Accelerated implementation for testing IID assumption of NIST SP 800-90B using GPU

Yewon Kim [Corresp., 1], **Yongjin Yeom** [1, 2]

[1] Department of Financial Information Security, Kookmin University, Seoul, South Korea

[2] Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul, South Korea

Corresponding Author: Yewon Kim
Email address: fdt150@kookmin.ac.kr

In cryptosystems and cryptographic modules, insufficient entropy of the noise sources that serve as the input into random number generator (RNG) may cause serious damage, such as compromising private keys. Therefore, it is necessary to estimate the entropy of the noise source as precisely as possible. The National Institute of Standards and Technology (NIST) published a relevant standard document known as Special Publication (SP) 800-90B, which describes the method for estimating the entropy of the noise source that is the input into an RNG. The principles and statistical tests in SP 800-90B have been analyzed theoretically; however, it is hard to find research on the efficient implementation thereof. The NIST offers two programs for running the entropy estimation process of SP 800-90B, written in Python and C++. The running time for estimating the entropy is more than one hour for each noise source. As an RNG tends to use several noise sources, the times of the NIST estimation are a burden for developers as well as evaluators working for the Cryptographic Module Validation Program. In this study, we propose a GPU-based parallel implementation of the most time-consuming part of the entropy estimation, namely the process of the independent and identically distributed assumption testing. To achieve maximal GPU performance, we propose a scalable method that adjusts the optimal size of the global memory allocations depending on GPU capability and balances the workload between streaming multiprocessors. The experimental results demonstrate that our method is up to 33 times faster than that of the NIST package.

Accelerated implementation for testing IID assumption of NIST SP 800-90B using GPU

Yewon Kim[1] **and Yongjin Yeom**[1, 2]

[1]**Department of Financial Information Security, Kookmin University, Seoul, Korea**
[2]**Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul, Korea**

Corresponding author:
Yewon Kim[1]

Email address: fdt150@kookmin.ac.kr

ABSTRACT

In cryptosystems and cryptographic modules, insufficient entropy of the noise sources that serve as the input into random number generator (RNG) may cause serious damage, such as compromising private keys. Therefore, it is necessary to estimate the entropy of the noise source as precisely as possible. The National Institute of Standards and Technology (NIST) published a relevant standard document known as Special Publication (SP) 800-90B, which describes the method for estimating the entropy of the noise source that is the input into an RNG. The principles and statistical tests in SP 800-90B have been analyzed theoretically; however, it is hard to find research on the efficient implementation thereof. The NIST offers two programs for running the entropy estimation process of SP 800-90B, written in Python and C++. The running time for estimating the entropy is more than one hour for each noise source. As an RNG tends to use several noise sources, the times of the NIST estimation are a burden for developers as well as evaluators working for the Cryptographic Module Validation Program. In this study, we propose a GPU-based parallel implementation of the most time-consuming part of the entropy estimation, namely the process of the independent and identically distributed assumption testing. To achieve maximal GPU performance, we propose a scalable method that adjusts the optimal size of the global memory allocations depending on GPU capability and balances the workload between streaming multiprocessors. The experimental results demonstrate that our method is up to 33 times faster than that of the NIST package.

INTRODUCTION

A random number generator (RNG) generates the random numbers required to construct the cryptographic keys, nonce, salt, and sensitive security parameters used in cryptosystems and cryptographic modules. In general, an RNG produces random numbers (output) via a deterministic algorithm, depending on the noise sources (input). Hence, if its input is affected by the low entropy of the noise sources, the output may be compromised. It is easy to find examples that show the importance of entropy in operating systems. Heninger et al. (2012) describes they can obtain the RSA/DSA private keys for some TLS/SSH hosts due to insufficient entropy of Linux PRNG during the key generation process. Ding et al. (2014) investigated the amount of the entropy of Linux PRNG running on Android in boot-time. Kaplan et al. (2014) demonstrated an IPv6 Denial of Service attack and a stack canary bypass with the weaknesses of insufficient entropy in the boot-time of Android. Also, Kim et al. (2013) presents a technique to recover PreMasterSecret (PMS) of the first SSL session in Android by 2^{58} complexity since PMS is generated from insufficient entropy of OpenSSL PRNG at boot time. In addition, Ristenpart and Yilek (2010); Bernstein et al. (2013); Michaelis et al. (2013); Schneier et al. (2015); Yoo et al. (2017) describe the attacks caused by weakness of entropy collectors or wrong estimations of the entropy that are exaggerated or too conservative.

Insufficient entropy of the noise source that is the input into the RNG may cause serious damage in cryptosystems and cryptographic modules. Thus, it is necessary to estimate the entropy of the noise source as precisely as possible. The United States National Institute of Standards and Technology (NIST) Special Publication (SP) 800-90B (Barker and Kelsey, 2012;

Sönmez Turan et al., 2016, 2018) is a standard document for estimating the entropy of the noise source. This document is currently used in the Cryptographic Module Validation Program (CMVP) and has been cited as a recommendation for entropy estimation in an ISO standard document ISO/IEC-20543 (2019) for test and analysis methods of random bit generators. The principles of entropy estimators in SP 800-90B have been investigated and analyzed theoretically (Kang et al., 2017; Zhu et al., 2017, 2019). However, it is difficult to find research on the efficient implementation of the entropy estimation process of SP 800-90B. The general flow of the entropy estimation process in the final version of SP 800-90B (Sönmez Turan et al., 2018) is summarized in Figure 1.

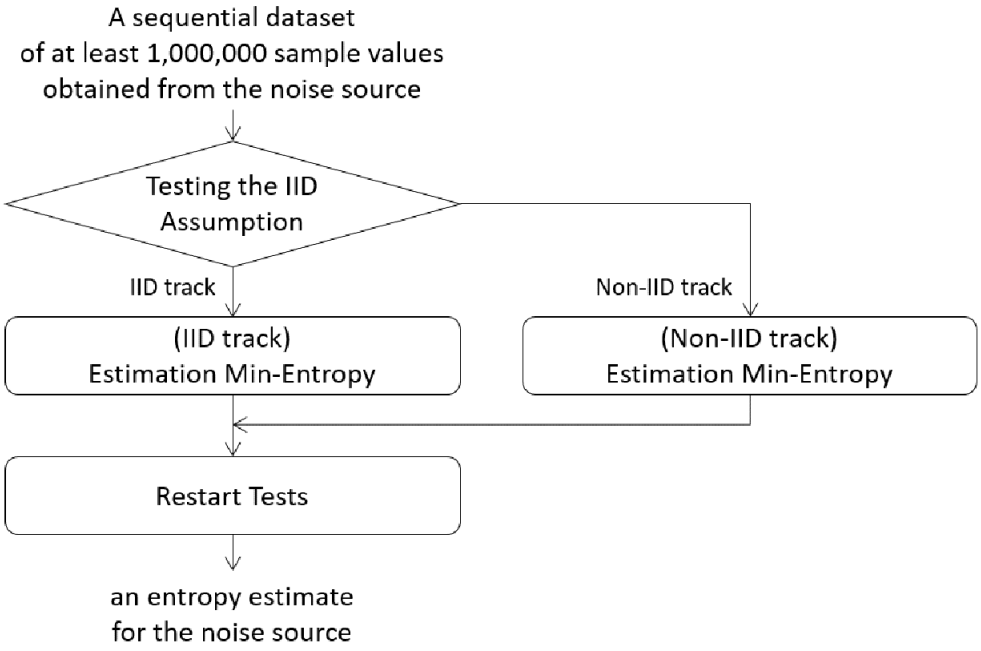

Figure 1. Flow of entropy estimation process of SP 800-90B.

The NIST provides two programs on GitHub (NIST, 2015) for the entropy estimation process of SP 800-90B. The first program is for the entropy estimation process of the second draft of SP 800-90B (Sönmez Turan et al., 2016), written in Python. The second program is for the entropy estimation process of the final version (Sönmez Turan et al., 2018) of SP 800-90B, written in C++. Table 1 displays the execution times of two single-threaded NIST programs on the central processing unit (CPU). The noise source used as input is GetTickCount, with a sample size of 8 bits. GetTickCount can be collected through the `GetTickCount()` function in the Windows environment. In Table 1, the process of testing the independent and identically distributed (IID) assumption, hereinafter referred to as the IID test, consumes the majority of the total execution time in both NIST programs.

As recommended by the CMVP, the RNG applied in cryptosystems and cryptographic modules should use at least one noise source as the input for security. Therefore, the entropy of each noise source used as the RNG input should be estimated to analyze the security of the RNG. As the noise sources are affected by the environment from which they are collected, the entropy of each noise source should be estimated repeatedly to increase the confidence in the estimating results. For example, suppose that a cryptographic module developer analyzes the security of the RNG in his/her module using the NIST program written in C++. Assume that the module supports two operating systems, and 10 noise sources are used as input into the RNG in each operating system. Moreover, assume that he/she estimates the entropy of a noise source by repeating 5 times to increase the confidence in the results. According to Table 1, the

	NIST program written in Python	NIST program written in C++
Testing IID assumption (IID test)	17 h	1 h 10 min
[IID track] Estimation entropy	< 1 s	1 s
[Non-IID track] Estimation entropy	15 min	20 s
Restart tests	2 s	2 min
Total execution time	17 h 16 min	1 h 13 min

Table 1. Execution time of each single-threaded NIST program for entropy estimation process (noise source: GetTickCount; noise sample size: 8 bits).

NIST program requires approximately 1 h to estimate the entropy of one noise source. Therefore, at least 20 h are required to analyze the security of the developer's RNG. However, since the entropy of each noise source should be estimated several times (assumed five times), it may need more than 100h that is four days. As this runtime may be burdensome for developers, it can be tempting to use an RNG without security analysis. Thus, if the developer's RNG is vulnerable, this vulnerability is likely to affect the overall security of the cryptographic module.

Graphics processing units (GPUs) are excellent candidates to perform the acceleration of this IID test. GPUs were initially designed for accelerating computer graphics and image processing. In recent years, GPUs have also become more flexible, even allowing them to be used for general computations. The use of GPUs for performing computations handled by CPUs is known as general-purpose computing on GPUs (GPGPUs). New parallel computing platforms and programming models, such as the computing unified device architecture (CUDA) released by NVIDIA, enable software developers to leverage GPGPUs for various applications. GPGPUs are used in cryptography as well as areas including signal processing and artificial intelligence. Numerous studies have been conducted on the parallel implementations of cryptographic algorithms such as AES, ECC, and RSA (Neves and Araujo, 2011; Li et al., 2012; Pan et al., 2016; Ma et al., 2017; Li et al., 2019) and on the acceleration of cryptanalysis, including hash collision attacks using GPUs (Stevens et al., 2017).

In this study, we propose a parallel implementation of the IID test by using multiple optimization techniques. To process the entire IID test in parallel, approximately 9 GB or more of the global memory of the GPU are required. We implement the IID test in parallel by setting the adaptive sizes of the global memory used in the kernel function so that maximal performance improvement can be obtained from the GPU specification in use. Our experiments support the finding that our parallel implementation can achieve optimized results with up to 33 times higher performance than that of the NIST.

The remainder of this paper is organized as follows. Section 2 introduces the CUDA GPU programming model and the IID test of SP 800-90B. Section 3 outlines our GPU-based parallel implementation of the IID test. In section 4, the experimental results on the optimization and performance of our method are presented and analyzed. Finally, Section 5 summarizes and concludes the paper.

PRELIMINARIES

CUDA programming model

NVIDIA CUDA (NVIDIA, 2020b) is the most widely used programming model for GPUs. CUDA uses the single instruction multiple thread (SIMT) model. A *kernel* is a function that performs the same instruction on the GPU in parallel. A *thread* is the smallest unit operating the instructions of the kernel function. Multiple threads are grouped into a *CUDA block*, and multiple blocks are grouped into a *grid*.

A CUDA-capable GPU contains numerous *CUDA cores*, that are fundamental computing units and execute the threads. CUDA cores are collected into groups *streaming multiprocessors*

(SMs).

A kernel is launched from the host (CPU) to run on GPU and generate a collection of threads organized into blocks. Each CUDA block is assigned to one of the SMs on the GPU and execute independently on GPU. The mapping between blocks and SMs is done by a CUDA scheduler (Vaidya, 2018). An SM can concurrently execute the smaller group of threads, which is called a *warp*. All threads in a warp execute the same instruction, and the number of threads in a warp is 32 on most CUDA-capable GPUs. The latency can occur, such as data required for computation have not yet been fetched from global memory that the access is slow. To hide the latency, an SM can take place *context-switching*, which transfers control to another warp while waiting for the results.

The memory of CUDA-capable GPU includes global memory, local memory, shared memory, register, constant memory, and texture memory. Table 2 shows the memory types listed from top to bottom by access speed from fast to slow, and the principal characteristics.

Memory	Location on/off chip	Access	Scope	Lifetime
Register	On	R/W	1 thread	Thread
Local	Off	R/W	1 thread	Thread
Shared	On	R/W	All threads in block	Block
Global	Off	R/W	All threads + host	Host allocation
Constant	Off	R	All threads + host	Host allocation
Texture	Off	R	All threads + host	Host allocation

Table 2. Memory of CUDA-capable GPU (NVIDIA, 2020a).

A basic frame of the program using the CUDA programming model is as follows; allocate memory in the device (GPU) and transfer data from the host to the device (if necessary); launch the kernel; transfer data from the device to the host (if required).

IID test for entropy estimation

The IID test of SP 800-90B consists of permutation testing and five additional chi-square tests. The permutation testing is the most time-consuming step in the entire IID test. Therefore, we only focus on the permutation testing in this study.

We define several terms before introducing the permutation testing. A *sample* is data obtained from one output of the (digitized) noise source and the *sample size* is the size of the (noise) sample in bits. For example, we collect a sample of the noise source GetTickCount in Windows by calling the `GetTickCount()` function once. In this case, the sample size is 32 bits. However, as certain estimators of SP 800-90B do not support samples larger than 8 bits, it is necessary to reduce the sample size. GetTickCount is the elapsed time (in milliseconds) since the system was started. Thus, it is thus easy to conclude that the low-order bits in the sample of GetTickCount contain most of the variability. Therefore, it would be reasonable to reduce the 32-bit sample to an 8-bit sample by using the lowest 8 bits. The tests of SP 800-90B are performed on input data consisting of one million samples, where each sample has a reduced size of 8 bits. Furthermore, the maximum of the min-entropy per sample is 8.

Algorithm 1 presents the algorithm of the permutation testing described in SP 800-90B. The permutation testing is the step that involves identifying evidence against the null hypothesis that the noise source is IID. The permutation testing first performs statistical tests on one million samples of the noise source, namely the original data. We refer to the results of the statistical tests as the original test statistics. Thereafter, permutation testing is carried out $10,000$ iterations, as follows: In each iteration, the original data are shuffled, the statistical tests are performed on the shuffled data, and the results are compared with the original test statistics. After $10,000$ iterations, the ranking of the original test statistics among the shuffled test statistics is computed. If the rank belongs to the top 0.05% or bottom 0.05%, the permutation testing

Algorithm 1 Permutation testing (Sönmez Turan et al., 2018).

Input: $S = (s_1, ..., s_L)$, where s_i is the noise sample and $L = 1,000,000$.
Output: Decision on the IID assumption.

1: **for** statistical test i **do**
2: Assign the counters $C_{i,0}$ and $C_{i,1}$ to zero.
3: Calculate the test statistic $TEST_i^{\mathrm{IN}}$ on S.
4: **end for**
5: **for** $j = 1$ to $10,000$ **do**
6: Permute S using the Fisher–Yates shuffle algorithm.
7: Calculate the test statistic $TEST_i^{\mathrm{Shuffle}}$ on the shuffled data.
8: **if** $(TEST_i^{\mathrm{Shuffle}} > TEST_i^{\mathrm{IN}})$ **then**
9: Increment $C_{i,0}$.
10: **else if** $(TEST_i^{\mathrm{Shuffle}} = TEST_i^{\mathrm{IN}})$ **then**
11: Increment $C_{i,1}$.
12: **end if**
13: **end for**
14: **if** $((C_{i,0} + C_{i,1} \leq 5)\,\mathrm{or}\,(C_{i,0} \geq 9,995))$ for any i **then**
15: Reject the IID assumption.
16: **else**
17: Assume that the noise source outputs are IID.
18: **end if**

determines that the original data (input) are not IID. That is, it is concluded that the original data are not IID if Equation 1 is satisfied for any i that is the index of the statistical test. For any i, the counter $C_{i,0}$ is the number of j in step 5 of Algorithm 1 satisfying the shuffled test statistic $TEST_i^{\mathrm{Shuffle}} >$ the original test statistic $TEST_i^{\mathrm{IN}}$. The counter $C_{i,1}$ is the number of j satisfying $TEST_i^{\mathrm{Shuffle}} = TEST_i^{\mathrm{IN}}$, whereas the counter $C_{i,2}$ is the number of j satisfying $TEST_i^{\mathrm{Shuffle}} < TEST_i^{\mathrm{IN}}$.

$$(C_{i,0} + C_{i,1} \leq 5)\ \mathrm{or}\ (C_{i,0} \geq 9,995) \tag{1}$$

Equivalently, the permutation testing determines that the original data are IID if Equation 2 is satisfied for all i that is the index of the statistical test.

$$(C_{i,0} + C_{i,1} > 5)\ \mathrm{and}\ (C_{i,1} + C_{i,2} > 5) \tag{2}$$

The NIST optimized the permutation testing of the NIST program written in C++ using Equation 2. Thus, even if each statistical test is not performed $10,000$ times completely, the permutation testing can determine that the input data are IID. Algorithm 2 is the improved version of the permutation testing optimized by the NIST.

We briefly introduce the shuffle algorithm and the tests used in the permutation testing. The shuffle algorithm is the Fisher–Yates shuffle algorithm presented in Algorithm 3. The permutation testing uses 11 statistical tests, the names of which are as follows:

- Excursion test
- Number of directional runs
- Length of directional runs
- Number of increases and decreases
- Number of runs based on the median
- Length of runs based on the median
- Average collision test statistic
- Maximum collision test statistic
- Periodicity test

Algorithm 2 Permutation testing of NIST program written in C++.

Input: $S = (s_1, ..., s_L)$, where s_i is the noise sample and $L = 1,000,000$.
Output: Decision on the IID assumption.

for statistical test i **do**
  Assign the counters $C_{i,0}$ and $C_{i,1}$ to zero.
  Calculate the test statistic $TEST_i^{\mathrm{IN}}$ on S.
end for
for $j = 1$ to $10,000$ **do**
  Permute S using the Fisher–Yates shuffle algorithm.
  for statistical test i **do**
    if $status_i = true$ **then**
      Calculate the test statistic $TEST_i^{\mathrm{Shuffle}}$ on the shuffled data.
      if $(TEST_i^{\mathrm{Shuffle}} > TEST_i^{\mathrm{IN}})$ **then**
        Increment $C_{i,0}$.
      else if $(TEST_i^{\mathrm{Shuffle}} = TEST_i^{\mathrm{IN}})$ **then**
        Increment $C_{i,1}$.
      else
        Increment $C_{i,2}$.
      end if
      if $((C_{i,0} + C_{i,1} > 5) \mathrm{and} (C_{i,1} + C_{i,2} > 5))$ **then**
        $state_i = false$.
      end if
    end if
  end for
end for
if $((C_{i,0} + C_{i,1} \leq 5) \mathrm{or} (C_{i,0} \geq 9,995))$ for any i **then**
  Reject the IID assumption.
else
  Assume that the noise source outputs are IID.
end if

Algorithm 3 Fisher–Yates shuffle (Sönmez Turan et al., 2018).

Input: $S = (s_1, ..., s_L)$, where s_i is the noise sample and $L = 1,000,000$.
Output: Shuffled $S = (s_1, ..., s_L)$.

for i from L downto 1 **do**
  Generate a random integer j such that $1 \leq j \leq i$.
  Swap s_j and s_i.
end for

- Covariance test
- Compression test*

The aim of the periodicity test is to measure the number of periodic structures in the input data. The aim of the covariance test is to measure the strength of the lagged correlation. Thus, the periodicity and covariance tests take a lag parameter as input and each test is repeated for five different values of the lag parameter: 1, 2, 8, 16, and 32 (Sönmez Turan et al., 2018). Therefore, a total of 19 statistical tests are used in the permutation testing.

If the input data are binary (that is, the sample size is 2), one of two conversions is applied to the input data for some of the statistical tests. The descriptions of each conversion and the names of the statistical tests using that conversion are as follows (Sönmez Turan et al., 2018):

Conversion I

Conversion I divides the input data into 8-bit non-overlapping blocks and counts the number of 1s in each block. If the size of the final block is less than 8 bits, zeroes are appended. The numbers and lengths of the directional runs, numbers of increases and decreases, periodicity test, and covariance test apply Conversion I to the input data.

Conversion II

Conversion II divides the input data into 8-bit non-overlapping blocks and calculates the integer value of each block. If the size of the final block is less than 8 bits, zeroes are appended. The average collision test statistic and maximum collision test statistic apply Conversion II to the input data.

As an example of the conversions, let the binary input data be $(0,1,1,0,0,1,1,0,1,0,1,1)$. For Conversion I, the first 8-bit block includes four 1s and the final block, which is not complete, includes three 1s. Thus, the output data of Conversion I are $(4,3)$. For Conversion II, the integer value of first block is 102 and the final block becomes $(1,0,1,1,0,0,0,0)$ with an integer value of 88. Thus, the output of Conversion II is $(102,88)$.

PROPOSED GPU IMPLEMENTATION

Target of parallel processing

Steps 5 to 22 of Algorithm 2, with $10,000$ iterations, consume most of the processing time of the permutation testing. The shuffle algorithm and 19 statistical tests are performed on the data with one million samples of the noise source in each iteration. Hence, it is natural to consider the GPU-based parallel implementation of $10,000$ iterations, which are processed sequentially in the permutation testing.

The implementation of the compression test* differs from those of the other statistical tests used in the permutation testing. The compression test* uses bzip2 (Seward, 2019), which compresses the input data using the Burrows–Wheeler transform (BWT), the move-to-front (MTF) transform, and Huffman coding. Research on the parallel implementation of bzip2 using a GPU is still ongoing. In Patel et al. (2012), all three main steps, namely the BWT, the MTF transform, and Huffman coding, were implemented in parallel using a GPU. However, the performance was 2.78 times slower than that of the CPU implementation. In Shastry et al. (2016), only the BWT was computed on a GPU and a performance improvement of 1.4 times that of the standard CPU-based algorithm was achieved. However, this approach is not applicable in this case, because our parallel test should be implemented in the GPU together with other permutation tests. Moreover, the compression test does not play a key role in Algorithm 2. That is, it is infrequent for a noise source to be determined as non-IID only by the compression test results among the 19 statistical tests used in the permutation testing. Therefore, we design the GPU-based parallel implementation of the permutation testing consisting of the shuffle algorithm and 18 statistical tests, without the compression algorithm.

Overview of parallel permutation testing

Approximately 9.3 GB ($= 10,000 \times$ one million bytes of data) of the global memory of the GPU is required for the CPU to invoke a CUDA kernel to process $10,000$ iterations of the permutation

testing in parallel on the GPU. Some GPUs do not have more than 9 GB of global memory. Therefore, we propose parallel implementation of the permutation testing, which processes N iterations in parallel on the GPU according to the user's GPU specification and repeats this process $R = \lceil 10{,}000/N \rceil$ times.

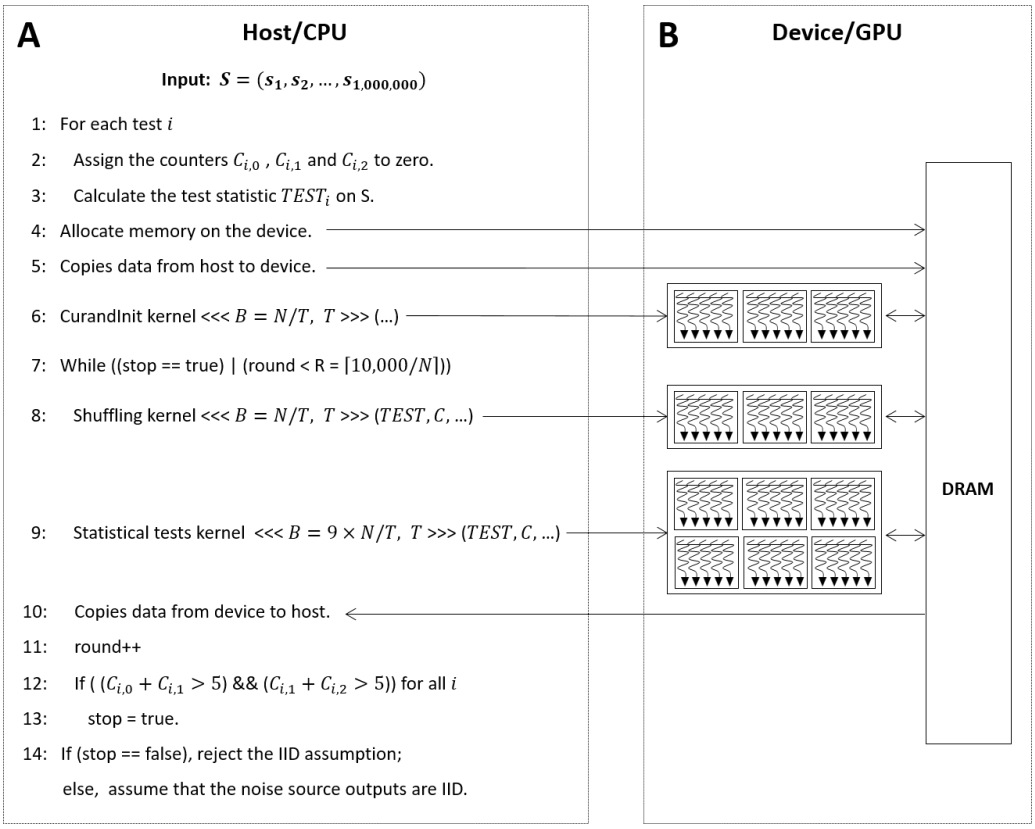

Figure 2. CPU/GPU workflow of permutation testing.
(A) Code running on the host/CPU. (B) Code running on the device/GPU.

Figure 2 presents the workflow of the CPU and GPU. The *host* refers to a general CPU that executes the program sequentially, whereas the *device* refers to a parallel processor such as a GPU. In steps 1 to 3 of Figure 2, the host performs 18 statistical tests on one million bytes of the input data (*without shuffling*) and holds the results. In step 4, the host calls a function that allocates the device memory required to process N iterations in parallel on the device. The use and sizes of the variables are listed in Table 3. In step 5, the input data (No. 1 in Table 3), and the results of the statistical tests in steps 1 to 3 (No. 4 in Table 3) are copied from the host to the device. In step 6, the host launches a CUDA kernel `CurandInit`, which initializes the N seeds used in the `curand()` function. The `curand()` function that generates random numbers using seeds on the device is invoked by the CUDA kernel `Shuffling`. When the host receives the completion of the kernel `CurandInit`, the host proceeds to steps 7 to 13, in which N iterations are divided into R rounds and each round processes in parallel on the device. To process N iterations, the host launches the CUDA kernel `Shuffling` (step 8) and then launches the CUDA kernel `Statistical test` (step 9) as soon as the host receives the completion of the kernel `Shuffling`. When the host receives the completion of the kernel `Statistical test`, in step 10, the counters $C_{i,0}$, $C_{i,1}$, and $C_{i,2}$ for $i \in \{1, 2, \ldots, 18\}$, which indicate the indices of the statistical tests, are copied from the device to the host. Following the operations in steps 17 to 19 of Algorithm 2, which correspond to those in steps 12 and 13 of Figure 2, the host moves on to step 14 if Equation 2 is satisfied for all i. Finally, in step 14, the host determines whether or not the input data are IID.

When the input data is binary, two conversions should be considered when designing the

No.	Use of variable	Size of variable (bytes)
1	Original data (input)	$1,000,000$
2	N shuffled data	$N \times 1,000,000$
3	N seeds used by `curand()` function	$N \times$ sizeof(curandState) $= N \times 48$
4	18 Original test statistics	$N \times$ sizeof(double) $= 144$
5	Counter $C_{i,0}, C_{i,1}, C_{i,2}$ for $1 \leq i \leq 18$	$18 \times$ sizeof(int) $\times 3 = 216$
6	N shuffled data after Conversion II (Only used if the input is binary)	$N \times 125,000$

Table 3. Use and sizes of variables allocated to GPU.

CUDA kernels. Therefore, we describe the CUDA kernels designed to process N iterations in parallel on the GPU depending on whether the input data is binary. The descriptions of the CUDA kernels `Shuffling` and `Statistical test` for non-binary noise sample are as follows:

CUDA kernel `Shuffling`

The kernel `Shuffling` generates N shuffled data by permuting one million bytes of the original data N times in parallel. Thus, each of N CUDA threads permutes the original data using the Fisher–Yates shuffle algorithm and then stores the shuffled data in the global memory of the device. As the shuffle algorithm uses the `curand()` function, each thread uses its unique seed that is initialized by the kernel `CurandInit` with its index, respectively.

CUDA kernel `Statistical test`

The kernel `Statistical test` performs 18 statistical tests on each of N shuffled data, and compares the shuffled and original test statistics. The size of each shuffled data is one million bytes and N shuffled data are stored in the global memory of the device. In this section, we present two methods that can easily be designed to handle this process in parallel on the GPU, and finally, we propose an optimized method.

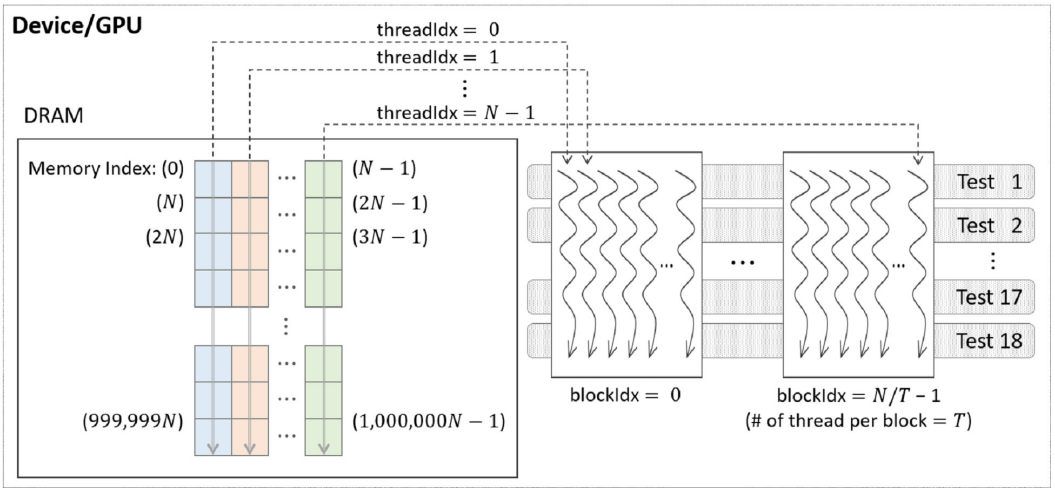

Figure 3. General parallel method 1 of kernel `Statistical test`.

Parallelization method 1 One CUDA thread performs 18 statistical tests sequentially on one shuffled dataset. This method is illustrated in Figure 3. If this method is applied to the kernel `Statistical test`, $B' = (N/T)$ CUDA blocks are used when the number of

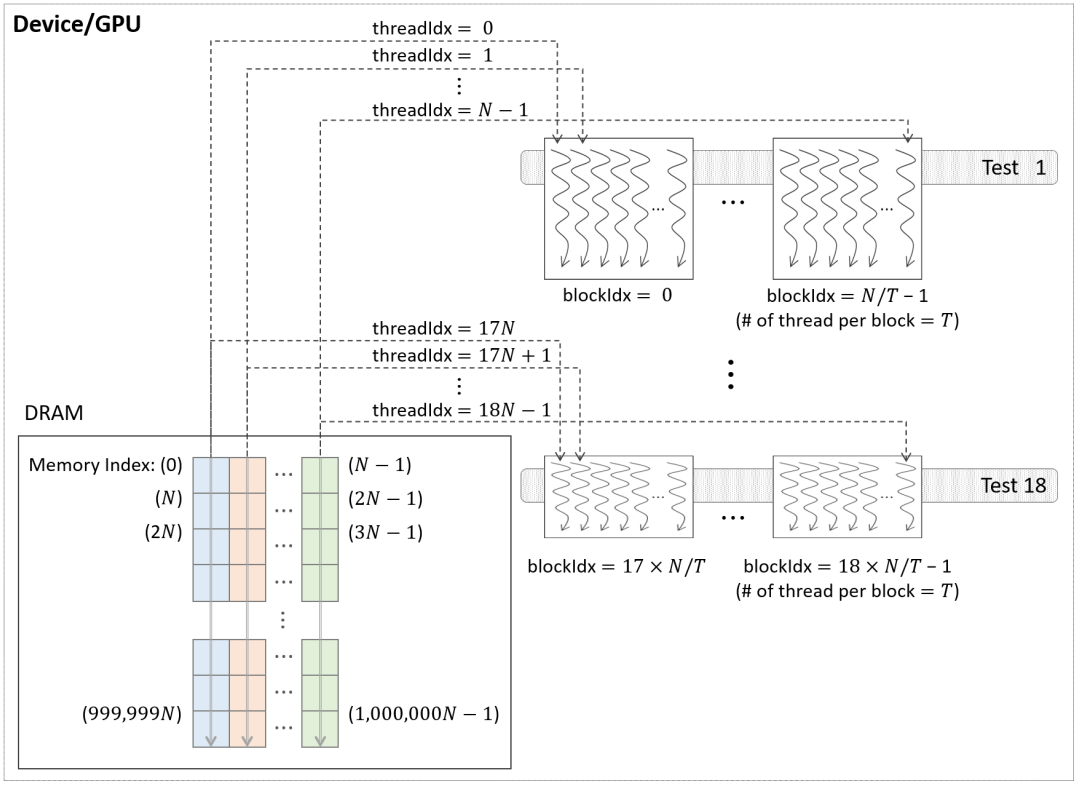

Figure 4. General parallel method 2 of kernel `Statistical test`.

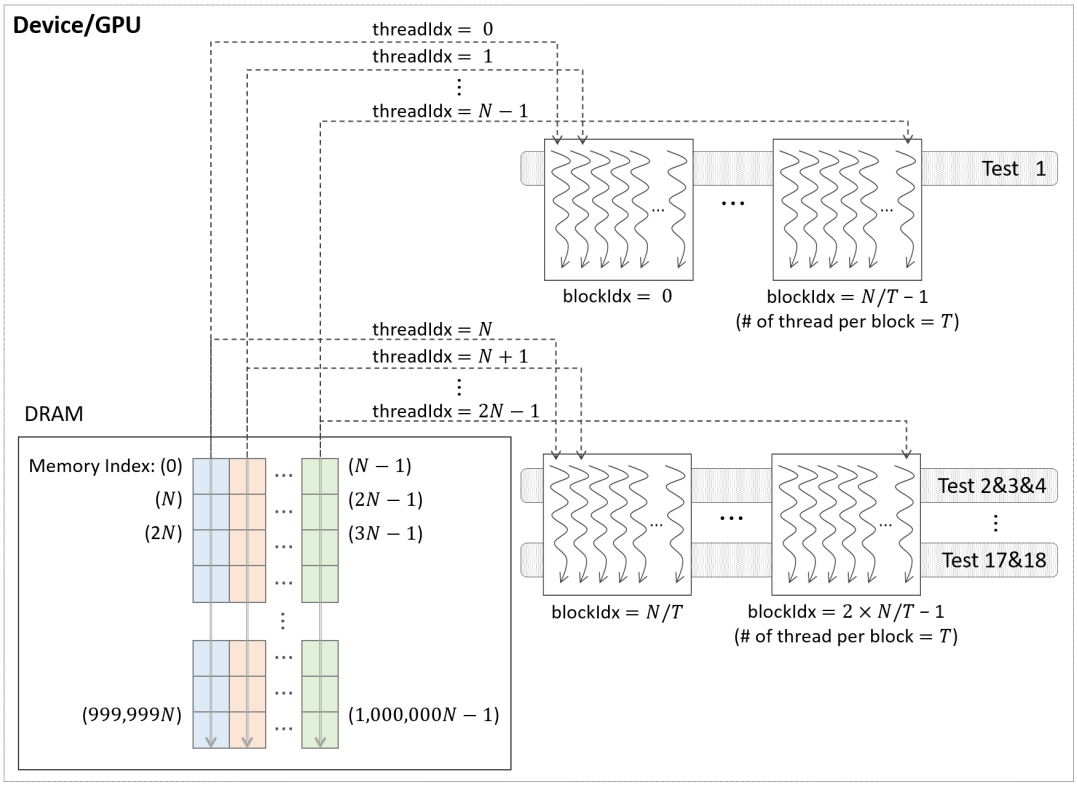

Figure 5. Proposed optimization method of kernel `Statistical test`.

CUDA threads is T. However, because each thread runs 18 tests in sequence, room for improvement is apparent in this method.

Parallelization method 2 In this method, each block performs its designated statistical test out of 18 tests on one shuffled dataset shared by 18 blocks. Thus, for one shuffled set, 18 statistical tests are run in parallel, and this method is a parallelization of the serial part in method 1 above. This method is illustrated in Figure 4, which indicates the kernel `Statistical test` with $B' = ((N/T) \times 18)$ CUDA blocks and T threads in a block.

Proposed optimization This method optimizes parallelization method 2 through two steps. (Step 1) To hide the latency in accessing the slow global memory of the GPU, we analyzed the runtime of 18 statistical tests from an algorithmic perspective. We merged several statistical tests with similar access patterns to the global memory into a single test. Therefore, 9 merged statistical tests replace 18 statistical tests. (Step 2) We configured one thread run at least one statistical test so that all blocks can finish their work within a similar time. Considering the execution time of each test, the thread in one block executes only the test that requires the longest time. This test is the merged of the average collision test statistic and the maximum collision test statistic. The thread in another block runs the rest of the tests. This method is depicted in Figure 5, where the kernel `Statistical test` uses $B' = ((N/T) \times 2)$ CUDA blocks, with T threads in each block.

If the noise sample size is 1 bit, one of two conversions is applied to certain statistical tests. With slight modifications to the kernels `Shuffling` and `Statistical test`, which are designed for non-binary samples, as described above, we can parallelize the permutation testing when the input data are binary. In the kernel `Shuffling`, N CUDA threads firstly generate N shuffled data in parallel. Thereafter, each thread proceeds to Conversion II for its own shuffled data and stores the results (No. 6 in Table 3) in the global memory of the GPU. The kernel `Statistical test` runs nine merged tests. As in the optimized method for non-binary data, the thread in the block executes at least one test so that the execution time of each block is similar. Therefore, $B' = (N/T) \times 4$ CUDA blocks are used when the number of CUDA threads is T. The data after Conversion I are the result of calculating the Hamming weight of the data following Conversion II. Both data after Conversion I and data after Conversion II can be stored separately in the global memory. However, we use a method to calculate the Hamming weight of the data after Conversion II in the merged statistical tests applied by Conversion I to minimize the use of the global memory.

EXPERIMENTS AND PERFORMANCE EVALUATION

In this section, we present the performance measurement of the proposed method and compare its performance with the NIST program written in C++. The performance was evaluated using two hardware configurations (Table 4).

There are two noise sources used in experiments. The first noise source is truerand provided by the NIST. The second noise source, GetTickCount, could be collected through the `GetTickCount()` function in the Windows environment. The sample size of each noise source is 1, 4, or 8 bits. As a result of confirming whether the input data is IID by repeating the IID test, truerand is determined as the IID noise source; however, GetTickCount is determined as the non-IID noise source.

The experimental result is the average of the results repeated 20 times. The difference between the results of the experiments repeated 20 times was within 5 %.

GPU optimization concepts

We conducted experiments on the optimization concepts considered while parallelizing the permutation testing. The experimental data used in this section consisted of one million samples collected from the noise source GetTickCount, where the sample size was 8 bits.

Prior to the experiments, we set the values of the parameters used. To process N iterations in parallel on the GPU, we required $N \times 1{,}000{,}000$ bytes of the global memory of the GPU. Therefore, we set N to 2,048 (using about 2 GB of the global memory). Then we set T, the

Name	Device A	Device B
CPU model	Intel(R) Core (TM) i7-8086K	Intel(R) Core (TM) i7-7700
CPU frequency	4.00 GHz	3.60 GHz
CPU cores	6	4
CPU threads	12	8
Accelerator type	NVIDIA GPU	NVIDIA GPU
Models	TITAN Xp	GeForce GTX 1060
Multiprocessors (MPs)	30	10
CUDA cores/MP	128	128
CUDA capability major	6.1	6.1
Global memory	12,288 MB	6,144 MB
GPU Max clock rate	1,582 MHz	1,709 MHz
Memory clock rate	5,750 MHz	4,004 MHz
Memory bus width	384 bits	192 bits
Registers/block	65,536	65,536
Threads/MP	2,048	2,048
Threads/block	1,024	1,024
Warp size	32	32
CUDA driver version	10.1	10.1

Table 4. Configurations of experimental platforms.

number of threads per block used in the CUDA kernel, to 256, which was a multiple of the warp size ($= 32$).

Merging statistical tests

Our optimization method consists of a step in which tests are merged (Step 1) and a step in which at least one test is allocated so that the working time of each thread is similar (Step 2). Therefore, this section confirms the validity of our merging tests.

We first designed new CUDA kernels for experimentation, where each of the N threads performed one statistical test on one shuffled data. We measured the execution time and the number of registers of each test kernel. Since we set the number of threads per block T to 256, each test kernel uses 8 CUDA blocks. The experimental results, the execution time of each statistical test on the GPU are shown in Table 5. Figure 6 presents the number of registers per thread of each statistical test as measured with Nsight. NVIDIA Nsight is a development environment that enables the developer to build and debug and examine the state of GPU and the memory.

From Table 5, it is expected that it takes about 4 seconds if one thread sequentially performs 18 statistical tests. However, if one thread performs 9 merged tests, it can be expected that it will take about 2.3 seconds. We improved the performance for all 18 statistical tests by about 1.7 times by combining the tests.

Since the number of registers used by each thread is one of the key factors for performance improvement (NVIDIA, 2020b), we analyzed whether there is any performance degradation due to the number of registers when using the merged tests, as follows; If the merged tests are not used, one of the optimizing methods is to apply Step 2 of our proposed technique to the parallelization method 2. We analyzed this method and our proposed method in terms of occupancy and the number of active blocks. The more threads (blocks) are allocated to the MP, the better the performance. Occupancy is calculated as ((number of active blocks $\times$ number

No.	Name of statistical test	Execution time (ms)	No.	Name of merged statistical test	Execution time (ms)
1	Excursion test	214	$1'$	Excursion test	214
2	Number of directional runs	75	$2'$	Directional runs and number of inc/dec	90
3	Length of directional runs	81			
4	Numbers of increases and decreases	38			
5	Number of runs based on median	103	$3'$	Runs based on median	143
6	Length of runs based on median	128			
7	Average collision test statistic	1,257	$4'$	Collision test statistic	1,258
8	Maximum collision test statistic	1,238			
9	Periodicity test (lag = 1)	50	$5'$	Per/Cov test (lag = 1)	129
10	Covariance test (lag = 1)	71			
11	Periodicity test (lag = 2)	94	$6'$	Per/Cov test (lag = 2)	137
12	Covariance test (lag = 2)	113			
13	Periodicity test (lag = 8)	93	$7'$	Per/Cov test (lag = 8)	134
14	Covariance test (lag = 8)	111			
15	Periodicity test (lag = 16)	93	$8'$	Per/Cov test (lag = 16)	134
16	Covariance test (lag = 16)	111			
17	Periodicity test (lag = 32)	93	$9'$	Per/Cov test (lag = 32)	134
18	Covariance test (lag = 32)	111			

Table 5. Left: execution time of each statistical test on GPU; right: execution time of each merged statistical test on GPU (Device A, number of CUDA blocks = 8, number of threads per block = 256).

of active warps) / maximum number of warps). As shown in Table 4, both devices have a warp size of 32 and a maximum of 2048 threads per block. The maximum number of warps is calculated as 64 ($= 2048/32$). To analyze occupancy, we calculated the number of active blocks. The number of active blocks is the minimum of the number of blocks calculated considering the amount of shared memory, the number of threads per block, and the number of registers (NVIDIA, 2020a). Since shared memory was not used, the number of active blocks calculated by the amount of shared memory is the device limit of 32. Since the number of threads per block is set to 256, the number of active blocks calculated by the number of threads is 8 ($= 2,048/256$). As shown in Figure 6, the maximum number of registers is 32, which is the same for both methods. Since 8,192 ($= 256 \times 32$) registers are used per block, and the number of registers that can be used in an SM is 65,536, the number of active blocks calculated by the number of registers is 8 ($= 65,536/8,192$). As a result, the number of active blocks is $min(32,8,8) = 8$. Since the number of active warps is 8 ($= 256/8$), occupancy of both methods is calculated by 100 % ($= 8 \times 8/64 \times 100$). If more than 33 registers need to be used, the number of active blocks

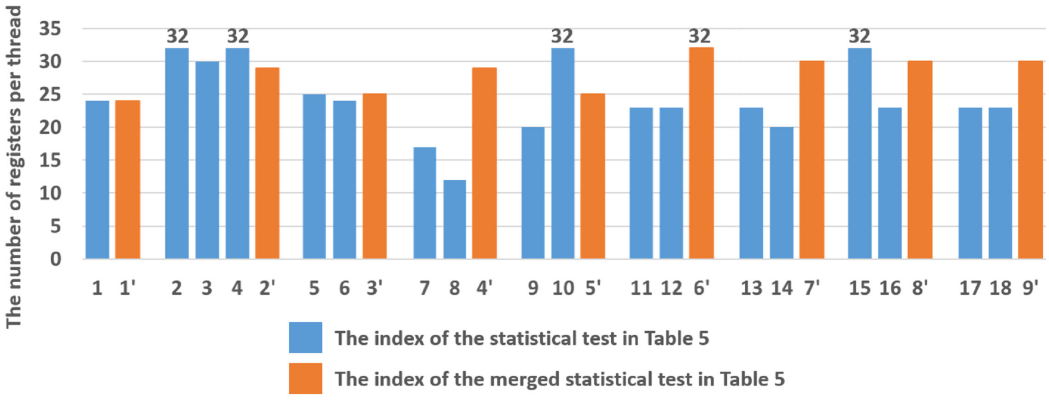

Figure 6. Number of registers used by each CUDA thread running each statistical test and each merged statistical test.

is reduced to 7, resulting in 88 % occupancy. Therefore, there is no performance degradation caused by register limitations. If we use 2 more registers, the performance consideration will be more complex including the spill-over to local-memory.

We measured the execution time of the parallelization method 2 applying Step 2, and our method. Referring to the results of Table 5, we designed each CUDA block to proceed with each of test $1 \sim 6$, test 7, test 8, and test $9 \sim 18$. The kernel `Statistical test` applying this method uses 32 $(= (N/T) \times 4)$ blocks; however, applying our proposed method uses 16 $(= (N/T) \times 2)$ blocks. Table 6 presents the execution time of a kernel `Statistical test` applying each method. As a result, our method is about 1.5 times speedup than the parallelization method 2 applying Step 2.

	Number of CUDA blocks	**Execution time (s)**
Parallelization method 2 **(**18 **tests) + Step** 2	32	2.24
Our method (9 **merged tests + Step** 2**)**	16	1.51

Table 6. Performance of parallelization method 2 applying Step 2 and our method (Device A, the number of threads per block = 256).

Parallelism methods

This section experimentally verifies whether the proposed optimized method 2 is better than other methods. We firstly confirmed the difference in the operation time of each CUDA thread in the kernel `Statistical test`, where each parallelization method is applied by drawing a figure. Figure 7 displays the operation times of the CUDA threads, assuming that the GPU had three SMs and considering the results of Table 5. It is the task of the GPU scheduler to allocate the CUDA blocks to the SMs; however, we assigned arbitrarily for visualization as Figure 7. As indicated in Table 5, the statistical tests had different execution times. Therefore, we expressed the different lengths of the threads in the CUDA blocks running each statistical test, as illustrated in Figure 7 (left). In the proposed method, several statistical tests were merged for optimization. The execution time of the merged statistical test (Table 5 (right)) was equal to or slightly longer than each execution time of the original statistical tests prior to merging (Table 5 (left)). Suppose that we merged Test 1 and Test 2. Then the lengths of the threads in the block running Test 1&2 were slightly longer than those of the threads in the block running Test 1 or Test 2, as indicated in Figure 7 (right). As illustrated in Figure 7, we expected that our optimization outperformed parallelization methods 1 and 2.

We measured the execution time of a kernel `Statistical test` according to the parallel

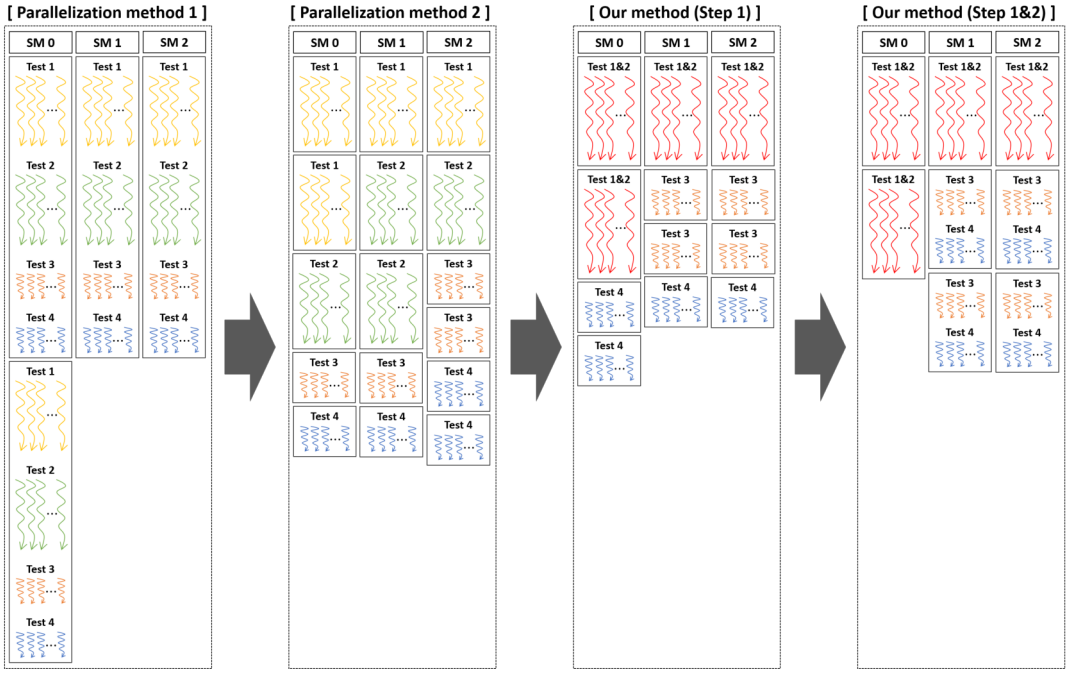

Figure 7. Operation time of CUDA threads in kernel `Statistical test` when applying each method on device.

Method	Number of CUDA blocks	Execution time (s)	
		Device A	Device B
Parallelization method 1	8	4.53	6.39
Parallelization method 2	144	2.77	6.33
Our optimization (Step 1)	72	1.62	2.94
Our optimization (Step 1&2)	16	1.51	2.76

Table 7. Execution time of kernel `Statistical test` according to parallel method (number of threads per block = 256).

method. Table 7 shows the execution times of each kernel measured on both devices. If the occupancy of the kernel in our parallelization method is calculated using the calculation process described in the previous section, it reaches 100 %. It is the occupancy per SM. Since our method uses a small number of blocks, there might be idle SMs on high-performance GPU with many SMs. However, if the host calls the test kernel for each noise source simultaneously using a multi-stream technique, we can make use of almost full GPU capability.

Since 18 statistical tests were running in parallel, the parallelization method 2 has been improved by 1.6 times than method 1 in Device A; however, there was no improvement in the performance in Device B. In Device B, the number of SMs is 10, and the number of active blocks is calculated by 8. Thus, it is analyzed as the result that came out because the number of blocks generated by the kernel (= 144) is more than the number of blocks active in the device simultaneously (= 80). Our method Step 1 is about 1.7 times and 2.1 times speedup than the parallelization method 2 since merged statistical tests have improved the performance than before, as confirmed in the previous section. Since the work of each CUDA block was adequately balanced, it is analyzed that our method Step 1&2 has been slightly improved over our method Step 1. Furthermore, our method has been improved by 3 times and about 2.3 times than the parallelization method 1.

Next, we analyzed how each method affects the performance of permutation testing. As shown in Algorithm 2, the permutation testing has 10,000 iterations. Since implemented N iterations in parallel, the kernel `CurandInit` is called once, and the kernel `Shuffling` and `Statistical test` are called $(10{,}000/N)$ times. Since we set N to 2,048 and does not use Equation 2 in this experiment, the permutation testing consists of one `CurandInit`, five `Shuffling` and five `Statistical test`. Figure 8 shows the execution time of this permutation testing according to the parallelization method. As shown in Figure 8, the permutation testing applied our method has been improved maximum of about 1.8 times than applied method 1. Thus, our optimization method outperformed parallelization methods 1 and 2.

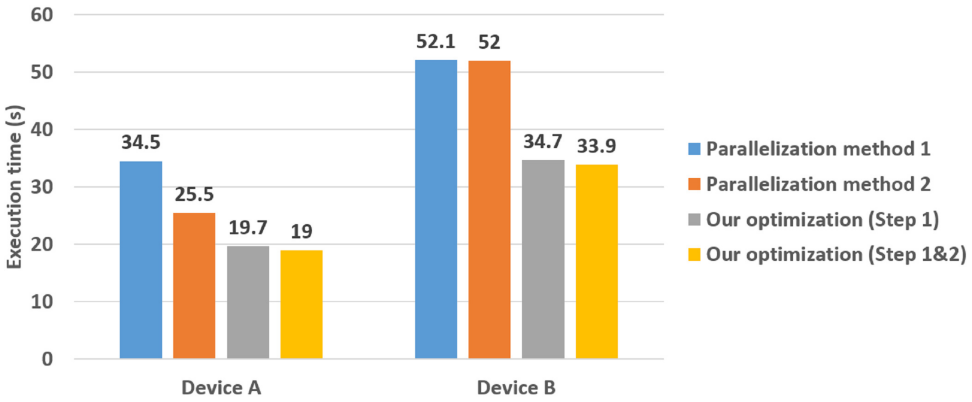

Figure 8. Execution time of permutation testing according to parallel method (number of threads per block = 256).

Coalesced memory access

In this study, we used the memory coalescing technique (Figure 9) to transfer data from slow global memory to the registers efficiently. Table 8 displays the performance of our parallel implementation of the permutation testing before and after using this technique. From this section, the permutation testing uses the kernel `Statistical test` with our optimization method. As a result, we obtained an improvement of 1.5 times.

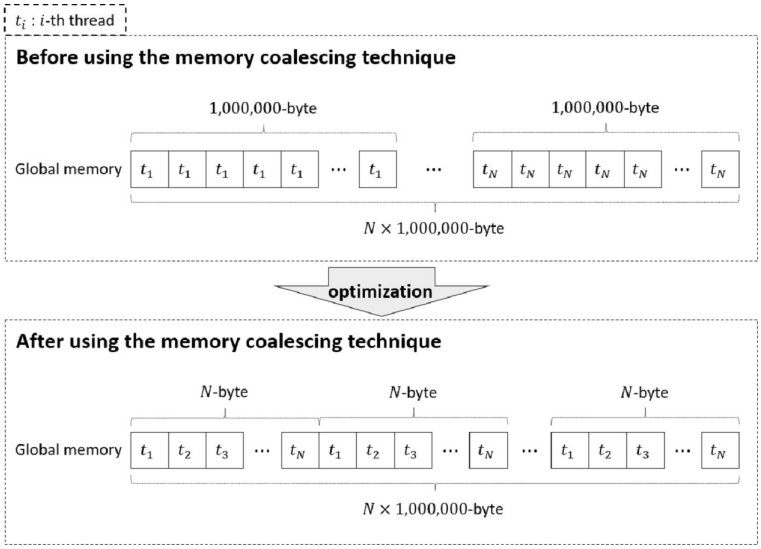

Figure 9. Memory coalescing technique.

	Before using memory coalescing technique (s)	After using memory coalescing technique (s)
Device A	27.2	19.0
Device B	54.1	33.9

Table 8. Performance of proposed parallel implementation of permutation testing depending on whether memory coalescing technique was used.

Performance evaluation according to the parameter

We measured the performance of the permutation testing according to the value of the parameter N. The parameter N is the number of iterations of the permutation testing to be processed in parallel. Through this result, we set the proper value of N.

The execution time of each kernel `CurandInit`, `Shuffling` and `Statistical test` according to the value of N is shown in Figure 10. We use data consisting of one million truerand samples, and the sample size is 8 bits. As shown in Figure 10, as the value of N increases, the execution time of each kernel increases linearly.

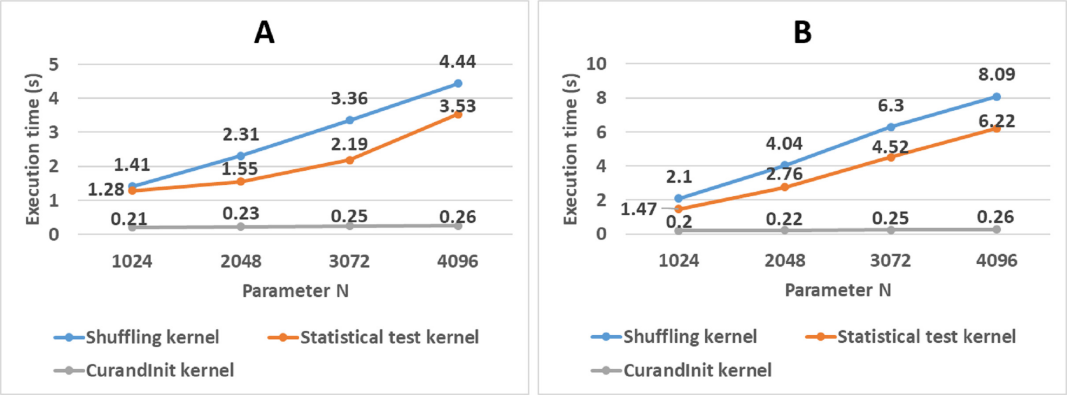

Figure 10. Execution time of each kernel according to parameter N.
(A) Execution time on Device A. (B) Execution time on Device B.

		Execution time (s)			
Parameter N		1,024	2,048	3,072	4,096
Device A	**truerand-8bit**	2.68	3.82	5.46	7.42
	GetTickCount-8bit	27.03	19.03	21.68	21.93
Device B	**truerand-8bit**	3.59	6.80	10.85	14.36
	GetTickCount-8bit	35.82	33.92	43.43	43.23

Table 9. Execution time of permutation testing according to the value of the parameter N.

As shown in Algorithm 2, the kernel `CurandInit` is called once, and the kernel `Shuffling` and `Statistical test` are called $(10,000/N)$ times. The process repeated $(10,000/N)$ times is as follows. After the kernel `Shuffling` and the kernel `Statistical test` are sequentially run once, if the results do not satisfy Equation 2, two kernels are called again. This call is repeated until all 10,000 iterations have been completed. If Equation 2 is satisfied, the calls to the two kernels are aborted. The execution time of this permutation testing according to the value of N is shown in Table 9. The noise sources, truerand and GetTickCount, are used in the experiment. The sample size of each noise source is 8 bits.

As a result of permutation testing, truerand is a noise source that is IID. If the noise source is IID, there is little evidence against the null hypothesis that the noise source is IID. Therefore, the probability of satisfying Equation 2 increases, and the number of kernel calls decreases. In the experiment on truerand data, each kernel was called only once. Thus, it was confirmed that execution time according to N written in Table 9 has a rate of increase similar to the rate of each kernel shown in Figure 10. GetTickCount is a non-IID noise source from a result of permutation testing. If the noise source is Non-IID, there is a lot of evidence against the null hypothesis that the noise source is IID. Therefore, contrary to the IID noise source case, the probability of satisfying Equation 2 decreases, and the number of kernel calls increases. In this experiment, the kernel `Shuffling` and `Statistical test` are each called $(10{,}000/N)$ times. As shown in Table 9, it took the least execution time of the permutation testing when N was 2048. It is analyzed that the increase rate of execution time and the number of kernel calls $(= 10{,}000/N)$ according to N affected the results of GetTickCount in Table 9. Therefore, it is appropriate to set N to 2048 in consideration of both the usage of the global memory and performance.

Performance evaluation with NIST program according to noise source

We measured the performances of the proposed parallel implementation of the permutation testing using the GPU. Two noise sources, truerand and GetTickCount, were used in the experiment and the sample size of each noise source is one of 1, 4, and 8 bits.

The NIST program, written in C++, is compatible with OpenMP and can make $10{,}000$ iterations work in a multi-threaded environment. In this experiment, the NIST program running on the CPU uses 12 CPU threads in Device A and 8 CPU threads in Device B (Table 4). Thus we compared our performances with those of the permutation testing in a single-threaded and multi-threaded NIST program. Since our GPU-based parallel implementation of the permutation testing is designed without the compression algorithm, we measured the performance of the NIST program except for the compression test.

Table 10 presents the execution times of the NIST program on the CPU and the proposed program on the GPUs, measured for each noise source. As indicated in Table 10, for truerand, the performance of the proposed program was up to about 21.5 times better than that of the single-threaded NIST program and up to about 15.3 times better than of the multi-threaded NIST program. In the case of GetTickCount, the performance of our program has been improved by about 33.6 times and about 24.4 times than the single-threaded and the multi-threaded NIST program.

		Execution time (s)					
Name of noise source		**truerand**			**GetTickCount**		
Sample size (bit)		1	4	8	1	4	8
Device A	**NIST program (CPU single-thread)**	43.42	77.52	24.94	434.42	485.58	638.89
	NIST program (CPU multi-thread)	37.53	54.91	23.66	331.76	339.79	347.68
	Proposed program (GPU)	2.70	3.60	3.82	13.59	18.14	19.03
Device B	**NIST program (CPU multi-thread)**	41.35	50.15	23.18	361.23	347.15	353.52
	Proposed program (GPU)	4.64	5.95	6.80	23.19	30.08	33.92

Table 10. Performances of proposed program and NIST program written in C++ according to noise source.

As shown in Algorithm 2, the kernel `CurandInit` is called once, and the kernel `Shuffling`

and `Statistical test` are called 5 ($= 10{,}000/N$) times. If the noise source is likely to be determined as IID from the permutation testing, there is a high probability of satisfying Equation 2. From this, the number of kernel calls on the GPU will be fewer. In the NIST program on the CPU, as shown in Algorithm 2, if any statistical test satisfied Equation 2, that test was no longer performed in the iterations. Thus if a higher probability of meeting Equation 2, the number of iterations of each statistical test is fewer. However, if the noise source is less likely to be determined as IID from the permutation testing, there is a low probability of satisfying Equation 2. It increases the number of kernel calls on the GPU (maximum number = 5) and the number of iterations of each statistical test on the CPU (maximum number = 10,000).

In Table 10, the minimum performance improvement of the proposed program for truerand is not higher than that of the program for GetTickCount. As a result of the permutation testing on each noise source, truerand is determined as the IID noise source, and GetTickCount is determined as the non-IID noise source. Since the number of iterations of the permutation testing algorithm for IID noise source is fewer than that for non-IID noise source, the minimum performance improvement of the proposed program for truerand is not higher than that of the program for GetTickCount.

CONCLUSIONS

The security of modern cryptography is heavily reliant on sensitive security parameters such as encryption keys. RNGs should provide cryptosystems with ideal random bits, which are independent, unbiased, and most importantly, unpredictable. To use a secure RNG, it is necessary to estimate its input entropy as precisely as possible. The NIST offers two programs for entropy estimations, as outlined in SP 800-90B. However, a long time is required to manipulate several noise sources for an RNG.

This paper has proposed GPU-based parallel implementation of the permutation testing, which requires the longest execution time in the IID test of SP 800-90B. The proposed method is designed to use massive parallelism of the GPU by balancing the number of registers and the execution time for statistical tests, as well as optimizing the use of the global memory for data shuffling. We experimentally compared our GPU optimization with the NIST. The proposed program was 33 times faster than the single-threaded NIST program. Moreover, our proposal improved the performance up to 24 times than the multi-threaded NIST program. It is expected that the time required for analyzing the RNG security will be significantly reduced for developers and evaluators by using the proposed approach, thereby improving the validation efficiency in the development of cryptographic modules. For future work, we will implement the compression test excluded in this study in parallel on the GPU.

REFERENCES

Barker, E. and Kelsey, J. (2012). Recommendation for the entropy sources used for random bit generation. *National Institute of Standards and Technology.* NIST Special Publication (SP) 800-90B (Draft).

Bernstein, D. J., Chang, Y.-A., Cheng, C.-M., Chou, L.-P., Heninger, N., Lange, T., and Van Someren, N. (2013). Factoring RSA keys from certified smart cards: Coppersmith in the wild. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 341–360. Springer.

Ding, Y., Peng, Z., Zhou, Y., and Zhang, C. (2014). Android low entropy demystified. In *2014 IEEE International Conference on Communications (ICC)*, pages 659–664. IEEE.

Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. (2012). Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220.

ISO/IEC-20543 (2019). Information technology — Security techniques — Test and analysis methods for random bit generators within ISO/IEC 19790 and ISO/IEC 15408.

Kang, J.-S., Park, H., and Yeom, Y. (2017). On the Additional Chi-square Tests for the IID Assumption of NIST SP 800-90B. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, pages 375–3757. IEEE.

Kaplan, D., Kedmi, S., Hay, R., and Dayan, A. (2014). Attacking the Linux PRNG On Android: Weaknesses in Seeding of Entropic Pools and Low Boot-Time Entropy. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*.

Kim, S. H., Han, D., and Lee, D. H. (2013). Predictability of Android OpenSSL's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 659–668.

Li, P., Zhou, S., Ren, B., Tang, S., Li, T., Xu, C., and Chen, J. (2019). Efficient implementation of lightweight block ciphers on volta and pascal architecture. *Journal of Information Security and Applications.*, 47:235–245.

Li, Q., Zhong, C., Zhao, K., Mei, X., and Chu, X. (2012). Implementation and analysis of aes encryption on gpu. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 843–848. IEEE.

Ma, J., Chen, X., Xu, R., and Shi, J. (2017). Implementation and evaluation of different parallel designs of aes using cuda. In *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, pages 606–614. IEEE.

Michaelis, K., Meyer, C., and Schwenk, J. (2013). Randomly failed! the state of randomness in current java implementations. In *Cryptographers' Track at the RSA Conference*, pages 129–144. Springer.

Neves, S. and Araujo, F. (2011). On the performance of gpu public-key cryptography. In *ASAP 2011-22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 133–140. IEEE.

NIST (2015). EntropyAssessment. Available at https://github.com/usnistgov/SP800-90B_EntropyAssessment (accessed February 2020).

NVIDIA (2020a). CUDA C++ BEST PRACTICES GUIDE. *NVIDIA, Aug.*

NVIDIA (2020b). CUDA C++ PROGRAMMING GUIDE. *NVIDIA, Aug.*

Pan, W., Zheng, F., Zhao, Y., Zhu, W.-T., and Jing, J. (2016). An efficient elliptic curve cryptography signature server with gpu acceleration. *IEEE Transactions on Information Forensics and Security*, 12(1):111–122.

Patel, R. A., Zhang, Y., Mak, J., Davidson, A., and Owens, J. D. (2012). *Parallel lossless data compression on the GPU*. IEEE.

Ristenpart, T. and Yilek, S. (2010). When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*.

Schneier, B., Fredrikson, M., Kohno, T., and Ristenpart, T. (2015). Surreptitiously weakening cryptographic systems. *IACR Cryptol. ePrint Arch.*, 2015:97.

Seward, J. (2019). bzip2 and libbzip2, version 1.0.8: A program and library for data compression. *Avaliable at https://sourceware.org/bzip2/.*

Shastry, K., Pandey, A., Agrawal, A., and Sarveswara, R. (2016). Compression acceleration using GPGPU. In *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*, pages 70–78. IEEE.

Stevens, M., Bursztein, E., Karpman, P., Albertini, A., and Markov, Y. (2017). The first collision for full SHA-1. In *Annual International Cryptology Conference.*, pages 570–596. Springer.

Sönmez Turan, M., Barker, E., Kelsey, J., McKay, K., Baish, M., and Boyle, M. (2016). Recommendation for the entropy sources used for random bit generation. *National Institute of Standards and Technology.* NIST Special Publication (SP) 800-90B (2nd Draft).

Sönmez Turan, M., Barker, E., Kelsey, J., McKay, K., Baish, M., and Boyle, M. (2018). Recommendation for the entropy sources used for random bit generation. *National Institute of Standards and Technology.* NIST Special Publication (SP) 800-90B.

Vaidya, B. (2018). *Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA: Effective techniques for processing complex image data in real time using GPUs.* Packt Publishing Ltd.

Yoo, T., Kang, J.-S., and Yeom, Y. (2017). Recoverable random numbers in an internet of things operating system. *Entropy*, 19(3):113.

Zhu, S., Ma, Y., Chen, T., Lin, J., and Jing, J. (2017). Analysis and improvement of entropy estimators in NIST SP 800-90B for non-IID entropy sources. *IACR Transactions on Symmetric*

Cryptology, pages 151–168.

Zhu, S., Ma, Y., Li, X., Yang, J., Lin, J., and Jing, J. (2019). On the analysis and improvement of min-entropy estimation on time-varying data. *IEEE Transactions on Information Forensics and Security*.