[30] For this reason, we have utilized an automation technique to generate such a bit-level description. The bitsliced MICKEY implementation code (CUDA/C) is generated by a higher-level transcript (i.e., written in Python language).

To manually develop any bit-level functions, one should describe thousands of lines of codes that increase the error rate in the algorithm's overall functionality. Hence, the description of all functions and blocks in the bitsliced MICKEY algorithm, including the main loop of the circuit and feedback functions, is generated by a computer program.

4.5 Shared Memory and Coalesced Access

Like any parallel implementation algorithm in GPU platforms, efficient memory access is crucial to minimize execution delay. The bitsliced implementation gives the advantage of full utilization of local registers available in the threads. The fact that proposed implementations, do not require any additional memory rather than registers makes the overall expectation very efficient in terms of access delay. However, a single memory access should be performed by each thread to write their output at each cycle. By exploiting the hierarchical architecture of toady's modern GPUs, the *Shared Memory* is employed to accelerate this procedure. In all of our implementations, a method is carried out to store the data temporarily in the device to minimize host-device communication. For instance, in MICKEY 2.0 Bitsliced, each thread, stores the output of each loop (32-bits) in the Shared Memory. After filling the shared memory capacity, the entire data is moved to *Global Memory* at a single memory access instruction where can be retrieved by the host. As could be expected, this intermediate access to Shared Memory decreases the run-time considerably compared to direct write access to Global Memory. Note that the suitable size to occupy shared memory is determined experimentally and is highly correlated to the device's specification in hand. In our CUDA implantation, simple try and error reveal the suitable shared memory occupancy, which yields a fair performance gain.

Furthermore, coalesced memory access is another crucial tweak for performance gain in GPUs. We have also implemented the proposed algorithm to have coalesced write accesses. The substantial parallel write executions in such implementations often cause a significant delay due to several problems regarding non-optimal addressing configuration. A performance gain could be achieved in both shared memory and global memory access by organizing a suitable addressing scheme via thread identifications and memory labeling.

5 EVALUATION

This section will present the evaluation results of the performance and the performance per cost metrics achieved from the execution of our proposed CPRNG implementations on several different CUDA-enabled GPUs. In this study, six Nvidia GPUs are carefully selected for evaluation purposes. The employed GPUs each have different structural characteristics such as different single and double precision throughputs and memory bandwidths. These features are carefully selected to represent a wide range of execution platforms. We selected these GPU platforms because firstly, the range

of the selected GPUs ultimately represents the platforms available to a wide range of users spanning home and enterprise users. Secondly, these GPUs give us a fair comparison with the previously proposed methods. Moreover, we demonstrate the randomness robustness and reliability of the generated bits by discussing the NIST statistical test results for our sample implementation.

5.1 Setup

This section elaborates on the specification of the hardware platforms used for the evaluation of our proposed method. GPU platforms GTX 480, GTX 980 Ti, and GTX 1050 Ti on systems with two Intel XEON E5 2697 V3 CPUs clocked at 2.6 GHz, and 128 GB of DDR3 RAM were used for the evaluation process [18]. Moreover, to prove the scalability of the proposed method Tesla V 100 and GTX 2080 Ti GPUs are also utilized for evaluation and employed in a Virtual Machine environment with the same virtually dedicated specifications. Table II shows the GPU platforms' specification details in terms of the processing power and memory bandwidth.

GPU	Single Precision (GFlops)	Double Precision (GFlops)	Mem. BW (GB/s)
GTX 480	1344	168	177
GTX 980 Ti	5632	176	337
GTX 1050 Ti	1981	62	112
GTX 1080 Ti	10609	332	484
Tesla V100	14028	7014	900
GTX 2080 Ti	11750	367	616

Table 2: Specification of the GPU platforms used for evaluation

5.2 Performance

Figure 10 illustrates the achieved performance for our proposed method based on three cryptosystems (AES, Mickey, Grain). In this Figure, we have compared our results with NVIDIA's cuRAND library (on the same platform) since all other previously proposed methods have failed to reach the cuRAND performance in PRNGs. The best result obtained on GPU V100 is acquired in the following manner of executing the implemented CUDA kernel code with fixed parameters of *thread blocks* and *thread per block* set to 64 and 256, respectively. The code's loop size is varying between 4,400 to 13,000, yielding to a different performance throughput. The cuRand results here are evaluated using the Mersenne Twister algorithm as the default *cuRand* method for RNG. Note that the peak AES performance is limited compared to the stream ciphers here. The complex bitsliced S-box mainly causes this in the AES. Also, the LFSR based structure of the stream ciphers is more compatible with the proposed bitslice technique.

5.3 Normalized Performance Evaluation

Due to the lack of access to some of the GPU platforms used in previous works that are currently outdated platforms and to deliver a fair comparison, we follow the method of normalizing the results of our proposed method and related works on parameters of performance per processing power which is shown in Figure 11. However, as already indicated, the most important available library to compare, is the cuRAND which has been considered in our evaluation.

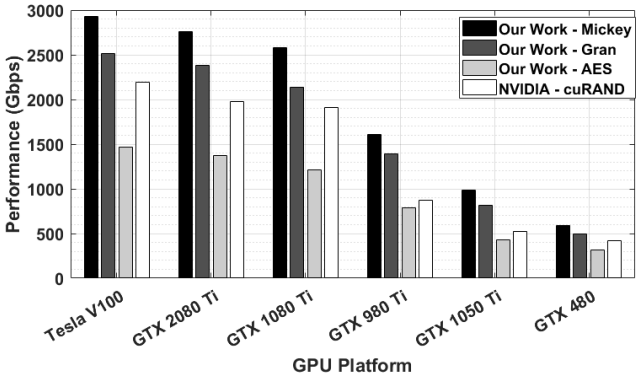

Figure 10: Comparing the performance of the proposed method on different GPU platforms

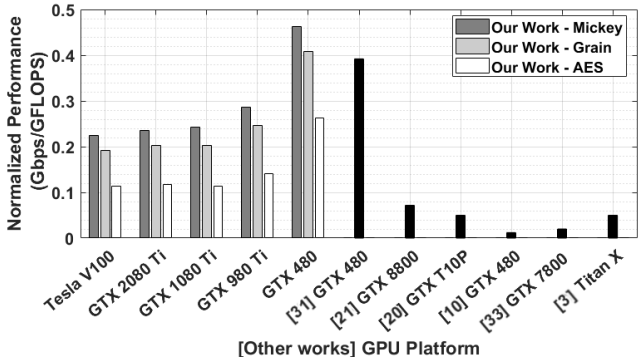

Figure 11: Comparison of normalized performance of the proposed method with previous works

5.4 Multi-GPU implementation

By taking advantage of more than one GPU in a single machine, the throughput of proposed CPRNGs can further be scaled up on a more significant number of GPUs within a single standalone machine. This approach could certainly be considered a suitable replacement for expensive Tbps optical solutions in output performance. For this goal, firstly, the input parameters (e.g., the seed, nonce, and counter) are shared and partitioned amongst all of the available GPUs. In this phase, if the GPUs are alike in the processing power metric, the input data is equally broken down into the same sized partitions. Then, each GPU executes the algorithm separately and in parallel. For example, different counter values used as the input vector with a unique key in the AES-CTR algorithm are passed to GPUs. Then each GPU generates a portion of data in parallel, which could be reconstructed sequentially. In a multi-GPU approach, a single CPU thread is required to invoke the kernel code on each of the available GPUs. OpenMP threads can manage the GPUs in parallel. The rest of the process is handled, as discussed in the single-GPU implementation. Our evaluation setup includes two GTX 1080

Ti GPUs running the Mickey 2 Bitsliced algorithm. In this setup in which two similar GPUs are used, the performance achieves a near-linear throughput (1.92 x). However, it is known that by increasing the number of GPUs to 4 or 8, the overall performance descends due to the cost of data scheduling latency, data concatenation.

Another exciting aspect of a multi-GPU implementation by the use of this approach is that the same output sequence of random bits could be generated identically in a single GPU sequentially. This feature could be handy in two-way communication where the sequence should be reconstructed at the receiver.

5.5 Statistical Tests

We use the NIST SP 800-22 version sts-2.1.2 for testing the statistical and cryptographical properties of our generated random sequences in the Mickey algorithm. Bitstreams generated by our implementation successfully pass all statistical tests. As recommended by the NIST tests, the items are executed using 1,000 instances of 1Mb of random bits generated by our solution. Note that the *significant value* here is considered to be $\alpha = 0.01$, and the results of $P-Value$ verify the randomness of the input bitstream.

Test	P-value	Proportion	Result
Frequency	0.251741	0.9982	Success
BlockFrequency	0.350485	0.9947	Success
CumulativeSums	0.4766135	0.9751	Success
Runs	0.534146	0.9781	Success
LongestRun	0.350485	0.9562	Success
Rank	0.213309	0.9950	Success
FFT	0.534146	0.9971	Success
NonOverlappingTemplate	0.4821360	0.9885	Success
OverlappingTemplate	0.739918	0.9912	Success
ApproximateEntropy	0.350485	0.9721	Success
Serial	0.7227795	0.99982	Success
LinearComplexity	0.739918	0.9840	Success

Table 3: Evaluation results of NIST statistical suite. The results are the average of 1,000 samples of 1,000,000 bit streams of random numbers generated by the proposed MICKEY algorithm.

6 DISCUSSION & CONCLUSION

High-performance, secure, and efficient random generators are required in many industrial and research applications in different fields. We propose a high-throughput, fully parallel cryptographically secure pseudo-random number generator using the bitslicing technique in this work. In this technique, the data from the conventional row-major representation is altered into column-major representation to fully utilize the computation datapath of the employed device. Using the bitslicing technique on the LFSR-based cryptographically secure MICKEY 2.0 stream cipher and other crypto-systems is implemented. This allows for high-performance random number generation by the elimination of the shift and masks operations. Various supplementary techniques, such as the utilization of shared memory and coalesced memory access are also employed to further increase performance. One of the main concerns of employing GPUs for the generation of random numbers is the delay, which, compared to the similar computational methods including ASIC, FPGA, and physical methods such as optical circuits, may be considered the major drawback of these relatively general-purpose computational platforms. The proposed method can prove advantageous when employed on applications where a slight delay is not a matter of great concern, and the performance and the cost-efficiency of the solution are considered. Our proposed methodology achieves the throughput of 2.90 Tb/s on Nvidia V 100, outperforming the Nvidia's proprietary cuRAND library while striking a notable balance in performance criteria per cost.

REFERENCES

[1] Rudolf Ahlswede and Imre Csiszár. 1993. Common randomness in information theory and cryptography. part i: secret sharing. *IEEE Transactions on Information Theory* 39, 4 (1993).

[2] Armin Ahmadzadeh, Omid Hajihassani, and Saeid Gorgin. 2018. A high-performance and energy-efficient exhaustive key search approach via GPU on DES-like cryptosystems. *The Journal of Supercomputing* 74, 1 (2018), 160–182.

[3] Mohammed Abdul Samad AL-khatib and Auqib Hamid Lone. 2018. Acoustic lightweight pseudo random number generator based on cryptographically secure LFSR. *International Journal of Computer Network and Information Security* 11, 2 (2018), 38.

[4] Steve Babbage, C Canniere, Anne Canteaut, Carlos Cid, Henri Gilbert, Thomas Johansson, Matthew Parker, Bart Preneel, Vincent Rijmen, and Matthew Robshaw. 2008. The eSTREAM portfolio. *eSTREAM, ECRYPT Stream Cipher Project* (2008), 1–6.

[5] Steve Babbage and Matthew Dodd. 2006. The stream cipher MICKEY 2.0. *ECRYPT Stream Cipher* (2006).

[6] Eli Biham. 1997. A fast new DES implementation in software. In *International Workshop on Fast Software Encryption*. Springer, 260–272.

[7] Kurt Binder, Dieter Heermann, Lyle Roelofs, A John Mallinckrodt, and Susan McKay. 1993. Monte Carlo simulation in statistical physics. *Computers in Physics* 7, 2 (1993), 156–157.

[8] Ihsan Cicek, Ali Emre Pusane, and Gunhan Dundar. 2014. A novel design method for discrete time chaos based true random number generators. *INTEGRATION, the VLSI journal* 47, 1 (2014), 38–47.

[9] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M LaConte. 2013. Medical image processing on the GPU–Past, present and future. *Medical image analysis* 17, 8 (2013), 1073–1094.

[10] Shuang Gao and Gregory D Peterson. 2013. GASPRNG: GPU accelerated scalable parallel random number generator library. *Computer Physics Communications* 184, 4 (2013), 1241–1249.

[11] Benedikt Gierlichs, Lejla Batina, Christophe Clavier, Thomas Eisenbarth, Aline Gouget, Helena Handschuh, Timo Kasper, Kerstin Lemke-Rust, Stefan Mangard, Amir Moradi, et al. 2008. Susceptibility of eSTREAM candidates towards side channel analysis. (2008).

[12] Chunye Gong, Jie Liu, Lihua Chi, Qingfeng Hu, Li Deng, and Zhenghu Gong. 2010. Accelerating Pseudo-Random Number Generator for MCNP on GPU. In *AIP Conference Proceedings*, Vol. 1281. AIP, 1335–1337.

[13] Antonio Gulli and Sujit Pal. 2017. *Deep Learning with Keras*. Packt Publishing Ltd.

[14] O. Hajihassani, S. Khalaj Monfared, S. H. Khasteh, and S. Gorgin. 2019. Fast AES Implementation: A High-throughput Bitsliced Approach. *IEEE Transactions on Parallel and Distributed Systems* (2019), 1–1.

[15] Martin Hell, Thomas Johansson, and Willi Meier. 2007. Grain: a stream cipher for constrained environments. *IJWMC* 2, 1 (2007), 86–93.

[16] Benjamin Jun and Paul Kocher. 1999. The Intel random number generator. *Cryptography Research Inc. white paper* 27 (1999), 1–8.

[17] Ido Kanter, Yaara Aviad, Igor Reidler, Elad Cohen, and Michael Rosenbluh. 2010. An optical ultrafast random bit generator. *Nature Photonics* 4, 1 (2010), 58.

[18] Mohammad Sina Kiarostami, Mohammad Reza Daneshvaramoli, Saleh Khalaj Monfared, Dara Rahmati, and Saeid Gorgin. 2019. Multi-Agent non-Overlapping Pathfinding with Monte-Carlo Tree Search. In *2019 IEEE Conference on Games (CoG)*. IEEE, 1–4.

[19] Philip Koopman. 2002. 32-bit cyclic redundancy codes for internet applications. In *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 459–468.

[20] William B Langdon. 2008. A fast high quality pseudo random number generator for graphics processing units. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 459–465.

[21] William B Langdon. 2009. A fast high quality pseudo random number generator for nVidia CUDA. In *Proceedings of the 11th Annual Conference Companion on*

Genetic and Evolutionary Computation Conference: Late Breaking Papers. ACM, 2511–2514.

[22] Pierre L'Ecuyer. 1990. Random numbers for simulation. *Commun. ACM* 33, 10 (1990), 85–97.

[23] Pierre L'Ecuyer and Richard Simard. 2007. TestU01: AC library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)* 33, 4 (2007), 22.

[24] Pu Li, Ya Guo, Yanqiang Guo, Yuanlong Fan, Xiaomin Guo, Xianglian Liu, Kunying Li, K Alan Shore, Yuncai Wang, and Anbang Wang. 2018. Ultrafast fully photonic random bit generator. *Journal of Lightwave Technology* 36, 12 (2018), 2531–2540.

[25] Yang Liu, Qi Zhao, Ming-Han Li, Jian-Yu Guan, Yanbao Zhang, Bing Bai, Weijun Zhang, Wen-Zhao Liu, Cheng Wu, Xiao Yuan, et al. 2018. Device-independent quantum random-number generation. *Nature* 562, 7728 (2018), 548.

[26] George Marsaglia et al. 2003. Xorshift rngs. *Journal of Statistical Software* 8, 14 (2003), 1–6.

[27] Michael Mascagni. 1999. SPRNG: A scalable library for pseudorandom number generation. In *Recent Advances in Numerical Methods and Applications II*. World Scientific, 284–295.

[28] Michael Mascagni and Ashok Srinivasan. 2000. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software (TOMS)* 26, 3 (2000), 436–461.

[29] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8, 1 (1998), 3–30.

[30] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, and Gilles Muller. 2018. Usuba: optimizing & trustworthy bitslicing compiler. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*. 1–8.

[31] Nimalan Nandapalan, Richard P Brent, Lawrence M Murray, and Alistair P Rendell. 2011. High-performance pseudo-random number generation on graphics processing units. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 609–618.

[32] Naoki Nishikawa, Hideharu Amano, and Keisuke Iwai. 2017. Implementation of Bitsliced AES Encryption on CUDA-Enabled GPU. In *Network and System Security*, Zheng Yan, Refik Molva, Wojciech Mazurczyk, and Raimo Kantola (Eds.). Springer International Publishing, Cham, 273–287.

[33] Wai-Man Pang, Tien-Tsin Wong, and Pheng-Ann Heng. 2008. Generating massive high-quality random numbers using GPU. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 841–847.

[34] Saeid Rahmani, Armin Ahmadzadeh, Omid Hajihassani, S Mirhosseini, and Saeid Gorgin. 2016. An efficient multi-core and many-core implementation of k-means clustering. In *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 128–131.

[35] Vincent Rijmen and Joan Daemen. 2001. Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology* (2001), 19–22.

[36] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. 2001. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*. Technical Report. Booz-Allen and Hamilton Inc Mclean Va.

[37] Guido Di Patrizio Stanchieri, Andrea De Marcellis, Elia Palange, and Marco Faccio. 2019. A True Random Number Generator Architecture Based on a Reduced Number of FPGA Primitives. *AEU-International Journal of Electronics and Communications* (2019).

[38] Berk Sunar. 2009. True random number generators for cryptography. In *Cryptographic Engineering*. Springer, 55–73.

[39] Myles Sussman, William Crutchfield, and Matthew Papakipos. 2006. Pseudorandom number generation on the GPU. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. ACM, 87–94.

[40] Robert Szerwinski and Tim Güneysu. 2008. Exploiting the power of GPUs for asymmetric cryptography. In *International Workshop on Cryptographic hardware and embedded systems*. Springer, 79–99.

[41] Je Sen Teh, Azman Samsudin, Mishal Al-Mazrooie, and Amir Akhavan. 2015. GPUs and chaos: a new true random number generator. *Nonlinear Dynamics* 82, 4 (2015), 1913–1922.

[42] NVIDIA Corporation. [n.d.]. The NVIDIA CUDA Random Number Generation library (cuRAND). https://developer.nvidia.com/curand

[43] David Barrie Thomas, Lee Howes, and Wayne Luk. 2009. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 63–72.

[44] John von Neumann. 1963. Various techniques used in connection with random digits. *John von Neumann, Collected Works* 5 (1963), 768–770.

[45] Hesong Xu, Nicola Massari, Leonardo Gasparini, Alessio Meneghetti, and Alessandro Tomasi. 2019. A SPAD-based random number generator pixel based on the arrival time of photons. *Integration* 64 (2019), 22–28.