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

It has been established that 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 in cryptosystems and cryptographic modules. 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 challenging 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 improvement from the user GPU performance, we propose a scalable method that adjusts the optimal size of the global memory occupancy in the proposed GPU kernel function according to the GPU specifications. Moreover, our method improves the performance by merging two statistical tests without increasing the number of registers used by the kernel. The experimental results demonstrate that our method is at least 23 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

It has been established that 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 in cryptosystems and cryptographic modules. 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 challenging 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 improvement from the user GPU performance, we propose a scalable method that adjusts the optimal size of the global memory occupancy in the proposed GPU kernel function according to the GPU specifications. Moreover, our method improves the performance by merging two statistical tests without increasing the number of registers used by the kernel. The experimental results demonstrate that our method is at least 23 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 which show the importance of the entropy in operating systems. Yilek et al. (2009) describes that a pseudo-random number generator (PRNG) of Debian OpenSSL gathers entropy insufficiently and thereby the private keys generated by the PRNG are predictable. 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 and 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, Yoo et al. (2017); Nguyen and Shparlinski (2002); Bernstein et al. (2013); Michaelis et al. (2013) describe the attacks caused by wrong estimations of the entropy, exaggeratedly or too conservatively.

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.

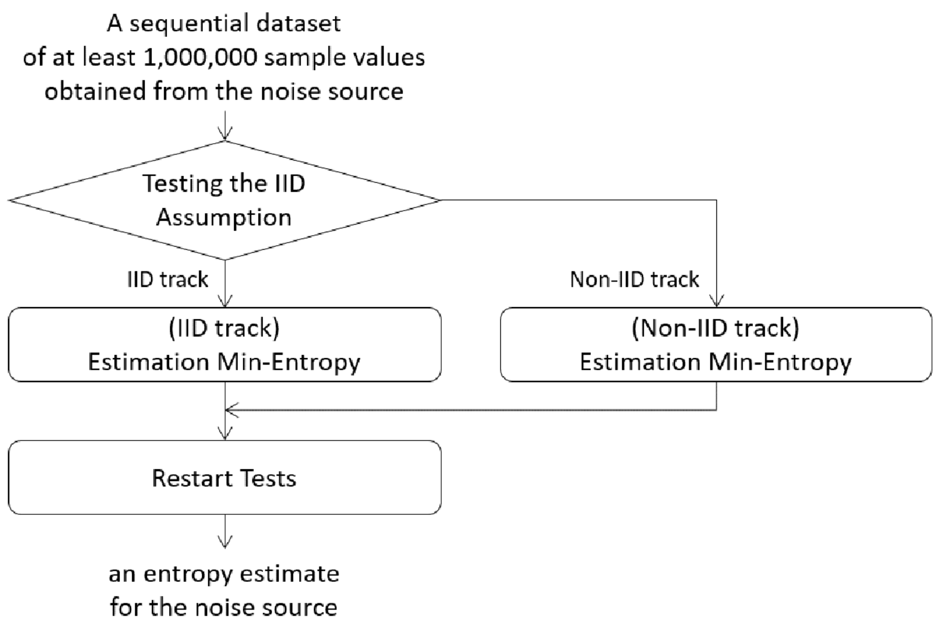

Figure 1. Flow of entropy estimation process of SP 800-90B.

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 the two NIST programs for estimating the entropy of the noise source. GetTickCount, which can be collected through the `GetTickCount()` function in the Windows environment, has a sample size of 8 bits. 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. 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++. Moreover, assume that the module supports two operating systems and 10 noise sources are used as input into the RNG in each operating system. According to Table

	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 NIST program for entropy estimation process (noise source: GetTickCount; noise sample size: 8 bits).

1, the 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, because the entropy of each noise source should be estimated several times, over 200 h may be necessary, or three days when the number of iterations is set to 10. 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) were initially designed for accelerating computer graphics and image processing. In recent years, GPUs have been used for general computations in addition to graphics processing. The use of GPUs for performing computations handled by central processing units (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 PRESENT (Manavski, 2007; Szerwinski and Güneysu, 2008; 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. Furthermore, we merge two statistical tests without increasing the number of registers used by the kernel so that the proposed method can provide a performance improvement. Our experiments support the finding that our parallel implementation can achieve optimized results with over 20 times higher performance than that of the NIST.

The remainder of this paper is organized as follows. Section 2 introduces 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.

IID TEST

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,

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 T_i^{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 T_i^{Shuffle} on the shuffled data.
8: **if** $(T_i^{\text{Shuffle}} > T_i^{\text{IN}})$ **then**
9: Increment $C_{i,0}$.
10: **else if** $(T_i^{\text{Shuffle}} = T_i^{\text{IN}})$ **then**
11: Increment $C_{i,1}$.
12: **end if**
13: **end for**
14: **if** $((C_{i,0} + C_{i,1} \leq 5) \,\text{or}\, (C_{i,0} \geq 9,995))$ for any i **then**
15: Reject the IID assumption.
16: **else**
17: Assume that the noise source outputs are IID.
18: **end if**

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 and 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 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 $T_i^{\text{Shuffle}} >$ of the original test statistic T_i^{IN}. The counter $C_{i,1}$ is the number of j satisfying $T_i^{\text{Shuffle}} = T_i^{\text{IN}}$, whereas the counter $C_{i,2}$ is the number of j satisfying $T_i^{\text{Shuffle}} < T_i^{\text{IN}}$.

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

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.

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

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 10,0000 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.

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 T_i^{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 T_i^{Shuffle} on the shuffled data.
      if $(T_i^{\text{Shuffle}} > T_i^{\text{IN}})$ **then**
        Increment $C_{i,0}$.
      else if $(T_i^{\text{Shuffle}} = T_i^{\text{IN}})$ **then**
        Increment $C_{i,1}$.
      else
        Increment $C_{i,2}$.
      end if
      if $((C_{i,0} + C_{i,1} > 5) \,\text{and}\, (C_{i,1} + C_{i,2} > 5))$ **then**
        $state_i = false$.
      end if
    end if
  end for
end for
if $((C_{i,0} + C_{i,1} \leq 5) \,\text{or}\, (C_{i,0} \geq 9,995))$ 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

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
- 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 ongoing. In Patel et al. (2012), all three main steps, namely the BWT, MTF transform, and Huffman coding, were implemented in parallel using a GPU, but 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, it is extremely rare 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. Considering the total amount of the global memory of the GPU, which depends on the hardware specifications, we do not allocate more than 2 GB at once. 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.

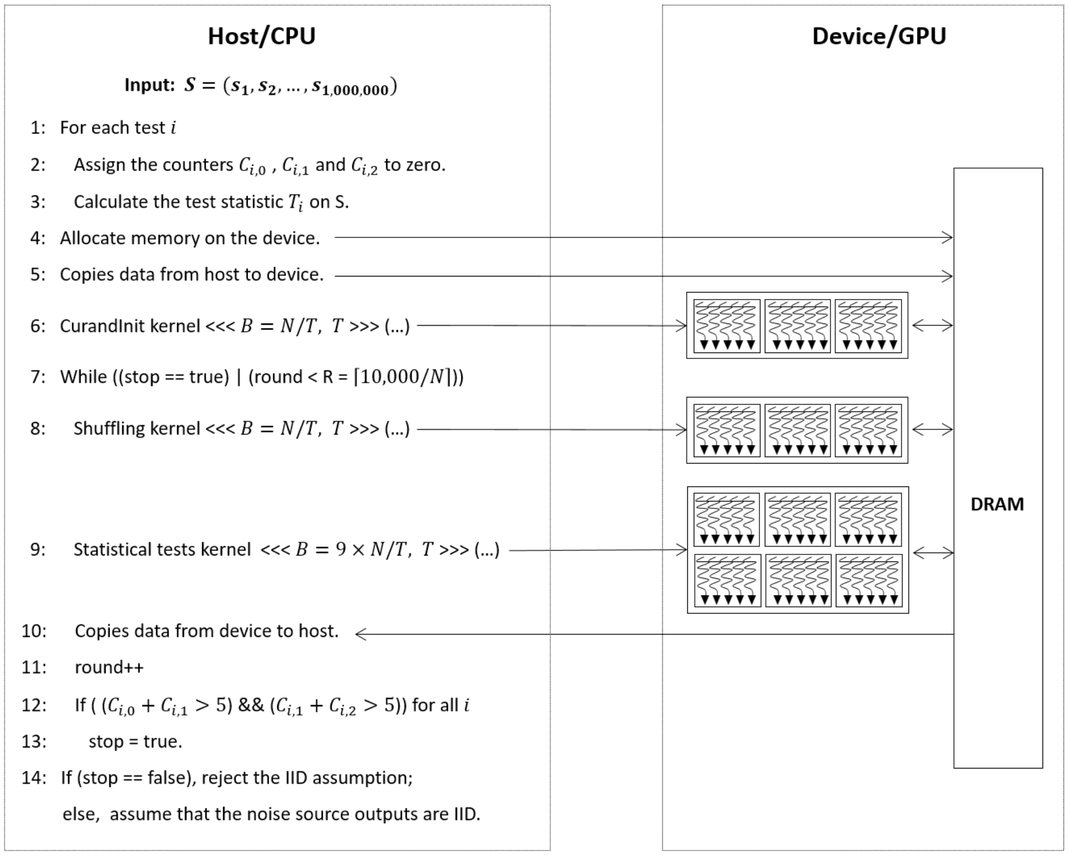

Figure 2. CPU/GPU workflow of permutation testing.

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 2. Use and sizes of variables allocated to 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*). In step 4, the host calls a function that allocates the device memory required to process N iterations in parallel on the device. The usage and sizes of the variables are listed in Table 2. In step 5, the input data (No. 1 in Table 2), and the results of the statistical tests in steps 1 to 3 (No. 4 in Table 2) 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 used in 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 processed in parallel on the device, and this process is repeated R times. 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.

The descriptions of the CUDA kernels `Shuffling` and `Statistical test` designed for processing N iterations in parallel on the GPU 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.

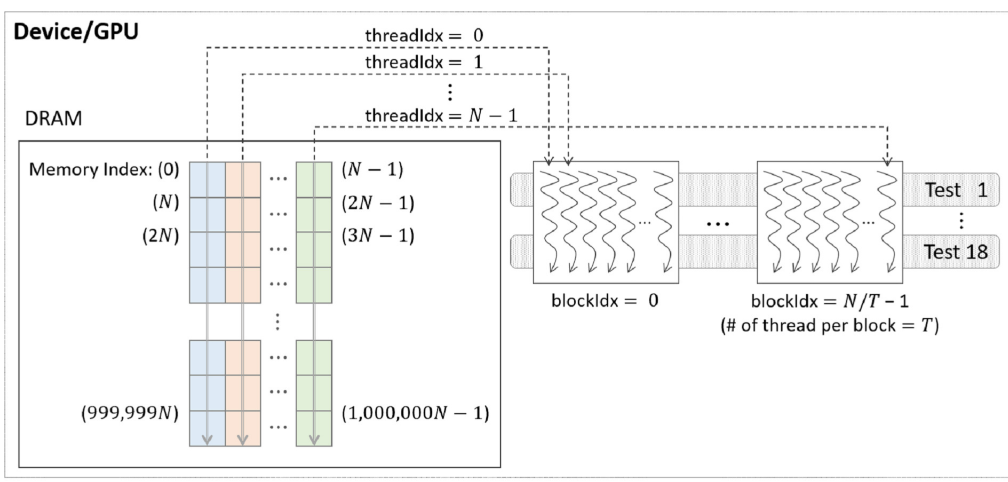

Figure 3. General parallel method 1 of kernel `Statistical test`.

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

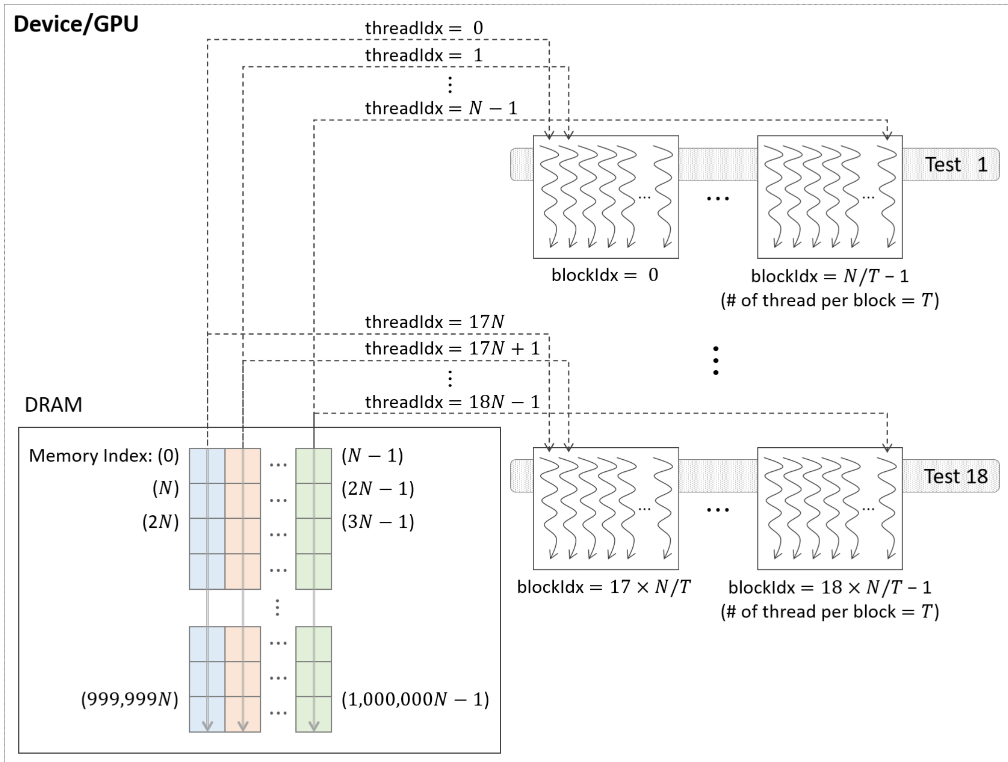

Figure 4. General parallel method 2 of kernel `Statistical test`.

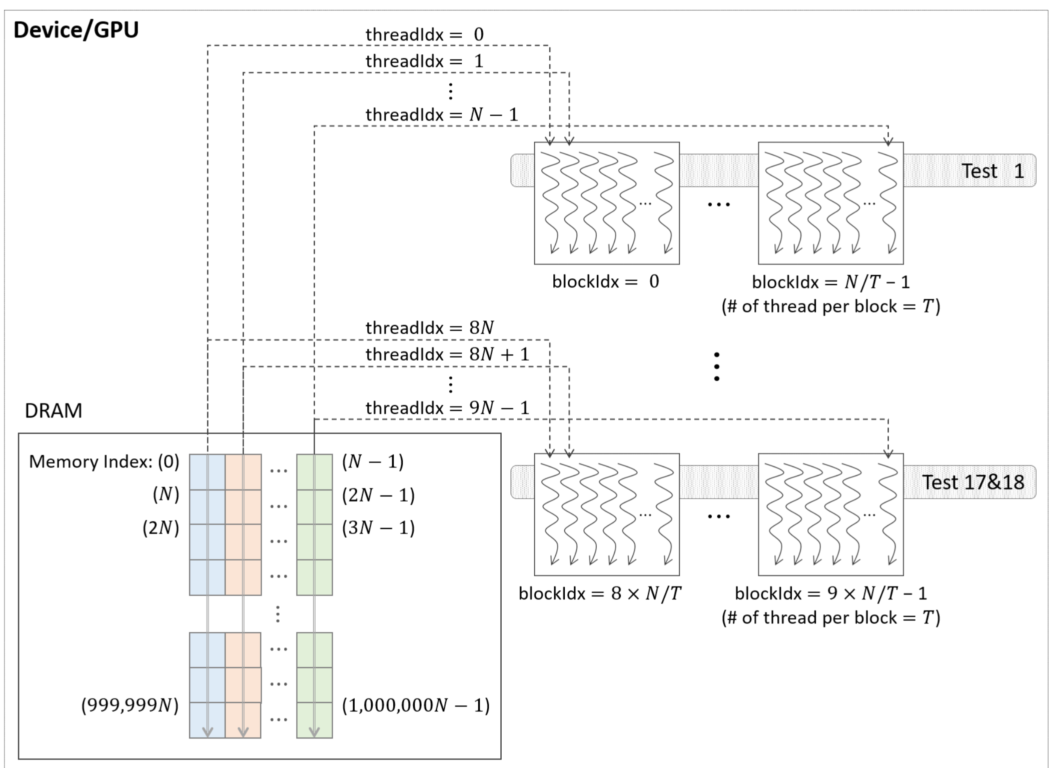

Figure 5. Optimized parallel method of kernel `Statistical test`.

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.

Optimized parallelization This method optimizes parallelization method 2. To hide the latency in accessing the slow global memory of the GPU, we analyze the runtime of 18 statistical tests from an algorithmic perspective and merge several statistical tests with similar access to the global memory into a single test. Therefore, 9 merged statistical tests replace 18 statistical tests. This method is depicted in Figure 5, where the kernel `Statistical test` uses $B' = ((N/T) \times 9)$ 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 8-bit 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. As no conversions are applied to the excursion test and runs based on the median test, each thread performs these two tests sequentially on the shuffled data designated for processing. In the runs based on the median test, two statistical tests, namely the number of runs based on the median and the length of the runs based on the median, are merged. Thereafter, each thread proceeds to Conversion II for its own shuffled data and stores the results (No. 6 in Table 2) in the global memory of the GPU. The kernel `Statistical test` runs seven merged tests in parallel, with the exception of two tests that are already performed in the kernel `Shuffling`. Therefore, $B' = (N/T) \times 7$ 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. Instead of storing the data after Conversion II as well as the data after Conversion I separately in the global memory, to minimize the use of the global memory, we use a method to calculate the Hamming weight of the data after Conversion II in the merged statistical tests applied by Conversion I.

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 3).

Prior to the experiment, 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. Both devices used in the experiment had a global memory of more than 2 GB; however, to minimize the size of the global memory used in our proposed method by considering a common device with a specification lower than that used in the experiment, we set N to $2,048$ (≈ 2 GB$/1,000,000$ bytes). Then we set T, the number of threads per block used in the CUDA kernel, to 256, which was a multiple of the warp size ($= 32$). As N and T were determined, B (the number of blocks in the kernel `Shuffling`), was set to $8 (= N/T)$. In the same manner, B' (the number of blocks in the kernel `Statistical test`), was set to $72 (= N/T \times 9)$.

GPU optimization concepts

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

Parallelism and merging statistical tests

To verify that the proposed optimized parallel method was optimal compared to parallelization methods 1 and 2, we conducted an experiment and measured the execution times, as indicated in Table 4, which presents the performance of the kernel `Statistical test` for each method. It

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
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
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 3. Configurations of experimental platforms.

Method	Execution time (s)	
	Device A	Device B
Parallelization method 1	19.83	27.81
Parallelization method 2	13.55	30.89
Our optimization	9.38	13.53

Table 4. Execution time of kernel `Statistical test` according to parallel method (number of CUDA blocks = 8; number of threads per block = 256).

can be observed from the table that our optimization technique was effective on both Device A and Device B.

When the operation time of each CUDA thread in the kernel where each parallel method was applied is represented graphically, it can be confirmed that the difference in the execution times between each method presented in Table 4 is reasonable. Figure 6 displays the operation times of the CUDA threads with each parallelization method on the GPU, assuming that the GPU had four MPs. The task of the GPU scheduler was to allocate the CUDA blocks to the MPs, however, we allocated arbitrarily for visualization as Figure 6. When each statistical test was run in parallel on the GPU (Device A) for N shuffled data, the 18 statistical tests had different execution times, as indicated in Table 5(left). Therefore, we expressed the different lengths of the threads in the CUDA blocks running each statistical test, as illustrated in Figure 6 (left and center). In the proposed method, several statistical tests were merged for optimization so that 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)). Thus, 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 6(right). As illustrated in Figure 6, we confirmed that our optimization outperformed parallelization methods 1 and 2.

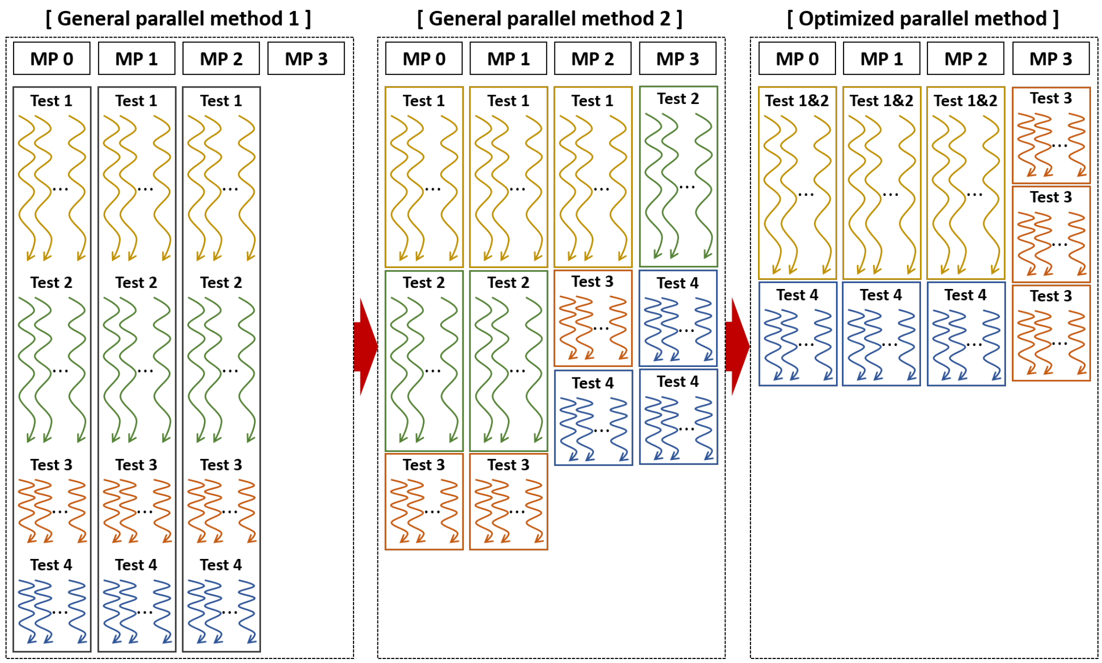

Figure 6. Operation time of CUDA threads in kernel `Statistical test` when applying each method on device.

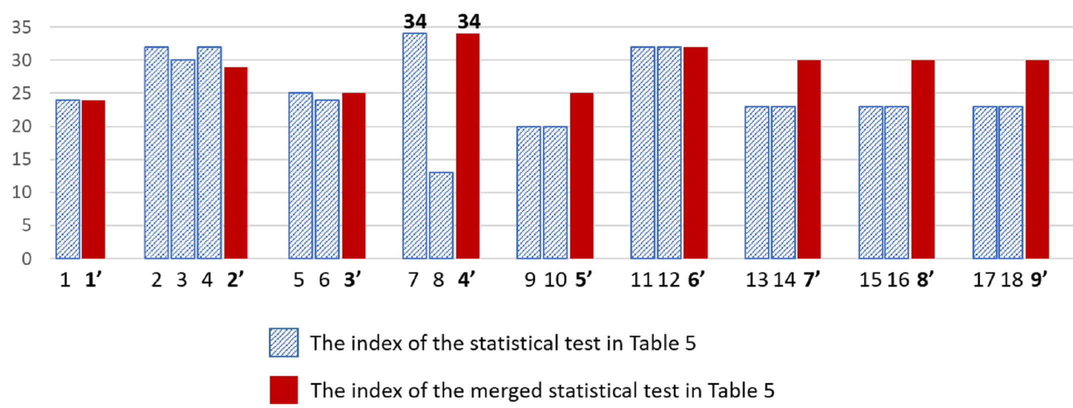

Figure 7. Number of registers used by each CUDA thread running each statistical test and each merged statistical test.

As more threads and thread blocks are likely to reside on an MP when a kernel uses fewer registers, which may improve the performance, the number of registers used by each thread is one of the key factors for performance improvement(NVIDIA, 2019). To provide an analysis in terms of the number of registers with which the optimized method performance was superior to the others, we firstly measured the number of registers used by each thread running each statistical test and each merged test in the kernel `Statistical test`, respectively. Figure 7 presents the measured numbers of registers per thread. In Figure 7, the numbers 1 to 18 on the x-axis represent the tests indicated on the left side of Table 5, whereas the numbers $1'$ to $9'$ represent the tests on the right. According to Figure 7, the maximum number of registers in the merged statistical tests was equal to the maximum number of registers in the statistical tests. Therefore, we can confirm that the statistical tests we merged did not degrade the performance by using the same maximum number of registers as the tests before being merged. The maximum number of registers in the kernel to which each method was applied was 34 in all cases and

No.	Name of statistical test	Execution time (s)	No.	Name of merged statistical test	Execution time (s)
1	Excursion test	0.20	$1'$	Excursion test	0.20
2	Number of directional runs	0.04	$2'$	Directional runs and number of inc/dec	0.04
3	Length of directional runs	0.04			
4	Numbers of increases and decreases	0.04			
5	Number of runs based on median	0.10	$3'$	Runs based on median	0.11
6	Length of runs based on median	0.10			
7	Average collision test statistic	9.09	$4'$	Collision test statistic	9.32
8	Maximum collision test statistic	9.09			
9	Periodicity test (lag = 1)	0.06	$5'$	Per/Cov test (lag = 1)	0.11
10	Covariance test (lag = 1)	0.08			
11	Periodicity test (lag = 2)	0.05	$6'$	Per/Cov test (lag = 2)	0.11
12	Covariance test (lag = 2)	0.07			
13	Periodicity test (lag = 8)	0.06	$7'$	Per/Cov test (lag = 8)	0.11
14	Covariance test (lag = 8)	0.08			
15	Periodicity test (lag = 16)	0.06	$8'$	Per/Cov test (lag = 16)	0.11
16	Covariance test (lag = 16)	0.08			
17	Periodicity test (lag = 32)	0.06	$9'$	Per/Cov test (lag = 32)	0.11
18	Covariance test (lag = 32)	0.08			

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).

each block had 256 threads. Therefore, up to 7 blocks could reside on the MPs as they required $7 \times 256 \times 34$ registers, which was almost $65,536$: the maximum number of registers available on an MP. The CUDA kernel `Statistical test` used 8, 144, and 72 CUDA blocks for parallelization methods 1 and 2, and our method, respectively. In Device A, which had 30 MPs (Table 3), the numbers of active blocks per MP were 1, $3 \sim 4$, and $2 \sim 3$ for the three methods, respectively. These numbers of active blocks per MP were less than the maximum number of blocks per MP, which was 7. By analyzing the number of registers per MP and the operation time of each block for each method, as indicated in Figure 6, we could confirm that the optimized method on Device A was superior. In Device B, which had 10 MPs (Table 3), the numbers of active blocks were 1 and 7 for parallelization method 1 and our method, respectively. When method 2 was applied in the kernel `Statistical test`, the number of active blocks was greater than 7. However, the maximum number of blocks per MP was 7, which explains why method 2 was slower than method 1, as indicated in Table 4. Thus, we also found that the optimized method performed better on Device B with fewer MPs.

Coalesced memory access

In this study, we used the memory coalescing technique (Figure 8) to transfer data from slow global memory to the registers efficiently. Table 6 displays the performance of our parallel implementation of the permutation testing before and after using this technique. As a result, we obtained an improvement of 1.1 times.

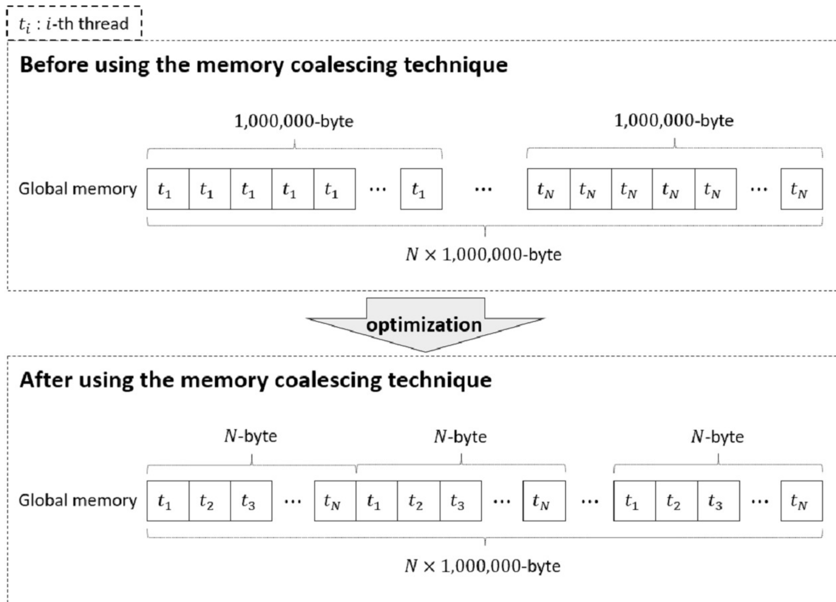

Figure 8. Memory coalescing technique.

	Before using memory coalescing technique (s)	After using memory coalescing technique (s)
Device A	67	60
Device B	190	176

Table 6. Performance of proposed parallel implementation of permutation testing depending on whether memory coalescing technique was used.

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 for both the IID and non-IID noise sources. Moreover, we compared these performances with those of the permutation testing in the NIST program written in C++.

Two noise sources were used in the experiment. The first noise source, truerand, was provided by the NIST. This noise source was IID and the estimated min-entropy was 7.2 bits when the noise sample size was 8 bits. The second noise source, GetTickCount, could be collected through the `GetTickCount()` function in the Windows environment, and its estimated min-entropy was 1.6 bits when the noise sample size was 8 bits.

Table 7 presents the execution times of the NIST program on the CPU and the proposed program on the GPUs, measured for each noise source. Each execution time in Table 7 was the average time required for 50 executions. In the case of truerand (the IID noise source), it was unlikely that each of the 18 statistical tests would run all 10,000 iterations in the permutation testing of Algorithm 2, where Equation 2 was used. In the NIST program, if any statistical test satisfied Equation 2, that test was no longer performed in the iterations. However, because the proposed program processed N iterations of the 18 statistical tests in parallel on the GPU, it verified whether Equation 2 was satisfied using the results of N iterations, and if this was the case, it did not proceed with N iterations any further. Therefore, when the noise source was IID, the performance of the proposed program was up to 10 times better than that of the NIST program, as indicated in Table 7. However, if the noise source was non-IID, it was more likely that the 18 statistical tests would run all $10,000$ iterations. Thus, in the case of non-IID, from Table 7, the proposed program was up to 23 times faster than the NIST program.

Name of noise source	**Sample size**	**NIST program written in C++ (s)**	**Proposed program (s)**	
			Device A	**Device B**
truerand	1	37	4	6
	4	60	6	13
	8	23	12	19
GetTickCount	1	428	19	30
	4	467	25	39
	8	605	60	91

Table 7. Performances of proposed program and NIST program written in C++ according to noise source.

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. When applied to an IID noise source, the proposed program was 10 times faster than the NIST program written in C++. Moreover, for a non-IID noise source, our proposal improved the performance up to 23 times. 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.

Manavski, S. A. (2007). CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *2007 IEEE International Conference on Signal Processing and Communications.*, pages 65–68. 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.

Nguyen, P. Q. and Shparlinski, I. E. (2002). The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *Journal of Cryptology*, 15(3).

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

NVIDIA (2019). CUDA C++ PROGRAMMING GUIDE. *NVIDIA, Nov.*

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

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.

Szerwinski, R. and Güneysu, T. (2008). Exploiting the power of GPUs for asymmetric cryptography. In *International Workshop on Cryptographic Hardware and Embedded Systems.*, pages 79–99. 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.

Yilek, S., Rescorla, E., Shacham, H., Enright, B., and Savage, S. (2009). When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 15–27.

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.*