applied sciences

MDPI

Article

Highly Efficient Implementation of Block Ciphers on Graphic Processing Units for Massively Large Data

SangWoo An [1,†] and Seog Chung Seo [1,2,*,†]

[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
* Correspondence: scseo@kookmin.ac.kr; Tel.: +82-02-910-4742
† These authors contributed equally to this work.

Received: 30 April 2020; Accepted: 22 May 2020; Published: 27 May 2020



Abstract: With the advent of IoT and Cloud computing service technology, the size of user data to be managed and file data to be transmitted has been significantly increased. To protect users' personal information, it is necessary to encrypt it in secure and efficient way. Since servers handling a number of clients or IoT devices have to encrypt a large amount of data without compromising service capabilities in real-time, Graphic Processing Units (GPUs) have been considered as a proper candidate for a crypto accelerator for processing a huge amount of data in this situation. In this paper, we present highly efficient implementations of block ciphers on NVIDIA GPUs (especially, Maxwell, Pascal, and Turing architectures) for environments using massively large data in IoT and Cloud computing applications. As block cipher algorithms, we choose AES, a representative standard block cipher algorithm; LEA, which was recently added in ISO/IEC 29192-2:2019 standard; and CHAM, a recently developed lightweight block cipher algorithm. To maximize the parallelism in the encryption process, we utilize Counter (CTR) mode of operation and customize it by using GPU's characteristics. We applied several optimization techniques with respect to the characteristics of GPU architecture such as kernel parallelism, memory optimization, and CUDA stream. Furthermore, we optimized each target cipher by considering the algorithmic characteristics of each cipher by implementing the core part of each cipher with handcrafted inline PTX (Parallel Thread eXecution) codes, which are virtual assembly codes in CUDA platforms. With the application of our optimization techniques, in our implementation on RTX 2070 GPU, AES and LEA show up to 310 Gbps and 2.47 Tbps of throughput, respectively, which are 10.7% and 67% improved compared with the 279.86 Gbps and 1.47 Tbps of the previous best result. In the case of CHAM, this is the first optimized implementation on GPUs and it achieves 3.03 Tbps of throughput on RTX 2070 GPU.

Keywords: AES; CHAM; LEA; Graphic Processing Unit (GPU); CUDA; Counter (CTR) mode; Parallel Processing

1. Introduction

With the development of IoT and Cloud computing service technology, the amount of data produced by many users and growing IT devices has also increased significantly. Thus, servers have to provide stable services for the greatly increased data without compromising their services. In addition, as the importance of information protection has emerged, encryption is considered as being essential to protect the privacy of users' data. Thus, servers have to be able to rapidly encrypt users' data and transmitted files. In other words, servers require a technology capable of encrypting a large amount of data while maintaining their own service smoothly in real-time. A server that manages IoT devices receives and transmits data from a number of IoT devices. To protect the data transmitted during

this process, massively-large data, from MB to GB, must be quickly encrypted in real-time on the server. The amount of data that the server needs to encrypt in real-time will increase over time, and, according to these demands, continuous optimization research on the encryption algorithm on the Graphic Processing Unit (GPU) is essential. For example, in 2017, Amazon corp. provided Amazon EC2 P3 instances powered by NVIDIA Tesla V100 GPUs for cloud computing services.

Since using only CPU for encrypting a large amount of data has a limitation, high-speed encryption technologies have been required. GPU has been considered as a promising candidate for a crypto accelerator for encrypting a huge amount of data in this situation. Unlike the latest CPUs that have dozens of cores, GPUs have hundreds to thousands of cores, making them suitable for computation-intensive works such as cryptographic operations.

Block cipher algorithms such as AES [1], CHAM [2], and LEA [3] have been optimized in the GPU environment [4–12]. In the case of AES, the possibility of parallel encryption using GPU CUDA has been presented [4]. Since then, a method of implementing GPU memory optimization using shared memory has been proposed [5,6]. AES has also been studied various optimization implementations such as bitslice technique depending on the GPU architecture [7,8]. Recently, an optimization study was conducted to reduce GPU memory copy time by using CTR operation mode [9]. LEA also has optimized research using GPU parallel encryption process [10]. Research has also been proposed to optimize AES and LEA through fine-grain and coarse-grain methods [11,12]. However, several optimization studies have not considered the characteristics of the GPU architecture and the features of the target block cipher algorithms at the same time.

We present optimized implementations together from two perspectives: GPU architecture and block cipher algorithm. We optimize the target algorithms AES, CHAM, and LEA in GTX 970 (Maxwell), GTX 1070 (Pascal), and RTX 2070 (Turing) GPU architectures, respectively. Although lightweight block ciphers such as CHAM and LEA were originally proposed for use in constrained environments such as IoT, the server must use the same cipher to encrypt or decrypt the data being sent and received from the IoT device. Therefore, optimizing the lightweight block cipher algorithm on the GPU needs be studied considering the IoT server environment. Recently, since devices that communicate through the network are also becoming smaller, we present the possibility of applying large-capacity data encryption by optimizing computation-based cipher algorithms that do not use tables. We encrypted data up to 1024 MB, which is the size of massively-large data transmitted by users. In addition, we compared the performance according to the capacity difference while increasing the data size from at least 128 MB.

The contributions of our paper can be summarized as follows.

1. **Proposing several optimization techniques for block ciphers on parallel GPU environment**
   We propose several optimization techniques for AES, LEA, and CHAM in NVIDIA CUDA GPU platforms. First, we introduce a method that can significantly reduce the memory copy time between the CPU and GPU by customizing CTR mode on the GPUs. Next, we handle various memory issues that cause a slowdown in the GPU environment. Finally, we present an additional method to efficiently manage the GPU kernel.
2. **Maximizing throughput of block ciphers on various GPU architectures**
   We implemented the core part of each block ciphers with handcrafted inline PTX codes. By optimizing the core part with inline PTX, we could control more closely the registers in GPU, which led to a huge performance improvement. When all operations were implemented with inline PTX, excellent performance throughput was achieved: 3.03 Tbps and 2.47 Tbps in CHAM and LEA. This performance is 598% and 548% faster in CHAM and LEA, respectively, compared to the previously implemented algorithm without optimization.
3. **Achieving excellent performance improvement compared with previous implementations**
   We measured the performance of proposed implementations on various GPU architecture: GTX 970 (Maxwell), GTX 1070 (Pascal), and RTX 2070 (Turing). In the order of AES, CHAM and LEA, GTX 970 showed 123 Gbps, 936 Gbps, and 868 Gbps, respectively; GTX 1070 showed 214 Gbps, 1.67 Tbps, and 1.54 Tbps, respectively; and RTX 2070 showed 310 Gbps, 3.03 Tbps, and 2.47 Tbps,

respectively. In particular, the proposed optimization methods showed an effective improvement in a lightweight block encryption algorithm composed of simple operations. To the best of our knowledge, CHAM has never been optimized for GPU environments. The best implementation of CHAM in this paper can encrypt terabits of data per second.

The composition of this paper is as follows. Section 2 introduces related optimization studies on GPUs thus far. Section 3 briefly reviews the target algorithms and overview of GPU architecture. Section 4 describes the proposed optimization techniques including efficient parallel encryption, optimizing GPU memory usage, pipeline encryption methods, and application of inline PTX assembly codes. Section 5 presents the implementation results. The performance difference according to the data size and the number of threads is presented in a table, and the results including memory copy time are further examined. Finally, Section 6 concludes this paper with suggestions for our future works.

2. Related Works

Many optimization studies have been conducted for optimizing AES thus far. As non-graphic operation becomes possible using GPU CUDA, research has been carried out to implement AES on GPU using CUDA [4]. As the possibility of AES research in the GPU environment was suggested, an implementation study was also conducted in which the tables used in AES are copied to shared memory [5]. Later, there was a study of optimizing AES on Kepler GPU architecture using the micro-benchmark analysis method [6]. Recently, a bitslice method that encrypts AES with simple computations advantageous to GPUs has been studied while excluding the use of tables with a large computational load [7,8]. As new GPU architectures, Maxwell and Pascal have been developed; some studies optimized AES in the environment [9]. Author1 [9] presented an optimization method for memory copy between CPU and GPU as well as the implementation of parallel encryption. As with AES, there is a study that LEA has been optimized through parallel encryption in the GPU environment [10]. In addition, studies on parallel optimization of AES and LEA using fine-grain and coarse-grain implementation techniques have been conducted [11,12].

Although CHAM and LEA are designed with a lightweight block cipher algorithm that operates on small IoT devices, the characteristics of using simple bit operation as the main encryption operation have excellent computational efficiency in GPU. Since CHAM is a recently proposed cipher algorithm, its performance was not extensively studied including GPU environments. Although some optimization studies have been conducted in the microcontroller environment [13,14], which is the operating environment of the lightweight cryptographic algorithm, there is no prior implementation of CHAM in the GPU environment as far as we know. Similar to CHAM, most of the studies conducted on LEA were optimization implementation studies in an embedded environment [10], and few studies were conducted on GPUs [12,15].

Table 1 summarizes the results of studies conducted on AES, CHAM, and LEA.

Table 1. Throughput (Gbps) table with existing implementation.

	Platform	AES	CHAM	LEA
N. Nishikawa et al. [6]	GTX 680	63.9	-	-
	Radeon HD 7970	205.8	-	-
N. Nishikawa et al. [7]	Tesla P100-PCIe	605.9	-	-
R.K. Lim et al. [8]	GTX 480	78.6	-	-
A.A. Abdelrahman et al. [9]	GTX 780	80.52	-	-
	GTX TITAN X	207.25	-	-
	GTX 1080	279.86	-	-
H. Seo et al. [10]	GTX 680	-	-	138.8
	K40c	-	-	260.28 (scaled)
W.K. Lee et al. [11]	GTX 980	149.4	-	-
W.K. Lee et al. [12]	K40c	79	-	368
	GTX 980	154	-	678
	GTX 1080	256	-	1,478
T. Kim et al. [13]	i5 Skylake@2.3GHz (CPU)	-	1.33	-

3. Background

This section provides information on how AES [1], CHAM [2], and LEA [3] block ciphers work and features of GPUs. Among the features of the GPU, points to be considered during implementation in the GPU environment are also described.

3.1. Notations

Table 2 defines notations used in this paper. The contents of the notation describe variables and operations used in the encryption process of the proposed algorithm.

Table 2. Notations.

Symbol	Meaning
$X \oplus Y$	Bit-wise exclusive OR operation
$X \boxplus Y$	$X + Y \bmod 2^{32}$
$X * Y$	Multiply X and Y in $GF(2^8)$ in AES
e_j	jth columns in AES *block*
x_i, y_i, z_i, w_i	32-bit word in $i - th$ rounds of CHAM/LEA *block*
$x \lll i$	Rotating x left-side as i bits
$x \ggg i$	Rotating x left-side as i bits

3.2. Overview of Target Algorithms

Block cipher algorithms generally are based on different structures and the number of rounds depending on the key length. Table 3 shows the number of rounds according to the key length of the encryption algorithm. To match the same experimental environment, we set the plaintext and key size of AES, CHAM, and LEA as 128 bits.

Table 3. Table of rounds according to key length.

Algorithm	Rounds	Key Length (bit)
AES-128	10	128
AES-196	12	196
AES-256	14	256
CHAM-64/128	88	128
CHAM-128/128	112	128
CHAM-128/256	120	256
LEA-128	24	128
LEA-196	28	196
LEA-256	32	256

3.2.1. AES

The AES block cipher algorithm is based on the SPN structure and has a block length of 128 bits. The key lengths used for AES are 128, 196, and 256 bits, and the number of rounds according to the key length is 10, 12, and 14 rounds. Each round consists of AddRoundKey, SubBytes, ShiftRows, and MixColumns functions. In the last round, there is no MixColumns function involved. The result of grouping SubBytes, ShiftRows, and MixColums and tabulating the results is called T-table and has shown significant performance improvements [16]. The T-table consists of four tables and each table consists of 256 32-bit values. Encryption is performed while referring to the table value 16 times in a round. In this paper, we implement the AES version using T-table. The creation of T-table and encryption operation using T-table are as follows.

$$T_0[a] = \begin{bmatrix} Sbox[a] * 02 \\ Sbox[a] \\ Sbox[a] \\ Sbox[a] * 03 \end{bmatrix} T_1[a] = \begin{bmatrix} Sbox[a] * 03 \\ Sbox[a] * 02 \\ Sbox[a] \\ Sbox[a] \end{bmatrix} T_2[a] = \begin{bmatrix} Sbox[a] \\ Sbox[a] * 03 \\ Sbox[a] * 02 \\ Sbox[a] \end{bmatrix} T_3[a] = \begin{bmatrix} Sbox[a] \\ Sbox[a] \\ Sbox[a] * 03 \\ Sbox[a] * 02 \end{bmatrix} \quad (1)$$

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-C_1}] \oplus T_2[a_{2,j-C_2}] \oplus T_3[a_{3,j-C_3}] \quad (2)$$

3.2.2. CHAM

CHAM is a lightweight block cipher algorithm based on the Feistel structure [2]. CHAM is divided into three types according to the length of the plaintext and the length of the key, CHAM-64/128, CHAM-128/128, and CHAM-128/256. In CHAM-n/m, n means the length of the plaintext and m means the key length. The length of the plaintext can be 64 and 128 bits, and the key length can be 128 and 256 bits. The first suggested number of rounds per block and key length of CHAM was 80, 80, and 96 rounds, but, to prevent the threat of related-key differential attacks, in the revised version of CHAM presented in 2019 [2], the number of rounds per block and key length was revised to 88, 112, and 120 rounds. Unlike AES, CHAM does not use a table, and it encrypts plaintext using only `XOR`, `ADD`, and `ROTATE` operations. One plaintext block consists of four words, each word size being 16 bits when the plaintext length is 64 bits and 32 bits when the plain text length is 128 bits. The encryption process of CHAM is shown in Figure 1. The block divided into four words, and words are rotated for each round. At this time, the number of bits in the bit rotation differs depending on round i is even or odd.

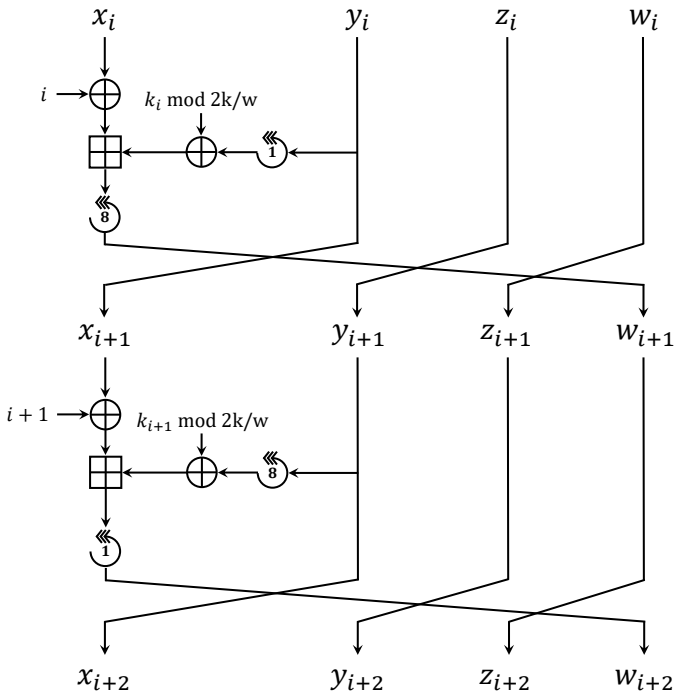


Figure 1. Two consecutive rounds beginning with the even *i*th round in CHAM.

3.2.3. LEA

LEA is a lightweight block cipher algorithm developed considering high-speed environments such as big data and cloud computing services and lightweight environments such as mobile devices [3]. LEA has a fixed 128-bit plaintext block size, and the key length is 128, 196, and 256 bits. LEA has a feature that does not use S-box, and the round function consists of simple operations in 32-bit units. The number of rounds is 24, 28, and 32 for 128-, 196-, and 256-bit keys. The main operations are `XOR`, `ADD`, and `ROTATE`. Unlike CHAM using only left-rotates, LEA uses both right rotate and left rotate operations. Six 32-bit round keys are used for encryption per round. Figure 2 shows the encryption process of LEA.

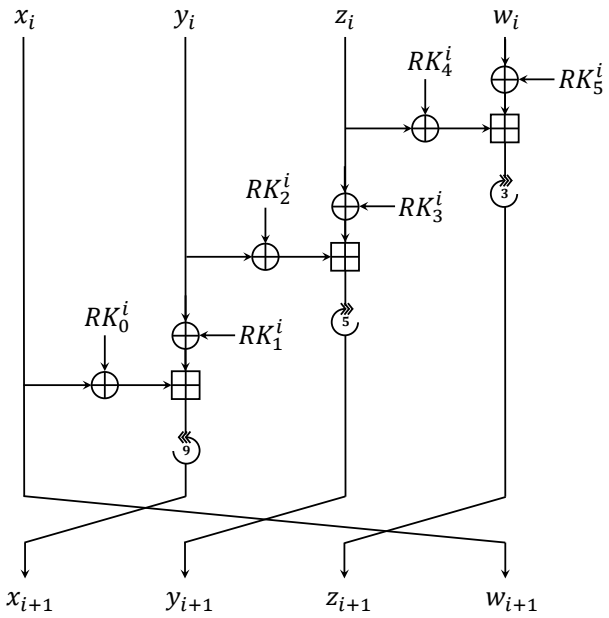


Figure 2. Round function of LEA.

3.2.4. Counter Modes of Operation

The block cipher operation method refers to a procedure that repeatedly and safely uses a block cipher in one key. Since block ciphers operate in blocks of a specific length, to encrypt variable-length data, they must first be divided into unit blocks. The method used to encrypt the blocks is called an operation method. There are various block cipher operation methods, and typical block cipher operation methods are ECB, CBC, and CTR modes. The ECB operation mode has the simplest structure, and the message to be encrypted is divided into several blocks to encrypt each. In the CBC operation mode, each block is `XORed` with the previous block's encryption result before being encrypted, and, in the case of the first block, an initialization vector is used. In CTR operation mode, the counter value for each block is combined with nonce and used as the input of the block cipher. The encrypted input is `XORed` with plaintext. The structure of the CTR operation mode can be seen in Figure 3. In the case of CTR mode, since the plaintext is not encrypted, it is `XORed` with the encrypted counter value to make the ciphertext, thus using this feature enables efficient encryption on the GPU. In addition, due to the nature of CTR mode, encryption can be performed in parallel for each block, making it easy to implement on GPUs. Therefore, many studies have been conducted to operate the block cipher in the CTR mode, and it is utilized in various fields using the cipher, such as CTR_DRBG and Galois/Counter mode.

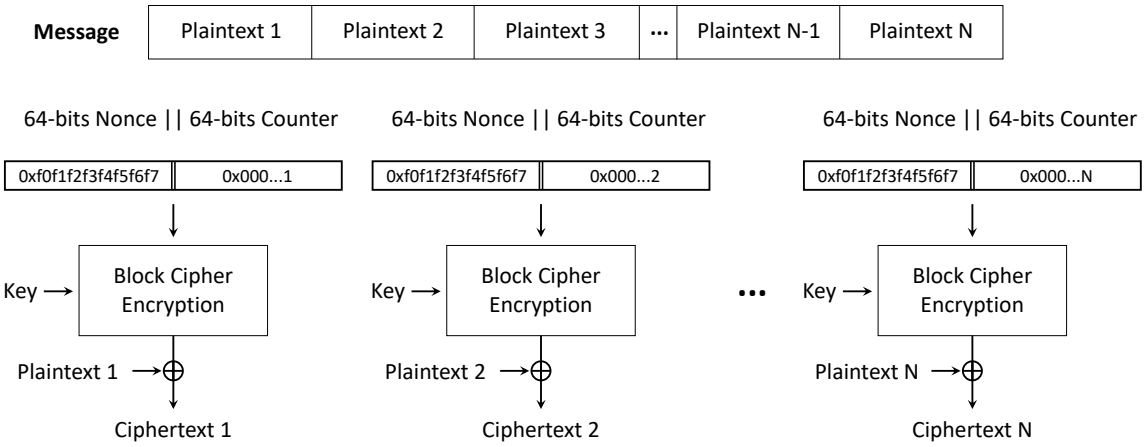


Figure 3. CTR operation mode structure.

3.3. GPU Architecture Overview

As the size of the data to be processed increases, the CPU has a limit to process such data quickly. As such, expectations for a GPU capable of parallel computing are gradually increasing. As GPUs are used for General-Purpose computing on Graphics Processing Units (GPGPU), cryptography can also be used to perform computations. To accurately and quickly process calculations using these GPUs, users need to know more about the structure and operation of GPU.

To perform NVIDIA GPU programming, data must be copied and operated using an instruction set such as the CUDA library. CUDA supports different versions for each GPU architecture, thus more efficient computation methods can be used in the latest architecture. By using CUDA, GPUs that were used only in a limited range can be utilized for efficient computation processing of large-scale data. CUDA can be written using industry-standard languages such as C so that developers who do not know the graphic API through CUDA can utilize the GPU. A function called kernel works through CUDA instructions, and the GPU can be operated through kernel.

The GPU consists of a number of Streaming Multiprocessors (SM), most of which are composed of ALU (Arithmetic Logic Units). While the CPU uses a large portion of the chip area as a cache to use a small number of high-performance cores, the GPU uses most of the area for ALU. The GPU uses threads to use the ALU, and utilizes a huge number of threads to perform independent parallel

processing. Even in the case of encryption, excellent encryption performance can be expected because each thread performs encryption in parallel.

The GPU consists of a grid, the grid consists of many blocks, and each block consist of a number of threads. The threads enable parallel computation of the GPU by performing independent operations. Inside the GPU, there is a variety of memory used by these threads, including registers used by each thread and shared memory independent of each block, and global memory shared by all threads in the GPU. Global memory is the largest memory on the GPU, but global memory has a disadvantage that it takes a lot of time to access, so using shared memory can make use of memory faster. Shared memory is shared by all threads in a block, and each block is independent. When using shared memory, the data of global memory must first be uploaded to shared memory as a cache, and then, when each thread copies data to shared memory, all threads must complete the copying through synchronization and then perform the next operation. This is because there is no problem of getting the garbage value by referring to the memory value in the state that is not all raised.

One thing to watch out for when using shared memory is to avoid bank conflicts. A bank conflict occurs when threads in a block access the same memory bank. Shared memory consists of a certain number of banks. If multiple threads try to access the same bank at the same time, requests for memory access are serialized, and waiting threads are put on hold. Figure 4 shows a case where a bank conflict occurs when multiple threads access a memory bank at the same time.

In addition to memory considerations, there are structural features of GPU that need to be taken into account when programming GPUs, to prevent warp divergence. The threads in the block are grouped by 32 to form a warp, and the warp is a unit for executing operation instructions. All threads belonging to warp perform the same operation. If threads perform different operations due to conditional statements, other threads in the warp are idle until the operation of the thread ends. In Figure 5, it can be seen that the threads in the warp cause warp divergence when different operations are performed depending on conditions.

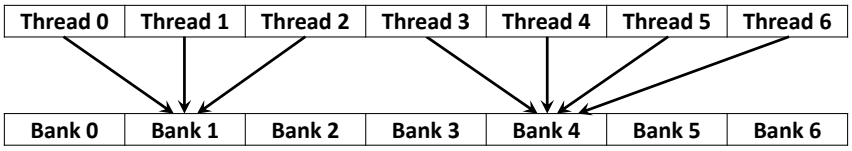


Figure 4. Bank conflict conditions.

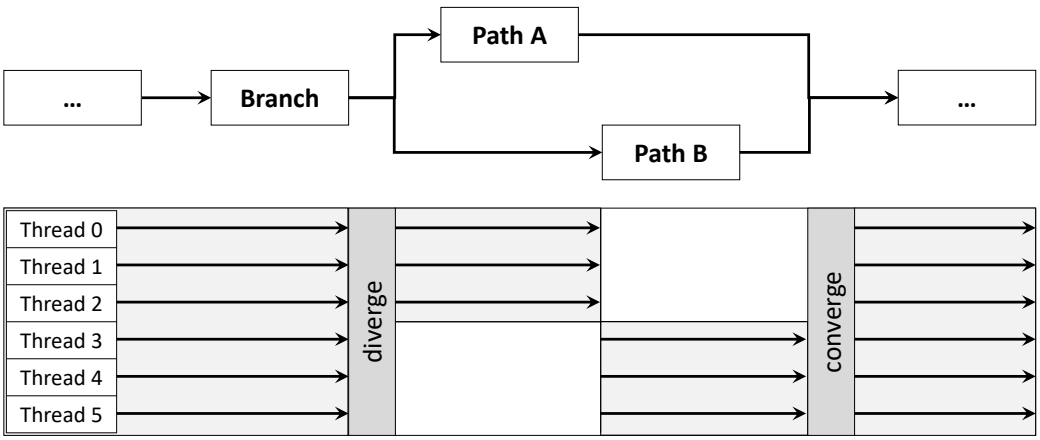


Figure 5. Warp divergence conditions.

3.4. Target GPU Architecture

For the evaluation of block cipher performance on various architecture, we choose NVIDIA GTX 970, GTX 1070, and RTX 2070 GPU. Table 4 shows the differences between the three GPUs. The GTX 970 is Maxwell architecture, supports 4 GB GDDR5 memory and CUDA compute capability 5.2. The GTX 1080 is Pascal architecture, supports GDDR5 memory and CUDA compute capability 6.1. The RTX2070 is a Turing architecture, the base clock is lower than the GTX1070, but, it has more cores, especially the Tensor core and RT core, which are not found in the Pascal architecture, allowing much faster computation. Turing architecture also supports GDDR6 memory and CUDA compute capability 7.5, showing better performance.

Table 4. Target CUDA-enabled NVIDIA GPUs.

GPU	GTX 970	GTX 1070	RTX 2070
Architecture	Maxwell	Pascal	Turing
SM count	13	15	36
Core count	1664	1920	2304
Memory Size	4 GB	8 GB	8 GB
Base clock	1050 MHz	1506 MHz	1410 MHz
CUDA	5.2	6.1	7.5

4. Proposed Implementation Techniques

In this section, we describe various optimization methods that have been attempted in this study for the cipher algorithm described above. The proposed optimization method was applied to all of AES, CHAM, and LEA. Other optimization methods for each algorithm are also described.

4.1. Parallel Encryption in CTR Mode

To encrypt plain text on the GPU, we must first copy the plain text data on the CPU to the GPU. Data copy between CPU and GPU is done through PCIe. However, the speed of each CPU and GPU is very fast, but PCIe is slower than both. Thus, it takes a significant amount of time to copy data between the CPU and GPU.

However, if we use the CTR mode of the block cipher operation mode, there is no need to copy the plain text to the GPU. All we have to do is copy the encrypted counter value generated by the GPU to the CPU and `XOR` it with the plain text. Threads create counter value using each thread's unique number and perform encryption in parallel. This optimization idea greatly improves performance by significantly reducing the copy time between the CPU and GPU. Algorithm 1 shows that CTR mode encryption is performed without passing plaintext as an argument to the encryption function.

Algorithm 1 CTR parallel encryption.

Require: Original Key
Ensure: Result 128-bit Ciphertext C
1: Generate *roundkey* with Original Key
2: Generate a 64-bit random value *nonce*
3: Memory copy round key and nonce from CPU to GPU
4: Compute $C \leftarrow Encryption_kernel(roundkey, nonce)$
5: Memory copy C from GPU to CPU
6: Compute $C \leftarrow C \oplus plaintext$
7: $(Return\ C)$

Figure 6 illustrates the process of encryption on the GPU using CTR mode. When encrypting on the GPU, it can be seen that only the round key and nonce are used on the GPU.

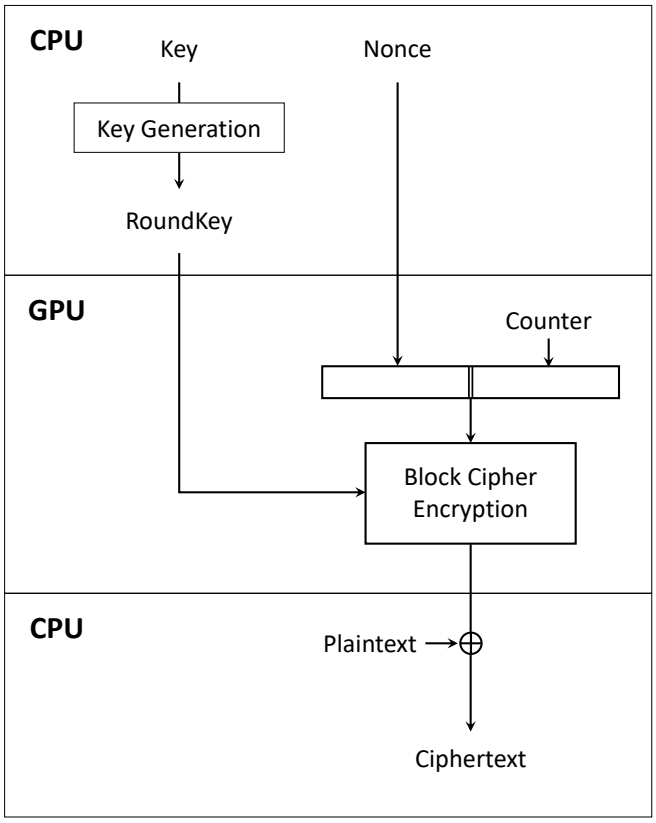


Figure 6. Block cipher counter modes of operation in GPU platform.

4.2. Optimizing GPU Memory Usage

4.2.1. Common Memory Management in GPU Kernel

Numerous threads inside the GPU try to access a number of memory. In this environment, if there is a problem with memory access, it cannot perform well. There are a few memory issues to consider among GPU programming and some suggestions for how to avoid them.

We first used the frequently used round key in shared memory. Because shared memory has faster access latency than global memory, better performance can be achieved by using shared memory. Since shared memory is shared among the threads in the block, threads in the block are implemented to use the round key or table value in shared memory when performing encryption. At this time, the bank size was adjusted with the CUDA command to reduce access to the same memory bank by different threads. In addition to prevent warp divergence, we implemented that one thread encrypts one 128 bits block. Each thread is designed to encrypt the counter value by accessing each independent memory through its own independent number.

When accessing data, rather than reading the data one by one, the memory in chunk units is accessed to read the values in the chunks. It is the fastest when we can find all the data we want to use by accessing the chunk once. However, if the data are far from each other, multiple chunk accesses are required. This causes a significant speed drop. The coalesced memory access and otherwise can be seen in Figure 7. To access the memory in a coalesced way, each thread must consider the data to be loaded, and when reading a chunk of data, the data of the chunk read should be available to as many threads as possible. Therefore, when reading table values or round keys that are frequently used in cipher operations, the threads in the warp fetch values near each other according to the warp, which is a bundle of multiple threads. This consideration also applies to data copying between the CPU and GPU. It is implemented to copy the ciphertext inside the GPU to the CPU as efficiently as possible using CUDA instructions.

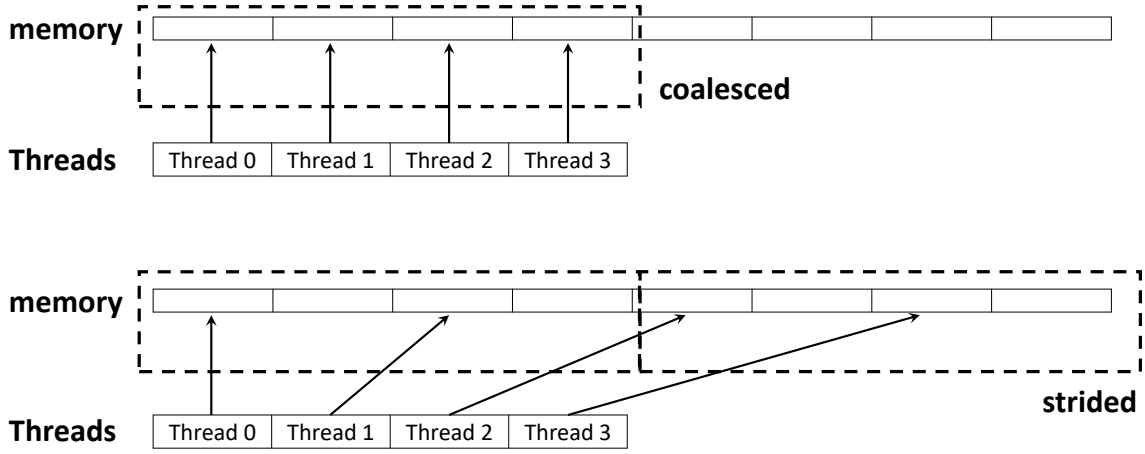


Figure 7. Memory coalesced and strided.

4.2.2. Memory Optimization Technique in AES

In the case of AES, to store the T-table in shared memory, each thread is implemented to copy the table value at the location corresponding to each id. Since the table value copied to shared memory can be shared by all threads in the block, we implemented threads to copy global memory efficiently to shared memory using warp. After copying the table, the threads were synchronized to prevent an accidental progression of encryption before the copy was completed. Plaintext and round keys are implemented to be allocated to registers while avoiding being stored in local memory. Shared memory space was allocated to the table as much as possible.

4.2.3. Memory Optimization Technique in CHAM and LEA

In the case of CHAM, unlike AES, there is no table to copy; thus, when copying from CPU to GPU, only round key was copied to the GPU, and encryption was started. Because nonce is a random constant, only the constant value is transferred, not copied to the GPU. AES requires 44 32-bit round keys to encrypt, but CHAM has only 8 32-bit round keys required when using 128-bit plaintext and 128-bit key length. Eight round keys are used up to 112 rounds repeatedly, thus we can use them efficiently by assigning them to registers. By using the small round key size of CHAM, registers can be used as efficiently as possible.

Since LEA does not use tables, efficient memory optimization is possible. Similar to CHAM, plaintext is stored in a register and used. However, unlike CHAM, the round key of LEA requires 144 32-bit values, so the round key is stored and used in shared memory. When copying the round key, as in AES, each thread in the warp is implemented to efficiently copy a contiguous memory area. LEA also synchronized after completing the round key copy.

4.3. Pipelining Encryption with CUDA Stream

For encryption on the GPU, data are first copied from the CPU to the GPU, and while encrypting on the GPU, the CPU is set to idle by default and waits for GPU kernel operations to complete. To operates CPU and GPU efficiently, the CUDA stream can be used to implement memory copy and encryption in parallel. By using the CUDA stream, GPUs are used to copy and encrypt memory in parallel for multiple streams. Therefore, if the data are divided by the number of streams to perform copying and encryption asynchronously, the CPU latency can be reduced to improve performance. Figure 8 shows the time improvement through the asynchronous execution.

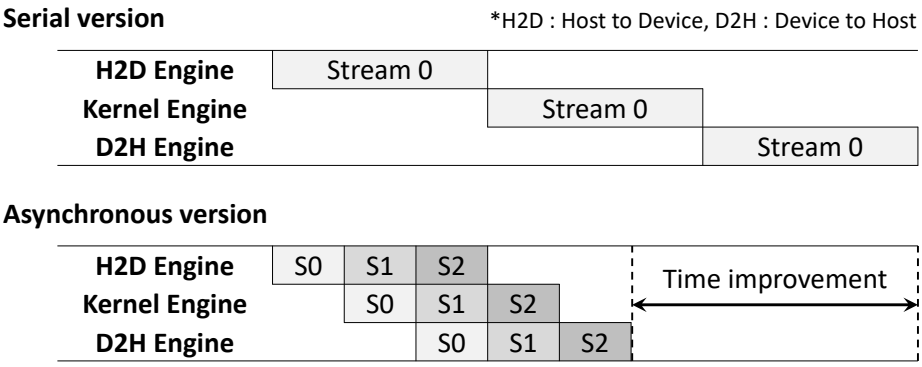


Figure 8. Asynchronous execution.

4.4. Application of Inline PTX Assembly Codes

Assembly language is a programming language that is one level up from machine language and belongs to low level languages along with machine language. Writing code in the C language is a high-level language, and assembly language is located between the machine language and the high-level language, making it easier to understand machine languages that are difficult to understand. Because assembly language has a small binary size and the program is fast, we can also consider how to write existing code in assembly language when trying to optimize. An inline assembly technique was developed to use common code and assembly language together, and inline assembly can now be used in GPU, called inline PTX. The functions used in inline PTX are similar in form and usage to functions used in inline assembly.

Here is how to use inline PTX function using CUDA library in C programming language. We can specify the use of inline PTX by using the *asm*() command at the place where inline PTX is used in the general coding area. The inline PTX code declared as *asm*() is divided into three areas: the instruction part, the input part, and the output part. The instruction part written with several functions such as *xor*.*b*32 is executed by selecting the input register and the output register for the command. The *n*% to the right of the instruction means a register, and the input part and output part under the instruction part are combined and pointed in order from 0%. For example, "+r" (x) at the bottom of Figure 9 becomes 0%, "+r" (y) is 1%, "+r" (z) is 2%, and "+r" (w) becomes 3%, which points to the output register. The following variables "r" (x), "r" (y), ..., "r" (rk[7]) become 4%, 5%, ..., 15%, respectively, which point to the input register. Each output register and input register is called through an inline PTX function to perform the operation.

```
asm("{\n\t"
      " .reg.u32          t0,t1,t2,t3,t4,t5;  \n\t"  # register t0,t1,t2,t3,t4,t5
      " shf.l.clamp.b32   t0, %5, %5, 0x1;    \n\t"  # t0 = y <<< 1
      " xor.b32           t1, t0, %8;         \n\t"  # t1 = t0 ⊕ rk[0]
      " xor.b32           t2, %4, 0x0;        \n\t"  # t2 = x ⊕ 0
      " add.cc.u32        t3, t1, t2;         \n\t"  # t3 = t1 + t2 mod 2^32
      " shf.l.clamp.b32   t4, t3, t3, 0x8;    \n\t"  # t4 = t3 <<< 8
      " shf.l.clamp.b32   t0, %6, %6, 0x8;    \n\t"  # t0 = z <<< 8
      " xor.b32           t1, t0, %9;         \n\t"  # t1 = t0 ⊕ rk[1]
      " xor.b32           t2, %5, 0x1;        \n\t"  # t2 = y ⊕ 1
      " add.cc.u32        t3, t1, t2;         \n\t"  # t3 = t1 + t2 mod 2^32
      " shf.l.clamp.b32   t5, t3, t3, 0x1;    }\n\t" # t5 = t3 <<< 1
      : "+r"(x), "+r"(y), "+r"(z), "+r"(w)           # output x,y,z,w
      : "r"(x), "r"(y), "r"(z), "r"(w),              # input x,y,z,w,rk[0],...,rk[7]
      "r"(rk[0]), "r"(rk[1]), "r"(rk[2]), "r"(rk[3]), # rk(= roundkey)
      "r"(rk[4]), "r"(rk[5]), "r"(rk[6]), "r"(rk[7])
);
```

Figure 9. Inline PTX code of CHAM round function.

In this study, encryption operation was performed using inline PTX. We implemented the optimization to store counter value in a register through inline PTX and reduce unnecessary operations using assembly language. For example, one of the main operations of CHAM and LEA, the ROTATE operation was originally a function consisting of two-bit SHIFT operations, one bit OR operation, and one minus operation, but it was implemented to be processed in a register with one instruction using inline PTX.

In addition, in the case of CHAM and LEA that do not use a table, it is possible to obtain very fast performance by moving all values necessary for encryption to a register and then performing simple operations only. Moreover, since the word rotate process existing in the CHAM encryption operation can be performed by changing only the parameters of the corresponding data register, the number of operations required for encryption can be further reduced. Figures 9 and 10 show the inline PTX code of the CHAM and LEA round function implemented in the actual C language. In the figure, only the first two rounds in CHAM and the first round in LEA are counted, but, in the actual implementation, the code up to 112/24 rounds operates as a single inline PTX function without repetition. Normally, the output is read-only, written as $''=r''$, or $''+r''$ as read-write. Looking at the inline PTX operation, *reg* is a register variable declaration, and *shf.l.clamp*, *shf.r.clamp* is a bit ROTATE. *xor.b*32 is 32-bit XOR operation, add.cc.u32 is 32-bit ADD operation. Lightweight ciphers such as CHAM and LEA can be encrypted with only three commands in inline PTX because the operation consists of XOR, ADD, and ROTATE.

```
asm("{\n\t"
      " xor.b32            %2, %6, %12;       \n\t"  # Y = Y ⊕ rk[i + 4]
      " xor.b32            %3, %7, %13;       \n\t"  # Z = Z ⊕ rk[i + 5]
      " add.cc.u32         %3, %6, %7;        \n\t"  # Z = Y + Z mod 2^32
      " shf.r.clamp.b32    %3, %7, %7, 0x3;   \n\t"  # Z = Z <<< 3
      " xor.b32            %1, %5, %10;       \n\t"  # X = X ⊕ rk[i + 2]
      " xor.b32            %2, %6, %11;       \n\t"  # Y = Y ⊕ rk[i + 3]
      " add.cc.u32         %2, %5, %6;        \n\t"  # Y = X + Y mod 2^32
      " shf.r.clamp.b32    %2, %6, %6, 0x5;   \n\t"  # Y = Y >>> 5
      " xor.b32            %0, %4, %8;        \n\t"  # W = W ⊕ rk[i]
      " xor.b32            %1, %5, %9;        \n\t"  # X = X ⊕ rk[i + 1]
      " add.cc.u32         %1, %4, %5;        \n\t"  # X = W + X mod 2^32
      " shf.l.clamp.b32    %1, %5, %5, 0x9;   \n\t"  # X = X <<< 9
      " xor.b32            %3, %7, %14;       \n\t"  # Z = Z ⊕ rk[i + 10]
      " xor.b32            %0, %4, %15;       \n\t"  # W = W ⊕ rk[i + 11]
      " add.cc.u32         %0, %7, %4;        \n\t"  # W = Z + W mod 2^32
      " shf.r.clamp.b32    %0, %4, %4, 0x3;  }\n\t"  # W = W >>> 3
      : "+r"(W), "+r"(X), "+r"(Y), "+r"(Z)           # output W,X,Y,Z
      : "r"(W), "r"(X), "r"(Y), "r"(Z),              # input W,X,Y,Z,rk[i],...,rk[i+11]
      "r"(rk[i]), "r"(rk[i + 1]), "r"(rk[i + 2]), "r"(rk[i + 3]),    # rk(= roundkey)
      "r"(rk[i + 4]),"r"(rk[i + 5]),"r"(rk[i + 10]), "r"(rk[i + 11])
);
```

Figure 10. Inline PTX code of LEA round function.

5. Implementation Results

This section presents the results for each optimization method, introduces performance improvement according to each optimization, and explains differences between data and number of threads.

5.1. Experiment Environment

The experimental environment used for performance measurement is shown in Table 5. The performance measurement was measured several times according to the number of threads and data size, and the throughput value was set as the average measurement time through 10 iterations. In the case of inline PTX, the available operation differs according to the GPU architecture, and the

computations available above the Kepler architecture were mainly used. The experiment was configured so that only the GPU was different in the same software environment. In addition to the CPU and GPU configurations, the driver and CUDA library versions used can be found in Table 5. The proposed optimization methods were all implemented using the C programming language. The CUDA Toolkit works on the basis of the C programming language, and the presented CUDA version is compatible with the GPU architecture of all platforms used in the experiment. When running the kernel on the GPU, the operation was monitored through the NVIDIA Visual Profiler and NVIDIA Nsight. The code we implemented can be referenced at: https://github.com/dannster95/GPU_BlockCipher/blob/master/kernel.cu.

Table 5. Experimental Environment.

	Platform 1	**Platform 2**	**Platform 3**
CPU	AMD Ryzen 5 3600		
Memory	32GB		
GPU	GTX 970	GTX 1070	RTX 2070
OS	Windows 10 64bit		
Library	CUDA Toolkit v10.2.89		
GPU driver	NVIDIA GeForce v445.87		

5.2. Experiment Results

The experimental results are shown in Tables 6 and 7. Table 6 measures the performance throughput of each optimization method on GTX 970, GTX 1070, and RTX 2070. This table measures performance excluding memory copy between CPU and GPU. Basic in the table means a simple parallel encryption implementation without optimization. CTR Opt, Memory Opt, and Async Opt in the table represent parallel encryption in CTR mode, avoid GPU memory problems, pipeline encryption with CUDA stream, and the proposed optimization methods.

Each value represents throughput when only the optimization method is applied in the basic implementation. For example, looking at the AES cipher algorithm in the RTX 2070 platform in Table 6, it can be confirmed that the basis implementation without the optimization method shows 74 Gbps performance. When applying only the CTR operation mode optimization method to this basis implementation, the performance of 94 Gbps can be shown. Similarly, when applying only the memory optimization implementation to the basis implementation, the performance of 237 Gbps can be performed. `All` indicates throughput when all optimization methods are applied in basis implementation. By applying all the proposed optimization methods to the AES cipher algorithm on the RTX 2070 platform, the throughput of 310 Gbps was measured. In Table 6, $(+n\%)$ on the right side of throughput shows the performance improvement compared to the basis implementation.

In AES, even within the same GPU parallel encryption implementation, the implementation method using CTR mode to reduce the copy time between the CPU and the GPU was 61% faster than the implementation performance including copying plaintext from the CPU to the GPU using the existing ECB mode. It was 94% faster in CHAM and 73% faster in LEA.

In the case of copying and encrypting the table and round key to shared memory, a significant performance improvement was obtained. In AES, the performance of implementation using shared memory was 57% better than that of the implementation that processed all data on global memory. In addition, as a result of using the coalesced memory access technique and CUDA stream instruction to reduce the memory copy time between the CPU and the GPU, more performance improvement compared to the previous implementation was achieved. In the case of CHAM and LEA, performance improvement was confirmed through the proposed memory optimization method.

Computational optimization using inline PTX also has improved performance dramatically. When all the operations of the encryption process were implemented using inline PTX, it showed about

20% higher speed than the previous implementation in AES kernel. The inline PTX showed a high performance improvement in CHAM, unlike the AES that use the table frequently. In case of using inline PTX in CHAM, it shows up 209% performance improvement. LEA, similar to CHAM, showed an excellent speed boost in GPU kernel. LEA shows 97% performance improvement when applying inline PTX. Including all the proposed optimization methods, AES, CHAM, and LEA show a speed up to 4.18, 6.98, and 6.48 times faster than the implementation without any optimization, respectively.

The performance improvement according to each optimization method is shown in Table 6. Commonly, the speed of encryption functions due to memory optimization has increased a lot, and in CHAM and LEA excluding AES, performance due to inline PTX implementation has improved significantly. In particular, in the case of CHAM, all variables used in the encryption function could be allocated and used in registers, so the performance improvement of the GPU encryption kernel according to the proposed optimization method was very high.

Table 6. Throughput by each optimization implementation (throughput is measured by Gbps).

Platform	Algorithm	Basic	CTR Opt.	Memory Opt.	Async Opt.	Inline PTX	All
GTX 970	AES	29	34 (+17%)	104 (+259%)	31 (+7%)	37 (+28%)	123 (+324%)
	CHAM	131	139 (+6%)	277 (+111%)	132 (+1%)	345 (+163%)	936 (+615%)
	LEA	122	124 (+2%)	348 (+185%)	122	271 (+122%)	868 (+611%)
GTX 1070	AES	51	59 (+16%)	171 (+235%)	57 (+12%)	58 (+14%)	214 (+320%)
	CHAM	221	247 (+12%)	473 (+114%)	241 (+9%)	594 (+169%)	1671 (+656%)
	LEA	202	219 (+8%)	615 (+204%)	226 (+12%)	460 (+128%)	1546 (+665%)
RTX 2070	AES	74	94 (+27%)	237 (+220%)	80 (+8%)	89 (+20%)	310 (+318%)
	CHAM	434	476 (+10%)	889 (+105%)	446 (+3%)	1342 (+209%)	3,033 (+598%)
	LEA	381	431 (+13%)	985 (+159%)	395 (+4%)	753 (+97%)	2472 (+548%)

Table 7 shows the difference in speed depending on the size of the memory and the number of threads per block. All performance in the table was measured on the RTX 2070. The best performance for the optimization implementation of AES, CHAM, and LEA is 310 Gbps, 3.03 Tbps, and 2.47 Tbps, respectively. In the case of AES, the highest performance was confirmed when encrypting 512 MB data with 512 threads per block. In the case of CHAM and LEA, unlike AES, they do not use a table, thus it takes less time to access the memory and shows higher throughput. In this table, the performance was measured ignoring data copy time.

Additionally, for GTX 1070, the highest performance of AES, CHAM, and LEA was 214 Gbps, 1.67 Tbps, and 1.54 Tbps, respectively, and, for GTX 970, the performance of 123 Gbps, 936 Gbps, and 868 Gbps was confirmed, respectively.

Table 7. Throughput by data size and the number of threads in RTX 2070 (Gbps).

		128 MB	256 MB	512 MB	1024 MB
AES	256 threads	300	307	296	280
	512 threads	309	306	**310**	301
	1024 threads	306	309	309	308
CHAM	256 threads	2886	**3033**	2952	2859
	512 threads	2827	2851	2870	2844
	1024 threads	2511	2539	2526	2501
LEA	256 threads	2437	**2472**	2412	2419
	512 threads	2429	2447	2452	2450
	1024 threads	2238	2251	2256	2261

5.3. Comparison

Table 8 compares throughput performance of our implementation and the other work. For comparison in the same environment, we implemented the method presented in [12] on RTX 2070, and the throughput of 278 Gbps was measured. In [13], even though the performance was given in cycles per byte, we convert it as the throughput using the proposed CPU clock count.

Table 8. Throughput comparison with existing implementations (Gbps).

	Platform	AES	CHAM	LEA
N. Nishikawa et al. [6]	GTX 680	63.9	-	-
R.K. Lim et al. [8]	GTX 480	78.6	-	-
A.A. Abdelrahman et al. [9]	GTX TITAN X	207.25	-	-
	GTX 1080	279.86	-	-
W.K. Lee et al. [12]	GTX 980	154	-	678
	GTX 1080	256	-	1478
T. Kim et al. [13]	i5 Skylake@2.3GHz (CPU)	-	1.33	-
H. Seo et al. [10]	GTX 680	-	-	138.8
Our work	GTX 970	123	936	868
	GTX 1070	214	1601	1505
	RTX 2070	310	3033	2472

6. Conclusions

Many studies have been conducted to speed up the cipher algorithm using various optimization methods. Currently, high-speed research is being conducted not only in CPU, but also in the GPU environment, and such research is essential for data encryption technology that is continuously evolving.

In this paper, we present various optimization techniques in Maxwell, Pascal, and Turing GPU architecture in order to encrypt a large amount of data. Our optimization studies were conducted for AES, CHAM, and LEA. AES is a well-known block cipher algorithm, but the lightweight block cipher algorithms CHAM and LEA have not been actively researched in the GPU environment. In particular, this is the first GPU optimization study for CHAM to the best of our knowledge.

First, it was implemented to reduce the time to copy from the first CPU to the GPU by using the characteristics of the CTR mode. Next, an optimization implementation was attempted to reduce the number of accesses to global memory with slow access latency by actively using shared memory and registers inside the GPU. In addition, we tried to prevent and reduce various problems that can occur while implementing in the GPU environment. Additionally, by implementing inline PTX that is performed inside the GPU, an optimization implementation method that simplifies heavy-duty computation or reduces unnecessary computation is applied.

As the proposed optimization methods, in the case of AES, a performance improvement of up to 4.18 times in RTX 2070, and throughput of 310 Gbps was obtained excluding the memory copy time. In our implementation, CHAM shows 3.03 Tbps throughput when applying all of the proposed optimization methods. This implementation is 6.98 times faster than a basic implementation without optimization. In particular, CHAM, which consists of simple operations that do not use tables, can significantly reduce kernel operating time through inline PTX. CHAM performance increased by 3.09 times when only the inline PTX optimization implementation method was applied. Since LEA is composed of simple operations like CHAM, it is possible to have an effective performance improvement through the inline PTX optimization method. When LEA applied only the inline PTX optimization implementation, the performance increased by 97%. LEA was able to obtain up to 6.48 times faster speed through all of the proposed optimization methods.

We studied the optimization of block cipher algorithms on various GPU platforms. In the future, we will conduct optimization research on public-key algorithms in the GPU environment. Through the results of this paper, it can be considered to be useful in environments such as cloud computing services and IoT edge computing services.

Author Contributions: Writing-original draft, S.A. and S.C.S.; Writing-review and editing, S.A. 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 Publication (FIPS) 197. Available online: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf (accessed on 30 April 2020).
2. Roh, D.; Koo, B.; Jung, Y.; Jeong, I.W.; Lee, D.G.; Kwon, D.; Kim, W.H. Revised Version of Block Cipher CHAM. In Proceedings of the 22nd Annual International Conference on Information Security and Cryptology, Seoul, Korea, 4–6 December 2019; pp. 1–19.
3. Hong, D.; Lee, J.; Kim, D.; Kwon, D.; Ryu, K.; Lee, D. LEA: A 128-Bit Block Cipher for Fast Encryption on Common Processors. In Proceedings of the International Workshop on Information Security Applications 2013, Jeju Island, Korea, 19–21 August 2013; pp. 3–27.
4. Biagio, A.D.; Barenghi, A.; Agosta, G.; Pelosi, G. Design of a parallel AES for graphics hardware using the CUDA framework. In Proceedings of the International Symposium on Parallel Distributed Processing, Lisbon, Portugal, 30 June–4 July 2009.
5. Zhao, K.; Mei, X.; Chu, X. Implementation and Analysis of AES Encryption on GPU. In Proceedings of the IEEE 14th International Conference on High Performance Computing and Communications, Liverpool, UK, 25–27 June 2012; pp. 843–848.
6. Nishikawa, N.; Iwai, K.; Tanaka, H.; Kurokawa, T. Throughput and Power Efficiency Evaluation of Block Ciphers on Kepler and GCN GPUs Using Micro-Benchmark Analysis. *IEICE Trans. Inf. Syst.* **2014**, *E97.D*, 1506–1515. [CrossRef]
7. Nishikawa, N.; Amano, H.; Iwai, K. Implementation of bitsliced AES encryption on CUDA Enabled GPU. In Proceedings of the Network and System Security—11th International Conference, Helsinki, Finland, 21–23 August 2017; pp. 273–287.
8. Lim, R.K.; Petzold, L.R.; Koç, Ç.K. Bitsliced High-Performance AES-ECB on GPUs. *Lncs Essays New Codebreakers* **2015**, *9100*, 125–133.
9. Abdelrahman, A.A.; Fouad, M.M.; Dahshan, H. High Performance CUDA AES Implementataion: A Quantitative Performance Analysis Approach. In Proceedings of the IEEE 2017 Computing Conference, London, UK, 18–20 July 2017; pp. 1077–1085.
10. Seo, H.; Park, T.; Seo, H.; Bae, B.; Hu, Z.; Zhou, L.; Nogami, Y.; Zhu, Y. Parallel Implementations of LEA, Revisited. In Proceedings of the International Workshop on Information Security Applications, Jeju Island, Korea, 25–27 August 2016; pp. 318–330.
11. Lee, W.K.; Cheong, H.S.; Phan, R.C.W.; Goi, B.M. Fast Implementation of block ciphers and PRNGs in Maxwell GPU architecture. *Clust. Comput.* **2016**, *19*, 335–347. [CrossRef]
12. 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.* **2019**, *31*, e5048. [CrossRef]
13. Kim, T.; Hong, D.; Software Implementation of Lightweight Block Cipher CHAM for Fast Encryption. *Korea Soc. Comput. Inf.* **2018**, *23*, 111–117. [CrossRef]
14. Lee, S.; Kang, J.; Hong, D.; Seo, C. Research for Speed Improvement Method of Lightweight Block Cipher CHAM using NEON SIMD. *Korean Inst. Inf. Sci. Eng.* **2019**, *46*, 485–491. [CrossRef]

15. Park, M.; Yoon, J. Optimization of Lightweight Encryption Algorithm (LEA) using Threads and Shared Memory of GPU. *J. Koread Inst. Inf. Secur. Cryptol.* **2015**, *25*, 719–726.
16. Gladman, B. A Specification for Rijndael, the AES Algorithm. Available online: https://pdfs.semanticscholar.org/9798/dbce96185ddc14f5da684c1cc4cb8642a276.pdf?_ga=2.98532575.191228374.1589876278-2102828541.1587714240 (accessed on 30 April 2020).



© 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/).