Article

Parallel Implementations of ARX-Based Block Ciphers on Graphic Processing Units

SangWoo An [1], YoungBeom Kim [2], Hyeokdong Kwon [3], Hwajeong Seo [3] and Seog Chung Seo [2,*]

1 Department of Financial Information Security, Kookmin University, Seoul 02707, Korea; pinksnail06@kookmin.ac.kr
2 Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul 02707, Korea; darania@kookmin.ac.kr
3 Division of IT Convergence Engineering, Hansung University, Seoul 02876, Korea; hyeok@hansung.ac.kr (H.K.); hwajeong@hansung.ac.kr (H.S.)
* Correspondence: scseo@kookmin.ac.kr; Tel.: +82-02-910-4742

Received: 23 August 2020; Accepted: 13 October 2020; Published: 31 October 2020

Abstract: With the development of information and communication technology, various types of Internet of Things (IoT) devices have widely been used for convenient services. Many users with their IoT devices request various services to servers. Thus, the amount of users' personal information that servers need to protect has dramatically increased. To quickly and safely protect users' personal information, it is necessary to optimize the speed of the encryption process. Since it is difficult to provide the basic services of the server while encrypting a large amount of data in the existing CPU, several parallel optimization methods using Graphics Processing Units (GPUs) have been considered. In this paper, we propose several optimization techniques using GPU for efficient implementation of lightweight block cipher algorithms on the server-side. As the target algorithm, we select high security and light weight (HIGHT), Lightweight Encryption Algorithm (LEA), and revised CHAM, which are Add-Rotate-Xor (ARX)-based block ciphers, because they are used widely on IoT devices. We utilize the features of the counter (CTR) operation mode to reduce unnecessary memory copying and operations in the GPU environment. Besides, we optimize the memory usage by making full use of GPU's on-chip memory such as registers and shared memory and implement the core function of each target algorithm with inline PTX assembly codes for maximizing the performance. With the application of our optimization methods and handcrafted PTX codes, we achieve excellent encryption throughput of 468, 2593, and 3063 Gbps for HIGHT, LEA, and revised CHAM on RTX 2070 NVIDIA GPU, respectively. In addition, we present optimized implementations of Counter Mode Based Deterministic Random Bit Generator (CTR_DRBG), which is one of the widely used deterministic random bit generators to provide a large amount of random data to the connected IoT devices. We apply several optimization techniques for maximizing the performance of CTR_DRBG, and we achieve 52.2, 24.8, and 34.2 times of performance improvement compared with CTR_DRBG implementation on CPU-side when HIGHT-64/128, LEA-128/128, and CHAM-128/128 are used as underlying block cipher algorithm of CTR_DRBG, respectively.

Keywords: CHAM; LEA; HIGHT; Graphic Processing Unit (GPU); CUDA; Counter (CTR) mode; parallel processing

---

1. Introduction

As the era of the 4th industrial revolution enters, the amount of information processed in real-time is increasing exponentially. In particular, as the number of users sharply increases due

to the development of the Internet of Things (IoT) technology and cloud computing services, the need to protect user's personal information also increases. Accordingly, various encryption technologies have been studied and applied to protect user's personal information.

However, from the server's point of view, it is very burdensome to process data encryption as well as basic services that the server has to provide. Since the encryption operation creates an additional time load, it is necessary to optimize the cipher used for encryption to provide data encryption services to many users in real-time.

Various optimization studies have been conducted on existing ciphers such as Advanced Encryption Standard (AES) [1] in the Central Processing Unit (CPU) environment. However, AES has limitations in optimizing on small devices such as microcontrollers. Therefore, it is efficient to optimize the lightweight block cipher algorithm considering the operation in a constrained environment. However, few optimization studies have been conducted on lightweight ciphers yet. Therefore, in this paper, we propose several methods of optimizing lightweight block ciphers using the Graphics Processing Unit (GPU) from the standpoint of the server that encrypts a large amount of data. The reason that the server uses the same lightweight block cipher used by the IoT device is that the server and the device must use the same cipher to decrypt each other's encrypted data.

By optimizing the lightweight block encryption algorithm in the server, data encryption can be provided quickly to multiple IoT devices. Using a GPU specialized for parallel computing, the server can quickly encrypt data transmitted from multiple IoT devices.

In this paper, lightweight block cipher algorithms to be optimized are HIGHT [2], LEA [3], and CHAM [4,5]. We present various methods that can implement operations and memory access methods inside lightweight block ciphers in a direction optimized for the GPU platform, and introduce several techniques to reduce time load by utilizing the features of the counter (CTR) operation mode. In addition, an asynchronous execution technique has been proposed to reduce the memory copy time between the CPU and GPU. In addition, an additional method to efficiently use registers and eliminate unnecessary operations has been suggested by using the GPU's inline assembly language inside the encryption operations.

Based on the optimization method for several lightweight block ciphers in CTR operating mode, we propose several optimization methods of Counter Mode Based Deterministic Random Bit Generator (CTR_DRBG) [6].

The contributions of our paper can be summarized as follows.

1. Proposing General Optimization Methods for Add-Rotate-Xor (ARX)-based lightweight block ciphers on GPU.
   We optimize not only the high security and low weight (HIGHT) [2], which was established as the ISO/IEC international block encryption algorithm standard in 2010, but also the Lightweight Encryption Algorithm (LEA) [3] established as the ISO/IEC international lightweight block encryption algorithm standard in 2019, and the CHAM algorithm proposed in 2017 [4] and revised in 2019 [5]. In this paper, we propose several optimization methods that can be applied commonly to the following Add-Rotate-Xor (ARX) operation based lightweight block ciphers. We propose an optimization method that can reduce unnecessary operations by taking advantage of the fact that the nonce value does not change in CTR mode. In addition, we introduce some methods to efficiently proceed with encryption using the counter value in CTR mode by utilizing the characteristics of the GPU. Rather than performing simple parallel encryption, we introduce some methods that allow multiple GPU threads to effectively access and use registers and shared memory inside the GPU. By these optimizations, we present excellent encryption speeds for target lightweight block cipher algorithms. GPU encryption kernels show encryption speeds of 468 Gbps for HIGHT, 2593 Gbps for LEA, and 3063 Gbps for CHAM. This result was measured on the RTX 2070, one of NVIDIA's Turing architecture products. These results have been shown better throughput performance even compared to other existing studies.

2. **Proposing Optimization Methods for actual encryption service provision environment**
   We not only optimize the GPU kernel time of the encryption algorithms but also proposed several optimization methods for memory copying time that must be performed between the CPU and GPU. The encryption speed operating inside the GPU is important, but to provide an actual encryption service, data copying time between the CPU and the GPU in real-time is also important. Therefore, we propose some methods to eliminate unnecessary memory copy time between CPU and GPU by using CTR mode. In addition, we introduce a method that can reduce the idle state as much as possible while the CPU and GPU perform tasks asynchronously using Compute Unified Device Architecture (CUDA) stream. Through these optimization methods, when encryption performance includes memory copying time, performance improvements were achieved by 67% in HIGHT, 59% in LEA, and 96% in CHAM compared to the implementation without any optimization methods.
3. **Optimization of CTR_DRBG based on efficient CTR implementation**
   We propose several optimization methods for lightweight block ciphers by utilizing the characteristics of the CTR operation mode. Using these methods, the CTR_DRBG [6], a deterministic random number generator using the CTR operation mode, has additionally been optimized. The extract function in CTR_DRBG has been optimized through the optimization methods presented in this paper. In addition, various optimization methods have been proposed from the progress structure of CTR_DRBG. As a result of the optimization of CTR_DRBG on the GPU, when using HIGHT, LEA, and CHAM, the performance was achieved that was 52.2, 24.8, 34.2 times faster than the previous CPU implementation, respectively.

Abbreviations used in this paper are summarized in Table 1.

Table 1. Abbreviation table.

Abbreviations	Full Name
IoT	Internet of Things
CPU	Central Processing Unit
GPU	Graphics Processing Unit
ARX	Add-Rotate-Xor operations
HIGHT	HIGh security and light weigHT
AES	Advanced Encryption Standard
LEA	Lightweight Encryption Algorithm
ALU	Arithmetic Logic Units
CTR	CounTeR
ECB	Electronic CodeBook
DRBG	Deterministic Random Bits Generator
CUDA	Compute Unified Device Architecture
SM	Streaming Multiprocessors

2. Related Works

So far, various cryptographic algorithms have been developed and optimized. In the case of AES, optimized implementations have been proposed in various environments such as GPUs. However, only a few studies have been done on the optimization of the lightweight block cipher algorithm in GPU.

In the CPU and microprocessor platform, several optimization studies have been conducted. Some studies have been optimized on the CPU platforms. Others have been optimized on microprocessors such as AVR, MSP, and ARM. In [7], HIGHT was optimized in the MSP430 environment, which used in

sensor networks. In [8], efficient all-in-one implementation techniques considering the IoT environment were conducted. In [9], CHAM was optimized to perform a fast single block encryption in the CPU platform. In [10], the optimization methods in ARM Cortex-A53 using NEON SIMD for CHAM were proposed. In this paper, we have implemented parallel optimization in the CPU platform through OpenMP ourselves. As a result of implementing parallel optimization using OpenMP in the CPU, the throughput of HIGHT was measured 1.05, LEA was 4.89, and CHAM was 1.65 Gbps in the AMD Ryzen 5 3600 CPU environment. When parallel encryption was performed in ECB mode instead of CTR mode, the performance of CTR implementation and ECB implementation was not significantly different.

To the best of our knowledge, only a few studies have optimized HIGHT and CHAM on the GPU platform. In the case of LEA, a parallel optimization study for LEA was conducted at ICISC 2013 [11]. The LEA was optimized using the various optimization methods, such as coalesced memory access and inline PTX code. In JKIISC'2015, an LEA optimization study using GPU shared memory was conducted [12]. In [13], the excellent performance improvement was achieved through coarse-grained optimization using thread warp. By utilizing the characteristics of warp actively, terabit throughput was proposed for various block ciphers. In [14], CHAM and LEA were optimized in GPU environment. Terabit throughput was achieved by integrating and resolving various memory problems that could occur in the GPU environment.

In the CTR_DRBG, to the best of our knowledge, this is the first work to optimize CTR_DRBG in the GPU environment. In addition, since only CTR_DRBG among DRBGs generates random numbers using block ciphers, we present the first GPU CTR_DRBG implementation using target lightweight block cipher algorithms. The result of optimized CTR_DRBG can be an indicator of future studies.

3. Background

This section describes the optimization target lightweight block algorithms: HIGHT, LEA, and CHAM. This section also describes CTR mode, one of the block cipher operation modes, and CTR_DRBG, a deterministic random bits generator that utilizes CTR mode. In addition, the GPU, which is a platform to propose an optimization method, will be described. In GPU, we introduce the parallel computing characteristics and memory structure.

3.1. Overview of Target Algorithms

3.1.1. HIGHT

HIGHT [2] was developed in 2005 for portable devices and mobile environments and certified by ISO/IEC international standards in 2010. HIGHT supports 128-bit key and 64-bit plaintext. In the key scheduling, the master key is used to generate the whitening key and the round key. In the encryption, the plaintext is encrypted through a transformation function by the whitening key before and after of the round function. The round function consists of an *XOR* and a *Rotate* function. In Figure 1, the round function of HIGHT is given. Notations $X^{\lll i}(i \in [0,7])$ and SK indicate left *Rotate* and sub key, respectively. P_0 to P_7 constitute plaintext. At the beginning and end of the round, a conversion process with whitening keys is performed.

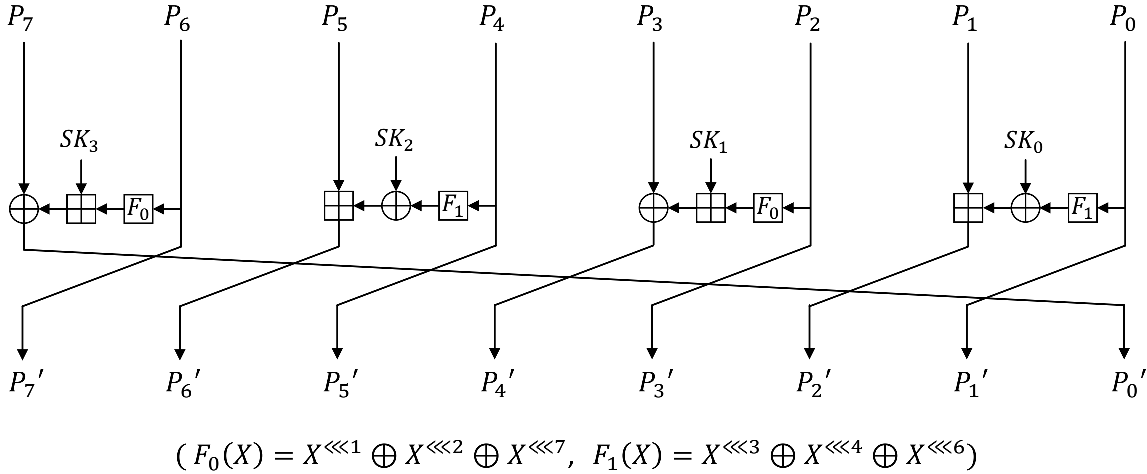

Figure 1. Round function of high security and light weight (HIGHT).

3.1.2. LEA

LEA [3] is a lightweight block cipher certified by ISO/IEC international standards in 2019. LEA was developed to provide fast encryption in cloud computing services or mobile devices. The size of the plaintext is 128 bits. LEA is classified according to the key size of 128, 192, and 256 bits. Depending on the key size, the number of rounds is set to 24, 28, and 32 rounds for 128, 192, and 256 bits, respectively. The round function follows the ARX structure, including *ADD*, *Rotate*, and *XOR* operations. Figure 2 shows the round function of the LEA. *RK* indicates the round key, and 6 round keys are used per round. In the Figure, the symbol appearing after the *ADD* operation indicates the *Rotate* operation, and the number inside the Figure indicates the number of bits to rotate.

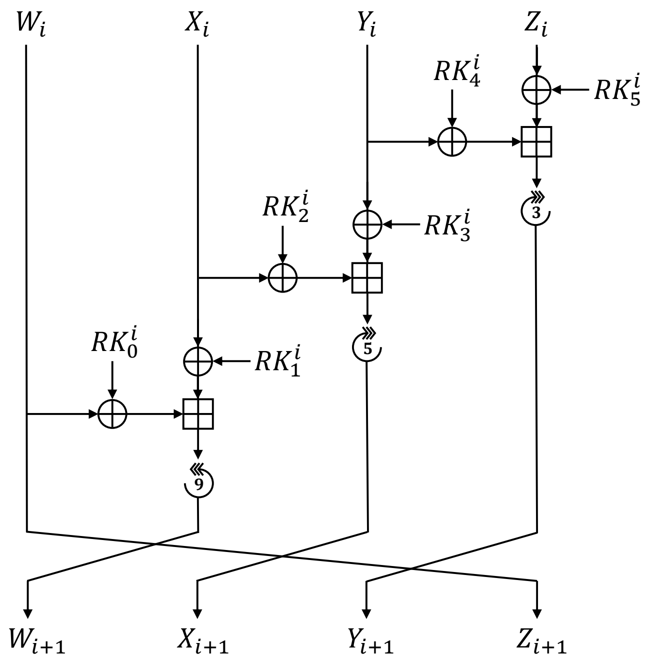

Figure 2. Round function of Lightweight Encryption Algorithm (LEA).

3.1.3. CHAM

CHAM [4,5] is a lightweight block cipher algorithm proposed in 2017 and revised in 2019. In this paper, revised CHAM has been described. CHAM follows the generalized four-branch Feistel structure

based on ARX operations. CHAM consists of three types, including CHAM-64/128, CHAM-128/128, and CHAM-128/256 (plaintext length/key length). Depending on each type, the number of rounds is repeated by 88, 112, and 120 for CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively. CHAM has a smaller round key size than other cryptographic algorithms. The identical round key is reused several times throughout the round function. Figure 3 shows the round function of CHAM, where the i indicates the order of round. In odd rounds of CHAM, one bit is left rotated first and eight bits left rotated last. In even rounds, eight bits are left rotated first and one bit left rotated last. The round key k is reused a certain number of rounds, and $2k/w$, the size of the repeated section, equals 16 in CHAM-64/128 and CHAM-128/256, and eight in CHAM-128/128.

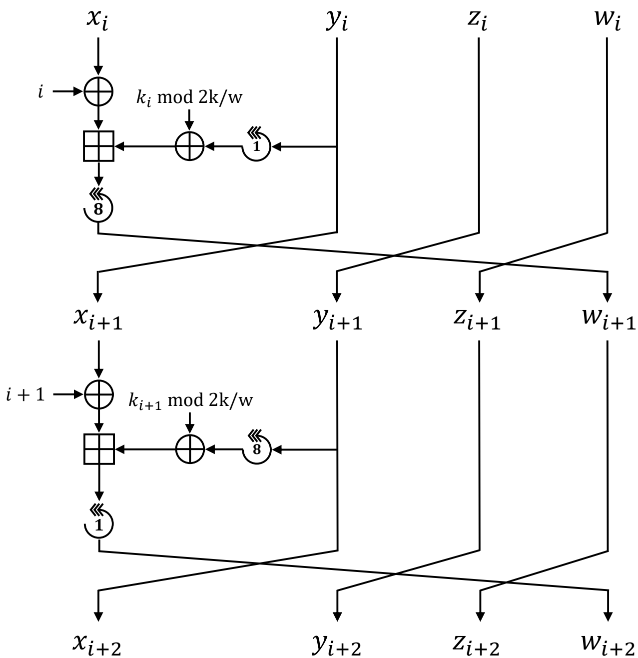

Figure 3. Round function of CHAM.

3.1.4. Counter Modes of Operation

There are various methods of operation of block ciphers. The most representative method is the ECB operation mode. In the ECB operation mode, data are divided by the size of the plaintext block size. Each block is encrypted with the same key. However, the ECB operation mode has security vulnerabilities. When the plaintext is the same, the ciphertext is always the same. On the other hand, the CTR operation mode does not encrypt plaintext, directly. This proceeds the encryption using a counter value, which is updated by adding one for each block. The counter value that has passed through the encryption algorithm is XORed with the plaintext to form a ciphertext.

3.2. Overview of CTR_DRBG

CTR_DRBG [6] is a type of deterministic random bit generator, which receives entropy as an input and generates a random number sequence according to a determined algorithm. CTR_DRBG generates a random number using the block encryption CTR operation mode, and the overall CTR_DRBG operation process is shown in Figure 4.

As shown in Figure 4, the entropy collected from the noise source is entered as inputs of CTR_DRBG with nonce and personalization string. The input values are used to initialize the internal state in the instantiate function, and the instantiate function consists of a derivation function and an internal update function. After that, the generate function is repeatedly called and extracts a random number sequence. At this time, when predict resistance is activated or when the reseed

counter value, which increases when random numbers are repeatedly output, reaches the threshold, the reseed function is called to update the external state. In other cases, the generate function directly outputs a random number through the extract function, and the internal state is updated through the update function.

The internal main functions of CTR_DRBG are derivation function, update function, and extract function. The derivation function is operated through the CBC_MAC operation mode, and the update function and the extract function are operated through the CTR operation mode. Figure 5 shows the derivation function progress.

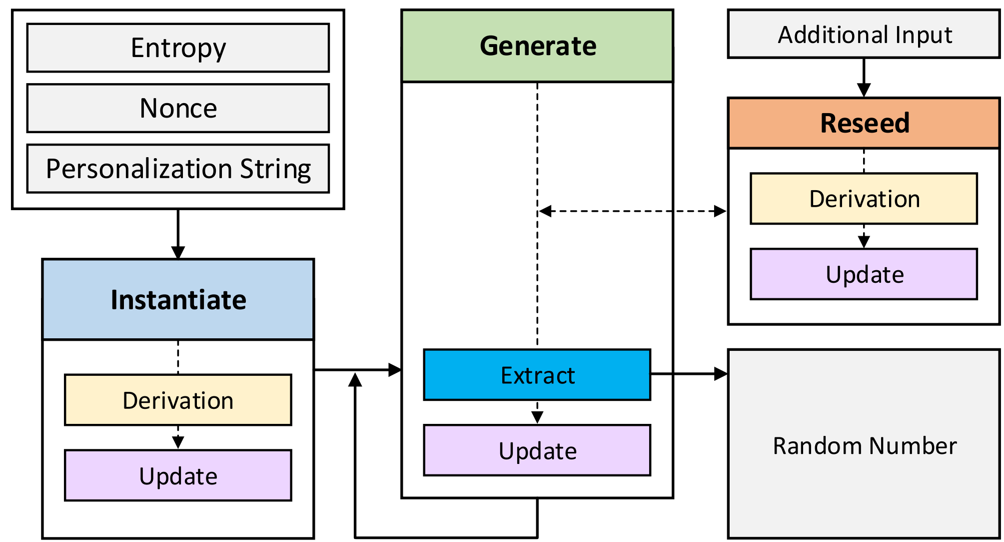

Figure 4. Counter Mode Based Deterministic Random Bit Generator (CTR_DRBG) overall process.

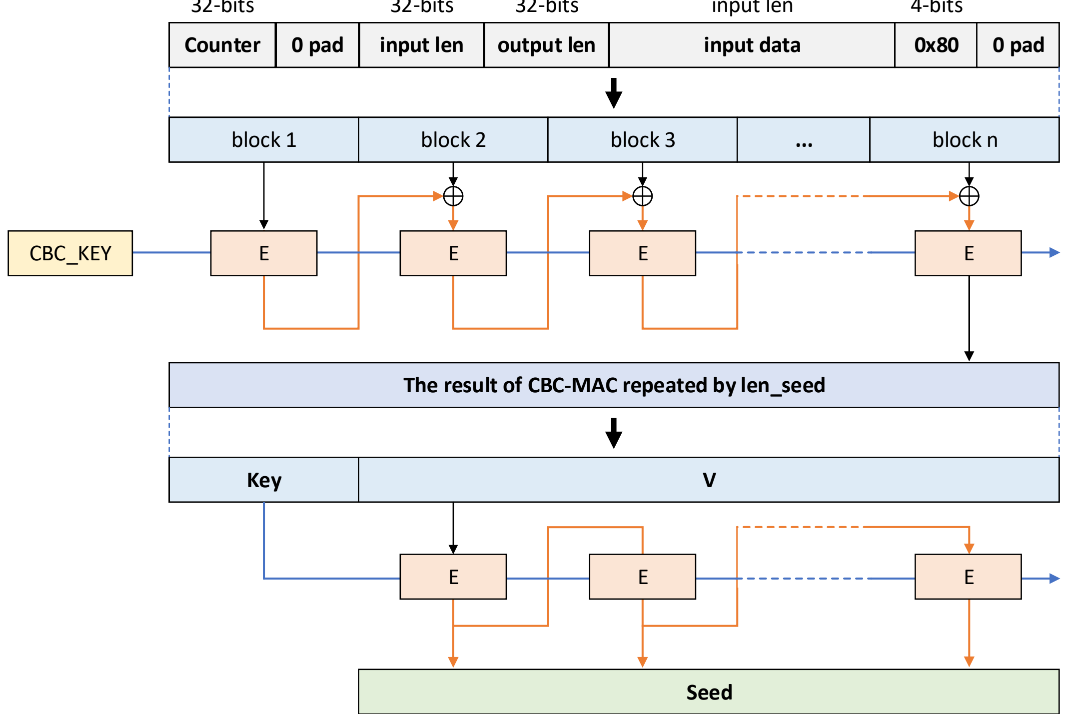

Figure 5. Derivation function.

The derivation function uses CBC_MAC, and the data that goes into the input of CBC_MAC are as follows: counter value, zero paddings to fit the block size including counter value, value for input length, value for output seed length, input variable, constant hexadecimal value 0×80, and zero paddings to fit the entire length to the block size. The input values of CBC_MAC are divided by block size and encrypted by the constant CBC_KEY. Each time CBC_MAC is executed, one block size result is output, and this is repeatedly accumulated by len_seed, which is a predetermined value for each block encryption algorithm. The resulting output as many times as the number of repetitions is again divided into a key part and a plaintext V part, and encrypted through the CBC operation mode. The result of combining each block output becomes the seed value.

Figure 6 shows the update and extract function progress. Both the update function and the extract function proceed through the CTR operation mode. However, in the extract function, the result of CTR encryption becomes the output random number, and n at this time becomes the random number output block length. In the update function, the result of CTR encryption is XORed with the input data to update the internal state. In update function, n becomes the len_seed value.

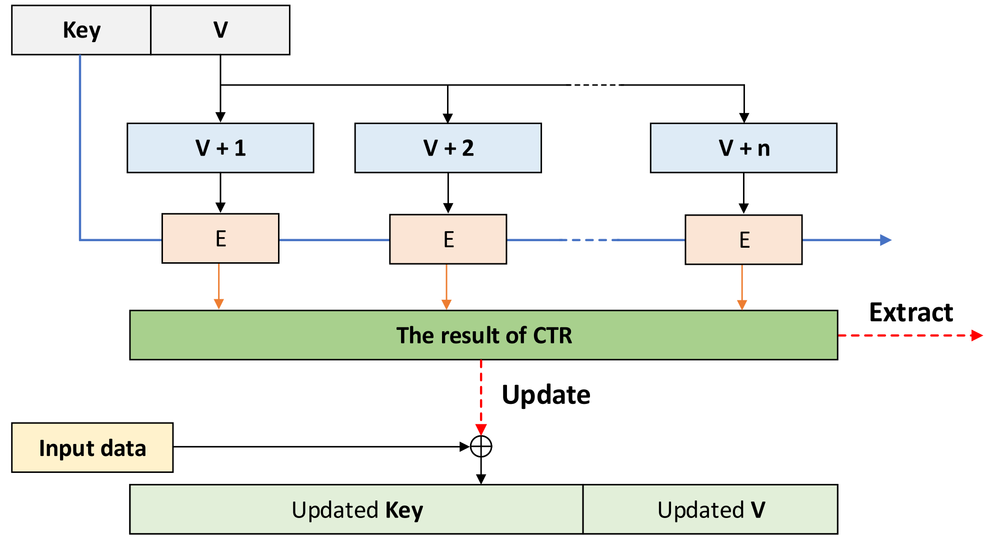

Figure 6. Update function and extract function.

3.3. GPU Architecture Overview

Even though GPU was originally developed for graphic and image processing, they are widely used for general purpose applications including acceleration of crypto operations, machine learning, and so on. The NVIDIA is a representative GPU manufacturer, and GPUs produced by NVIDIA are classified according to their architectures. NVIDIA TITAN RTX, the flagship GPU of Turing architecture released in 2018, among many architectures, has 4608 CUDA cores with a 1770 MHz boost clock. It also has 24 GB of GDDR6 graphics memory and has a memory clock of 1750 MHz.

The CPU uses most of the chip area for cache, while the GPU uses most of the chip area for arithmetic logic units (ALUs). GPUs use hardware threads that run the same instruction stream on different data sets. There are multiple streaming multiprocessors (SM) within the GPU, and a collection of threads running on one multiprocessor is called a block. GPUs utilize these numerous threads to perform high-level parallelism in their applications.

GPU memory is made up of many different types. Figure 7 shows the memory structure of GPU devices. GPU has a memory area named global memory, constant memory, and texture memory, which is shared by all threads. Since these memory types are the first memory areas to be accessed in data copy with the CPU using PCIe, so the memory size is very large. Global memory is enormous

because it uses the DRAM area of the GPU. However, it has the disadvantage that the memory access speed is very slow compared to other memory areas. To solve this shortcoming, from the Fermi architecture GPU, it is possible to cache and use global memory by adding a cache to the SM. However, because the cache size is very small, there are limitations to actively use it. A register is a memory area used by threads responsible for parallel operations in a block. Although small in size, it is very fast. If threads use a lot of registers, Since there is an upper limit to the size of registers per block, some of the memories in the registers are stored as local memory. Since local memory is a memory existing in the DRAM area, the access speed is slower than register. Shared memory is a memory shared by threads within a block, and has the advantage of fast access. Since the data are shared in shared memory, the values in the memory can be affected by other threads.

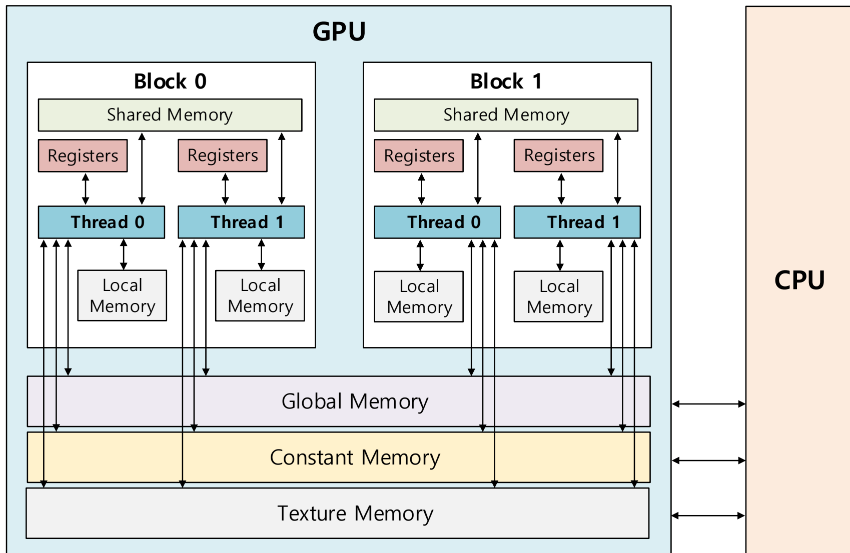

Figure 7. Memory structure of GPU devices.

In 2006, NVIDIA announced CUDA [15]. CUDA is a parallel computing platform and API model that enables the use of the general-purpose computing on graphics processing units (GPGPU) technique, which is used for general-purpose computation of applications that process GPUs used only for traditional graphics tasks. CUDA can be used in a various language such as C and C++, and new versions are updated whenever a new GPU or architecture is released. CUDA compute capability supported version varies depending on the GPU used.

CUDA programming uses a language that adds CUDA-specific syntax as a library. CUDA code consists of calling a function on the host CPU called a kernel that only works inside the GPU. When processing data in the GPU kernel, the memory on the host CPU is not immediately available, requiring an additional process of copying the required memory area from the host to the device in advance. Additional processes may also involve copying data back from the device to the host after completing the operation.

Currently, many types of the latest NVIDIA GPUs that can process GPGPU using the CUDA library have been released. As new GPUs are developed, new instructions are constantly being developed. NVIDIA GPUs have different features for different architecture generations, and different CUDA versions are available. Table 2 describes the examples and features from the latest Turing architecture to the Maxwell architecture. There is a minimum CUDA version that must be satisfied for each GPU architecture, and this is called CUDA compute capability.

Table 2. Specifications of Compute Unified Device Architecture (CUDA)-enabled NVIDIA GPU architectures.

Architecture	Maxwell	Pascal	Turing
GPU Chips example	GTX 980 Ti	GTX 1080 Ti	RTX 2080 Ti
SM count	24	28	68
Core count	2816	3584	4352
Memory Size	6 GB	11 GB	11 GB
Base clock	1000 MHz	1480 MHz	1350 MHz
CUDA compute capability	5.2	6.1	7.5

4. Proposed Implementation Techniques in Target Lightweight Block Ciphers

This section proposes several methods that can be optimized on the GPU through the target lightweight block cipher algorithm CTR operation mode. This section explains the part where optimization can be applied using the features of the CTR operation mode and presents some methods to effectively execute the CTR operation mode by utilizing the features of the GPU. Several common optimization methods applicable to all target algorithms are first introduced, followed by appropriate optimization implementation methods for each algorithm.

4.1. Common Optimization Method

4.1.1. Parallel Counter Mode of Operation in GPU

In the general block cipher, the plaintext and key are entered as input values of the encryption algorithm, and the ciphertext is output. In the ECB operation mode, the data to be encrypted must always be plaintext. However, to handle data in the GPU, CPU's data must be copied to the GPU in advance. To perform encryption on the GPU through the ECB operation mode, an additional process of copying plaintext values stored in the CPU in advance to the GPU is required.

The CPU and GPU have ultimate fast computation speed. However, PCIe, the transmission path between the CPU and GPU, is relatively slow. Therefore, copying data between the CPU and GPU is time-consuming. Reducing this data copy time can make a significant optimization contribution in GPU implementations. This heavy memory copy time can be reduced through the counter mode of operation. A characteristic of the CTR operation mode is that it encrypts the counter value instead of plaintext. Since the plaintext is not used while encrypting on the GPU, the copying time of the plaintext is reduced.

Each thread that is in charge of computation inside the GPU has a unique number, the thread ID. Each unique index of threads can be used as a counter value whose value increases by one for each encryption block. As a result, the result of encrypting each thread ID as a counter value is the same as the result of encrypting each block while increasing the counter value by one. The CPU performs encryption by one block while increasing the counter value by one, but the GPU can encrypt the counter values by the number of threads even if each thread encrypts only once. Thus, CTR operation mode encryption using the advantage of these GPUs can show very fast encryption speed. Figure 8 shows the parallel CTR operation mode encryption process on these GPUs.

The round keys are generated through the key expansion function. Generated round keys are copied and used in global memory by the GPU. Global memory is a memory space that all threads can refer to in common but has a disadvantage that it is very slow compared to other memory spaces. To use the round keys efficiently, the round keys must be stored in another memory space. Shared memory is a memory shared in block units composed of multiple threads, which is faster than global memory. Shared memory cannot be initialized, and after it is declared in the GPU kernel function, the data in global memory are copied to shared memory.

Encryption proceeds using the round key stored in the shared memory. Shared memory consists of several banks. When different threads access the same memory bank, a problem called a bank conflict occurs. Figure 9 shows the state of the bank conflict problem. When a bank conflict occurs, the thread does not perform parallel operation and waits for sequential access to the memory bank. This causes a big drop in speed. To prevent bank conflict, the shared memory bank size was adjusted to use a unique bank for each thread, and the round keys were stored in each bank location. Therefore, when encryption is performed, each thread can use the round key stored in shared memory within its bank without duplicate access.

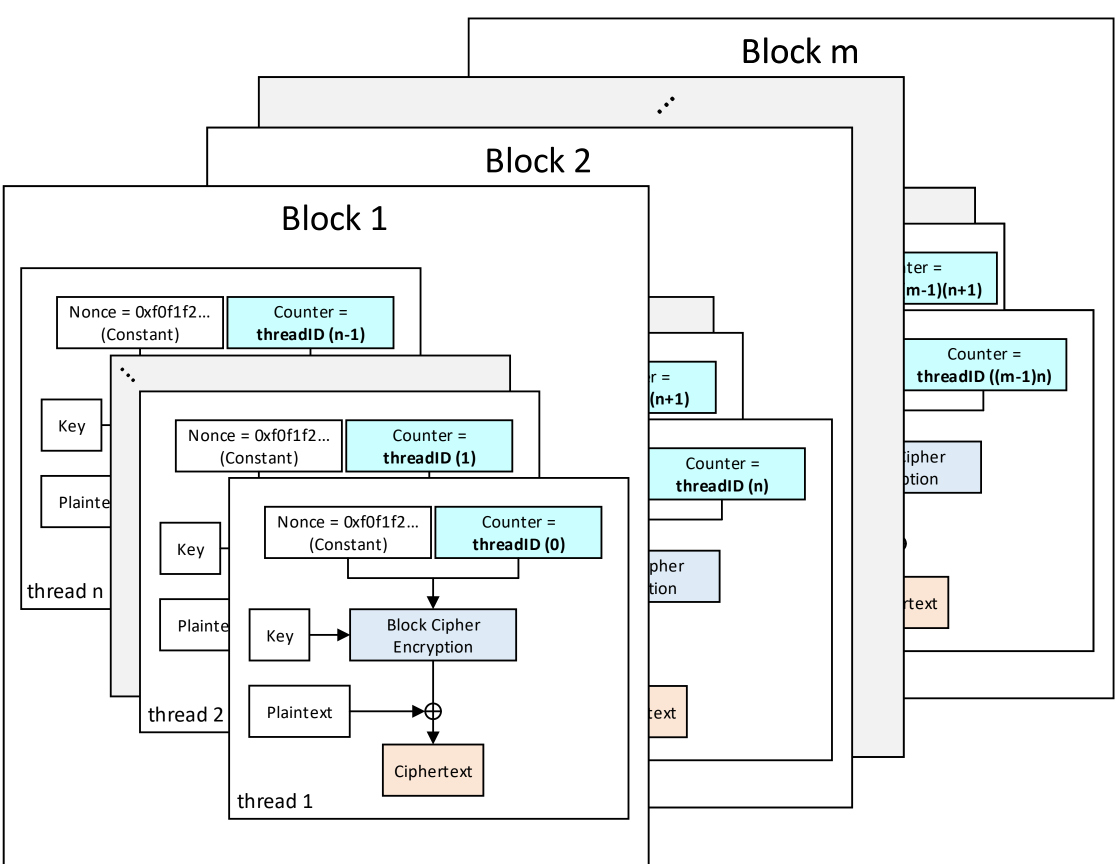

Figure 8. Parallel CTR encryption on GPU threads.

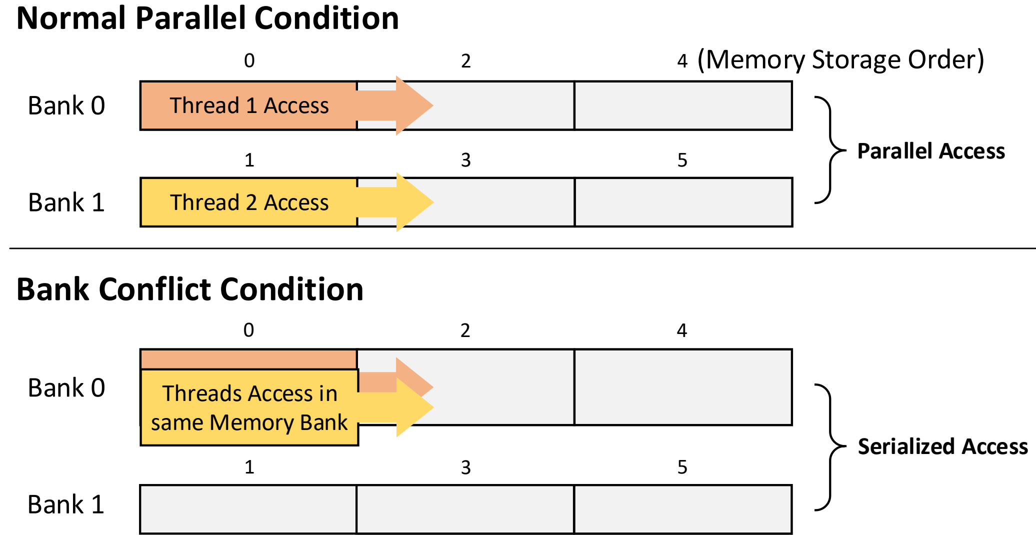

Figure 9. Bank conflict condition.

4.1.2. Reduce Memory Copy Time Using CUDA Stream

In addition to the optimization method through the CTR mode, an additional optimization method was applied to reduce the memory copy time between the CPU and GPU. As shown in Figure 10, the CPU enters the idle state and waits until the operation of the GPU is finished. When the GPU operation is finished, the GPU data are copied to the CPU. To reduce the CPU's idle time, asynchronous instructions can be executed using CUDA streams. The stream is composed of a single default stream, but CUDA instructions can be used to create multiple streams and divide data to perform operations. Since each stream is managed asynchronously, when the first stream finishes copying data from the CPU to the GPU and enters the kernel engine, the data copy process of the next stream proceeds immediately. In this case, the CPU can continuously process the CPU task while reducing the latency caused by the GPU operation. This optimization technique helps reduce memory copy time between the CPU and GPU.

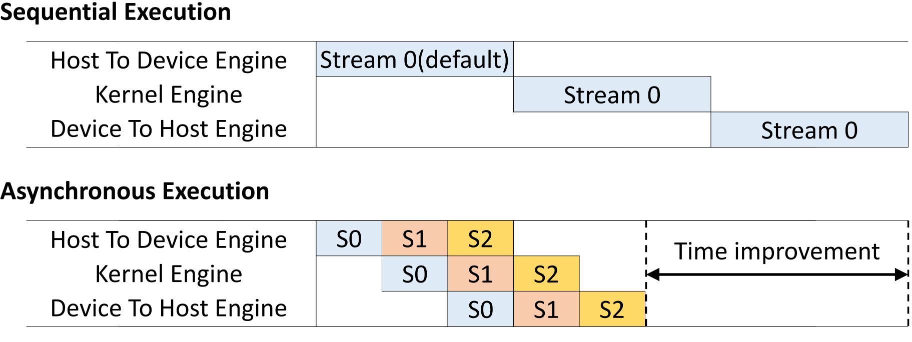

Figure 10. Asynchronous execution by using CUDA stream.

4.2. HIGHT

Unlike general block cipher algorithms like AES, HIGHT does not have a lookup table to suit the environment where hardware resources such as memory are insufficient. HIGHT uses eight 8-bit whitening keys and 128 8-bit subkeys from a 128-bit secret key. Since HIGHT's plaintext is only 8 bytes, not only the plaintext but also the whitening key can be stored and used in a register. In other words, one thread uses 16 bytes of the register, and one GPU block, which is a bundle of threads, can be implemented to store and use 128 bytes of subkey in shared memory.

In the CTR mode of operation, the nonce and counter values are entered into the encryption algorithm instead of plaintext. The counter value increases by 1 for each plaintext block, but the nonce value has the same value for all plaintext blocks. Since block cipher algorithms divide the plaintext into several words and perform the operation, the result of the operation for a fixed nonce value is the same for all blocks. By using these features, unnecessary operations for nonce value can be reduced. In the case of HIGHT, if 32 bits of 64 bits of plaintext are set as nonce and 32 bits as CTR value, the counter value from P_0 to P_3 in Figure 11 and the nonce value from P_4 to P_7 are used. There is no need to perform separate calculations for parts where only nonce values are used. In HIGHT, the operation of almost two rounds is reduced.

Accordingly, a table that can appear according to an 8-bit secret key is created in advance in the CPU, and unnecessary operations are eliminated in the GPU. Since all blocks use the same subkeys to perform CTR operation mode encryption, the precomputed table operation only needs to be performed once, and the result value is used for encryption as a constant value by all threads.

Additionally, by optimizing and implementing the existing cryptographic algorithm operation with the assembly language of the GPU, PTX assembly code, it is possible to utilize registers to the maximum and reduce unnecessary operations. In particular, by optimizing the rotation operation of the ARX-based block cipher algorithm through PTX assembly code, the optimization method to reduce unnecessary rounds in the CTR mode is applied. In addition, the operation itself is optimized to see an excellent optimization effect.

Figure 12 shows the PTX assembly code for a round of HIGHT on a CUDA GPU. Whole plaintext P_0 to P_7 can be stored in a register to perform encryption to achieve fast encryption performance. Entire subkeys have been stored and used in shared memory. ROL in the footnote means the left rotate. For example, $ROL(P_0, 3)$ means rotated by three bits to the left. Rotate can perform efficient computation using the funnel shift command of GPU.

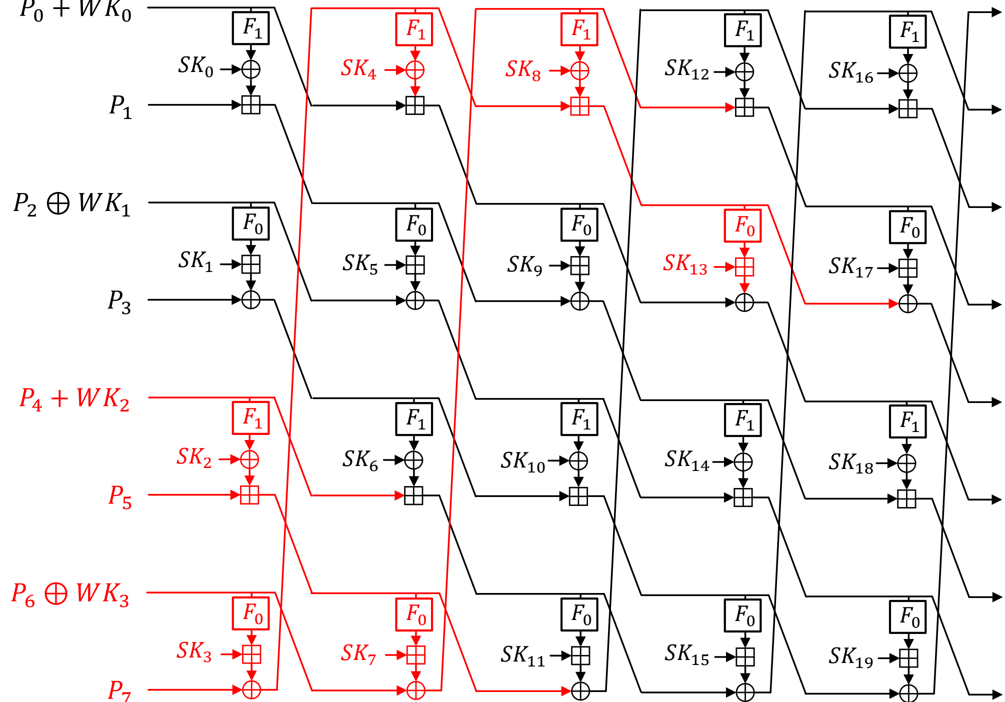

Figure 11. Rounds that can be efficiently progressed by a fixed nonce in HIGHT.

```
asm("{\n\t"
        " .reg.type          t0, t1, t2;              \n\t"  //Declare register temp
        " shf.l.mode.type    t0, %9, %9, 3;           \n\t"  //t0 = ROL(P0, 3) ┐
        " shf.l.mode.type    t1, %9, %9, 4;           \n\t"  //t1 = ROL(P0, 4) ├ = F1[P0]
        " shf.l.mode.type    t2, %9, %9, 6;           \n\t"  //t2 = ROL(P0, 6) ┘
        " xor.type           t0, t0, t1;              \n\t"  //t0 = t0 XOR t1
        " xor.type           t0, t0, t2;              \n\t"  //t0 = t0 XOR t2
        " xor.type           t0, t0, %16;             \n\t"  //t0 = t0 XOR sk[i*4]
        " add.mode.type      %0, t0, %9;              \n\t"  //P0 = t0 + P1
                ⋮                  ⋮                    ⋮
        " shf.l.mode.type    t0, %15, %15, 1;         \n\t"  //t0 = ROL(P6, 1) ┐
        " shf.l.mode.type    t1, %15, %15, 2;         \n\t"  //t1 = ROL(P6, 2) ├ = F0[P6]
        " shf.l.mode.type    t2, %15, %15, 7;         \n\t"  //t2 = ROL(P6, 7) ┘
        " xor.type           t0, t0, t1;              \n\t"  //t0 = t0 XOR t1
        " xor.type           t0, t0, t2;              \n\t"  //t0 = t0 XOR t2
        " add.mode.type      t0, t0, %19;             \n\t"  //t0 = t0 + sk[i*4+3]
        " xor.type           %6, t0, %15;             }\n\t" //P6 = t0 XOR P7
        " : "+r"(P0), "+r"(P1), ... , "+r"(P7)                //: output %0, %1, ... , %7
      : "r"(P0), "r"(P1), ... , "r"(P7),                     //: input %8, %9, ... , %19
      "r"(sk[i*4]), "r"(sk[i*4 + 1]), "r"(sk[i*4 + 2]), "r"(sk[i*4 + 3])   // sk : sub key
);                                                           // i = round
```

Figure 12. Example of inline PTX assembly codes of HIGHT round function.

4.3. LEA

Since LEA uses many round keys, the number of memory accesses is high. When the round key is directly stored and used from global memory, the total kernel speed could be slow by the memory access time. Using shared memory or constant memory can speed up memory access time. Using shared memory can take advantage of fast memory access speeds, but be cautious of bank conflicts in this case. Constant memory shows a fast memory access speed when copying previously cached data, but when accessing data in uncached constant memory, it has a similar access speed to global memory.

In the CTR operation mode of LEA, since 64 bits of the 128-bit plaintext are used as counter values and the remaining 64 bits are input as nonce values, certain operations can be omitted. In Figure 13, it can reduce the total amount of calculations for one round, and a fixed nonce affects even the beginning of the third round.

Figure 14 shows the PTX assembly code for a round of LEA on a CUDA GPU. The $ROL(Z, 3)$ of the footnote means right rotate by three bits. In the case of LEA, the round key size is very large, so the whole round key cannot be stored in the register. Therefore, part of the round key is stored in a register for use. Since LEA performs right rotate in addition to the left rotate, the left rotate and right rotate are performed using $shf.l$ and $shf.r$ instruction in the PTX code.

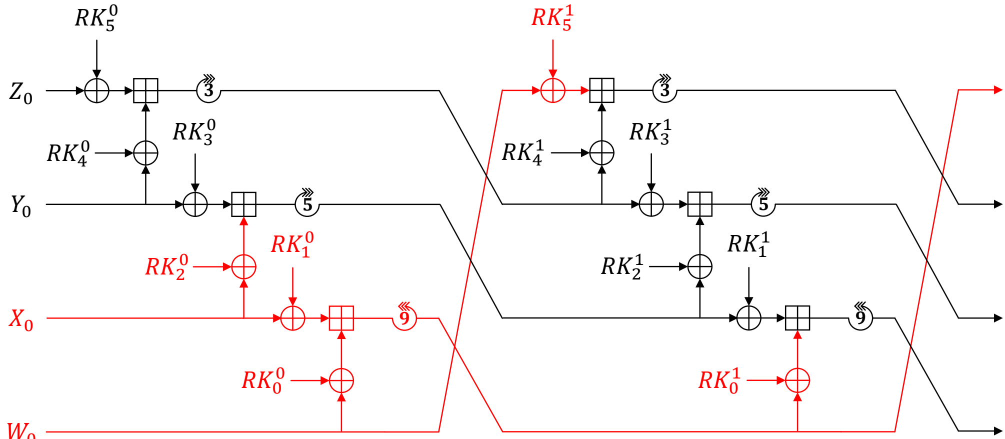

Figure 13. Rounds that can be efficiently progressed by a fixed nonce in LEA.

```
asm("{\n\t"                                                  //i = round
        " xor.type          %2, %6, %12;          \n\t"  //Y = Y XOR rk[i + 4]
        " xor.type          %3, %7, %13;          \n\t"  //Z = Z XOR rk[i + 5]
        " add.mode.type     %3, %6, %7;           \n\t"  //Z = Y + Z
        " shf.r.mode.type   %3, %7, %7, 0x3       \n\t"  //Z = ROR(Z, 3)
        " xor.type          %1, %5, %10;          \n\t"  //X = X XOR rk[i + 2]
        " xor.type          %2, %6, %11;          \n\t"  //Y = Y XOR rk[i + 3]
        " add.mode.type     %2, %5, %6;           \n\t"  //Y = X + Y
        " shf.r.mode.type   %2, %6, %6, 0x5;      \n\t"  //Y = ROR(Y, 5)
        " xor.type          %0, %4, %8;           \n\t"  //W = W XOR rk[i]
        " xor.type          %1, %5, %9;           \n\t"  //X = X XOR rk[i + 1]
        " add.mode.type     %1, %4, %5;           \n\t"  //X = W + X
        " shf.l.mode.type   %1, %5, %5, 0x9;      \n\t"  //X = ROL(X, 9)
        " xor.type          %3, %7, %14;          \n\t"  //Z = Z XOR rk[i + 10]
        " xor.type          %0, %4, %15;          \n\t"  //W = W XOR rk[i + 11]
        " add.mode.type     %0, %7, %4;           \n\t"  //W = Z + W
        " shf.r.mode.type   %0, %4, %4, 0x3;      }\n\t"//W = ROR(W, 3)
      : "+r"(W), "+r"(X), "+r"(Y), "+r"(Z)                          //: output %0, %1, %2, %3
      : "r"(W), "r"(X), "r"(Y), "r"(Z),                             //: input %4, %5, ... %15
      "r"(rk[i]), "r"(rk[i + 1]), "r"(rk[i + 2]), "r"(rk[i + 3]),   // W, X, Y, Z : word
      "r"(rk[i + 4]),"r"(rk[i + 5]),"r"(rk[i + 10]), "r"(rk[i + 11]) // rk : round key
);
```

Figure 14. Example of inline PTX assembly codes of LEA round function.

4.4. CHAM

The CHAM algorithm repeatedly uses a short round key contrast to other algorithms. Therefore, it is possible to have much faster memory access speed by storing all the CHAM plaintext and round keys in a register. However, if the number of threads is too large, the maximum register size that can be used by each thread is limited. Therefore, when applying the optimization technique using registers, the number of threads was appropriately selected based on the number of registers to be used for encryption.

In CHAM, when using the CTR operation mode, it is possible to omit the operation of the fixed nonce value. Figure 15 shows that a total of two rounds of computation can be reduced.

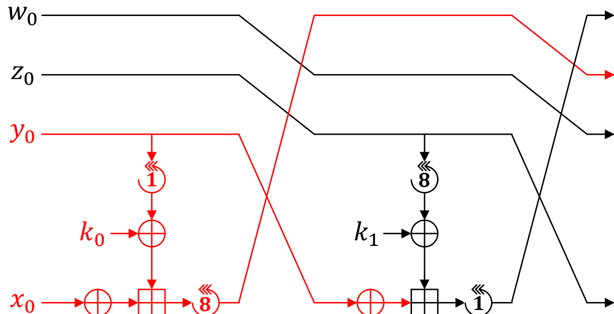

Figure 15. Rounds that can be efficiently progressed by a fixed nonce in CHAM.

Figure 16 shows the PTX assembly code for a round of CHAM on a CUDA GPU. "*r*" means register, and all inputs and outputs are entered into the PTX code through registers. In the case of CHAM, since all round keys can be stored in a register, whole round keys can be used as an input of a PTX code. On the GPU, the rotate operation on the existing CPU can be replaced with the funnel shift operation. *Type* is a variable data type, which is $u16$ for CHAM-64/128 and $u32$ otherwise. In the CHAM encryption process, the process of rotating the word is necessary at the end of each round. However, the word rotation process is unnecessary because the PTX code calls the corresponding register instead.

```
asm("{\n\t"
    " .reg.type          t0,t1,t2,t3,t4,t5;     \n\t"  //Declare register t0,t1,t2,t3,t4,t5
    " shf.l.mode.type    t0, %5, %5, 0x1;       \n\t"  //t0 = ROL(y, 1)
    " xor.type           t1, t0, %8;            \n\t"  //t1 = t0 XOR rk[0]
    " xor.type           t2, %4, 0x0;           \n\t"  //t2 = x XOR 0 (round number i = 0)
    " add.mode.type      t3, t1, t2;            \n\t"  //t3 = t1 + t2
    " shf.l.mode.type    t4, t3, t3, 0x8;       \n\t"  //t4 = ROL(t3, 8)
    " shf.l.mode.type    t0, %6, %6, 0x8;       \n\t"  //t0 = ROL(z, 8)
    " xor.type           t1, t0, %9;            \n\t"  //t1 = t0 XOR rk[1]
    " xor.type           t2, %5, 0x1;           \n\t"  //t2 = y XOR 1 (round number i = 1)
    " add.mode.type      t3, t1, t2;            \n\t"  //t3 = t1 + t2
    " shf.l.mode.type    t5, t3, t3, 0x1;       }\n\t" //t5 = ROL(t3, 1)
    : "+r"(x), "+r"(y), "+r"(z), "+r"(w)                //: output %0, %1, %2, %3
    : "r"(x), "r"(y), "r"(z), "r"(w),                   //: input %4, %5, ... %15
    "r"(rk[0]), "r"(rk[1]), "r"(rk[2]), "r"(rk[3]),     // x, y, z, w : word
    "r"(rk[4]), "r"(rk[5]), "r"(rk[6]), "r"(rk[7])      // rk : round key
);
```

Figure 16. Example of inline PTX assembly codes of CHAM round function.

4.5. CTR_DRBG Optimization Using Target Lightweight Block Ciphers

Since the core operation of CTR_DRBG is performed through the CTR operation mode, if the optimization method for the CTR operation mode is applied above, optimization implementation effects can be obtained in CTR_DRBG as well. In this paper, CTR_DRBG is optimized and implemented using HIGHT, LEA, and CHAM, which are target lightweight block algorithms that have proposed an optimization method. In this subsection, we propose a common optimization method using the structural features of CTR_DRBG.

4.5.1. Parallel Random Number Extraction

The main operation of CTR_DRBG is the CTR mode encryption process, so the method of optimizing block encryption in CTR operation mode can be applied to CTR_DRBG. In CTR_DRBG, the CTR mode of operation's encryption process is mainly used in the update function and extract

function. Figure 17 shows the use of GPU in the update and extract functions. Significant time improvement can be obtained by processing the CTR operation mode encryption process performed in parallel using GPU threads.

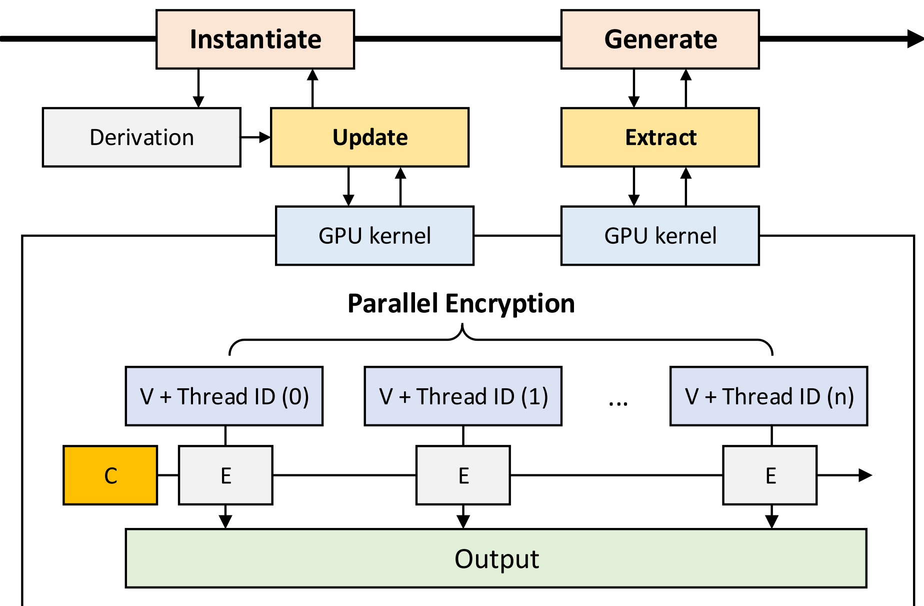

Figure 17. Parallel CTR_DRBG optimization on GPU.

When generating a random number sequence while calling multiple CTR_DRBGs, since the update function performs CTR mode encryption of 2 to 3 blocks depending on the type of the selected block cipher, 2 to 3 threads can implement one CTR_DRBG. In this paper, the optimization strategy for CTR_DRBG on GPU are two categories. The first thing that threads output a random number sequence by calling one CTR_DRBG function. In this case, since a different CTR_DRBG internal environment can be built by using a thread ID, which is a unique number for each thread, a more secure random number sequence can be output. However, memory and performance are limited because each thread has to bear the instantiate function, internal derivation function, and update function, which are processes until the random number sequence is output from CTR_DRBG. When each thread calls a number of CTR_DRBGs, the parallel CTR mode optimization method can be applied by encrypting one or two CTR_DRBGs in 2 to 3 threads according to the CTR block lenseed value of the update function.

Another optimization direction is to call one CTR_DRBG but set the length of the random number sequence very large inside to process many CTR mode blocks in parallel. That is, the CTR mode encryption process performed by the extract function is optimized and used as efficiently as possible. In this case, the same process is performed in the CPU until inside the CTR_DRBG just before the extract function but is implemented to process only the extract function with the GPU. This method applies only the CTR mode optimization method in one CTR_DRBG. The CPU only needs to manage one internal state, so it has the advantage of less memory load than the optimization method that calls multiple CTR_DRBGs. Outputting a large number of random numbers through one internal state can cause safety issues. However, the maximum value of the seed counter that calls the seed function is 2^{48}. This value is an extremely big number that is hard to reach. Therefore, even in a method of outputting a large number of random numbers through the GPU, the corresponding seed counter value is not transmitted.

In the extract function, the number of CTR mode blocks is determined according to the output length of the random number sequence. In the case of the optimization method that calls multiple CTR_DRBGs, since 2 to 3 threads are in charge of one CTR_DRBG in the update function, the random number sequence output length is also implemented to output 2 to 3 blocks according to the number

of threads. In the case of the optimization method of outputting a large number of random number sequences in one CTR_DRBG, the number of threads can be set to the number of CTR mode blocks according to the random number size to be output, so that each thread can encrypt each block in parallel. For example, when outputting a random number sequence having a block size of 16 bytes, to output a random number sequence of 1 MB, 65,536 threads can encrypt each CTR mode block once.

4.5.2. Method for Omitting Operations Using Constant Results

In the first CTR_DRBG operation, the internal state is initialized by the instantiate function. The derivation function and the update function in the instantiate function are slightly different from the derivation function and the update function in the generate function. Some input values of the instantiate functions are fixed.

As shown in Figures 18 and 19, the key and plaintext used for encryption are all fixed, so the ciphertext output is always fixed. In the first derivation function in the instantiate function, a counter value in the first block and a zero-padded value are entered as inputs. By the way, the counter value only increases by one while repeating CBC_MAC as much as len_seed, and since CBC_KEY is a constant value, the ciphertext has a fixed value. In the derivation function, the range of the constant value is not so large by the nature of the CBC operation mode.

In the update function inside the instantiate function, V and the key are all zeros. Therefore, all the output values by the CTR operating mode are fixed. By using the result of this operation as a constant value, the number of encryption times as much as len_seed can be reduced.

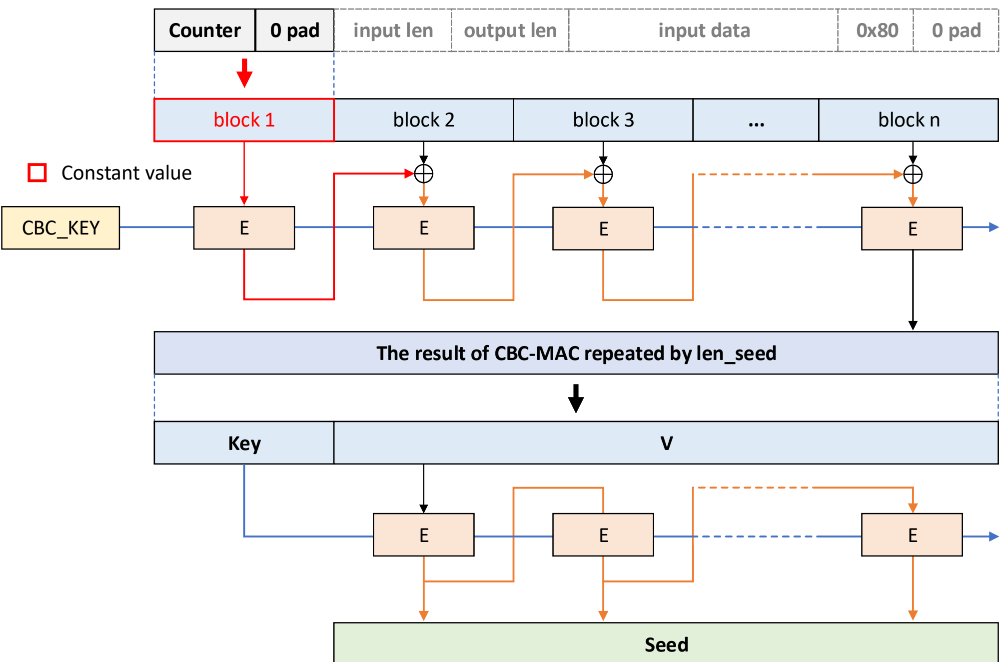

Figure 18. The part that can be constant value in the derivation function.

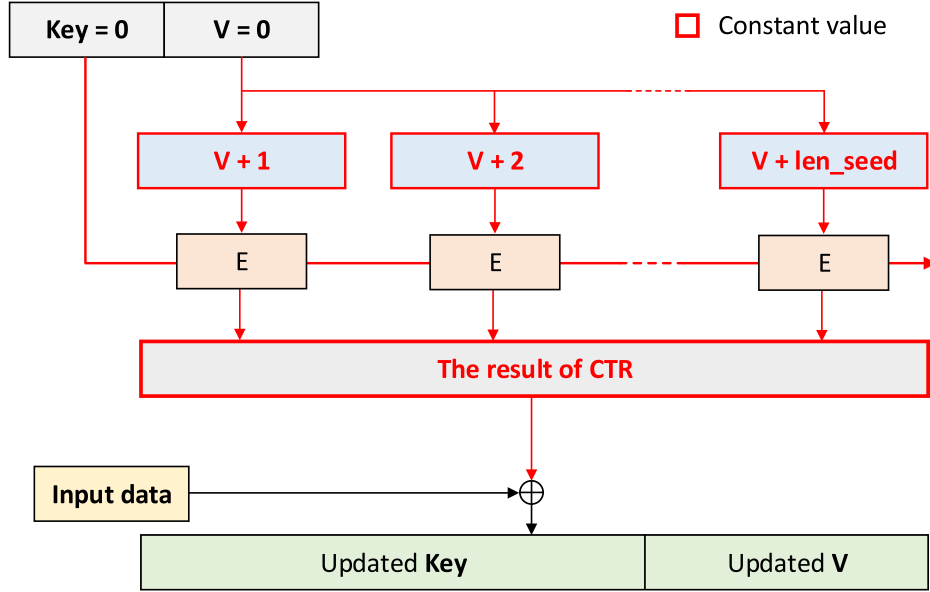

Figure 19. The part that can be constant value in the update function.

5. Implementation Results

5.1. Experiment Environment

The results implemented according to the proposed GPU optimization method were measured in the environment presented in Table 3. In this paper, we present the performance of the target block cipher algorithms and the performance of CTR_DRBG.

The optimization results for the block encryption algorithms were measured based on the time taken to encrypt data of a fixed size. Experimental results were measured while encrypting data of a minimum size of 128 MB and a maximum size of 1024 MB. The number of threads per GPU block used while performing encryption on the GPU is from 256 to 1024. The performance results including the memory copy time between the CPU and GPU are presented first, and the GPU kernel performance results without the memory copy time are presented.

In the performance results including memory copy time, the results for three types of implementations are presented. First, the parallel encryption performance in the ECB operation mode is presented, followed by the parallel encryption performance in the CTR operation mode. In addition, the performance to which the CUDA stream optimization method was applied is presented. In the ECB mode implementation, all input plaintext data were filled with random values generated by the random number generator. In the case of the CTR mode implementation, only the nonce value was filled with a random number value. In the case of the CUDA stream optimization implementation, the total plaintext data were divided by the number of streams, and encryption was performed asynchronously. The performance result was measured based on the time from when the first stream started copying data from the CPU to the GPU until the last stream copied the ciphertext from the GPU to the CPU. The performance was averaged after measuring the repeated time for a total of 100 iterations.

Table 3. GPU optimization implementation test environment.

CPU	AMD Ryzen 5 3600	
GPU	GTX 1070	RTX 2070
GPU Architecture	Pascal	Turing
GPU Core count	1920	2304
GPU Memory Size	8 GB	8 GB
GPU Base clock	1506 MHz	1410 MHz
CUDA Version	11.0	
OS	Windows 10	

5.2. Experiment Results in Block Cipher Encryption

Figures 20–22 show the performance improvement according to each optimization method for CHAM, LEA, and HIGHT. Each figure compares the performance of each algorithm, and within each figure, compares each optimization performance according to the key size of the algorithm. Figures 20–22 all show performance results including memory copy time between CPU and GPU. The performances shown in Figures 20–22 were measured on the RTX 2070. The encryption speed inside the GPU kernel also affected the encryption service provision time, but because the memory copy time load occupied a much larger portion than the kernel's computation time, the performance results including the memory copy time was directly affected for IoT or cloud computing servers.

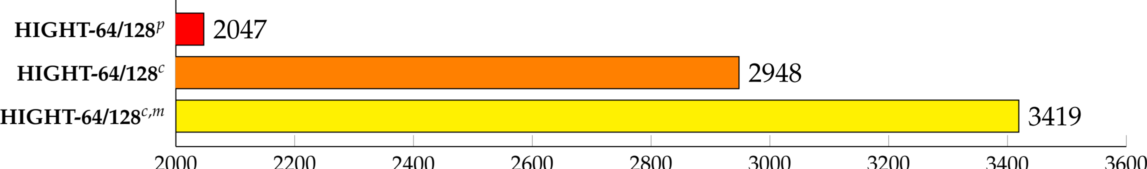

Figure 20. Comparison of throughput (MB/s) for HIGHT implementations on GPU platform including memory copy time, where p, c, and m represent parallel ECB mode of operation, parallel CTR mode of operation, and GPU memory optimization, respectively.(HIGHT-n/m means n-bit plaintext encryption with the m-bit key).

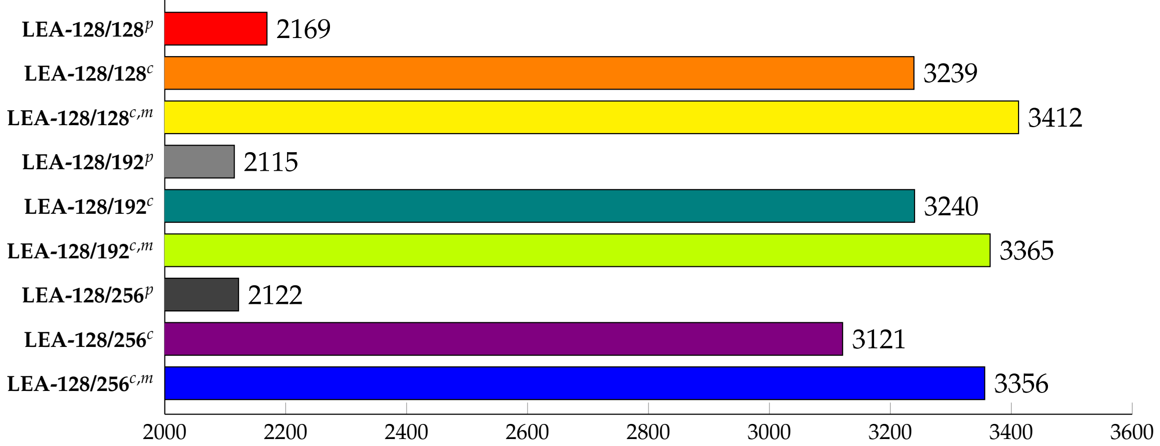

Figure 21. Comparison of throughput (MB/s) for LEA implementations on GPU platform including memory copy time, where p, c, and m represent parallel ECB mode of operation, parallel CTR mode of operation, and GPU memory optimization, respectively.(LEA-n/m means n-bit plaintext encryption with the m-bit key).

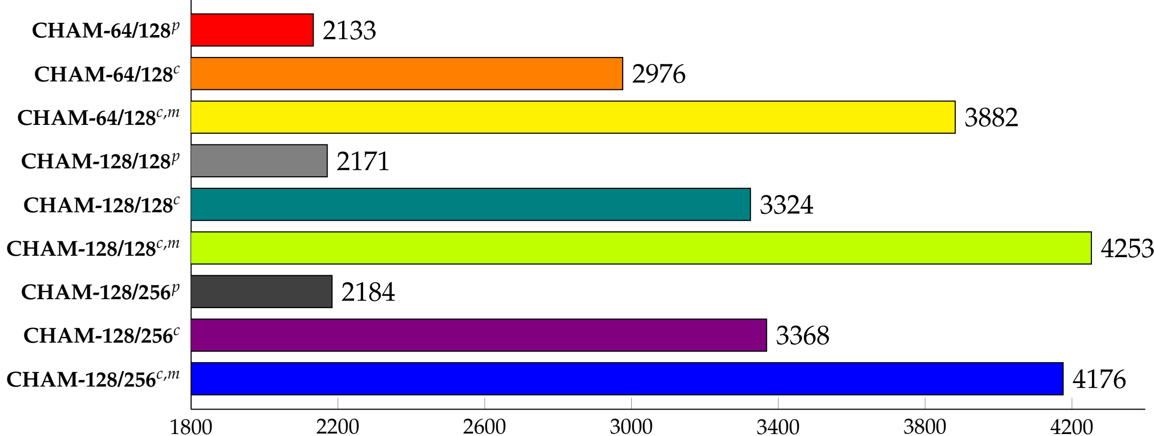

Figure 22. Comparison of throughput (MB/s) for CHAM implementations on GPU platform including memory copy time, where *p*, *c*, and *m* represent parallel Electronic CodeBook (ECB) mode of operation, parallel CTR mode of operation, and GPU memory optimization, respectively.(CHAM-n/m means n-bit plaintext encryption with the m-bit key).

5.2.1. HIGHT CTR

In the case of HIGHT shown in Figure 20, the performance of applying only the simple ECB mode parallel operation implementation was measured in 2047 MB/s for HIGHT-64/128. In addition, the performance of 2948 MB/s was confirmed in CTR mode parallel implementation. Finally, when the encryption was split asynchronously using the CUDA stream, the performance was 3419 MB/s, up to 67% performance improvement over the simple parallel implementation without optimization. The number of CUDA streams used in the experiment showed the highest performance when the maximum number was available in the GPU architecture. In RTX 2070, highest performances were shown when the number of CUDA stream was 32.

5.2.2. LEA CTR

In the case of LEA shown in Figure 21, the performance of applying only the simple ECB mode parallel operation implementation was measured in 2169, 2115, and 2122 MB/s for LEA-128/128, LEA-128/192, and LEA-128/256 respectively. In addition, when the optimization was implemented in the CTR mode that did not copy plaintext from the existing ECB operation mode, it can be seen that the performance of each increased to 3239, 3240, and 3121 MB/s. Finally, when the encryption was split asynchronously using the CUDA stream, the performance for each was 3412, 3365, and 3356 MB/s, up to 57, 59, and 58% performance improvement over the simple parallel implementation without optimization.

5.2.3. CHAM CTR

In the case of CHAM shown in Figure 22, the performance of applying only the simple ECB mode parallel operation implementation was measured in 2133, 2171, and 2184 MB/s for CHAM-64/128, CHAM-128/128, and CHAM-128/256 respectively. In addition, when the optimization was implemented in the CTR mode that does not copy plaintext from the existing ECB operation mode, it can be seen that the performance of each increased to 2976, 3324, and 3368 MB/s. Finally, when the encryption was split asynchronously using the CUDA stream, the performance for each was 3882, 4253, and 4176 MB/s, up to 81, 95, and 91% performance improvement over the simple parallel implementation without optimization.

5.3. Experiment Results in CTR_DRBG

CTR_DRBG was also performed in the same environment in which the CTR mode optimization implementation was tested. The performance of the GPU CTR_DRBG optimization implementation was measured based on when the CTR_DRBG was called and ended when the random number sequence was output. The performance measurements were measured by various experimental environment variables, and the set experimental environment variables were divided into four types: block cipher type, output random number sequence size, prediction resistance, and additional input. Prediction resistance and additional input, which had little effect on the performance measurement, was turned off in the test environment.

When the optimization was implemented, the progressing function varied depending on the predict resistance and whether additional input was performed, but the extract function, which is the main element of CTR_DRBG operating as the GPU, was not affected. Therefore, the results provided in Figure 23 measured the performance while changing the size of the random number output while the environment was fixed as one. In the performance measurement, the result of repeating the entire process 1000 times is presented as an average.

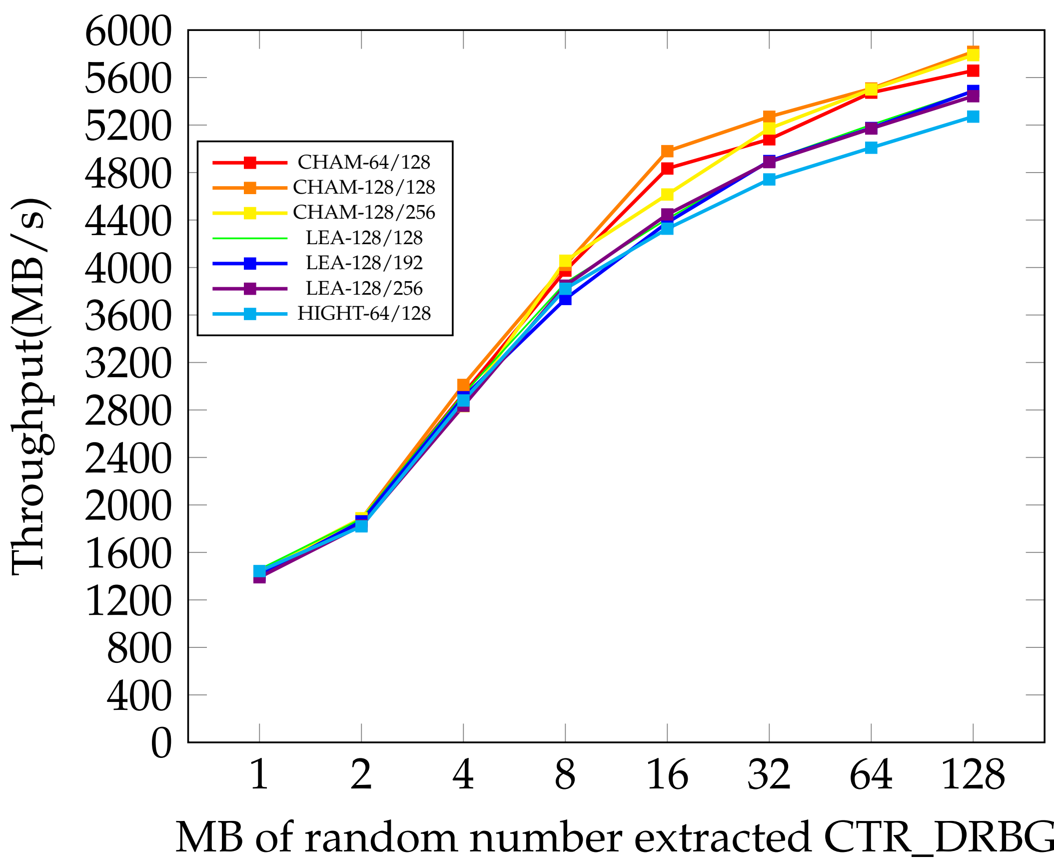

Figure 23. Comparison of throughput (MB/s) for CHAM, LEA, and HIGHT implementations on GPU platform. (cipher-n/m means n-bit plaintext encryption with the m-bit key).

Figure 23 shows the result of throughput(MB/s) for the CTR_DRBG optimization implementation on the GPU according to the random number output size. It can be seen that the performance of all the cryptographic algorithms implemented increases as the output random number increased in common. When outputting a 128 MB random number compared to the random number output size of 1 MB, it can be seen that the performance increased up to five times.

Based on the 128 MB random number output, the throughput of the CPU is 100.9, 235.8, and 169.6 MB/s respectively for HIGHT-64/128, LEA-128/128, and CHAM-128/128 and when

the random number was output through the GPU, up to 52.2, 24.8, and 34.2 times performance improvement was shown.

5.4. Comparison

Table 4 shows the GPU kernel performance for the optimized target algorithm by the number of threads and data size. The fastest performance result was obtained when the number of threads per block was 256.

Table 4. Throughput by data size and the number of threads (Gbps) in RTX 2070.

		128 MB	256 MB	512 MB	1024 MB
HIGHT	256 threads	443	468	432	393
	512 threads	437	439	419	402
	1024 threads	401	414	409	373
LEA	256 threads	2486	2593	2564	2497
	512 threads	2498	2509	2492	2504
	1024 threads	2317	2301	2325	2309
CHAM	256 threads	2897	3063	2966	2870
	512 threads	2843	2890	2887	2874
	1024 threads	2535	2558	2536	2513

Table 5 shows the kernel performance for the target algorithm implemented in the GPU. Kernel operating performance excluding memory copy time was measured by throughput (Gbps). The performance was measured in CHAM-128/128, LEA-128/128, and HIGHT-64/128, and the number of threads was fixed to 256 to properly utilize the GPU memory space. Compared to the existing CPU implementation, it was possible to obtain 445, 530, and 1856 times improved performance for each of CHAM, LEA, and HIGHT when encrypted with RTX 2070 GPU. This result showed improved performance for other studies conducted in the past, and in the case of HIGHT and CHAM, few optimization studies were conducted in the existing GPU environment.

Table 5. Comparison of encryption kernel on GPU in terms of throughput (Gbps).

Method	Platform	HIGHT	LEA	CHAM
Seo et al. [11]	GTX 680	-	139	-
Lee et al. [13]	GTX 980	-	678	-
	GTX 1080		1478	
An et al. [14]	RTX 2070	-	2472	3033
Beak et al. [16]	GTX 470	46	-	-
This work	GTX 1070	281	1576	1620
	RTX 2070	468	2593	3063
	Ryzen 5 3600(CPU)	1.05	4.89	1.65

6. Conclusions

Along with many optimization studies on cryptographic algorithms, interest in optimization studies in a GPU environment is also increasing. However, research on the optimization of lightweight block ciphers in a GPU environment has not been actively conducted yet.

In this paper, we introduced the optimization methods for the lightweight block encryption algorithms HIGHT, LEA, and CHAM in the GPU environment. Several methods of reducing unnecessary memory copying and operations are proposed by linking the characteristics of the block cipher operation mode, CTR mode, with the characteristics of the GPU. In addition, various methods to efficiently use the internal memory of the GPU are presented. Inside the GPU, we proposed

an implementation method that can more efficiently process existing operations while actively utilizing registers using inline PTX assembly codes.

In the RTX 2070 GPU environment, our implementations with HIGHT, LEA, and revised CHAM provide 445, 530, and 1,856 times of improved encryption throughput compared with our best OpenMP CPU block cipher encryption implementations. In addition, the deterministic random number generator CTR_DRBG was optimized by applying the excellent optimization results in the CTR operation mode. In the RTX 2070 GPU environment, our CTR_DRBG implementations using HIGHT, LEA, and revised CHAM provide 52.2, 24.8, and 34.2 times enhanced throughput over CTR_DRBG implementations in the CPU, respectively.

By using the various optimization methods in this paper, the CPU can perform other tasks while the GPU processes encryption in parallel. By utilizing this, the server provides basic server functions from the CPU, while in the case of tasks that require encryption, the GPU is called to perform encryption asynchronously and provide it to the user.

In the case of CTR_DRBG, a random number generator, a very large random number sequence can be output at the same time according to the optimization methods proposed in this paper. Public-key cryptography or quantum-resistant cryptography often requires a very long random number sequence, so it can be actively used in a server environment that performs encryption using a public key.

In future works, we will conduct research that can operate resources and workloads distributedly considering multiple GPUs.

Author Contributions: Investigation, S.A. and H.K.; Software, H.K. and Y.K. and S.C.S.; Supervision, S.A. and Y.K. and H.S.; Writing–original draft, S.A. and H.K. and Y.K.; Writing–review and editing, H.S. and S.C.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1F1A1058494).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Federal Information Processing Standards Publications. *197(FIPSPUBS): Announcing the ADVANCED ENCRYPTION STANDARD(AES)*; National Institute of Standards and Technology (NIST): Gaithersburg, ML, USA, 2001.
2. Hong, D.J.; Sung, J.C.; Hong, S.H.; Lim, J.G.; Lee, S.J.; Koo, B.S.; Lee, C.H.; Chang, D.H.; Lee, J.S.; Jeong, K.T.; et al. HIGHT: A new block cipher suitable for low-resource device. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Yokohama, Japan, 10–13 October 2006*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 46–59.
3. Hong, D.J.; Lee, J.K.; Kim, D.C.; Kwon, D.S.; Ryu, K.H.; Lee, D.G. LEA: A 128-bit block cipher for fast encryption on common processors. In *Proceedings of the International Workshop on Information Security Applications, Jeju Island, Korea, 19–21 August 2013*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 3–27.
4. Koo, B.W.; Roh, D.Y.; Kim, H.J.; Jung, Y.H.; Lee, D.G.; Kwon, D.S. CHAM: A family of lightweight block ciphers for resource-constrained devices. In *Proceedings of the International Conference on Information Security and Cryptology, Xi'an, China, 3–5 November 2017*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 3–25.
5. Roh, D.Y.; Koo, B.W.; Jung, Y.H.; Jeong, I.W.; Lee, D.G.; Kwon, D.S.; Kim, W.H. Revised Version of Block Cipher CHAM. In *Proceedings of the International Conference on Information Security and Cryptology, Nanjing, China, 6–8 December 2019*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 1–19.
6. Information Technology Laboratory Computer Security Division. *The NIST SP 800-90A Deterministic Random Bit Generator Validation System (DRBGVS)*; National Institute of Standards and Technology(NIST): Gaithersburg, ML, USA, 2015.
7. Seo, H.J.; Kim, H.W. Optimized implementations of HIGHT algorithm for sensor network. *J. Korea Inst. Inf. Commun. Eng.* **2011**, *15*, 1510–1516. [CrossRef]
8. Seo, H.J. High Speed Implementation of LEA on ARMv8. *J. Korea Inst. Inf. Commun. Eng.* **2017**, *21*, 1929–1934.
9. Kim, T.U.; Hong, D.J. Software Implementation of Lightweight Block Cipher CHAM for Fast Encryption. *J. Korea Soc. Comput. Inf.* **2018**, *23*, 111–117.

10. Lee, S.J.; Kang, J.Y.; Hong, D.W.; Seo, C.H. Research for Speed Improvement Method of Lightweight Block Cipher CHAM using NEON SIMD. *J. Korea Inst. Inf. Secur. Cryptol.* **2019**, *5*, 485–491. [CrossRef]
11. Seo, H.J.; Liu, Z.; Park, T.H.; Kim, H.J.; Lee, Y.C.; Choi, J.S.; Kim, H.W. Parallel Implementations of LEA. In *Proceedings of the International Conference on Information Security and Cryptology, Okinawa, Japan, 1–5 April 2013*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 256–274.
12. Park, M.K.; Yoon, J.W. Optimization of Lightweight Encryption Algorithm (LEA) using Threads and Shared Memory of GPU. *J. Korea Inst. Inf. Secur. Cryptol.* **2013**, *25*, 719–726.
13. Lee, W.K.; Goi, B.M.; Phan, R.C.-W. Terabit encryption in a second: Performance evaluation of block cipher in GPU with Kepler, Maxwell, and Pascal architectures. *Concurr. Comput. Pract. Exp.* **2018**, *31*, e5048. [CrossRef]
14. An, S.W.; Seo, S.C. Highly Efficient Implementation of Block Ciphers on Graphic Processing Units for Massively Large Data. *Appl. Sci.* **2020**, *10*, 3711. [CrossRef]
15. NVIDIA. CUDA Toolkit-Develop, Optimize and Deploy GPU-Accelerated Apps. Available online: https://docs.nvidia.com/cuda/ (accessed on 21 August 2020)
16. Baek, E.T.; Lee, M.K. Speed-optimized Implementation of HIGHT Block Cipher Algorithm. *J. Korea Inst. Inf. Secur. Cryptol.* **2012**, *22*, 495–504.

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).