applied sciences

MDPI

Article

Parallel Implementations of ARIA on ARM Processors and Graphics Processing Unit

Siwoo Eum, Hyunjun Kim, Hyeokdong Kwon, Minjoo Sim, Gyeongju Song and Hwajeong Seo *

Division of IT Convergence Engineering, Hansung University, Seoul 02876, Republic of Korea
* Correspondence: hwajeong@hansung.ac.kr; Tel.: +82-760-8033

Abstract: The ARIA block cipher algorithm is Korean standard, IETF standard (RFC 5794), and part of the TLS/SSL protocol. In this paper, we present the parallel implementation of ARIA block cipher on ARMv8 processors and GPU. The ARMv8 processor is the latest 64-bit ARM architecture and supports ASIMD for parallel implementations. With this feature, 4 and 16 parallel encryption blocks are implemented to optimize the substitution layer of ARIA block cipher using four different Sboxes. Compared to previous works, the performance was improved by 2.76× and 8.73× at 4-plaintext and 16-plaintext cases, respectively. We also present optimal implementation on GPU architectures. GPUs are highly parallel programmable processors featuring maximum arithmetic and memory bandwidth. Optimal settings of ARIA block cipher implementation on GPU were analyzed using the Nsight Compute profiler provided by Nvidia. We found that using shared memory reduces the execution timing when performing substitution operations with Sbox tables. When using many threads with shared memory instead of global memory, it improves performance by about 1.08∼1.43×. Additionally, techniques using table expansion to minimize bank conflicts have been found to be inefficient when tables cannot be copied by the size of the bank. We measured the performance of ARIA block ciphers implemented with various settings. This represents an optimized GPU implementation of the ARIA block cipher.

Keywords: ARIA block cipher; ARMv8; GPU; parallel computation; software implementation; Nsight compute

Citation: 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. https://doi.org/10.3390/app122312246

Academic Editor: Paris Kitsos

Received: 21 October 2022
Accepted: 25 November 2022
Published: 30 November 2022

1. Introduction

Today, the size of data is getting bigger and the internet speed is getting faster. This causes a lot of data to be encrypted quickly. In line with this, hardware is developing rapidly, and the development of hardware enables fast operation and provides various functions.

ARMv8 is the latest 64-bit ARM architecture. ARMv8 supports Advanced Single Instruction Multiple Data (ASIMD), which is also known as NEON engine. ASIMD is an instruction that can perform arithmetic operations on multiple data in parallel. In [1], the parallel encryption of AES block ciphers [2] is performed, showing a 5% performance improvement over the ASIMD-based Linux kernel implementation. In [3], they presented optimized format alignment and round function layer for SM4 block cipher on ARMv8 architectures. In [4], they utilized TBL/TBX instructions to perform fast multiplication for format-preserving encryption on ARMv8 architectures.

Graphics processing units (GPUs) have become an integral part of today's computing systems. Parallel implementations of block ciphers using GPU capabilities are steadily progressing. Recently, the parallel implementation of block cipher using ARX structure was introduced in [5], and parallel implementation of AES using SPN structure was introduced in [6,7].

The ARIA block cipher [8] was developed in 2003. The algorithm is Korean standard, IETF standard (RFC 5794), and part of the TLS/SSL protocol. The ARIA block cipher is designed with an SPN (substitution permutation network) structure. It is designed as

a substitution layer using four different Sboxs, a diffusion layer, and an addroundkey layer [9]. Previous studies implemented ARIA block ciphers on low-end processors. Few implementation works have been carried out on high-end processors.

In this paper, we firstly present parallel implementations of the ARIA block cipher on ARMv8 processor. For the efficient implementation of the substitution layer, we explored two approaches of 4-PT (plaintext) and 16-PT, where 4-PT and 16-PT indicate that when the block size of the encryption algorithm is 128-bit, 4 (4 × 128-bit) and 16 (16 × 128-bit) blocks are encrypted in parallel. Secondly, we optimized parallel implementations of the ARIA block cipher on GPU (Nvidia GTX 3060). GPUs provide several types of memory space. In this paper, each implementation is evaluated by loading and using Sbox in different memory types, including global, shared, and constant. Furthermore, a method using extended Sbox to solve the bank conflict problem is introduced in [6]. We also explored this technique for ARIA block cipher in this paper.

1.1. Contributions

1.1.1. Parallel ARIA Implementation on ARMv8 Processor

We are the first to implement a parallel implementation of an ARIA block cipher on an ARMv8 processor. In order to efficiently use TBL instructions in an ARIA substitution process that uses an Sbox where each byte is different, 4 or 16 plaintext blocks are implemented in parallel. The LD4 instruction provided by ARMv8 optimizes the process of sorting plaintext blocks for parallel implementation.

1.1.2. Parallel ARIA Implementation on GPU

We are the first to implement and analysis a parallel implementation of an ARIA block cipher on a GPU. In the parallel implementation, Sbox is loaded into shared memory provided by the GPU for comparative analysis. The latency of the memory provided by the GPU is different. Depending on the implementation, high-latency memory may perform better than low-latency memory. Types of memory used are global memory, shared memory, and constant memory. We investigated the optimal environment by comparing and analyzing various factors (memory types, threads, blocks) that can affect performance in GPU implementations.

1.2. Previous Implementations of ARIA Block Cipher

There are many ARIA implementations on various environments. We firstly explore previous implementations, especially on other embedded processors.

Yang et al. [10] presented hardware architecture of ARIA block cipher. It divided plaintext into eight16-bit blocks to make smaller hardware size. They proposed a new design for the substitution and the memory block. The presented implementation used Verilog-HDL, and the proposed ARIA-128 implementation took 400 cycles at encryption step.

Ryu et al. [11] showed a 32-bit structure small hardware implementation of ARIA block cipher. Since it uses a 32-bit input value unit, they redesigned four kinds of Sboxes. The proposed 32-bit ARIA operator is on 0.25 µm standard CMOS cell process. The result takes 278 clock cycles for the ARIA-128 operation.

Lee and Choi [12] proposed a 16-bit optimization design for ARIA block cipher. They proposed a 16-bit computation for ARIA diffusion layer from 32-bit optimized technique [13]. It mainly used matrix multiplication of matrix form 16 × 16 involutional block diagonal matrix and 16 × 16 involutional matrix. Additionally, it has its own Sbox with 8 × 32 lookup table. The proposed implementation takes about 600 microseconds for ARIA-128 encryption on the target platform, Atmel ATmega2560 microcontrollers.

Seo et al. [14] targeted two processors, 16-bit MSP430 and 32-bit ARM Cortex-M3. It provided two optimized implementations. First, on the MSP430 processor, they mainly used a 16-bit word-wise operator to implement ARIA block cipher. Second, on the ARM Cortex-M3, implementation used an 8 × 32 lookup-table-based implementation, but was further optimized by effective memory access. The proposed method rescheduled memory

access to fully utilize three-stage pipelining. In addition, they proposed optimized implementation with counter mode of operation that applied precomputation techniques. These proposed ARIA-128 implementations took 209 and 96 cycles per byte on MSP430 and ARM Cortex-M3 processors, respectively.

Kwak et al. [15] shows several kinds of block cipher implementations, but on the same target platform as the 32-bit RISC-V processor. The RISC-V processor has limited registers, so they proposed efficient registers scheduling. To implement optimized ARIA block cipher, it also used a lookup table at the substitution layer. The result of optimized ARIA-128 implementation on RISC-V took 295 cycles per byte for encryption.

Lee et al. [16] also targeted the 32-bit RISC-V processor, but its approach is different. The proposed method used only 10 kinds of RISC-V instructions. It did not use a lookup table for the substitution step. However, it made new architecture of ARIA substitution. For this implementation, most of the operations used composite fields. The operation was performed on SPIKE simulator, and its ARIA-128 implementation took 319 clock cycles.

2. Related Work

2.1. ARIA Block Cipher

The block length of ARIA block cipher is 128 bits, and the key length is 128, 192, and 256 bits. Depending on the length of each key, the encryption process consists of 12, 14, and 16 rounds. Each round of the ARIA block cipher consists of the following three parts. First, the round key addition layer is XORed with the 128-bit round key. Second, the substitution layer uses two types of substitution layers. Each substitution layer uses precalculated values of S_1, S_2, and their inverses (i.e., S_1^{-1}, S_2^{-1}). Figure 1 shows ARIA block cipher structure and two types of substitution layers. Odd rounds use Type 1 and even rounds use Type 2. Lastly, the diffusion layer is a simple linear map which is an involution. The diffusion layer is given by

$$(x_0, x_1, \cdots, x_{15}) \rightarrow (y_0, y_1, \cdots, y_{15}),$$

where

$$\begin{aligned}
y_0 &= x_3 \oplus x_4 \oplus x_6 \oplus x_8 \oplus x_9 \oplus x_{13} \oplus x_{14}, & y_8 &= x_0 \oplus x_1 \oplus x_4 \oplus x_7 \oplus x_{10} \oplus x_{13} \oplus x_{15},\\
y_1 &= x_2 \oplus x_5 \oplus x_7 \oplus x_8 \oplus x_9 \oplus x_{12} \oplus x_{15}, & y_9 &= x_0 \oplus x_1 \oplus x_5 \oplus x_6 \oplus x_{11} \oplus x_{12} \oplus x_{14},\\
y_2 &= x_1 \oplus x_4 \oplus x_6 \oplus x_{10} \oplus x_{11} \oplus x_{12} \oplus x_{15}, & y_{10} &= x_2 \oplus x_3 \oplus x_5 \oplus x_6 \oplus x_8 \oplus x_{13} \oplus x_{15},\\
y_3 &= x_0 \oplus x_5 \oplus x_7 \oplus x_{10} \oplus x_{11} \oplus x_{13} \oplus x_{14}, & y_{11} &= x_2 \oplus x_3 \oplus x_4 \oplus x_7 \oplus x_9 \oplus x_{12} \oplus x_{14},\\
y_4 &= x_0 \oplus x_2 \oplus x_5 \oplus x_8 \oplus x_{11} \oplus x_{14} \oplus x_{15}, & y_{12} &= x_1 \oplus x_2 \oplus x_6 \oplus x_7 \oplus x_9 \oplus x_{11} \oplus x_{12},\\
y_5 &= x_1 \oplus x_3 \oplus x_4 \oplus x_9 \oplus x_{10} \oplus x_{14} \oplus x_{15}, & y_{13} &= x_0 \oplus x_3 \oplus x_6 \oplus x_7 \oplus x_8 \oplus x_{10} \oplus x_{13},\\
y_6 &= x_0 \oplus x_2 \oplus x_7 \oplus x_9 \oplus x_{10} \oplus x_{12} \oplus x_{13}, & y_{14} &= x_0 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_9 \oplus x_{11} \oplus x_{14},\\
y_7 &= x_1 \oplus x_3 \oplus x_6 \oplus x_8 \oplus x_{11} \oplus x_{12} \oplus x_{13}, & y_{15} &= x_1 \oplus x_2 \oplus x_4 \oplus x_5 \oplus x_8 \oplus x_{10} \oplus x_{15}.
\end{aligned}$$

In [8], an efficient 8-bit-based diffusion layer implementation method was introduced. It reduces the number of operations to 76 XOR operations using four additional variables $(T_1, \cdots, T_4)$ as follows.

$$
\begin{aligned}
T_1 &= x_3 \oplus x_4 \oplus x_9 \oplus x_{14}, & T_2 &= x_2 \oplus x_5 \oplus x_8 \oplus x_{15} \\
y_0 &= x_6 \oplus x_8 \oplus x_{13} \oplus T_1, & y_1 &= x_7 \oplus x_9 \oplus x_{12} \oplus T_2 \\
y_5 &= x_1 \oplus x_{10} \oplus x_{15} \oplus T_1, & y_4 &= x_0 \oplus x_{11} \oplus x_{14} \oplus T_2 \\
y_{11} &= x_2 \oplus x_7 \oplus x_{12} \oplus T_1, & y_{10} &= x_3 \oplus x_6 \oplus x_{13} \oplus T_2 \\
y_{14} &= x_0 \oplus x_5 \oplus x_{11} \oplus T_1, & y_{15} &= x_1 \oplus x_4 \oplus x_{10} \oplus T_2
\end{aligned}
$$

$$
\begin{aligned}
T_3 &= x_1 \oplus x_6 \oplus x_{11} \oplus x_{12}, & T_4 &= x_0 \oplus x_7 \oplus x_{10} \oplus x_{13} \\
y_2 &= x_4 \oplus x_{10} \oplus x_{15} \oplus T_3, & y_3 &= x_5 \oplus x_{11} \oplus x_{14} \oplus T_4 \\
y_7 &= x_3 \oplus x_8 \oplus x_{13} \oplus T_3, & y_6 &= x_2 \oplus x_9 \oplus x_{12} \oplus T_4 \\
y_9 &= x_0 \oplus x_5 \oplus x_{14} \oplus T_3, & y_8 &= x_1 \oplus x_4 \oplus x_{15} \oplus T_4 \\
y_{12} &= x_2 \oplus x_7 \oplus x_9 \oplus T_3, & y_{13} &= x_3 \oplus x_6 \oplus x_8 \oplus T_4
\end{aligned}
$$

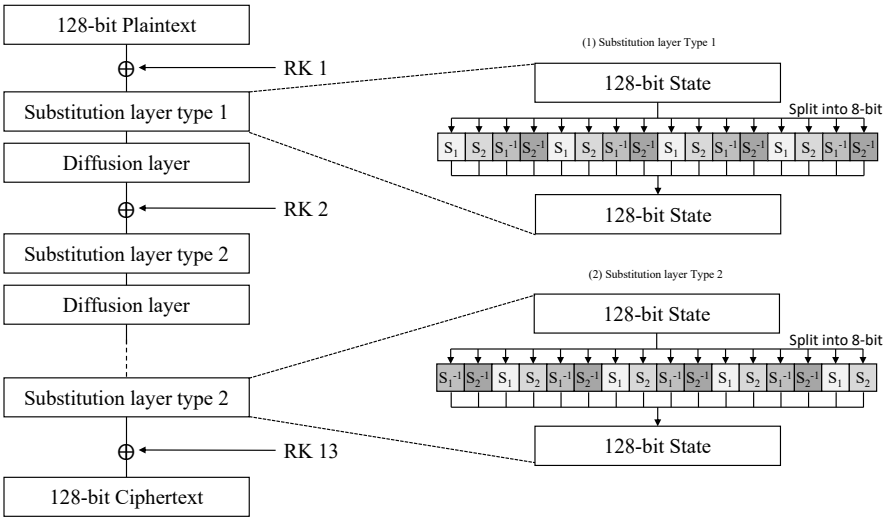


Figure 1. ARIA block cipher algorithm and two types of substitution layers.

2.2. ARMv8 Architecture

ARMv8 is a high-performance embedded 64-bit architecture that supports both 64-bit (i.e., AArch64) and 32-bit (i.e., AArch32) architectures. ARMv8 provides 31 general-purpose registers; x0-x30 can be used in 64-bit units, and w0-w30 can be used in 32-bit units. In addition, ARMv8 provides 32 128-bit vector registers (v0-v31). The ARMv8 processor has shown great influence in the smartphone and it is also widely used in various laptops and smartphones. In Table 1, the instruction set for ARM processors used in the parallel implementation of ARIA block cipher are given.

Table 1. Instruction set for optimized parallel implementation ARIA block cipher. Xd: destination scalar register, Xn: source scalar register, Vd: destination vector register, Vt: transferred vector register, Vn, Vm: source vector register [17].

asm	Operands	Description	Operation
EOR	Vd, Vn, Vm	Bitwise exclusive OR	Vd ← Vn ⊕ Vm
SUB	Vd, Vn, Vm	Subtract	Vd ← Vn − Vm
LD1R	Vt, (Xn)	Load single-element and replicate to all lanes	Vt ← (Xn)
LD4	Vd1–4, (Xn)	Load multiple single-element structures	Vd1–4 ← (Xn)

Table 1. *Cont.*

asm	Operands	Description	Operation
ST4	Vt1–4, (Xn)	Store multiple 4-element structures from four registers.	(Xn) ← Vt1–4
MOVI	Vt, #imm	Move immediate	Vt ← #imm
TBL	Vd, Vn, Vm	Table vector Lookup	Vd ← Vn[Vm]
TBX	Vd, Vn, Vm	Table vector lookup extension	Vd ← Vn[Vm]

2.3. GPU Architecture

GPU has become an integral part of today's computing systems. A modern GPU is a highly parallel programmable processor featuring maximum arithmetic and memory bandwidth that far exceeds CPU [18]. We used an Nvidia RTX 3060 laptop GPU. This GPU has 3840 cores and shows a clock rate of 1702 Mhz. Additionally, CC is 8.3, and it is designed with Ampere architecture. CC refers to compute capability of the device. Note that clock rates might vary depending on the GPU manufacturer [19].

Compute unified device architecture (CUDA) is a GPGPU technology that enables parallel processing performed by GPU to be written using C language. CUDA is developed and maintained by Nvidia, and this architecture requires an Nvidia GPU and stream processing driver. The CUDA GPU architecture includes functional kernel, thread, block, grid, and warp (a bundle of 32 threads) running on the GPU, with one warp running and a streaming multiprocessor (SM) running threads, concurrently [20,21].

Types of memory provided by GPU are register, shared memory, local memory, constant memory, texture memory, and global memory. Global memory is the largest memory on the GPU. Most of the data are stored and used in global memory. Global memory is the largest memory but is the slowest memory. Local memory is memory used to temporarily store register values when the number of registers used by a thread is too large. A lot of local memory usage is not good for speed, because local memory actually uses global memory. Texture memory is read-only memory used when visualizing data values. Constant memory is read-only memory. However, it is possible to initialize the kernel function before executing it. Constant memory actually uses global memory, but there is a separate constant cache. For this reason, if all threads use the stored value of the same address, it can access faster than global memory. Shared memory is memory shared and used by threads within a block. Although it provides small memory, it has the advantage of fast memory access speed. In shared memory, the concept of a bank is introduced, and 32 threads executed in warp units can access it at the same time, which shows low latency. CUDA manages GPU memory by dividing it into on-chip and off-chip. Register and shared memory are on-chip and the others are off-chip. In order to reduce the transmission delay of the memory, it is helpful to maximize the on-chip memory to improve the performance. It is important to use a small size of the on-chip memory efficiently [22]. The detailed structure of GPU memory is shown in Figure 2. In Figure 2, global, constant, and texture memory are indicated by arrows as data can move to/from the CPU.

NVIDIA provides a profiler tool for performance analysis. Among them, Nsight Tools provides three tools: Nsight Compute, Nsight Graphics, and Nsight Systems. Nsight Compute is a CUDA application interactive kernel profiler. By using Nsight Compute, performance analysis data on kernel operation such as kernel operation time, data throughput, and computation throughput can be obtained. We use the Nsight Compute profiler for performance analysis [23].

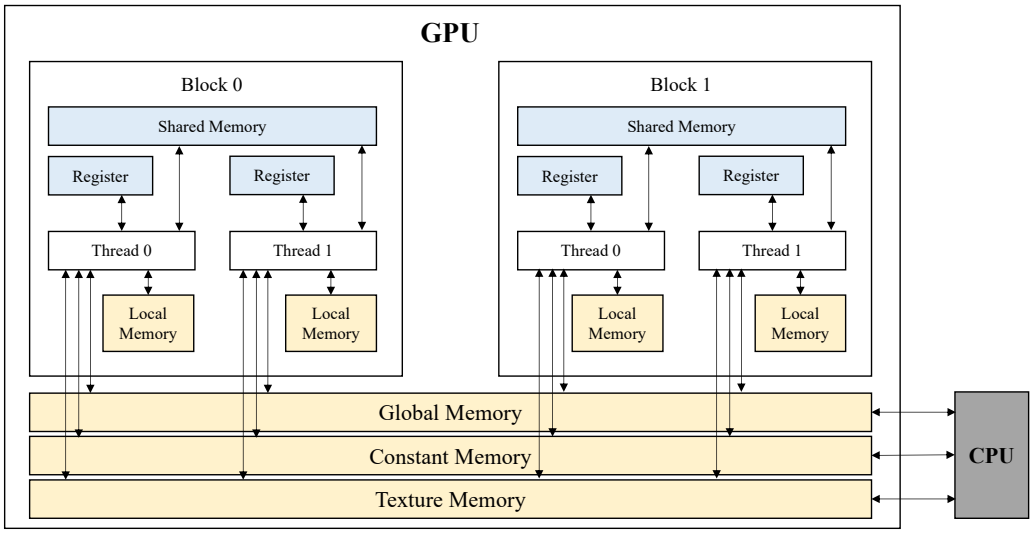


Figure 2. Structure of GPU memory.

3. Parallel ARIA Implementation

In this paper, a parallel ARIA block cipher is implemented on both ARMv8 and GPU architectures. Since implementations were performed on different processors, two parts are described separately in this paper.

3.1. Parallel ARIA Implementation on ARMv8

Instructions used for optimized implementation are described as follows:

- TBL instruction: TBL instruction performs table vector lookup. Substitution and permutation can be implemented efficiently by using the TBL instruction. An example of both implementations can be seen in Figure 3.
  Figure 3a is an example of the operation process of substitution. The value of the vector stored in the vn register is read, and the value is used as the index of the vm register (Sbox is stored in vm). The value stored in the corresponding index of vm is stored in vd. The location stored in vd is the index when reading a value from vn.
  Figure 3b is an example of the operation process of permutation. By using vn and vm inversely, we can implement efficient permutation. The permutation pattern is stored in vm, and the permutation result is stored in vd according to the operation of the TBL instruction described above [1,3].
- Load and store instructions: ARMv8 supports various load and store instruction. Among them, we use LD4 and ST4 instructions for the parallel implementation. A parallel implementation requires that the input value be aligned in registers. If we adjust an arrangement specifier (S and B), it can align the input value without additional works.
  As shown in Figure 1, the ARIA block cipher utilizes four different Sboxs. In order to use the TBL instruction, indexes using the same Sbox must be stored in the same register. This is why we implement parallel with 4 and 16 blocks.

In the 4-PT parallel implementation, the input value is loaded through the LD4.S instruction. Four plaintext blocks are loaded in the register, as shown in Figure 4a. In the state (a), it is a state suitable for implementing the round key addition layer and the diffusion layer. For the implementation of the substitution layer using the TBL instruction, the index using the same Sbox must be in the same register as in Figure 4b. Since the state Type 2 is not suitable for implementing the round key addition and the diffusion layers, in this implementation, the state Type 1 is used in the round key addition layer and the diffusion layer, and the state Type 2 is used in the substitution layer. In other words, the task of converting to state Type 2 before the substitution operation is added, and the operation of converting back to state Type 1 after the substitution operation is added.

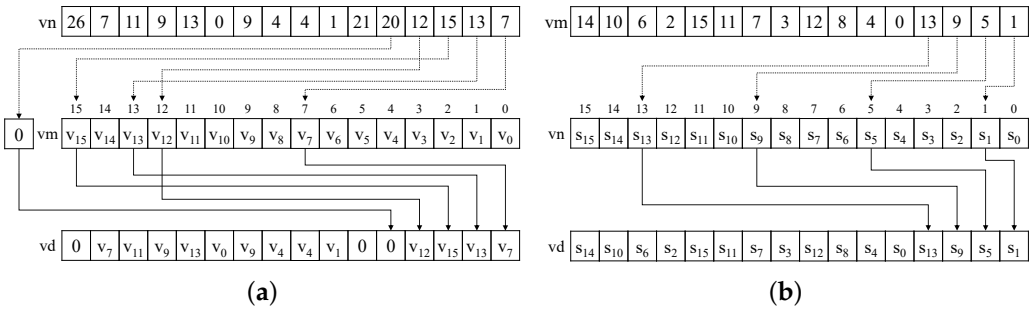
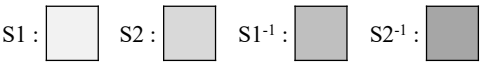


Figure 3. Usage of the TBL instruction to substitute or permutation (vn: input vector, vd: destination register). (**a**) TBL.16B vd, {vm}, vn; (vm: lookup table is stored). (**b**) TBL.16B vd, {vn}, vm; (vm: the permutation pattern is stored).

The diffusion layer is implemented by the 8-bit-based implementation method introduced in Section 2. In state Type 1, values required for operation exist in different registers. If values required for operation are adjusted to be located at the same index, it can be implemented simply through the EOR instruction. Figure 5 shows the operation process of the T variables by simplifying the register for easy understanding.

Although only one block is expressed for simple expression, four blocks do not affect each other and the operation is possible, as shown in Figure 5. The parallel operation of the diffusion layer is possible through optimal format alignments. The value of y, which indicates the result after the operation of the diffusion layer, can be implemented only with the XOR operation after index adjustment through the REV instruction, similar to the operation process of the variable T.



V0	D3	D2	D1	D0	C3	C2	C1	C0	B3	B2	B1	B0	A3	A2	A1	A0
V1	D7	D6	D5	D4	C7	C6	C5	C4	B7	B6	B5	B4	A7	A6	A5	A4
V2	D11	D10	D9	D8	C11	C10	C9	C8	B11	B10	B8	B09	A11	A10	A9	A8
V3	D15	D14	D13	D12	C15	C14	C13	C12	B15	B14	B13	B12	A15	A14	A13	A12

(**a**)

V0	D12	D8	D4	D0	C12	C8	C4	C0	B12	B8	B4	B0	A12	A8	A4	A0
V1	D13	D8	D5	D1	C13	C8	C5	C1	B13	B8	B5	B1	A13	A8	A5	A1
V2	D14	D10	D6	D2	C14	C10	C6	C2	B14	B10	B6	B2	A14	A10	A6	A2
V3	D15	D11	D7	D3	C15	C11	C7	C3	B15	B11	B7	B3	A15	A11	A7	A3

(**b**)

Figure 4. Two types of state for 4-PT parallel implementation. (**a**) 4-PT state Type 1. (**b**) 4-PT state Type 2.

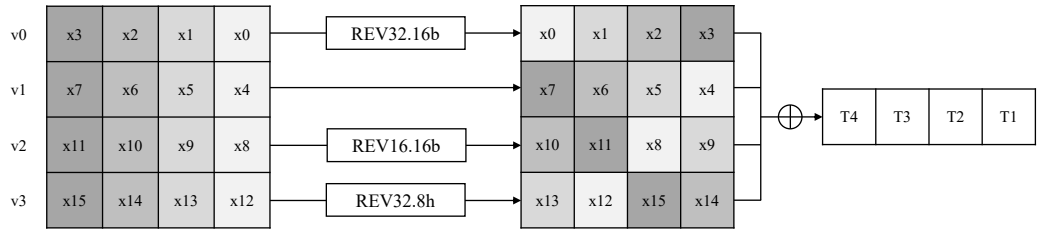


Figure 5. Simplified operation of the variable *T*.

16-PT Parallel Implementation on ARMv8

In the parallel implementation, it is important to align the input blocks of plaintext. Implementing a round function is simply performed, but there are cases where the register alignment is complicated. A 16-PT parallel implementation is such a case. As the number of blocks to be implemented in parallel increases, alignment becomes more complex than 4-PT parallel implementation. Conversely, the round function implementation is simplified because each byte is loaded into a different register. Due to the various load instructions in ARMv8 architecture, aligning is possible at the same time as load without the aligning process.

The 16-PT loads the plaintext block with the LD4.B instruction. In 4-PT, an arrangement specifier is used as S. In 16-PT, it is used as B. When one block is loaded, 4 LD4.B instructions are used, and 16 $\times$ 4 (64) times are used to load all plaintext blocks. The implementation code is the same as Algorithm 1. If the index of the implemented macro is operated 16 times from 0–15, the aligned plaintext block is loaded into the v0-v15 vector register. If the macro is operated once with PT-load 0, the first plaintext block is divided into bytes and loaded at the 0-th index of v0-v15.

In the implementation of the substitution layer, there is no difficulty in implementation because values using the same Sbox are combined in one register. In the implementation of the diffusion layer, 8-bit based implementation method is used. Since each register is divided in units of bytes, parallel operation is possible if registers required for each operation are used.

Algorithm 1 Plaintext load macro in 16-PT parallel implementation (x0: plaintext address).

```
.macro  PT-load   index
1:  LD4.B v0-v3[\index], [x0], #4
2:  LD4.B v4-v7[\index], [x0], #4
3:  LD4.B v8-v11[\index], [x0], #4
4:  LD4.B v12-v15[\index], [x0], #4
.endm
```

3.2. Parallel ARIA Implementation on GPU

The implementation using the T-table extended Sbox was used to implement ARIA on GPU. The original size of one Sbox is 256 bytes. In the case of the T-table, the size of 1 KB is used. Since ARIA block cipher uses four different Sboxes, the size of the entire table increases from 1 KB to 4 KB. By using T-table, memory usage increases, but it has the advantage of reducing operations in the diffusion layer.

When implementing ARIA block ciphers in parallel on GPU, we compare and analyze them using various memory types. The substitution layer introduces a lot of memory access for substitution. Memory accesses cause long latency, and as a result, many memory accesses have a significant impact on performance. The implementation performance depends on the memory access. The Sbox is loaded into different memories showing different latency [24].

In this section, we present four implementations in CUDA codes. Only one Sbox is covered in detail (in code level), but the remaining three tables operate with the same code.

First, the basic implementation is to load the Sbox in the *global memory* (CUDA code is Listing 1). Global memory has the slowest memory access speed, but it has the largest capacity. In CUDA programming, memory is allocated to the GPU (device) through *cudaMalloc*(). The memory allocated in this way uses global memory. The value of the PC (i.e., host) memory is copied through *cudaMemcpy*().

Listing 1. Parallel implementation of ARIA using global memory.

```
// Global Memory Code
constant uint32_t s1[256]={...};
...
__global__ Aria_device(in, out, s1,...){...}
__host__ Aria_host(in, out, key){
    roudnkey_gen(key, rk);
    ...
    cudaMalloc(&dev_s1, sizeof(uint32_t)*256);
    ...
    cudaMemcpy(s1, dev_s1, sizeof(uint32_t)*256);
    ...
    Aria_device<<<block, thread>>>(in, out, dev_s1, ...);
    ...
}
```

The second is *shared memory* implementation (CUDA code is Listing 2). Shared memory is a memory space provided per block that can be shared and used by multiple threads. In this case, it is possible to access the shared memory in unit of Warp (i.e., 32 threads) at once. Shared memory can be accessed with faster access speed than global memory. However, the memory space provided is small at 48 KB. When using shared memory, it should be implemented with caution against bank conflicts. A bank conflict is a problem in which multiple threads access the same bank. This leads to sequential processing in parallel machine. For this reason, it may show slow performance even with fast access speeds. Shared memory can be initialized through __*shared*__ in the device function. Afterwards, an additional operation is performed to copy the values from global memory to shared memory. A copy process must be performed for each block. In this case, *threadIdx.x*, which indicates the thread index of each block, is used [25].

Listing 2. Parallel implementation of ARIA using shared memory.

```
// Shared Memory Code
constant uint32_t s1[256]={...};
...
__global__ Aria_device(in, out, dev_s1,...){
    __shared__ uint32_t shared_s1[256];
    ...
    if(threadIdx.x < 256) {
        shared_s1[threadIdx.x] = dev_s1[threadIdx.x];
        ...
    }
    ...
}
__host__ Aria_host(in, out, key){
    roudnkey_gen(key, rk);
    ...
    cudaMalloc(&dev_s1, sizeof(uint32_t)*256);
    ...
    cudaMemcpy(s1, dev_s1, sizeof(uint32_t)*256);
    ...
    Aria_device<<<block, thread>>>(in, out, dev_s1, ...);
    ...
}
```

We also utilized the technique of copying the Sbox to minimize bank conflicts (CUDA code is Listing 3). Since the Sbox of ARIA is 4 KB, copying the Sbox by the number of banks will exceed the available shared memory size. Therefore, the maximum number of copies of the Sbox to shared memory is 12. However, the number of banks is 32. If we copy it to 12, the table is mixed when sorted in the bank. The bank conflict cannot be controlled if they are sorted as shuffled state. Thus, the best copy is the divisor of 32. In this paper, the implementation was carried out when Sbox was simply used as a shared memory without copying, when it was copied to 8, and, finally, when it was copied to 12.

Listing 3. Extended Sbox ARIA using shared memory.

```
// Extended Sbox code using shared memory
constant uint32_t s1[256]={...};
...
__global__ Aria_device(in, out, dev_s1,...){
    __shared__ uint32_t shared_s1[256][N];
    ...
    for(int i=threadIdx.x; i<256; i+=blockDim.x) {
        for(int j=0; j<N; j++){
            shared_s1[threadIdx.x][j] = dev_s1[i];
            ...
        }
    }
    ...
}
__host__ Aria_host(in, out, key){
    roudnkey_gen(key, rk);
    ...
    cudaMalloc(&dev_s1, sizeof(uint32_t)*256);
    ...
    cudaMemcpy(s1, dev_s1, sizeof(uint32_t)*256);
    ...
    Aria_device<<<block, thread>>>(in, out, dev_s1, ...);
    ...
}
```

4. Evaluation

4.1. Evaluation on ARMv8 Implementation

In this section, we show the performance evaluation of the ARIA block cipher on ARMv8 architectures. The difference in key length is only the number of rounds, so we measured performance based on 128-bit key length. Performance is measured on a MacBook Pro 13 with the Apple M1, one of the latest ARMv8 processors. We used the Xcode framework, set the optimization option to -Os, and measured the performance. Since there are no existing studies on ARIA implementations for ARMv8 architectures, comparative analysis is performed with reference implementation. In addition, although the target processor is different, the performance of the previous study was also included for comparison. This can compare performance differences due to differences in hardware. In our works, CPB (cycle per byte) is calculated and analyzed using the following formula:

$$\text{CPB} = \text{Milliseconds}/1000/\text{Number of iteration}/\text{Input Byte} \times \text{Operating frequency}$$

After running the encryption function 10,000,000 times, it is counted as the operating frequency (3.2 Ghz) and input bytes. Performance results are shown in Table 2. Existing studies also show the performance results of the encryption function.

Table 2. Performance comparison of implementation of the ARIA block cipher.

Imple.	Target	Parallel	CPB
Seo et al. [14]	32-bit ARM Cortex-M3	1-PT	147
Kwak et al. [15]	32-bit RISC-V HiFive1 rev b	1-PT	295
Reference C	64-bit ARMv8 Apple M1	1-PT	4.77
This work	64-bit ARMv8 Apple M1	4-PT	1.73
This work	64-bit ARMv8 Apple M1	16-PT	0.57
This work	64-bit ARMv8 A12 Bionic	4-PT	2.17
This work	64-bit ARMv8 A12 Bionic	16-PT	0.96

The performance difference from previous studies clearly shows the large difference due to the difference in processors. Compared to the reference C implementation, which is not an assembly-optimized implementation, the difference is about 60×. This means that hardware differences lead to large performance differences.

In 4-plaintext of our implementation, the performance was 1.73 cpb, which was 2.76× higher than that of reference implementation. The 16-plaintext showed a performance of 0.57 cpb, and 8.73× improved performance compared to reference implementation. It also showed 3.04× better performance compared to 4-plaintext. The result shows that the highly optimized ARIA block cipher in a parallel way can achieve much higher throughput than that of sequential implementation. In addition, the same ARMv8 architecture was used, but the effect of the operating frequency was investigated by measuring the performance in an environment with different operating frequencies. Performance was measured on an Apple IPad Air (3rd) with an A12 Bionic chip. The operating frequency speed of Apple M1 is 3.2 Ghz, whereas in A12 it is 2.49 GHz. As a result, it shows 1.6× and 1.25× higher performance in M1 with high operating frequency. From this, it can be seen that the higher the operating frequency, the better the performance.

4.2. Evaluation on GPU Implementation

In this section, we show the performance of ARIA block cipher implementations depending on various types of memory in the GPU. As with the ARMv8 performance evaluation, the difference in key length is only the number of rounds, so we measured performance based on 128-bit key length. Factors that affect GPU implementation performance include number of threads and number of blocks. Performance measurements were taken while correcting for these factors. The Nvidia GeForce GTX 3060 laptop GPU was used for the testing. It was implemented in the Visual Studio, and the CUDA 11.8 Runtime template was used. Performance is presented in several tables for more convenient performance comparison. The size of the input data for each implementation is (number of blocks × number of threads × block size). That is, if the number of blocks is 1024 and the number of threads is 32, the size of the input data is 0.5 MB (1024 × 32 × 16). The input data are used as a random value. Our goal is to find the optimal implementation environment (e.g., type of memory, number of threads, number of blocks). The Roundkey was used by storing it in the GPU's constant memory. When all threads refer to the same memory, the Roundkey is stored in constant memory because it is better for performance to use constant memory.

First, for performance comparison according to memory types, the number of blocks was fixed to 1024 × 32 and the number of threads to 256 to show the implementation performance according to the type of memory. Shared[256] is an implementation of Listing 2 that uses Sbox tables copied to shared memory. Shared[4][256] and Shared[256][4] are an implementation of Listing 3 that uses Sbox tables copied to shared memory. The performance results are shown in Table 3.

Table 3. Performance comparison by memory type (C.: compute; M.: memory).

Memory Type	Block	Thread	Duration (ms)	C. Throughput (%)	M. Throughput (%)
Global	32,768	256	7.95	55.92	93.94
Shared[256]	32,768	256	7.32	61.81	99.77
Shared[4][256]	32,768	256	8.15	58.43	98.52
Shared[256][4]	32,768	256	8.14	55.14	92.80

As shown in Table 1, it can be seen that the use of shared memory improves computation and memory throughput and, as a result, shortens the kernel operation time. In terms of kernel execution time, performance is improved by 1.08× using shared memory. However, we found that implementations of extending Sbox tables to avoid bank conflicts performed worse than using global memory. While the size of the bank is 32, it seems that it cannot prevent a complete bank collision, because only four copies were copied. To analyze this in more detail, we compare the performance according to the number of table copies. We use a fixed number of blocks and threads as above (block: 1024 × 32, thread: 256). The performance results are shown in Table 4.

Table 4. Performance by number of Sbox table copies (C.: compute, M.: memory, block: 1024 × 32, thread: 256).

Type	Duration (ms)	C. Throughput (%)	D. Throughput (%)	Bank Conflicts
Sbox[4][256]	8.15	58.43	98.52	123,863,597
Sbox[8][256]	8.57	59.21	96.70	126,228,907
Sbox[12][256]	9.16	58.77	91.87	126,287,694
Sbox[256][4]	8.14	58.42	98.94	124,919,451
Sbox[256][8]	9.50	53.32	98.73	156,685,446
Sbox[256][12]	9.93	54.18	94.59	152,467,045

Due to the size limit of shared memory, up to 12 tables can be copied. As a result of the measurement, it was confirmed that the higher the number of copies of the table, the lower the performance. When performing front table expansions ([4][256]), the increased number of copies slowed down memory throughput, resulting in poor performance. Increasing the number of copies only increases the number of copies of the table from global memory to shared memory and degrades performance because the bank conflicts cannot be resolved. When performing back table expansions ([256][4]), the increased number of copies slowed down compute throughput, resulting in poor performance. An increase in bank collisions is considered to be the cause of a decrease in the computational throughput. As a result, it is inefficient to apply this technique when it is impossible to copy as much as the bank size.

Next, we checked the performance comparison according to the number of blocks, which is one of the factors affecting performance. In this case, we fixed only the number of threads (256) and increased the number of blocks during the measurement. We did not compare the performance of table extension implementations, we only compared the performance of global and shared memory implementations, because we saw above that table expansion is inefficient. The performance results are shown in Table 5.

Table 5. Performance as the number of blocks increases (C.: compute, M.: memory, thread: 256).

Memory Type	Block	Duration (ms)	C. Throughput (%)	D. Throughput (%)
Global	1024	0.28	49.24	85.43
	1024 × 8	1.98	55.00	93.33
	1024 × 16	3.96	54.68	92.94
	1024 × 32	7.95	55.29	93.94
	1024 × 64	15.83	55.11	93.46
Shared	1024	0.24	58.99	95.32
	1024 × 8	1.84	61.54	99.34
	1024 × 16	3.67	61.73	99.63
	1024 × 32	7.32	61.81	99.77
	1024 × 64	14.63	61.86	99.84

In this case, when the same memory type is used, the kernel operation time (duration) cannot be compared because the input data increases as the number of blocks increases. However, the kernel operation time is also included for comparison according to memory types. In fact, increasing the number of blocks did not affect performance, but here we can see that the number of blocks does not affect performance (in implementation of ARIA block cipher). Additionally, we can reaffirm that using shared memory can help improve performance.

Finally, we compare the performance according to the number of threads. The number of blocks is fixed (1024 × 32) and performance is measured by increasing the number of threads. We compare the performance of global and shared memory implementations, such as comparing performance by number of blocks. The performance results are shown in Table 6.

Table 6. Performance as the number of threads increases (C.: compute, M.: memory, blocks: 1024 × 32).

Memory Type	Thread	Duration (ms)	C. Throughput (%)	D. Throughput (%)
Global Memory	32	0.97	56.21	94.82
	64	1.94	56.41	95.16
	128	3.87	56.53	95.39
	256	7.95	55.29	93.94
	512	17.94	48.55	83.42
	1024	47.65	36.69	62.24
Shared Memory	32	1.01	69.46	97.85
	64	1.91	65.27	98.99
	128	3.70	63.25	99.56
	256	7.32	61.81	99.77
	512	15.64	56.97	93.15
	1024	33.09	53.37	87.52

It can be seen that the lower the number of threads (32), the better the performance is to use global memory. This is because the process of copying from global to shared memory is higher than the improvements achieved using shared memory. The greater the number of threads, the greater the performance difference that can be achieved using shared memory. In the performance of the shared memory implementation, it can be seen that as the number of threads increases, the performance becomes better than that of global memory, but the computational throughput decreases. This is because as the number of threads increases, more bank conflicts occur.

Tables 7–9 show the overall performance of global, shared memory, and extended Sbox using shared memory implementations. Overall, depending on the implementation, it is more efficient to use global memory in implementations with fewer than 32 threads, and more efficient to use shared memory when using more than 32 threads. Increasing the number of blocks can improve memory throughput, but does not significantly improve performance as compute throughput cannot support it. Therefore, a large number of blocks is not always efficient, so it is recommended to use an appropriate number of blocks depending on the size of the data. Memory throughput has always been higher than computational throughput. Therefore, for the number of threads, it is recommended to use the number of threads 64, 128 because the compute throughput is highest when using shared memory.

Table 7. Performance of all the global memory implementation (C.: compute, M.: memory).

Memory Type	Block	Thread	Duration (ms)	C. Throughput (%)	D. Throughput (%)
Global	1024	32	0.04	41.60	73.96
Global	1024	64	0.07	47.45	82.93
Global	1024	128	0.13	51.34	88.00
Global	1024	256	0.28	49.24	85.43
Global	1024	512	0.65	41.70	71.86
Global	1024	1024	1.53	35.68	60.60
Global	1024 × 8	32	0.25	54.23	92.05
Global	1024 × 8	64	0.49	55.44	93.91
Global	1024 × 8	128	0.98	55.98	94.64
Global	1024 × 8	256	1.98	55.00	93.33
Global	1024 × 8	512	4.62	47.70	81.90
Global	1024 × 8	1024	11.95	36.60	62.08
Global	1024 × 16	32	0.49	55.50	93.84
Global	1024 × 16	64	0.98	56.09	94.76
Global	1024 × 16	128	1.94	56.35	95.11
Global	1024 × 16	256	3.96	54.68	92.94
Global	1024 × 16	512	9.09	47.95	82.35
Global	1024 × 16	1024	23.84	36.64	62.15
Global	1024 × 32	32	0.97	56.21	94.82
Global	1024 × 32	64	1.94	56.41	95.16
Global	1024 × 32	128	3.87	56.53	95.39
Global	1024 × 32	256	7.95	55.29	93.94
Global	1024 × 32	512	17.94	48.55	83.42
Global	1024 × 32	1024	47.65	36.69	62.24
Global	1024 × 64	32	1.93	56.54	95.27
Global	1024 × 64	64	3.86	56.60	95.44
Global	1024 × 64	128	7.72	56.63	95.47
Global	1024 × 64	256	15.83	55.11	93.46
Global	1024 × 64	512	35.13	49.65	85.29
Global	1024 × 64	1024	95.22	36.70	62.25

Table 8. Performance of all the shared memory implementation (C.: compute, M.: memory).

Memory Type	Block	Thread	Duration (ms)	C. Throughput (%)	D. Throughput (%)
Shared	1024	32	0.04	54.83	76.93
Shared	1024	64	0.07	58.64	88.69
Shared	1024	128	0.12	59.48	93.58
Shared	1024	256	0.24	58.99	95.32
Shared	1024	512	0.51	55.27	90.43
Shared	1024	1024	1.06	52.47	86.10
Shared	1024 × 8	32	0.26	67.74	95.40
Shared	1024 × 8	64	0.48	64.61	97.96
Shared	1024 × 8	128	0.93	62.88	98.98
Shared	1024 × 8	256	1.84	61.54	99.34
Shared	1024 × 8	512	3.93	56.80	92.89
Shared	1024 × 8	1024	8.30	53.13	87.12
Shared	1024 × 16	32	0.51	68.89	97.03
Shared	1024 × 16	64	0.96	65.10	98.71
Shared	1024 × 16	128	1.85	63.13	99.37
Shared	1024 × 16	256	3.67	61.73	99.63
Shared	1024 × 16	512	7.82	56.93	93.09
Shared	1024 × 16	1024	16.52	53.53	87.78
Shared	1024 × 32	32	1.01	69.46	97.85
Shared	1024 × 32	64	1.91	65.27	98.99
Shared	1024 × 32	128	3.70	63.25	99.56
Shared	1024 × 32	256	7.32	61.81	99.77
Shared	1024 × 32	512	15.64	56.97	93.15
Shared	1024 × 32	1024	33.09	53.37	87.52
Shared	1024 × 64	32	2.01	69.76	98.28
Shared	1024 × 64	64	3.82	65.40	99.19
Shared	1024 × 64	128	7.39	63.31	99.66
Shared	1024 × 64	256	14.63	61.86	99.84
Shared	1024 × 64	512	31.24	56.94	93.10
Shared	1024 × 64	1024	66.14	53.27	87.36

Table 9. Performance of the extended Sbox using shared memory implementation (C.: compute, M.: memory).

Memory Type	Block	Thread	Duration (ms)	C. Throughput (%)	D. Throughput (%)
Shared[4][256]	1024 × 32	32	1.57	59.57	74.37
Shared[4][256]	1024 × 32	64	2.24	66.16	95.84
Shared[4][256]	1024 × 32	128	4.14	62.09	98.77
Shared[4][256]	1024 × 32	256	8.15	58.43	98.52
Shared[4][256]	1024 × 32	512	17.09	53.38	92.14
Shared[4][256]	1024 × 32	1024	33.25	53.74	93.82
Shared[8][256]	1024 × 32	32	2.65	47.01	50.10
Shared[8][256]	1024 × 32	64	2.83	63.25	81.70
Shared[8][256]	1024 × 32	128	4.48	64.33	95.72
Shared[8][256]	1024 × 32	256	8.57	59.21	96.70
Shared[8][256]	1024 × 32	512	17.77	53.13	90.94
Shared[8][256]	1024 × 32	1024	33.92	53.59	93.56
Shared[12][256]	1024 × 32	32	3.96	39.31	39.31
Shared[12][256]	1024 × 32	64	3.89	54.02	63.20
Shared[12][256]	1024 × 32	128	4.85	65.82	91.44
Shared[12][256]	1024 × 32	256	9.16	58.77	91.87
Shared[12][256]	1024 × 32	512	18.68	52.10	86.87
Shared[12][256]	1024 × 32	1024	34.26	53.97	93.10
Shared[256][4]	1024 × 32	32	1.75	53.34	90.09
Shared[256][4]	1024 × 32	64	2.54	58.25	98.45
Shared[256][4]	1024 × 32	128	4.38	58.71	99.17
Shared[256][4]	1024 × 32	256	8.14	58.42	98.94
Shared[256][4]	1024 × 32	512	16.57	55.14	92.80
Shared[256][4]	1024 × 32	1024	31.81	56.15	94.11
Shared[256][8]	1024 × 32	32	4.14	30.11	81.52
Shared[256][8]	1024 × 32	64	4.37	41.03	96.72
Shared[256][8]	1024 × 32	128	6.06	47.62	98.51
Shared[256][8]	1024 × 32	256	9.50	53.32	98.73
Shared[256][8]	1024 × 32	512	17.30	54.40	93.56
Shared[256][8]	1024 × 32	1024	31.47	57.78	94.46
Shared[256][12]	1024 × 32	32	4.46	34.90	63.39
Shared[256][12]	1024 × 32	64	4.37	48.15	85.91
Shared[256][12]	1024 × 32	128	5.89	54.27	95.44
Shared[256][12]	1024 × 32	256	9.93	54.18	94.59
Shared[256][12]	1024 × 32	512	18.43	52.69	90.88
Shared[256][12]	1024 × 32	1024	33.22	55.64	94.99

5. Conclusions

In this paper, we present the parallel implementation of ARIA block cipher on both ARMv8 architectures and GPU. In ARMv8, 4 and 16 plaintext blocks are encrypted in parallel through TBL and LD4 instructions. Since this technique is also applicable to other block ciphers, it can be used for parallel implementations of other block ciphers. In GPU, optimal settings of ARIA block cipher implementation on GPU were analyzed using the Nsight Compute profiler provided by Nvidia. We found that using shared memory can help improve performance when performing substitution operations with Sbox tables. Additionally, techniques using table expansion to minimize bank conflicts were found to be inefficient when tables cannot be copied by the size of the bank. The performance results suggest that a large number of blocks and threads do not represent high performance. Therefore, it provides performance results so that it can be implemented by setting the appropriate number of blocks and threads according to the implementation environment. We believe that this work will be helpful in parallel implementation of other ciphers on both ARMv8 architectures and GPU.

Author Contributions: Software, S.E., H.K. (Hyunjun Kim), H.K. (Hyeokdong Kwon) and M.S.; Writing—original draft, S.E.; Writing—review & editing, G.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.

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

References

1. Fujii, H.; Rodrigues, F.C.; López, J. Fast AES implementation using ARMv8 ASIMD without cryptography extension. In Proceedings of the International Conference on Information Security and Cryptology, Nanjing, China, 8–9 December 2019; Springer: New York, NY, USA, 2019; pp. 84–101.
2. Daemen, J.; Rijmen, V. AES proposal: Rijndael. *Int. J. Commun. Netw. Syst. Sci.* **1999**, *1*, 1.
3. Kwon, H.; Kim, H.; Eum, S.; Sim, M.; Kim, H.; Lee, W.K.; Hu, Z.; Seo, H. Optimized Implementation of SM4 on AVR Microcontrollers, RISC-V Processors, and ARM Processors. *Cryptol. Eprint Arch.* **2021**, *10*, 80225–80233. [CrossRef]
4. Kim, H.; Sim, M.; Jang, K.; Kwon, H.; Uhm, S.; Seo, H. Masked Implementation of Format Preserving Encryption on Low-End AVR Microcontrollers and High-End ARM Processors. *Mathematics* **2021**, *9*, 1294. [CrossRef]
5. An, S.; Kim, Y.; Kwon, H.; Seo, H.; Seo, S.C. Parallel implementations of ARX-based block ciphers on graphic processing units. *Mathematics* **2020**, *8*, 1894. [CrossRef]
6. Tezcan, C. Optimization of Advanced Encryption Standard on Graphics Processing Units. *IEEE Access* **2021**, *9*, 67315–67326. [CrossRef]
7. Lee, W.K.; Seo, H.; Seo, S.; Hwang, S. 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]
8. Kwon, D.; Kim, J.; Park, S.; Sung, S.; Sohn, Y.; Yeom, Y.; Yoon, E.; Lee, S.; Lee, J.; Chee, S.; et al. New block cipher: ARIA. In Proceedings of the International Conference on Information Security and Cryptology, Perth, Australia, 30 November–2 December 2020; Springer: New York, NY, USA, 2003; pp. 432–445.
9. Seo, H.; Kwon, H.; Kim, H.; Park, J. ACE: ARIA-CTR Encryption for Low-End Embedded Processors. *Sensors* **2020**, *20*, 3788. [CrossRef] [PubMed]
10. Yang, S.; Park, J.; You, Y. The smallest ARIA module with 16-bit architecture. In Proceedings of the International Conference on Information Security and Cryptology, Busan, Republic of Korea, 30 November–1 December 2006; Springer: New York, NY, USA, 2006; pp. 107–117.
11. Ryu, G.H.; Koo, B.S.; Yang, S.W.; Chang, T.J. Area efficient implementation of 32-bit architecture of ARIA block cipher using light weight diffusion layer. *J. Korea Inst. Inf. Secur. Cryptol.* **2006**, *16*, 15–24.
12. Lee, W.Y.; Choi, Y.S. Optimization of ARIA Block-Cipher Algorithm for Embedded Systems with 16-bits Processors. *Int. J. Internet Broadcast. Commun.* **2016**, *8*, 42–52.
13. Sasi, S.B.; Sivanandam, N. A survey on cryptography using optimization algorithms in WSNs. *Indian J. Sci. Technol.* **2015**, *8*, 216. [CrossRef]
14. Seo, H.; Kim, H.; Jang, K.; Kwon, H.; Sim, M.; Song, G.; Uhm, S. Compact Implementation of ARIA on 16-Bit MSP430 and 32-Bit ARM Cortex-M3 Microcontrollers. *Electronics* **2021**, *10*, 908. [CrossRef]
15. Kwak, Y.; Kim, Y.; Seo, S.C. Benchmarking Korean block ciphers on 32-bit RISC-V processor. *J. Korea Inst. Inf. Secur. Cryptol.* **2021**, *31*, 331–340.
16. Lee, J.j.; Park, J.u.; Kim, M.j.; Kim, H.w. Efficient ARIA cryptographic extension to a RISC-V processor. *J. Korea Inst. Inf. Secur. Cryptol.* **2021**, *31*, 309–322.
17. ARMv8-A Instruction Set Architecture. Available online: https://documentation-service.arm.com/static/613a2c38674a052ae36ca307 (accessed on 26 June 2019).
18. Owens, J.D.; Houston, M.; Luebke, D.; Green, S.; Stone, J.E.; Phillips, J.C. GPU Computing. *Proc. IEEE* **2008**, *96*, 879–899. [CrossRef]
19. Furkan Altınok, K.; Peker, A.; Tezcan, C.; Temizel, A. GPU accelerated 3DES encryption. *Concurr. Comput. Pract. Exp.* **2022**, *34*, e6507. [CrossRef]
20. Choi, H.; Seo, S.C. Fast Implementation of SHA-3 in GPU Environment. *IEEE Access* **2021**, *9*, 144574–144586. [CrossRef]
21. Iwai, K.; Nishikawa, N.; Kurokawa, T. Acceleration of AES encryption on CUDA GPU. *Int. J. Netw. Comput.* **2012**, *2*, 131–145. [CrossRef] [PubMed]
22. Yeom, Y.J.; Cho, Y.K. High-Speed Implementations of Block Ciphers on Graphics Processing Units Using CUDA Library. *J. Korea Inst. Inf. Secur. Cryptol.* **2008**, *18*, 23–32.
23. Nsight Compute—NVIDA Documentation Center. Available online: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html (accessed on 24 August 2022).
24. CUDA C Programming Guide V6.0. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed on 11 May 2022).
25. Lee, W.K.; Goi, B.M.; Phan, R.C.W.; Poh, G.S. High speed implementation of symmetric block cipher on GPU. In Proceedings of the 2014 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS), Sarawak, Malaysia, 1–4 December 2014; IEEE: Piscatway, NJ, USA, 2014; pp. 102–107.