Designing a New XTS-AES Parallel Optimization Implementation Technique for Fast File Encryption

SANGWOO AN AND SEOG CHUNG SEO, (Member, IEEE)
Department of Financial Information Security, Kookmin University, Seoul 02707, South Korea
Corresponding author: Seog Chung Seo (scseo@kookmin.ac.kr)

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) funded by the Korean Government [Ministry of Science and ICT (MSIT)] through the Development of Fast Design and Implementation of Cryptographic Algorithms Based on Graphic Processing Unit/Application-Specific Integrated Circuit (GPU/ASIC) under Grant 2021-0-00540.

ABSTRACT XTS-AES is a disk encryption mode of operation that uses the block cipher AES. Several studies have been conducted to improve the encryption speed using XTS-AES according to the increasing disk size. Among them, there are researches on parallel encryption of XTS-AES using GPU. Although these studies focus on parallel encryption of AES, optimization for the entire XTS mode has not been performed. The reason is that the $\alpha_j$ computation process included in XTS mode is not suitable for parallel operation. Therefore, in this paper, we proposed several techniques for high-speed encryption in GPU by modifying XTS-AES into a form that is advantageous for parallel operation. The core idea is to pre-calculate the $\alpha_j$ calculation on the CPU into a form that is easy to operate on the GPU. To achieve this goal, we analyzed the $\alpha_j$ calculation process and present the parts that can be optimized. First, we presented a method that can replace multiple operations with a single table reference through the analyzed $\alpha_j$ computation progress. Thereafter, we proposed a method that can be calculated by partially skipping the entire $\alpha_j$ computation process that must be sequentially calculated through the table reference technique. For the proposed optimization implementation, we presented various results for evaluating the optimal implementation. In addition, we compared the performance of XTS-AES OpenSSL implementation on CPU and our proposed optimization implementation on GPU.

INDEX TERMS XTS, AES, GPU, CUDA, software optimization, disk encryption, full disk encryption.

I. INTRODUCTION Various security systems and cryptographic algorithms have been developed to protect user information. Disk encryption [1] is a type of technology that encrypts a computer’s hard disk to prevent information leakage caused by theft or loss. Representative, Bitlocker [2] on Windows performs a Full-Disk Encryption (FDE) function that encrypts the entire disk partition with one key. In addition, various disk encryption software, such as VeraCrypt and TrueCrypt, have been used in this area.

A common disk encryption method is the XTS operating mode using the block cipher algorithm AES [3]. XTS mode is a tweakable encryption method. The tweakable encryption method uses the sector address of the block in the sector and the tweak value, which is a combination of the index, which has the advantage of having different cryptographic statements depending on the location of the file.

Since the size of the disk increases, optimization of XTS-AES is required to effectively perform disk encryption. In XTS-AES, since the encryption process for each plaintext block is performed independently, a parallel computing device such as a GPU can be utilized. However, in the XTS mode, not only the AES encryption process but also the calculation process for the tweak value required for encryption is included. In XTS mode, the plaintext is encrypted according to the $\alpha$ defined in the Galois field. Depending on the total number $j$ of plaintext blocks, $\alpha$ is raised to a power of $j$, and encryption is performed using $\alpha^j$ in each $i$-th block.

Various optimization studies have been conducted on XTS-AES so far for fast file encryption. Although these studies contributed to the fast encryption of multiple plaintext
blocks by computing AES in parallel, they still have not proposed an optimization technique for the XTS mode itself. This is because the calculation process included in the XTS mode is not suitable for parallel operation. Since the calculation is operated while reading one most significant bit (MSB), it is a sequential structure in which the next operation cannot be performed before the previous operation is finished.

Therefore, in this paper, we have introduced a technique that can optimize the entire XTS mode by utilizing GPU. We have proposed a method to optimize the calculation process using a lookup table and intermediate values. This method is a new technique that goes through the pre-processing process on the CPU to efficiently perform operations on the GPU. Additionally, we have presented an efficient implementation technique that can parallelly compute AES block cipher algorithms used in XTS mode. The major contribution of this paper is to provide new insight regarding:

- **First entire XTS mode optimization implementation**
  In this paper, we have introduced an optimization technique for XTS-AES. However, this technique does not simply suggest a parallel encryption optimization method for AES. We have changed the operation structure of XTS mode, which is not suitable for parallel operation, to make it suitable for parallel operation, so that the entire XTS-AES process can be operated in parallel.

- **State-of-the-Art implementation for XTS-AES**
  The optimization implementation proposed in this paper showed the best computational performance than before. We could confirm that these results represent 240.1 times faster performance than the Naive CPU version and 21.96 times better performance than XTS-AES implemented in parallel with Naive GPU. This result was also 12.23 (XTS-AES-128) and 14.64 (XTS-AES-256) times faster than the most recent work on XTS-AES in OpenSSL.

- **Foundation technique available in various fields**
  We have proposed a fundamental optimization method for XTS-AES, and it can be utilized in various fields of research using XTS-AES. In addition to disk encryption, this optimization technique can be used for memory encryption using memory addresses, and can also be applied to encrypt data on the network or mobile devices.

This paper is organized as follows: Section II introduces optimization research papers conducted on existing XTS-AES. Section III looks at the structure and encryption procedures for XTS-AES. Section IV describes the optimization technique for XTS-AES that we intend to propose in this paper. Section V presents the implementation performance results for the optimization technique proposed in this paper and compares them with the results of the existing XTS-AES optimization study. Finally, we present significance for the findings of this paper in Section VI.

### II. RELATED WORKS

Researches that optimize the AES encryption process on GPUs have still been studied. Recently, various optimization methods and results of AES using GPUs have been presented in [4], [5] and [6].

However, not much research has been done on XTS mode yet. In [7] and [8], a method for parallel encryption of XTS mode using OpenMP [9] has been proposed. [8] have conducted the idea that multiple threads can encrypt blocks in parallel by utilizing the part that each plaintext can be encrypted independently like in CTR mode. [10] have presented several optimized techniques of XTS-AES that utilizes up to 32 processors simultaneously to rapidly encrypt large amounts of data in parallel using MPI [11]. In [12], A framework that can be utilized in mobile devices by speeding up XTS mode to GPU has been proposed. The proposed system was implemented in the form of a software module that performs parallel encryption for each 512-byte sector data. In the case of the hardware environment, studies on some XTS modes have been performed on FPGA. In [13], [14] and [15], various implementation techniques have been proposed to efficiently encrypt XTS-AES on FPGA.

Apart from research, XTS-AES has been provided by OpenSSL [16], an open-source implementation of TLS and SSL.

### III. BACKGROUND FOR XTS-AES

#### A. NOTATION

Table 1 summarizes the parameters for understanding XTS-AES.

#### B. XTS MODE

XTS [1] mode is a type of operation mode that operates block ciphers such as ECB, CBC, and CTR modes. XTS mode is a type of tweakable cipher specialized for encrypting block-oriented data, which was standardized by IEEE in 2007. XTS mode has been used in various disk encryption technologies so far, with the advantage of preventing vulnerabilities to existing CBC and XEX modes. The operating structure of the XTS mode is shown in Figure 1.

The encryption process via XTS mode is as follows. A total of two different keys are used in XTS mode. First of all, the tweak value is encrypted by the first key. This encrypted result is shared by all plaintext blocks with the same value. The encrypted tweak value is then multiplied by the $\alpha^j$ according to the number $i$ of each block. The multiplied values are different for each block and are utilized for a total of two XOR operations. The first XOR is performed with plaintext. The second XOR is performed on the result of encrypting the first XORred value with the second key.

The big difference between XTS mode and other block cipher operating modes is the multiplication operation with $\alpha^j$. The multiplication computation process with $\alpha^j$ is illustrated in detail in Figure 2. The first $\alpha^0$ value uses the tweak value encrypted with the first key. Alpha values are stored in the form of polynomials in the Galois field ($GF(2^{128})$) from the
TABLE 1. XTS-AES parameters.

| Variable | Description |
|----------|-------------|
| $\alpha$ | Primitive root in Galois field ($2^{128}$) |
| $i$      | Sector number of the plaintext block |
| $j$      | Total number of plaintext blocks |
| $P_i$    | i-th Plaintext |
| $C_i$    | i-th Ciphertext |
| Tweak    | 128-bit random string used as seed value |

FIGURE 1. Basic structure of XTS-AES.

The one-round process of AES can be integrated and saved as a 32-bit version of the table. It is called T-table [17]. Using T-table, encryption can be performed by referring to the T-table value 16 times in one round. The principle of T-table creation and usage for encryption is as follows. $s_{i,j}$ means the $j$-th word of the state in $i$-th round.

$$T_0[x] = \begin{bmatrix} \text{Sbox}[x] \ast 02 \\ \text{Sbox}[x] \\ \text{Sbox}[x] \ast 03 \end{bmatrix}$$

$$T_1[x] = \begin{bmatrix} \text{Sbox}[x] \\ \text{Sbox}[x] \\ \text{Sbox}[x] \\ \text{Sbox}[x] \ast 02 \end{bmatrix}$$

$$T_2[x] = \begin{bmatrix} \text{Sbox}[x] \\ \text{Sbox}[x] \ast 03 \\ \text{Sbox}[x] \ast 02 \end{bmatrix}$$

$$T_3[x] = \begin{bmatrix} \text{Sbox}[x] \\ \text{Sbox}[x] \ast 02 \end{bmatrix}$$

$$s_{i+1} = T_0[s_{i,j_0}] \oplus T_1[s_{i,j_1}] \oplus T_2[s_{i,j_2}] \oplus T_3[s_{i,j_3}]$$

D. GPU

GPU is a device developed to handle graphical operations. Recently, lots of General-Purpose computing on GPU (GPGPU) techniques that can utilize GPUs for general operations by utilizing Nvidia’s CUDA [18] library have been used. The advantage of GPUs is that they can handle multiple operations in parallel.

GPU consists of multiple blocks on a grid, and each block is composed of multiple threads. In NVIDIA GPU, the maximum number of threads that each block can utilize is 1024, but since there are limited resources per block, it is necessary to adjust the number of threads while considering the memory such as registers used by one thread.

Many cryptographic algorithms perform operations using not only basic bit-wise operations but also lookup tables. Since various types of memory exist in GPU, the performance difference greatly increases depending on which memory the reference table is stored and used. The overall memory structure of the GPU can be shown in Figure 4.

If the reference table is stored and used in the global memory that is not mounted on the GPU chip, which can be accessed by all threads, the load speed of the global memory is very slow, so it can show significantly slower performance compared to other memories. In the case of shared memory, since it is mounted on a chip, it has the advantage of being faster than the global memory and has the characteristic that it can be shared and used by threads within the same block. However, the shared memory has a limited size, and there is a disadvantage in that a bank conflict problem occurs when a plurality of threads access the same shared memory bank may occur. Registers show the fastest memory speed, but since the register size that a thread can utilize is greatly limited, efficient register design and use are required. Separately, the constant memory of the GPU has a slow memory access speed, but since frequently used values can be cached and used, it has a characteristic that it can show a memory access speed comparable to a register. Therefore,
when using memory in GPU, it is important to avoid using
global memory but to create the best reference environment
by properly distributing shared memory, constant memory,
and registers.

IV. XTS-AES OPTIMIZATION IMPLEMENTATION TECHNIQUES

A. PROBLEM

In XTS mode, each plaintext block can be encrypted inde-
dependently, as in CTR mode. But the XORed values in each
plain block are all different depending on the sector number
of the block. If the plaintext block is up to the $j$-th, the XTS
mode requires the sequential computation of the multiplica-
tion operations of 1 to $j$ power of $\alpha$ for the tweak value to
be encrypted. The problem is that the larger the size of the
data you want to encrypt, the larger the $j$, the greater the load
of the $j$-th power operation of $\alpha$. For example, if we encrypt
data in size 1 GB, each plain block has a size of 16 bytes,
so $j$ assigns a number from 1 to 67,108,864 to each plain
block. This method of computation needs to be improved
because the capacity of the storage devices used by the user
has increased significantly compared to the past. Therefore,
in this paper, we introduce an optimization technique that can
parallelly speed up the sequentially processed operations for
efficient XTS-AES encryption on large data.

B. MAIN IDEA

Our main idea is a technique that uses intermediate values for
parallel processing of the $j$-th power operations of $\alpha$, which
are computed sequentially. Rather than multiplying $\alpha$ one by
one, we present a method that allows the index to be calcu-
lated by skipping a certain interval, such as $\alpha$ to the 8th and
128th. With this calculation of the intermediate value on the
CPU, the remaining intervals that have been skipped can be
calculated in parallel through the GPU. Figure 5 summarizes
our main idea.

C. LOOKUP TABLE

During the power operation of $\alpha$, whole 128-bits data are
shifted to the left by 1 bit each time $\alpha$ is multiplied, and 135 is
XORed or not at the bottom of the data according to the msb.
135 is 8-bit data expressed in binary as 10000111 \( (2) \). While
considering the msb one by one, we decided to calculate the
final XORed value by considering the top 8 bits at once rather
than deciding whether to XOR at 135 or not. This is possible
because XOR values at the bottom of the data do not affect the
Most Significant Byte (MSB). For example, suppose the top
8 bits of data were 11111111 \( (2) \). If we operate the 8th power
of $\alpha$, XOR 135 and 1-bit shift to the left will be repeated a total of eight times. The first XOR 135 will finally be a 7-bit left shift, and the second XOR 135 will be a 6-bit left shift. The final results for a total of 8 bits are shown in Figure 6.

This XORed value depends only on msb, regardless of the lowest value of the data. Therefore, we can make a table of 256 results for the values $00000000_{(2)}$ to $11111111_{(2)}$ that the highest 8-bit can have. Results for 8-bit inputs can be found in Table 2.

### D. INTERMEDIATE VALUE

The use of tables reduces the operation of the 8th power of $\alpha$ to a single table lookup. In this case, the entire data needs to perform an 8-bit left shift, but since 8 bits is a single byte, it is only necessary to increase the byte index of the data one by one without having to shift. Using this, we can compute the next 128 bits, that is, to the 128th power of $\alpha$, using a table of all 16 bytes of data. Figure 7 shows the process of referencing the MSB data to a table and then XORing it to the least significant 2-byte. Figure 8 shows the process of referencing each byte of 16-byte data to a table and then XORing it to its location, considering the shift.

The primitive operation of repeated multiplication of $\alpha$ can be found in Algorithm 1. For an $\alpha^i$ of 128-bits, each operation is performed as follows: The value of the $\alpha^{i+1}$ is doubled for $mod^{128}_{2}$ from $\alpha^i$. In implementation, this can be implemented by shifting left by one bit. After that, the most significant bit of the alpha is checked, and if it is 1, the result of XORing 135 becomes the final $\alpha^{i+1}$ result.

The optimized operation that shortens the time to multiply the $\alpha$ one by one can be found in Algorithm 2 and 3. The difference from the Algorithm 1 is that $\alpha^{i+8}$ or $\alpha^{i+128}$ can be directly calculated through $\alpha^1$ instead of $\alpha^{i+1}$.

In the case of Algorithm 2, $\alpha^{i+8}$ is the result of multiplying $\alpha$ from $\alpha^8$ 8 times, so 2 is multiplied 8 times. Alpha is data composed of 16 bytes. Therefore, in byte representation, there is no need to multiply by $2^8$, just move the positions of the byte data array one by one. In Figure 7, it can be seen that byte data moves to the next byte position. The process of performing XOR 135 while reading the most significant bit 8 times is converted into a single table reference. The most significant byte goes into table $T(\alpha^8)$ and comes out as a 16-bit result, which is XORed on the result.
TABLE 2. Table of hexadecimal result values to be XORed by 16 bits according to top 8-bit input.

| a | b | c | d | e | f |
|---|---|---|---|---|---|
| 0 | 0000 | 0087 | 010e | 0189 | 021c | 029b | 0312 | 0395 | 0438 | 04bf | 0536 | 05b1 | 0624 | 06a3 | 072a | 07ad |
| 1 | 0870 | 087f | 097e | 09f9 | 0a6c | 0b62 | 0be5 | 0c48 | 0c6f | 0d46 | 0dc1 | 0e54 | 0ed3 | 0f5a | 0fd4 |
| 2 | 10e0 | 1067 | 11ee | 1169 | 12c2 | 127b | 13f2 | 1375 | 14d8 | 145f | 15d6 | 1551 | 16c4 | 1643 | 17ca | 174d |
| 3 | 1890 | 1817 | 199e | 1919 | 1a8c | 1a0b | 1b82 | 1b05 | 1ca8 | 1c2f | 1da6 | 1d21 | 1eb4 | 1e35 | 1fba | 1fd3 |
| 4 | 21c0 | 2147 | 20ce | 2049 | 23cd | 235b | 22d2 | 2255 | 2518 | 257f | 2416 | 2471 | 27c4 | 2763 | 26ea | 266d |
| 5 | 29b0 | 2937 | 28bc | 2839 | 2bac | 2b2b | 2aa2 | 2a25 | 2d88 | 2d0f | 2c86 | 2c01 | 2f94 | 2f13 | 2e9a | 2e1d |
| 6 | 3120 | 31a7 | 302e | 30a9 | 33cc | 33bb | 3232 | 32b5 | 3518 | 359f | 3416 | 3491 | 3704 | 3783 | 360a | 368d |
| 7 | 3950 | 39d7 | 385e | 3849 | 3b48 | 3bdc | 3a42 | 3ac5 | 3def | 3def | 3ce1 | 37f4 | 3f3f | 3e7a | 3efd |
| 8 | 4380 | 4307 | 428e | 4209 | 419c | 411b | 4092 | 4015 | 47b8 | 473f | 46b6 | 4631 | 45a4 | 4523 | 44aa | 442d |
| 9 | 4bf0 | 4b77 | 4a4e | 4a79 | 49ec | 4962 | 48e2 | 4865 | 4f08 | 4f4f | 4ec6 | 4e41 | 4dd4 | 4d53 | 4dca | 4e5d |
| a | 5360 | 53e7 | 526e | 52a9 | 517c | 51b2 | 5072 | 5015 | 5758 | 57df | 5656 | 56d1 | 5544 | 55c3 | 544a | 54cd |
| b | 5b10 | 5b97 | 5a1e | 5a99 | 590c | 598b | 5802 | 5885 | 5f68 | 5f6f | 5e26 | 5e81 | 5d34 | 5d6b | 5c3a | 5cbb |
| c | 6240 | 62c7 | 6c6e | 6c99 | 665c | 606b | 6152 | 61d5 | 6678 | 667f | 6767 | 67f1 | 6464 | 64e3 | 656a | 65ed |
| d | 6a30 | 6ab7 | 6b3e | 6b99 | 66b9 | 6922 | 69a5 | 6e08 | 6e8f | 6106 | 6181 | 61c4 | 6193 | 61da | 61d9 | 620d |
| e | 72a0 | 7227 | 73ae | 7329 | 70bc | 703b | 71b2 | 7135 | 7698 | 761f | 7796 | 7711 | 7484 | 7403 | 758a | 750d |
| f | 7a70 | 7a57 | 7bd0 | 7b59 | 7cc8 | 784b | 79c2 | 7945 | 7ee8 | 7ee6 | 7e6f | 7f61 | 7cf4 | 7c73 | 7d6a | 7d7d |

Algorithm 1 Primitive Operation of Repeated Multiplication of $\alpha^i$

Input: 128-bits data $\alpha^i$

Output: 128-bits data $\alpha^{i+1}$

1: $\alpha^{i+1} = (2 \times \alpha^i) \mod 2^{128}$
2: if Most Significant Bit of $\alpha^i$ is 1 then
3: $\alpha^{i+1} = \alpha^{i+1} \oplus 135$
4: end if

Algorithm 2 Optimized Operation of Repeated Multiplication of $\alpha^8$

Input: 128-bits data $\alpha^i \triangleright T = \text{Alpha Table}(Table 2)$

Output: 128-bits data $\alpha^{i+8}$

1: $\alpha^{i+8} = (2^8 \times \alpha^i) \mod 2^{128}$
2: $n = \text{Most Significant Byte of } \alpha^i$
3: $\alpha^{i+8} = \alpha^{i+8} \oplus T[n]$

In Algorithm 3, all 16 byte values constituting alpha go through table reference. The result values of each byte of data entered into the reference table must be XORed according to their byte positions. This is why the result values of each table are XORed as shown in Figure 8. However, since the most significant byte data performing table reference outputs a 16-bit table result value, an 8-bit carry occurs, and it must be XORed to the result value by putting it in the reference table once more.

E. PARALLEL OPERATION IN GPU

By changing the operation process for alpha into a form that is easy for parallel operation and transferring it to the GPU, the GPU can perform the powering operation on alpha in parallel and then independently encrypt each plaintext block. Inside the GPU, encryption proceeds in two stages. The first is the process of generating tweak values for all blocks from $\alpha^0$ to $\alpha^{j-1}$ using the received $\alpha^0$, $\alpha^{128}$, $\alpha^{256}$, ..., $\alpha^{j-128}$, and the second is the process of encrypting the plaintext using the tweak values.

F. PARALLEL ENCRYPTION PROCESS

Each GPU thread performs encryption using one tweak value and a plaintext block. In XTS mode, the tweak value is XORed with the data before and after the encryption process. Therefore, plaintext should be stored in GPU memory. To encrypt the plaintext after XORing it with the tweak value inside the GPU, it is necessary to copy the plaintext data from the CPU to the GPU in advance. We use CUDA streams to reduce the memory copy time between CPU and GPU. By dividing data by the number of streams, each stream can perform memory copy and operation asynchronously. We leveraged 32 CUDA streams to maximize the pipe-lining effect of encryption and memory copy. 32 is the maximum...
Algorithm 3 Optimized Operation of Repeated Multiplication of $\alpha^{128}$

**Input:** 128-bits data $\alpha^i$  \(\Rightarrow T = \text{Alpha Table(Table 2)}\)

**Output:** 128-bits data $\alpha^{i+128}$

1: \(\text{for } j = 0 \rightarrow 14 \text{ do}\)
2: \(n = j\)-th Least Significant Byte of $\alpha^i$\)
3: $\alpha^{i+128} = \alpha^{i+128} \oplus T[n] \cdot 2^j$\)
4: \(\text{end for}\)
5: \(n = 15\)-th Least Significant Byte of $\alpha^i$\)
6: $nm = \text{Most Significant Byte of } T[n]$\)
7: $nl = \text{Least Significant Byte of } T[n]$\)
8: $\alpha^{i+128} = \alpha^{i+128} \oplus nl \cdot 2^{120}$\)
9: $\alpha^{i+128} = \alpha^{i+128} \oplus T[nm]$\)

number of streams available on the GPU and shows the highest performance.

In XTS-AES, since all plaintext blocks use the same key value, the round key can be expanded in the CPU and copied to the GPU’s constant memory for use. GPU constant memory improves memory reference speed by caching frequently used values. In addition, we implemented AES with a 32-bit word size using T-box to speed up the AES encryption process by utilizing the 32-bit size register of the GPU.

In the case of the implementation method that uses the T-box by storing it in shared memory, if the threads access the same bank address in the shared memory, a bank conflict problem may occur. To avoid this problem, we implemented T-box to be copied as much as the bank size so that each thread refers to a different bank address in the same shared memory. In our implementation, 32 identical T-boxes corresponding to the bank size were stored in shared memory and used for encryption.

### V. EVALUATION

In this section, we evaluated the performance of our proposed XTS-AES optimization implementation. First, we profiled the computational weight required to calculate the tweak value in each environment of the CPU and GPU. Afterward, we compared the performance difference when the encryption was exclusively performed in each environment of the CPU and GPU. In addition, we summarized how performance differs for different implementations of optimizations. Finally, for performance comparison with other XTS-AES implementations, we compared the performance of XTS-AES provided by the open-source OpenSSL [16] with the performance of our proposed implementation. OpenSSL provides CPU multi-threading technology and parallel operation through AVX instructions [19]. This allows cryptographic operations to run very quickly even on the CPU.

The environment used to measure the implementation performance is as follows. In AMD Ryzen 9 5900X 4.7GHz OC CPU environment, we evaluated the performance of our Naive CPU implementation and benchmarked the XTS-AES implementation of OpenSSL 3.0.1. Performance of all GPU implementations was measured on NVIDIA GeForce RTX 3090 GPU. All performance measurement results were calculated as the average of the results of 1,000 iterations. Due to the computational characteristic of GPUs, there is a size of parallel computation data at which performance reaches a critical point. Therefore, Tables 3 and 4 present the performance results at the 128 MB data size, which is the performance saturation point.

The performance results of the GPU were measured based on the time it takes to copy all the plaintext data from the CPU to the GPU and then copy the ciphertext data from the GPU back to the CPU after encryption.

Each processing time for $\alpha^i$ computation and encryption in XTS-AES is shown in Table 3. Naive CPU is an implementation that sequentially encrypts AES using all the generated $\alpha^i$ values after all $\alpha^i$ computations are sequentially performed. Naive GPU is an implementation that performs $\alpha^i$ computation sequentially but encrypts AES in parallel using the generated $\alpha^i$. When comparing the Naive CPU and Naive GPU implementations, it could be seen that the $\alpha^i$ computation time of both is the same, but the encryption

| Implementation | Operation | Time(ms) | Ratio(%) |
|----------------|-----------|----------|----------|
| Naive CPU      | $\alpha^i$ computation | 32.72   | 85.7     |
|                | Encryption | 349.04  | 91.43    |
|                | Total      | 381.76  | 100      |

| Implementation | Operation | Time(ms) | Ratio(%) |
|----------------|-----------|----------|----------|
| Naive GPU      | $\alpha^i$ computation | 32.72   | 93.24    |
|                | Encryption | 0.3      | 6.76     |
|                | Total      | 35.72   | 100      |

| Implementation | Operation | Time(ms) | Ratio(%) |
|----------------|-----------|----------|----------|
| Optimized GPU ($\alpha^8$) | $\alpha^i$ computation | 4.08    | 77.33    |
|                        | Encryption | 1.2     | 22.67    |
|                        | Total      | 5.28    | 100      |

| Implementation | Operation | Time(ms) | Ratio(%) |
|----------------|-----------|----------|----------|
| Optimized GPU ($\alpha^{128}$) | $\alpha^i$ computation | 0.25    | 15.69    |
|                        | Encryption | 1.34    | 84.31    |
|                        | Total      | 1.59    | 100      |
operation time is greatly reduced in the GPU environment. Therefore, it was confirmed that the time to encrypt the entire
data through XTS-AES is reduced by about 1/11(10.69 times
faster) of the Naive GPU (35.72 ms) compared to the Naive
CPU (381.76 ms).

Table 4 shows the comparison of our several optimization
implementation. The optimized GPU is an implementation
that calculates the intermediate value $\alpha^8$ or $\alpha^{128}$ through
a lookup table so that the $\alpha^j$ computation process can be
performed in parallel, and then performs the rest $\alpha^i$
calculation and encryption in parallel. Unlike the Naive GPU
implementation, which computes only the encryption process
in parallel, it could be seen that the $\alpha^j$ computation time
is greatly reduced in the implementations that optimize the
$\alpha^j$ computation process in a form that is easy for parallel
operation. It could be seen that the encryption time increases
as the intermediate value range for $\alpha^j$ are set larger, but
the total operation time of XTS-AES gradually decreases.
Finally, it was confirmed that the computation time of the
implementation optimized to compute $\alpha^{128}$ as an interme-
diate value (1.59 ms) compared to the computation time of the
naive GPU implementation (34.91 ms) is reduced by about
1/22(21.96 times faster).

Table 5 compares the performance of XTS-AES in
OpenSSL with the performance of the optimization imple-
mentation proposed in this paper. The percentage figures
in the table are the performance improvement of the GPU
implementation compared to OpenSSL 3.0.1. It could be
seen that the overall performance of the GPU implementation
improves as the block size increases in XTS-AES. When the
block size is 8192, it was confirmed that the performance
improvement of XTS-AES-128 was about 12.23 times, and
that of XTS-AES-256 was about 14.64 times faster.

VI. CONCLUSION
In this paper, we propose several optimization techniques that
can efficiently compute XTS-AES, an encryption method
used for disk encryption. We proposed implementation tech-
niques that can change the tweak operation process, which
is not suitable for parallel operation, into a form that is
easy for parallel operation by using a lookup table and
intermediate values. As a result of this implementation,
it was possible to achieve about 12.23(XTS-AES-128) and
14.64(XTS-AES-256) times improvement in performance
compared to the implementation of OpenSSL. The tech-
niques and results proposed in this paper can be used for
various disk encryption functions and can be used not only
for disk encryption but also for mobile device encryption
or network encryption that can utilize location information
for encryption. In addition, since it is not a block cipher
AES optimization technique for the XTS operation mode and
does not depend on a specific algorithm, it can be used for
the XTS mode of various cryptographic algorithms. In the
future, we plan to compare the performance improvement by
applying our optimization techniques to Veracrypt, an open
source FDE software.

CONFLICT OF INTEREST
All authors have no conflict of interest

REFERENCES
[1] Standard for Cryptographic Protection of Data on Block-Oriented Storage
Devices, Standard IEEE P1619T/D16, 2007.
[2] Overview BitLocker Device Encryption Windows, Microsoft, Albuquerque,
NM, USA, 2021.
[3] Advanced Encryption Standard (AES), NIST, Gaithersburg, MD, USA,
2001.
[4] W. K. Lee, B.-M. Goi, and R. Phan, “Terabit encryption in a second:
Performance evaluation of block ciphers in GPU with Kepler, Maxwell,
and Pascal architectures,” Concurrency Comput., Pract. Exp., vol. 31,
p. e5048, Oct. 2018.
[5] O. Hajihassani, S. K. Monfared, S. H. Khasteh, and S. Gorgin, “Fast
AES implementation: A high-throughput bitsliced approach,” IEEE Trans.
Parallel Distrib. Syst., vol. 30, no. 10, pp. 2211–2222, Oct. 2019.
[6] S. An and S. C. Seo, “Highly efficient implementation of block ciphers
on graphic processing units for massively large data,” Appl. Sci., vol. 10,
no. 11, p. 3711, 2020.
[7] M. A. Alomari, K. Samsudin, and A. R. Ramli, “A parallel XTS encryption
mode of operation,” in Proc. IEEE Student Conf. Res. Develop. (SCoReD),
Nov. 2009, pp. 172–175.
[8] M. A. Alomari, K. Samsudin, and A. R. Ramli, “Implementation of a
parallel XTS encryption mode of operation,” Indian J. Sci. Technol., vol. 7,
no. 11, pp. 1813–1819, 2014.
[9] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald,
Parallel Program. OpenMP. Burlington, MA, USA: Morgan kaufmann,
2001.
[10] M. Shrestha, “Parallel implementation of AES using XTS mode of operation,” in *Culminating Projects in Computer Science and Information Technology*, USA: St. Cloud State Univ., 2018.

[11] *MPI: A Message-Passing Interface Standard*, Madhyanchal Forum, Madhya Pradesh, India, 1994.

[12] M. A. Alomari and K. Samsudin, “A framework for GPU-accelerated AES-XTS encryption in mobile devices,” in *Proc. IEEE Region 10 Conf.*, Nov. 2011, pp. 144–148.

[13] A. Shakil, K. Samsudin, A. Ramli, and F. Rokhani, “Effective implementation of AES-XTS on FPGA,” in *Proc. IEEE Region Conf.*, Nov. 2011, pp. 184–186.

[14] S. Ahmed, K. Samsudin, A. R. Ramli, and F. Z. Rokhani, “Advanced encryption standard-XTS implementation in field programmable gate array hardware,” *Secur. Commun. Netw.*, vol. 8, no. 3, pp. 516–522, Feb. 2015.

[15] Y. Wang, A. Kumar, and Y. Ha, “FPGA-based high throughput XTS-AES encryption/decryption for storage area network,” in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2014, pp. 268–271.

[16] *OpenSSL: The Open Source Toolkit for SSL/TLS*, OpenSSL, 2021.

[17] *CUDA Toolkit*, NVIDIA, Santa Clara, CA, USA, 2021.

[18] C. Lomont, *Introduction to Intel Advanced Vector Extensions*. Santa Clara, CA, USA: Intel, 2011.