Article

Efficient GPU Parallel Implementation and Optimization of ARIA for Counter and Exhaustive Key-Search Modes

Siwoo Eum [1], Minho Song [1], Sangwon Kim [2] and Hwajeong Seo [2,*]

[1] Department of Information Computer Engineering, Hansung University, Seoul 02876, Republic of Korea; shuraatum@hansung.ac.kr (S.E.); smino732@hansung.ac.kr (M.S.)
[2] Department of Convergence Security, Hansung University, Seoul 02876, Republic of Korea; 23613701@hansung.ac.kr
* Correspondence: hwajeong@hansung.ac.kr

Abstract: This paper proposes an optimized shared memory access technique to enhance parallel processing performance and reduce memory accesses for the ARIA block cipher in GPU environments. To overcome the limited size of GPU shared memory, we merged ARIA's four separate S-box tables into a single unified 32-bit table, effectively reducing the total memory usage from 4 KB to 1 KB. This allowed the consolidated table to be replicated 32 times within the limited shared memory, efficiently resolving the memory-bank conflict issues frequently encountered during parallel execution. Additionally, we utilized CUDA's built-in function *__byte_perm()*to efficiently reconstruct the desired outputs from the reduced unified table, without imposing additional computational overhead. In exhaustive key-search scenarios, we implemented an on-the-fly key-expansion method, significantly reducing the memory usage per thread and enhancing parallel processing efficiency. In the RTX 3060 environment, profiling was performed to accurately analyze shared memory efficiency and the performance degradation caused by bank conflicts, yielding detailed profiling results. The results of experiments conducted on the RTX 3060 Mobile and RTX 4080 GPUs demonstrated significant performance improvements over conventional methods. Notably, the RTX 4080 GPU achieved a maximum throughput of 1532.42 Gbps in ARIA-CTR mode, clearly validating the effectiveness and practical applicability of the proposed optimization techniques. On the RTX 3060, the performance of 128-bit ARIA-CTR was improved by 2.34× compared to previous state-of-the-art implementations. Furthermore, for exhaustive key searches on the 128-bit ARIA block cipher, a throughput of 1365.84 Gbps was achieved on the RTX 4080 GPU.

Keywords: ARIA; block cipher; parallel; GPU; counter mode; exhaustive key search; shared memory; bank conflict; CUDA; GPGPU; CUDA optimization; memory-bank conflict resolution; parallel cryptography

Academic Editors: Li Pan, Ning Liu and Conghui Zheng

Received: 10 April 2025
Revised: 12 May 2025
Accepted: 15 May 2025
Published: 15 May 2025

Citation: Eum, S.; Song, M.; Kim, S.; Seo, H. Efficient GPU Parallel Implementation and Optimization of ARIA for Counter and Exhaustive Key-Search Modes. *Electronics* **2025**, *14*, 2021. https://doi.org/10.3390/electronics14102021

1. Introduction

As data volumes increase and network speeds accelerate, modern computing demands fast and secure data encryption. The ARIA block cipher, a 128-bit block cipher developed by Korean cryptographers in 2003, has become a national standard (KS) [1] and an international standard (IETF RFC 5794) [2,3]. ARIA supports 128-, 192-, and 256-bit keys with 12, 14, or 16 rounds, respectively, similar in interface to that of AES (Advanced Encryption Standard) [4]. It has been widely adopted in Korea for government and public services and is included in security protocols (e.g., TLS/SSL) as a supported cipher. Given its growing use in practice, optimizing ARIA implementation for high throughput is an important research topic.

Graphics Processing Units (GPUs) offer a potent platform for accelerating cryptographic algorithms due to their massive parallelism and memory bandwidth. Unlike CPUs, which feature a few complex cores, GPUs consist of thousands of simpler cores that can execute thousands of threads concurrently in a Single-Instruction, Multi-Thread fashion. This makes GPUs well suited for data-parallel tasks like block cipher encryption, where many blocks can be processed independently. In particular, the counter (CTR) mode of operation is embarrassingly parallel: each block encryption uses a unique counter value and can be computed independently of others [5]. CTR mode is widely used in industry because of this parallelizability and because decryption simply reuses the encryption procedure (XORing the same keystream), avoiding the need for a separate decryption implementation. By exploiting GPU parallelism, an entire stream of CTR mode encryption can be performed simultaneously, significantly increasing throughput.

Another motivation for GPU-based ARIA optimization is to facilitate exhaustive key searches (brute-force attacks) in cryptanalysis scenarios. In theory, any block cipher with a k-bit key can be broken by trying 2^k possible keys [6]. For example, given a known plaintext–ciphertext pair, an attacker could encrypt the plaintext under every possible key until the matching ciphertext is found. Although a full 128-bit key search is computationally infeasible (requiring astronomically many trials), lower-bit security levels or portions of the key space can be explored using brute force. GPUs are attractive for this task because they can test many keys in parallel, offering orders-of-magnitude speedups over a single CPU core. In practice, the throughput of a brute-force search is bounded by how fast each trial encryption can be performed. Optimizing ARIA on GPUs, therefore, not only benefits legitimate encryption speeds but also allows security researchers to assess the cipher's resistance against brute-force attacks by reaching higher key-testing rates. Prior work on AES-128 demonstrated that a single NVIDIA RTX GPU can test on the order of 10^9 keys per second when fully optimized, illustrating both the power and limits of GPU-accelerated key searches (even at that speed, billions of GPUs and many years would be needed to exhaust 128-bit keys). This research targets similar optimizations for ARIA. Recent studies have begun to explore ARIA software optimizations on modern architectures. Eum et al. (2022) presented parallel implementations of ARIA on both ARMv8 CPUs and an NVIDIA GPU [7]. They reported that using ARM NEON vector instructions to process 4 or 16 blocks in parallel significantly improved ARIA encryption throughput, and on GPUs, they found that efficient usage of memory hierarchies is critical. In particular, loading ARIA S-box tables into fast on-chip shared memory (scratchpad memory) yielded a 1.08×–1.43× speedup over global memory access. They also experimented with an extended S-box (T-table) approach to merge substitution and diffusion operations, but noted that its benefit was limited by memory-bank conflicts in shared memory. An earlier study by Xiao et al. proposed merging the ARIA round function into lookup tables and carefully arranging data in different GPU memory spaces, achieving up to an 18×–45× speedup over a CPU implementation [8]. These works demonstrated the potential of GPU acceleration for ARIA, but they primarily focused on general encryption throughput (equivalent to ECB mode) with a fixed key. CTR mode encryption and exhaustive key searches pose different challenges—for example, CTR mode adds overhead for handling counters, and a brute-force search cannot amortize the key schedule across many blocks since the key changes every encryption. To date, there has been no dedicated study (to the best of our knowledge) on optimizing ARIA in CTR mode or on GPU-accelerated key searches.

Contributions. The main contributions of this paper are as follows:

1. We implement an optimized CTR mode for the ARIA algorithm. Previous research [7] could not employ a table-copying technique to avoid bank conflicts due to the large size of the S-box table. In this work, we reduce the size of the S-box table from 4 KB to 1 KB so that it can be replicated in shared memory as many times as the number of banks, effectively preventing bank conflicts and maximizing shared memory efficiency. To provide an accurate comparison of performance improvements, we also conduct a detailed profiling analysis based on the storage location of the S-box table. Specifically, we evaluate and compare performance when the table is stored in global memory, shared memory, and shared memory with our proposed bank conflict minimization technique.
2. We extensively utilize CUDA's built-in function *__byte_perm()*. Encryption processes often require state transformations such as permutations, which we efficiently implement using __byte_perm() instructions. Furthermore, we reconstruct the outputs of the reduced S-box table using *__byte_perm()*, achieving the same results as the original S-box with minimal additional overhead.
3. Exhaustive key searches (ES) in block cipher modes like CTR are suitable for parallel implementations due to their independent block computations. However, performing key expansion for each thread individually creates a memory burden for storing round keys. To address this, we implement an on-the-fly approach that computes round keys as needed. This method significantly reduces memory usage and minimizes memory accesses, resulting in performance improvements. To the best of our knowledge, this work presents the first optimization and performance analysis of ARIA ES operation.
4. Experiments on the RTX 3060 Mobile and RTX 4080 GPUs show notable performance gains. The RTX 4080 reaches 1532.42 Gbps in ARIA-CTR mode and 1365.84 Gbps in exhaustive search, while the RTX 3060 achieves a 2.34× speedup over prior implementations.

2. Background

2.1. Overview of the ARIA Block Cipher

ARIA is a 128-bit block cipher designed in 2003 by a team of South Korean cryptographers. It was selected as a national standard cipher algorithm (KS X 1213) in 2004 and later described in RFC 5794 for wider adoption. Algorithmically, ARIA is a substitution–permutation network (SPN) similar in structure to AES. It supports three key lengths—128, 192, or 256 bits—and the number of rounds varies accordingly (12, 14, or 16 rounds). This design allows ARIA to scale its security level while maintaining a 128-bit block size and overall structure.

Each round of ARIA consists of a sequence of transformations: a key addition, a substitution layer, and a diffusion layer. In the key addition step, a 128-bit round key (derived from the master key via the key schedule) is XORed with the data block. Next, the substitution layer applies byte-wise S-box transformations to introduce non-linearity. ARIA is unique in using four different 8×8-bit S-boxes (named $s1$, $s2$, $x1$, and $x2$) rather than a single S-box. These S-boxes are applied in an alternating pattern: in odd-numbered rounds, the substitution layer (denoted as $SL1$) applies S-boxes $s1, s2, x1$, and $x2$ to bytes $0, 1, 2$, and 3, respectively, and repeats this pattern for each group of four bytes. In even-numbered rounds, a different permutation (denoted as $SL2$) is used: bytes $0, 1, 2$, and 3 are transformed by $x1, x2, s1$, and $s2$, respectively. In essence, the cipher cycles through two S-box patterns, effectively utilizing all four S-box tables. One of these S-boxes is the Rijndael (AES) S-box, while the others are independent inverses or related transformations. This multi-S-box design increases cryptographic strength but also means that an implementation must handle four lookup tables instead of one, which has implications for optimization.

After the substitution layer, ARIA performs a linear diffusion layer (called Function A in the specification). This layer takes the 16-byte state and mixes the bits across bytes to achieve high diffusion (i.e., each output bit depends on many input bits). The diffusion layer can be seen as a matrix multiplication over a finite field, somewhat analogous to the AES MixColumns step, although the ARIA matrix is different. In practice, the diffusion involves a series of XORs and bit rotations on the state. Because ARIA substitution outputs are 8-bit values, the diffusion layer operates on these bytes (and their bits) to spread local changes across the 128-bit block. The specifics of the linear transformation are designed so that after a few rounds, each output byte is an affine function of all input bytes, improving security through the avalanche effect. The detailed structure of the ARIA algorithm is illustrated in Figure 1.

The key schedule of ARIA is also non-trivial. To generate the round keys from the master key, ARIA uses a 256-bit intermediate and processes it with a three-round Feistel network. The master key K (128, 192, or 256 bits) is split into two 128-bit values K_L and K_R, with K_R padded with zeros if needed. These are then alternately processed with two round functions (denoted as FO for "odd" and FE for "even" rounds) and fixed 128-bit constants derived from the fractional part of π. The FO and FE functions are structurally similar to the main cipher round: each includes an S-box layer ($SL1$ or $SL2$) and a diffusion layer A. As a result, the ARIA key schedule uses the same S-box operations as the encryption rounds, applied multiple times. This design produces an expanded set of round keys: one for each round, plus a whitening key for the final XOR. While the key schedule adds to the computational cost, especially for larger key sizes that entail more rounds of processing, it has the advantage of being an involution for certain key lengths, making the encryption and decryption key schedules identical and providing security against related-key attacks by introducing non-linearity in key expansion.

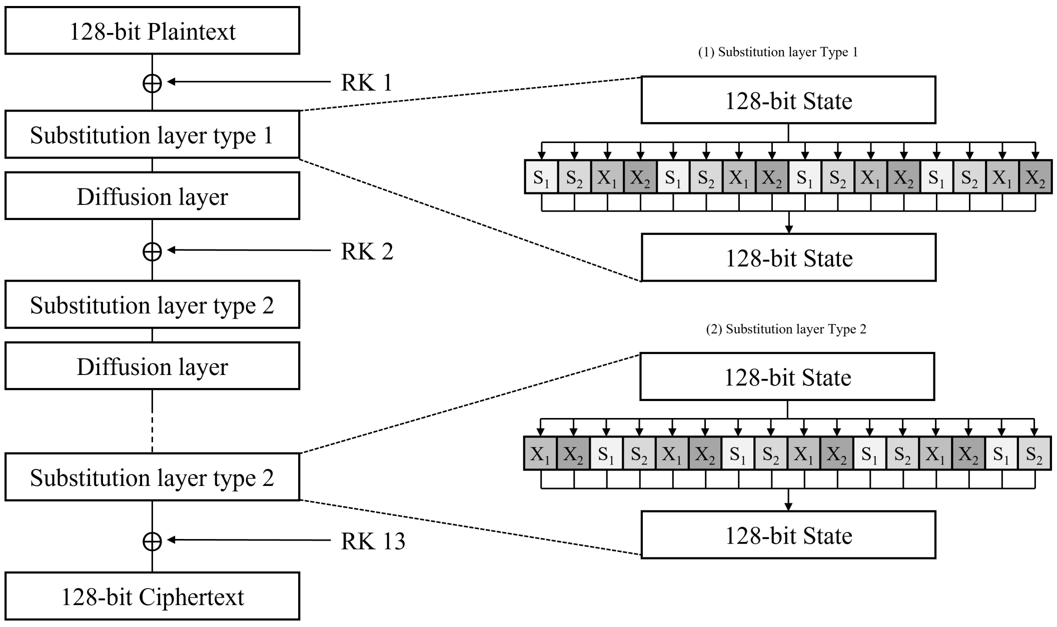
128-bit Plaintext
RK 1
Substitution layer type 1
Diffusion layer
RK 2
Substitution layer type 2
Diffusion layer
Substitution layer type 2
RK 13
128-bit Ciphertext
(1) Substitution layer Type 1
128-bit State
S1 S2 X1 X2 S1 S2 X1 X2 S1 S2 X1 X2 S1 S2 X1 X2
128-bit State
(2) Substitution layer Type 2
128-bit State
X1 X2 S1 S2 X1 X2 S1 S2 X1 X2 S1 S2 X1 X2 S1 S2
128-bit State

Figure 1. Overall operation process of the ARIA algorithm (128-bit key).

In summary, the ARIA encryption process involves repeated application of four S-box lookups per byte and a linear mixing of bytes, with a complex key schedule that itself invokes S-boxes. These characteristics mean that a straightforward implementation will be heavy in table lookups and XOR/rotate operations. Efficient implementation must handle multiple S-box tables and potentially significant memory access for both data and round keys. In contexts like CTR mode or parallel processing, one can take advantage of the fact that the same key (and hence round keys) is reused for many block encryptions, amortizing the key-expansion cost. However, in a brute-force scenario (trying many keys), the key schedule cost becomes a critical factor. Our optimizations consider these aspects, aiming to reduce the cost of S-box lookups and key schedule computations.

2.2. Graphics Processing Units and CUDA Basics

GPU architectures are fundamentally designed to achieve high throughput on parallel workloads. Modern GPUs consist of numerous Streaming Multiprocessors (SMs), each capable of running thousands of threads in parallel. GPU threads are extremely lightweight and organized into groups called warps, typically comprising 32 threads [9]. This Single-Instruction, Multi-Thread (SIMT) execution model executes the same instruction simultaneously across a warp, each thread handling different data.

NVIDIA's CUDA is a popular platform for general-purpose GPU computing. In CUDA, threads are grouped into blocks, and multiple blocks form a grid that executes on the device. A critical aspect of GPU performance is its ability to hide memory latency through rapid context switching between warps: while one warp waits for a memory operation to complete, another warp can execute immediately [10,11].

Memory hierarchy is an essential consideration in CUDA programming, as performance significantly depends on the type of memory used for storing data (such as S-box tables or round keys). NVIDIA GPUs offer several primary memory spaces: global memory, shared memory, constant memory, and registers. The overall memory structure is illustrated in Figure 2.

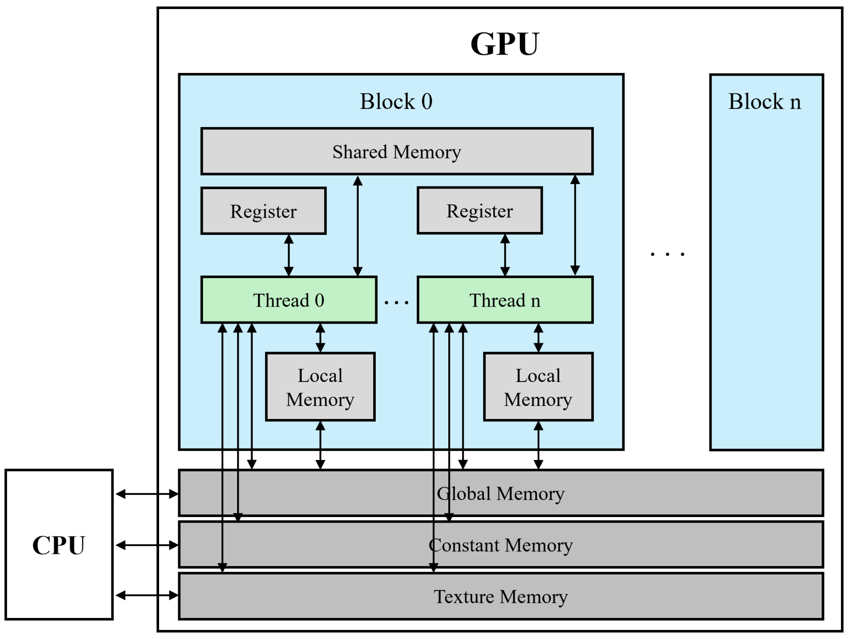

Figure 2. CUDA GPU memory architecture.

Global memory provides a large storage capacity but resides in VRAM and has the highest latency. Accessing global memory through the memory bus can take hundreds of clock cycles. Efficient use of global memory requires coalesced accesses, where consecutive threads within a warp access consecutive memory addresses, allowing hardware to combine these into fewer transactions, thus improving performance.

Shared memory, in contrast, is smaller (typically 48 KB per SM) on-chip memory offering significantly lower latency, comparable to an L1 cache. Threads within the same block share this memory, allowing faster access than global memory. However, shared memory is partitioned into multiple banks, and simultaneous access by multiple threads in a warp to the same bank results in bank conflicts, serializing the accesses. Hence, careful data layout in shared memory is necessary to avoid conflicts [9,12,13].

Constant memory is a read-only cache optimized for broadcast. Although small (around 64 KB) and residing physically in global memory, constant memory provides fast cached access if all threads within a warp read the same address simultaneously. It is particularly suitable for fixed data such as encryption round keys. For instance, in ARIA-CTR implementations, round keys totaling a few hundred bytes can be efficiently stored in constant memory, ensuring minimal access latency across all threads [14]. However, constant memory becomes less beneficial in scenarios like exhaustive key searches, where each thread tests different keys.

Finally, registers offer the fastest access speed for thread-local variable storage. CUDA attempts to keep frequently used variables, such as intermediate encryption state bytes, in registers. If the number of registers required exceeds availability, data spills into local memory, physically located in global memory, causing performance degradation. Therefore, minimizing register usage per thread to avoid spilling is crucial for performance optimization.

2.3. CTR Mode Encryption and Exhaustive Key Searches

Block cipher algorithms use a ’block cipher mode of operation’ when encrypting data, and one such mode is CTR (counter) mode [15]. CTR mode enables a block cipher to function similarly to a stream cipher. In this mode, a continuously increasing number (counter) is encrypted block by block to generate a keystream, which is then combined with the plaintext to produce ciphertext. Each block employs a distinct counter value, ensuring that even repeated plaintext blocks result in different ciphertext blocks, thus preventing repetitive data patterns from being exposed. Additionally, by setting an appropriate initial counter value (nonce), CTR mode prevents ciphertexts from overlapping, even if the same key is used across different encryption sessions. The detailed structure of CTR mode is illustrated in Figure 3.

A significant advantage of CTR mode is the ease of parallel processing due to the lack of dependency between blocks. Since the encryption results of previous blocks are not used to encrypt subsequent blocks, each block can be encrypted independently. This makes it highly efficient in high-speed parallel computing environments, such as multi-core CPUs or GPUs.

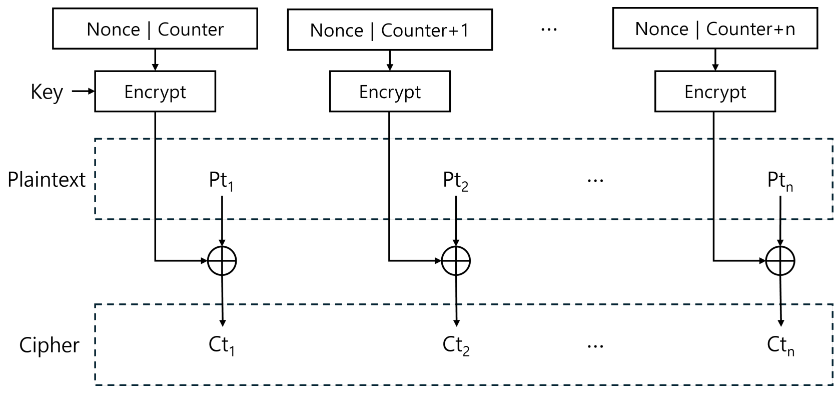

Figure 3. CTR mode among the block cipher operation modes.

On the other hand, exhaustive key search (also known as brute-force) attacks involve systematically testing all possible key values to discover the encryption key used in a cryptographic system. Attackers try to decrypt ciphertext or encrypt plaintext with each key candidate until they find the correct key by matching the result. Although theoretically guaranteed to succeed, this method becomes practically infeasible as the key length increases. For example, a 3-bit key has only $2^3 = 8$ possible combinations, making it trivial to test them all quickly. However, modern secure cryptographic algorithms such as ARIA, which uses 128-bit keys, yield 2^{128} (approximately 3.4×10^{38}) possible keys, rendering exhaustive searches practically impossible with current computing technology. Therefore, well-designed cryptographic algorithms utilize sufficiently long keys to minimize the success probability of exhaustive key-search attacks to negligible levels.

Nevertheless, advancements in computing power and parallelization techniques have significantly increased the speed of exhaustive key searches within certain practical boundaries. GPUs, in particular, are capable of performing thousands of operations simultaneously, substantially accelerating the key-search process by evaluating multiple key candidates concurrently. If a single CPU core can test a certain number of keys per second, a GPU with thousands of cores can theoretically test thousands of times more keys simultaneously. Due to this capability, GPU-based parallel processing, such as CUDA environments, is extensively utilized in situations requiring large-scale key searches, including cryptanalysis and password-cracking tasks, taking advantage of the exceptional parallel computing capabilities of GPUs.

2.4. Related Works

In [16], the ARIA and AES algorithms were implemented on a Nvidia GeForce 8800GTS GPU, and their parallel processing performance was compared. During the implementation, the GPU shared memory and registers were utilized efficiently, and performance was evaluated by dividing the data processed per thread into 8-bit and 32-bit units.

In the 8-bit implementation, each block consisted of 16 threads, with each thread designed to compute one byte of ciphertext by using intermediate states and round keys stored in shared memory. In contrast, the 32-bit implementation was structured so that each thread independently encrypted 32-bit units of data, allowing a comparative analysis of these two approaches.

The experimental results indicated that the 32-bit implementation achieved a throughput of 4.8 Gbps, significantly outperforming the 8-bit implementation, which reached only 214 Mbps. Additionally, when the same 32-bit implementation method was applied to the AES algorithm for performance comparison, ARIA demonstrated a higher throughput than AES.

Eum et al. [7] analyzed the optimal performance conditions for parallel implementations using an RTX 3060 GPU with the Nsight Compute profiler. They demonstrated a performance improvement of approximately 1.08 to 1.43 times for the substitution (S-box) operation when using shared memory. However, they pointed out that, due to the size characteristics of the ARIA algorithm's S-box, table expansion techniques aimed at avoiding GPU bank conflicts become inefficient when replicating tables according to bank size is not feasible.

In [6], although the authors did not focus on ARIA specifically, they optimized the AES algorithm for GPU implementations. They proposed reconstructing the AES T-table structure to be suitable for shared memory usage and suggested replicating the tables multiple times to distribute memory accesses, effectively addressing the issue of bank conflicts. These optimization techniques can also be effectively applied to GPU-based parallel implementations of the ARIA cipher.

3. Optimization Strategy of ARIA on GPU

3.1. Shared Memory Utilization Through Optimized S-box Table

The AES encryption algorithm fundamentally relies on a substitution operation known as the Substitution box, or S-box, to achieve non-linearity in encryption. An S-box specifically transforms an 8-bit input value into a predefined, pre-computed 8-bit output value. While this substitution step is effective, frequent computational repetitions of this operation can introduce inefficiencies. To address and optimize computational efficiency, AES implements an advanced method called T-table optimization. The T-table optimization method integrates the substitution operation with additional subsequent computations—such as Shiftrows and Mixcolumns—into a singular, unified pre-computed operation. Consequently, the original, relatively small 256-byte S-box is expanded significantly into a more substantial 4 KB T-table, enhancing processing speed by trading off increased memory usage [6,17].

Similarly, the ARIA encryption algorithm, which shares operational parallels with AES, also fundamentally utilizes substitution operations based on S-boxes to achieve security and complexity in encryption. Unlike AES, however, ARIA employs not just one, but four distinct and separate S-boxes to perform substitution. Due to this characteristic, ARIA can adopt a strategy analogous to AES's T-table optimization, wherein these substitution operations are expanded into pre-computed tables to achieve more efficient execution. Each of ARIA's four individual S-boxes expands from an initial size of 256 bytes to an expanded pre-computed table size of 1 KB. Consequently, considering ARIA's usage of four distinct S-boxes, the cumulative memory required for these expanded tables totals approximately 4 KB.

Given that substitution operations inherently involve frequent and repeated accesses to pre-computed tables, optimizing the efficiency of these memory accesses becomes critically important. Memory access operations, especially on GPU architectures, inherently have greater latency than simple register-to-register computations. Hence, strategies to minimize memory latency are of paramount importance in high-performance encryption implementations. A commonly employed strategy involves storing the S-box tables in GPU shared memory. However, a notable challenge when leveraging GPU shared memory is the occurrence of memory-bank conflicts. Bank conflicts become particularly problematic when threads within the GPU execute irregular or random memory access patterns—a typical scenario encountered during S-box lookups. Because substitution inputs for encryption processes are intrinsically random and unpredictable, multiple threads frequently attempt concurrent access to identical memory banks within shared memory. Such concurrent accesses lead to severe bank conflicts, substantially degrading the overall computational performance of the GPU.

To effectively address and alleviate these bank conflicts, we adopt the sophisticated method proposed by Tezcan C. [6], which involves replicating the lookup tables multiple times within the shared memory. By replicating the tables, each individual thread is afforded independent, conflict-free access to separate table instances. GPUs inherently execute threads in groups known as warps, each warp comprising 32 threads. Therefore, replicating lookup tables exactly 32 times ensures that each thread within a warp accesses a separate memory bank independently, ideally eliminating bank conflicts altogether. However, this replication strategy has practical limitations, primarily due to the restricted capacity of GPU shared memory. To overcome this constraint, Tezcan C. [6] introduced a practical solution, which involves downsizing the original 4 KB table to a smaller, more manageable 1 KB table, replicated 32 times. This downsizing strategy successfully fits within GPU shared memory limits, totaling only 32 KB, thus satisfying performance and memory constraints simultaneously.

In the context of ARIA, the expanded S-box tables occupy a total of 4 KB, making direct replication similar to AES initially impractical within a typical GPU's shared memory capacity. Therefore, this research proposes an optimization method similar to the approach described above by merging the four expanded ARIA S-box tables into a unified 32-bit optimized S-box table, significantly reducing the table size from 4 KB to 1 KB. By adopting this unified table strategy, it becomes practically feasible to replicate the optimized ARIA table exactly 32 times within the available shared memory of the GPU. This approach occupies a total of just 32 KB, efficiently conforming to standard GPU memory limitations while effectively mitigating bank conflicts. The detailed implementation process and structure of this optimized, unified ARIA S-box table are shown in Figure 4 and Algorithm 1.

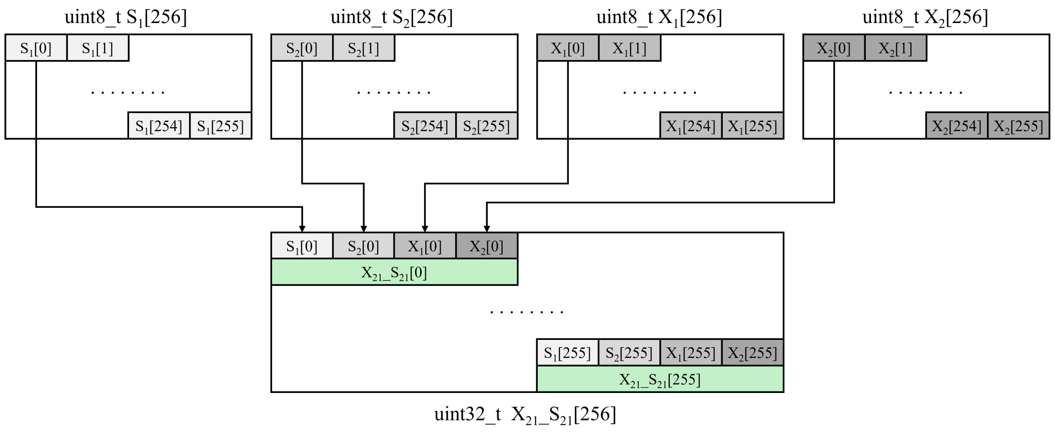

Figure 4. Generation process of the proposed uint32_t $X_{21}\_S_{21}[256]$ table.

Algorithm 1 x21_s21 Table construction

Input: $S_1[256], S_2[256], X_1[256], X_2[256]$ (uint8_t)
Output: $X_{21}\_S_{21}[256]$ (uint32_t)

```
1: for $i \leftarrow 0$ to 255 do
2:     $tmp \leftarrow (S_1[i]\ \&\ 0x000000FF)$
3:     $tmp \leftarrow tmp \mid (S_2[i] \ll 8)$
4:     $tmp \leftarrow tmp \mid (X_1[i] \ll 16)$
5:     $tmp \leftarrow tmp \mid (X_2[i] \ll 24)$
6:     $X_{21\_S_{21}}[i] \leftarrow tmp$
7: end for
```

3.2. Efficient Output Reconstruction Using the __byte_perm() Function

The proposed method provides significant memory efficiency, but directly using the integrated table causes an issue, as it outputs a single 8-bit result instead of the originally intended full 32-bit output. Thus, using the reduced table without additional processing leads to a mismatch between the intended and actual output forms. An additional post-processing step is required to resolve this discrepancy, and minimizing the overhead incurred during this step is important. The CUDA environment offers efficient support for byte-level operations and rearrangements through the *__byte_perm()* function. This function is capable of generating new 32-bit values by selectively rearranging bytes from two separate 32-bit inputs. It is particularly well suited for operations such as endian conversion, data packing, and unpacking, where data restructuring is required.

In this research, we leveraged the characteristics of the *__byte_perm()* function to accurately reconstruct the original expanded 32-bit output from the single 8-bit result obtained from the reduced table. Specifically, the single 8-bit output value extracted from the table was precisely expanded into the desired 32-bit format using the *__byte_perm()* function. This approach enables the rapid and efficient retrieval of the desired output values while minimizing computational overhead, without the need for complex arithmetic or logical operations. The detailed implementation process and data transfer paths employed in this step are illustrated in Figure 5.

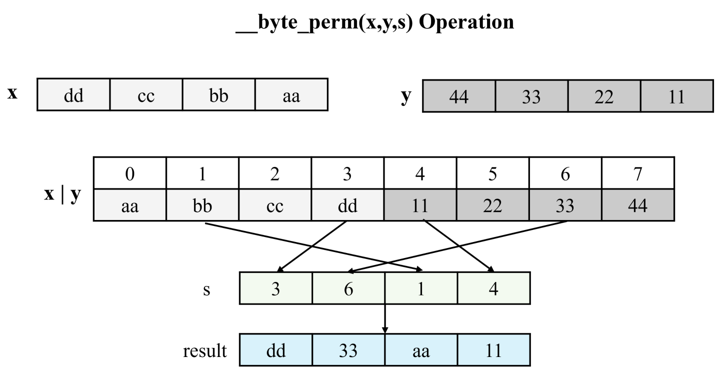

Figure 5. Operation process of the *__byte_perm()* function.

3.3. Overall Structure of the Proposed Optimization Technique

Figure 6 illustrates how each of the 32 threads within a warp is assigned a specific memory bank based on its thread index and accesses the *x21_s21 table* through this bank to perform substitution operations. The process of replicating the table into each bank can be observed in Algorithm 2. Specifically, if the thread index within each block (*threadIdx.x*, indicating the thread identifier within a CUDA block) is smaller than the table size (less than 256), the corresponding value from the global memory table (*tG[threadIdx.x]*) is copied 32 times into shared memory. The subsequent use of these replicated tables is further detailed in Algorithm 3.

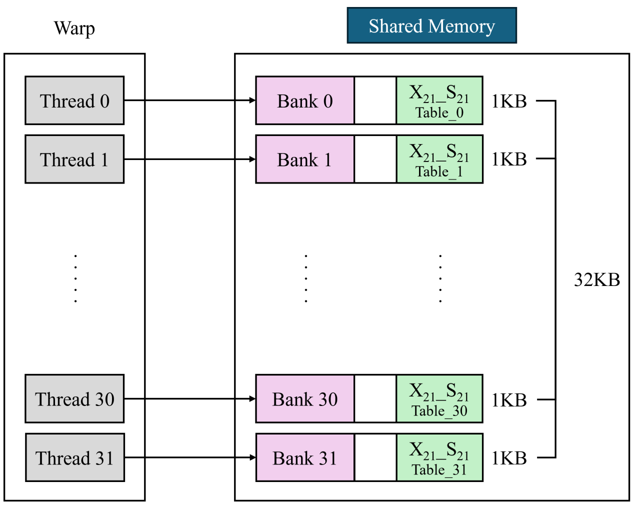
Warp
Shared Memory
Thread 0
Thread 1
Thread 30
Thread 31
Bank 0
Bank 1
Bank 30
Bank 31
X_{21}_S_{21} Table_0
X_{21}_S_{21} Table_1
X_{21}_S_{21} Table_30
X_{21}_S_{21} Table_31
1KB
32KB

Figure 6. Thirty-two threads in a warp accessing tables replicated across memory banks.

Algorithm 2 Shared memory table initialization

```
1: for threadIdx.x = 0 to TABLE_SIZE − 1 in parallel do
2:     for bankIndex = 0 to SHARED_MEM_BANK_SIZE − 1 do
3:         $tS$[threadIdx.x][bankIndex] ← $tG$[threadIdx.x]
4:     end for
5: end for
```

The subsequent use of these replicated tables is further detailed in Algorithm 3. The function performs substitution operations using the `x21_s21` table stored in shared memory. For instance, in the case of `pt[0]`, the 32-bit value is split into bytes, and each byte is used as an index in the `tS` table. The substitution values are then accessed through banks determined by the `wTindex`. For example, if the thread index is 3, then `wTindex` is also 3, and bank 3 is used. As a result, each thread accesses a different bank, enabling conflict-free parallel substitution.

The substituted values are then reassembled using the *__byte_perm()* function, producing the same result as would be obtained from a conventional substitution using the extended S-box. This process is illustrated in Figure 7.

Algorithm 3 Algorithmic description of `device_SBL1_M_func`

```
procedure DEVICE_SBL1_M_FUNC($pt, tS, wTindex$)
    for $i \leftarrow 0$ to 3 do
        $byte0 \leftarrow (pt[i] \gg 24)$
        $byte1 \leftarrow (pt[i] \gg 16)\&0xFF$
        $byte2 \leftarrow (pt[i] \gg 8)\&0xFF$
        $byte3 \leftarrow pt[i]\&0xFF$
        $pt[i] \leftarrow$ __byte_perm$(0, tS[byte0][wTindex], S1\_EXT) \oplus$
          __byte_perm$(0, tS[byte1][wTindex], S2\_EXT) \oplus$
          __byte_perm$(0, tS[byte2][wTindex], X1\_EXT) \oplus$
          __byte_perm$(0, tS[byte3][wTindex], X2\_EXT)$
    end for
end procedure
```

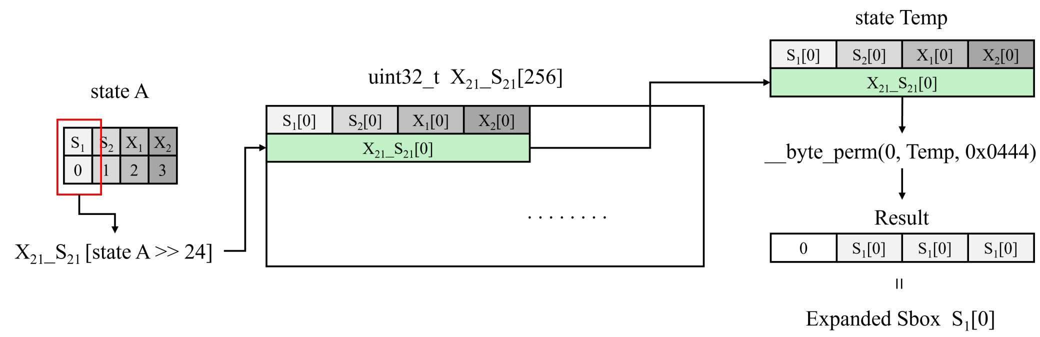
state A
S_1 S_2 X_1 X_2
0 1 2 3
$X_{21}\_S_{21}$ [state A >> 24]
uint32_t $X_{21}\_S_{21}[256]$
$S_1[0]$ $S_2[0]$ $X_1[0]$ $X_2[0]$
$X_{21}\_S_{21}[0]$
....
state Temp
$S_1[0]$ $S_2[0]$ $X_1[0]$ $X_2[0]$
$X_{21}\_S_{21}[0]$
__byte_perm(0, Temp, 0x0444)
Result
0 $S_1[0]$ $S_1[0]$ $S_1[0]$
=
Expanded Sbox $S_1[0]$

Figure 7. Substitution operation process using unified S-box and *__byte_perm()* function. The red box indicates the appropriate index of the look-up table.

3.4. Memory Optimization for Exhaustive Key Searches

An *exhaustive key search* is an approach in cryptanalysis that attempts encryption operations with all possible keys. For example, when using a 128-bit key, all 2^{128} possible keys must be searched exhaustively. In typical parallel implementations of CTR mode encryption, all threads share the same encryption key, so the key-expansion operation is performed only once, and the same round keys can be shared across all threads. However, in an exhaustive key-search scenario, each thread must use a different key. Thus, each thread independently performs the key-expansion operation, significantly increasing the computational overhead. Furthermore, since each thread needs to store its complete set of round keys, memory usage significantly increases.

To address this memory consumption issue, we applied an *on-the-fly* key-expansion method. Typically, key expansion is performed in advance to generate and store all required round keys, which are later referenced during the encryption process. In contrast, the *on-the-fly* method generates only the required round key immediately during each encryption round, applying it directly to the ongoing encryption operation. Consequently, this approach minimizes the storage of precomputed round keys, significantly reducing overall memory usage and enabling more efficient parallel execution in scenarios involving a large number of keys.

For 128-bit block encryption involving 12 rounds, one additional round key is needed, resulting in a total of 13 round keys (each 32 bits). In a conventional implementation, each thread stores all round keys, requiring $13 \times 4 = 52$ bytes of memory per thread. This implies that the total memory usage scales proportionally to the total number of threads utilized in kernel execution. In contrast, the *on-the-fly* method calculates and stores only one round key (32 bits, 4 bytes) at a time per thread. This reduces the per-thread memory usage to 4 bytes, significantly decreasing overall memory usage. As a result, more threads can be allocated and executed simultaneously, thereby improving the efficiency of exhaustive key-search operations.

4. Evaluation

In this section, we evaluate the GPU-based implementations of ARIA in both CTR and exhaustive key-search modes. We describe the experimental setup, present the profiling results and detailed performance measurements, and finally provide comprehensive comparative analyses.

4.1. Performance Measurement Environment and Measurement Method

In this paper, we measured the performance of GPU-based parallel implementations of two modes: exhaustive key-search (ES) mode and counter (CTR) mode encryption. Two different GPU environments were used for performance evaluation. The first environment was a notebook equipped with a Nvidia GeForce RTX 3060 Mobile GPU [18], characterized by lower power consumption and relatively limited memory capacity (6 GB), offering high portability and power efficiency. The second environment was a desktop setup with a Nvidia GeForce RTX 4080 GPU [19], providing high computational performance through higher power limits and abundant memory (16 GB).

To ensure accurate performance measurements, CUDA-based kernels running on the GPUs were utilized. Both modes performed a total of 2^{35} (approximately 34.36 billion) operations, which were parallelized using CUDA kernels configured with 1024 blocks and 512 threads per block. This resulted in a total thread count of 524,288, with each thread uniformly assigned 65,536 operations (key candidates or encryption blocks).

In CTR mode, the total execution time for encrypting 2^{35} sequential blocks was measured. Each CUDA thread incrementally increased its counter value within its assigned range, performing LEA encryption operations until all threads completed their tasks, marking the kernel's termination and capturing the total execution time.

In ES mode, a similar approach was taken to measure the time required for exhaustively searching through 2^{35} key candidates. Each thread was assigned a specific range of keys to test via encryption operations. The execution time was measured immediately before and after the kernel execution to evaluate overall performance.

The detailed configuration of the experimental environment is summarized in Table 1.

Table 1. GPU-based performance measurement configuration.

Parameter	Value
CUDA Blocks	1024
Threads per Block	512
Total Thread Count	524,288
Key Range (2^{power})	2^{35}
Key Range (decimal)	34,359,738,368
Key Range per Thread	65,536
Total Encryptions	34,359,738,368

The performance data obtained through these experiments quantitatively highlight the performance differences between the RTX 3060 Mobile GPU and RTX 4080 Desktop GPU, particularly under workloads associated with exhaustive key searches and CTR mode encryption. These results serve as critical metrics for selecting appropriate GPUs when designing high-performance parallel encryption systems, taking into account workload requirements and operational constraints.

The ARIA-CTR/ARIA-ES experimental results were measured based on encrypting a total of 2^{35} (34,359,738,368) blocks, corresponding to approximately 4.398×10^{12} bits (128 bits $\times$ 2^{35}), with performance expressed in Gbps (Gigabits per second).

4.2. Profiling Results Analysis

Table 2 summarizes the key performance metrics measured by Nsight Compute for the ARIA-128 CTR mode implementation. Three different approaches were compared for storing the S-Box: (1) in *global memory*, (2) in *shared memory*, and (3) in *shared memory with minimized bank conflicts*. Since the focus is on relative performance rather than the absolute execution time, the table excludes the kernel runtime and highlights GPU utilization metrics such as memory throughput, compute throughput, and occupancy.

Table 2. Key Nsight Compute metrics for ARIA-128 CTR mode (S-Box placement), excluding execution time.

Metric	Global Memory S-Box	Shared Memory S-Box	Shared Mem. + Bank Conflict Removal
Memory Throughput (%)	90.69	96.76	95.71
Compute (SM) Throughput (%)	67.27	73.53	95.71

As shown in Table 2, using global memory for the S-Box (left column) resulted in a high memory throughput (about 90.69%), indicating that the kernel was strongly memory-bound. However, the achieved SM throughput remained lower (67.27%), suggesting that the global memory transactions incurred significant stalls or uncoalesced accesses, ultimately limiting overall performance.

Moving the S-Box to shared memory (middle column) led to higher memory throughput and a modest increase in SM throughput. Yet, the profiling results show notable bank conflicts (around 60% of wavefronts), resulting in additional stalls and preventing the kernel from fully utilizing the SM resources.

Finally, replicating the S-Box across banks (right column) greatly reduced or nearly eliminated shared memory-bank conflicts. This optimization increased the SM throughput to 95.71%, and the achieved occupancy climbed to 96.60%.

In conclusion, while shared memory provides faster access than global memory, bank conflicts can severely degrade performance. Properly distributing the S-Box across banks is essential for removing these conflicts and maximizing GPU utilization.

4.3. ARIA-CTR Performance Evaluation

We further evaluated performance based on the execution time. Similar to the previous analysis, we compared three configurations: (1) in *global memory*, (2) in *shared memory*, and (3) in *shared memory with minimized bank conflicts*.

The execution time results aligned with our profiling analysis. Specifically, the configuration using *global memory* exhibited the lowest throughput (Gbps). Merely transitioning from *global memory* to *shared memory* resulted in a performance increase of approximately 15% to 19%, demonstrating that utilizing shared memory alone effectively enhances performance.

Furthermore, when comparing the *shared memory* and *shared memory with minimized bank conflicts* configurations, we observed an additional performance improvement of around 19% to 21%. This indicates the significant impact bank conflicts have on performance when using shared memory.

Consequently, when comparing *global memory* with the optimized shared memory configuration (with minimized bank conflicts), the overall performance improvement ranged from approximately 39% to 43%. The detailed measurement results are given in Figure 8 and Table 3.

Table 3. ARIA-CTR performance on the RTX 3060 for different S-box placement methods.

Placement	Key Size	Time (s)	Throughput (Gbps)
Global Memory	128-bit	18.27	240.73
	192-bit	20.41	215.48
	256-bit	22.98	191.39
Shared Memory	128-bit	15.31	287.27
	192-bit	17.66	249.04
	256-bit	19.98	220.12
Shared Memory (No Bank Conflicts)	128-bit	12.77	344.40
	192-bit	14.65	300.21
	256-bit	16.43	267.68

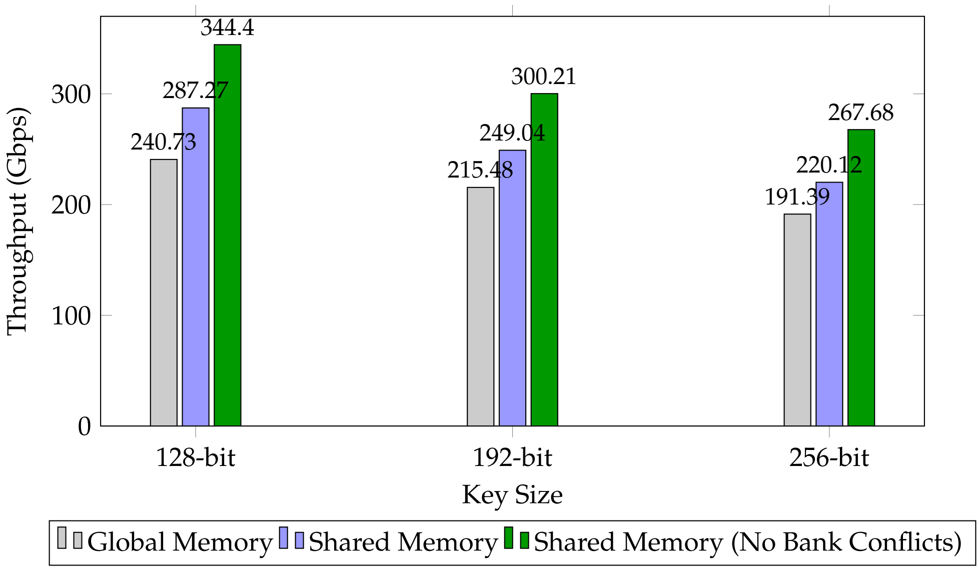

Figure 8. ARIA-CTR throughput on the RTX 3060 for different S-box placements.

4.4. ARIA-ES Performance Evaluation

We also evaluated performance based on the execution time for exhaustive key searches (ES), obtaining results consistent with those observed in CTR mode. Transitioning from *global memory* to *shared memory* yielded a performance improvement of approximately 11% to 13%. Additionally, by minimizing bank conflicts within *shared memory*, performance further improved by about 7% to 13%. In total, we observed an overall performance improvement ranging from approximately 21% to 24%. The detailed measurement results are presented in Figure 9 and Table 4.

Furthermore, since the ES implementation employs an *on-the-fly* key-expansion approach, only 4 bytes of memory are required per thread for round keys. The performance measurements used a block size of 1024 and 512 threads, resulting in a total memory requirement of 2,097,152 bytes for round keys during the kernel execution. In contrast, if all round keys were precomputed and stored for a 128-bit key length requiring storage for 13 rounds, the necessary memory would increase significantly to 27,262,976 bytes. Hence, our *on-the-fly* implementation theoretically reduces memory usage by approximately 92.31%.

Table 4. ARIA-ES performance on the RTX 3060 for different S-box placement methods.

Placement	Key Size	Time (s)	Throughput (Gbps)
Global Memory	128-bit	17.60	249.93
	192-bit	19.21	228.99
	256-bit	21.81	201.67
Shared Memory	128-bit	15.79	278.47
	192-bit	16.91	260.14
	256-bit	19.26	228.34
Shared Memory (No Bank Conflicts)	128-bit	13.96	315.04
	192-bit	15.79	278.53
	256-bit	17.54	250.74

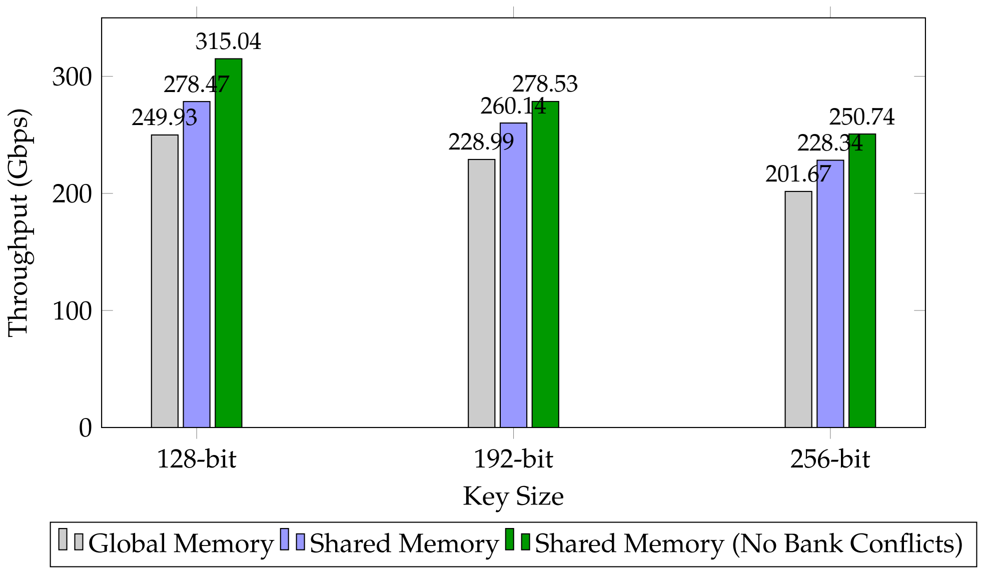

Figure 9. ARIA-ES throughput on the RTX 3060 for different S-box placements.

4.5. Overall Performance Comparison of ARIA-CTR/ES Modes

In this paper, we present the performance measurement results from existing ARIA implementations [7], along with those obtained using a modern GPU. This approach enables a clear comparison of performance improvements between prior research and our current study and suggests that performance results from contemporary high-performance GPUs can serve as valuable reference data for diverse application areas.

Previous studies also analyzed performance based on global and shared memory usage; however, they did not sufficiently address bank conflict issues associated with shared memory usage. In contrast, this study effectively resolves the bank conflict issue in shared memory, achieving a performance improvement of approximately 134.81% compared to previous implementations. These results clearly demonstrate that our proposed implementation method is more efficient and effective than earlier methods. The detailed performance comparison of ARIA-CTR is presented in Table 5.

Generally, a block cipher with a k-bit key can theoretically be broken by performing up to 2^k encryption operations. NIST recommends using key lengths of at least 112 bits until 2030. The total key space of ARIA-128 is 2^{128}.

The calculation of the key-search speed is as follows:

$$\text{Blocks processed per second} = \frac{\text{Throughput (Gbps)} \times 10^9}{\text{Block size (bits)}}$$

$$\text{Blocks processed per year} = \text{Blocks processed per second} \times \text{Seconds per year}$$

$$\text{Required number of GPUs} = \frac{\text{Total key space}}{\text{Blocks processed per year}}$$

According to the experimental results (Table 6), the RTX 3060 achieved a throughput of 315.04 Gbps in ARIA-ES mode with a 128-bit key. Given that one ARIA block is 128 bits, the RTX 3060 can process approximately 2.461×10^9 blocks per second, totaling around 7.762×10^{16} blocks annually. Considering the ARIA-128 total key space of 3.403×10^{38}, approximately 4.384×10^{21} RTX 3060 GPUs would be required to break ARIA-128 within one year.

Table 5. ARIA-CTR performance comparison (*: ours; ^{G}: global memory; ^{S}: shared memory).

GPU	Key Size	Time (s)	Throughput (Gbps)
RTX 3060 [7]	128-bit ^{G}	–	135.06
	128-bit ^{S}	–	146.67
RTX 3060 *	128-bit	12.77	344.40
	192-bit	14.65	300.21
	256-bit	16.43	267.68
RTX 4080 *	128-bit	2.87	1532.42
	192-bit	3.28	1340.87
	256-bit	3.70	1188.66

Table 6. ARIA-ES performance comparison (*: ours).

GPU	Key Size	Time (s)	Throughput (Gbps)
RTX 3060 *	128-bit	13.96	315.04
	192-bit	15.79	278.53
	256-bit	17.54	250.74
RTX 4080 *	128-bit	3.22	1365.84
	192-bit	3.67	1198.37
	256-bit	4.05	1085.93

For ARIA-192 with a throughput of 278.53 Gbps, the RTX 3060 can process approximately 2.176×10^9 blocks per second and 6.862×10^{16} blocks annually, requiring approximately 9.147×10^{40} GPUs to exhaustively search the entire key space in one year.

In the case of ARIA-256 with a throughput of 250.74 Gbps, the GPU can process approximately 1.959×10^9 blocks per second and 6.178×10^{16} blocks annually, necessitating approximately 1.874×10^{60} GPUs to break ARIA-256 within one year.

Using the RTX 4080, performance significantly improves. For ARIA-128, with a throughput of 1365.84 Gbps, the RTX 4080 can process about 1.067×10^{10} blocks per second and 3.365×10^{17} blocks annually, reducing the number of GPUs required to 1.011×10^{21}. For ARIA-192 at 1198.37 Gbps, it can process approximately 9.362×10^9 blocks per second and 2.952×10^{17} annually, requiring 2.126×10^{40} GPUs. Lastly, for ARIA-256 at 1085.93 Gbps, the RTX 4080 processes around 8.484×10^9 blocks per second and 2.675×10^{17} blocks annually, reducing the required GPUs to 4.328×10^{59}.

These comparisons clearly illustrate the substantial performance gains achievable with newer-generation GPUs. Nevertheless, given the current pace of technological advancements, using GPUs alone to perform exhaustive key searches on ARIA with key lengths of 128 bits or greater will remain practically infeasible for several decades. Consequently, ARIA implementations are considered sufficiently secure against GPU-based exhaustive key-search attacks in the foreseeable future.

5. Conclusions

In this paper, we proposed optimized implementations of ARIA-CTR and ARIA-ES modes on GPUs. Since CTR and ES modes can be computed independently, leveraging parallel computations on multi-core architectures such as GPUs effectively improved performance. To optimize CTR mode, we actively utilized shared memory. Due to its size, the original extended 4 KB S-box table in ARIA could not be replicated across all GPU shared memory banks. Therefore, we proposed a method that compresses four separate 1 KB S-box tables into a single 1 KB S-box. By employing the *__byte_perm()* function in this process, we achieved the same output values as the original, without incurring significant overhead. For ARIA-ES mode, we applied an on-the-fly technique, computing the round keys dynamically as needed rather than precomputing them in advance. This approach significantly reduced memory usage and minimized memory accesses. The experimental results confirmed that minimizing memory access directly influenced performance. Furthermore, as the key length increased, the performance gap between CTR and ES modes decreased. This trend occurred because the memory access costs and round key usage increase in CTR mode with longer keys, whereas ES mode maintains efficiency through minimal memory access. These results validated the effectiveness of our proposed optimization method. In conclusion, this study demonstrated that memory access frequency substantially impacts GPU performance, highlighting the need for more in-depth research into memory access optimization for GPU-based cryptographic algorithms. As future research directions, we will consider extending our optimization methods to other block ciphers and evaluating performance across various GPU architectures, such as AMD GPUs, to further generalize our findings.

Author Contributions: Software, S.E. and H.S.; Writing—original draft, S.E.; Writing—review & editing, S.E., M.S., S.K. and H.S.; Supervision, H.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was financially supported by Hansung University.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

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

References

1. *KS X 1213-1*; Information Technology—Security Techniques—128-Bit Block Cipher Algorithm ARIA—Part 1: General. Korean Agency for Technology and Standards (KATS): Seoul, Republic of Korea, 2004.
2. Kwon, D.; Kim, J.; Park, S.; Sung, S.H.; Sohn, Y.; Song, J.H.; Yeom, Y.; Yoon, E.J.; Lee, S.; Lee, J.; et al. New block cipher: ARIA. In Proceedings of the International Conference on Information Security and Cryptology, Seoul, Republic of Korea, 27–28 November 2003; pp. 432–445.
3. Kim, J.; Lee, J.; Kim, C.; Lee, J.; Kwon, D. A Description of the ARIA Encryption Algorithm. RFC 5794, 2010. Available online: https://www.rfc-editor.org/info/rfc5794 (accessed on 14 May 2025).
4. Daemen, J.; Rijmen, V. AES Proposal: Rijndael. 1999. Available online: https://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf (accessed on 14 May 2025).
5. Song, J.; Seo, S.C. Efficient parallel implementation of CTR mode of ARX-based block ciphers on ARMv8 microcontrollers. *Appl. Sci.* **2021**, *11*, 2548. [CrossRef]
6. Tezcan, C. Optimization of advanced encryption standard on graphics processing units. *IEEE Access* **2021**, *9*, 67315–67326. [CrossRef]
7. Eum, S.; Kim, H.; Kwon, H.; Sim, M.; Song, G.; Seo, H. Parallel implementations of ARIA on ARM processors and graphics processing unit. *Appl. Sci.* **2022**, *12*, 12246. [CrossRef]

8. Xiao, L.; Li, Y.; Ruan, L.; Yao, G.; Li, D. High performance implementation of aria encryption algorithm on graphics processing units. In Proceedings of the 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, Zhangjiajie, China, 13–15 November 2013; pp. 504–510.
9. Fang, M.; Fang, J.; Zhang, W.; Zhou, H.; Liao, J.; Wang, Y. Benchmarking the GPU memory at the warp level. *Parallel Comput.* **2018**, *71*, 23–41. [CrossRef]
10. NVIDIA. CUDA Toolkit Documentation 12.8, 2025. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory-3-0 (accessed on 1 April 2025).
11. Eum, S.; Kim, H.; Song, M.; Seo, H. Optimized Implementation of Argon2 Utilizing the Graphics Processing Unit. *Appl. Sci.* **2023**, *13*, 9295. [CrossRef]
12. Mei, X.; Zhao, K.; Liu, C.; Chu, X. Benchmarking the memory hierarchy of modern GPUs. In Proceedings of the Network and Parallel Computing: 11th IFIP WG 10.3 International Conference, NPC 2014, Ilan, Taiwan, 18–20 September 2014; Proceedings 11; Springer: Berlin/Heidelberg, Germany, 2014; pp. 144–156.
13. Mei, X.; Chu, X. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *28*, 72–86. [CrossRef]
14. Eum, S.W.; Kim, H.J.; Kwon, H.D.; Jang, K.B.; Kim, H.J.; Seo, H.J. Implementation of SM4 block cipher on CUDA GPU and its analysis. In Proceedings of the 2022 International Conference on Platform Technology and Service (PlatCon), Jeju, Republic of Korea, 22–24 August 2022; pp. 71–74.
15. Lipmaa, H.; Rogaway, P.; Wagner, D. CTR-mode encryption. In *First NIST Workshop on Modes of Operation*; Citeseer: College Park, MD, USA, 2000; Volume 39.
16. Yeom, Y.; Cho, Y.; Yung, M. High-Speed Implementations of Block Cipher ARIA Using Graphics Processing Units. In Proceedings of the 2008 International Conference on Multimedia and Ubiquitous Engineering (MUE 2008), Busan, Republic of Korea, 24–26 April 2008; pp. 271–275. [CrossRef]
17. Lee, W.K.; Seo, H.J.; Seo, S.C.; Hwang, S.O. Efficient implementation of AES-CTR and AES-ECB on GPUs with applications for high-speed FrodoKEM and exhaustive key search. *IEEE Trans. Circuits Syst. II Express Briefs* **2022**, *69*, 2962–2966. [CrossRef]
18. NVIDIA. GeForce RTX 30 Series Laptops, 2025. Available online: https://www.nvidia.com/en-us/geforce/laptops/30-series/#specs (accessed on 9 April 2025).
19. NVIDIA. GeForce RTX 4080 Series, 2025. Available online: https://www.nvidia.com/ko-kr/geforce/graphics-cards/40-series/rtx-4080-family/ (accessed on 9 April 2025).