applied sciences

MDPI

*Article*

# Secure and Robust Internet of Things with High-Speed Implementation of PRESENT and GIFT Block Ciphers on GPU

**Hyunjun Kim [1], Siwoo Eum [2], Wai-Kong Lee [3], Sokjoon Lee [3] and Hwajeong Seo [2,*]**

[1] Department of Information Computer Engineering, Hansung University, Seoul 02876, Korea
[2] Division of IT Convergence Engineering, Hansung University, Seoul 02876, Korea
[3] Department of Computer Engineering, Gachon University, Seongnam 13120, Korea
* Correspondence: hwajeong@hansung.ac.kr; Tel.: +82-760-8033

**Abstract:** With the advent of the Internet of Things (IoT) and cloud computing technologies, vast amounts of data are being created and communicated in IoT networks. Block ciphers are being used to protect these data from malicious attacks. Massive computation overheads introduced by bulk encryption using block ciphers can become a performance bottleneck of the server, requiring high throughput. As the need for high-speed encryption required for such communications has emerged, research is underway to utilize a graphics processor for encryption processing based on the high processing power of the GPU. Applying bit-slicing of lightweight ciphers was not covered in the previous implementation of lightweight ciphers on GPU architecture. In this paper, we implemented PRESENT and GIFT lightweight block ciphers GPU architectures. It minimizes the computation overhead caused by optimizing the algorithm by applying the bit-slicing technique. We performed practical analysis by testing practical use cases. We tested PRESENT-80, PRESENT-128, GIFT-64, and GIFT-128 block ciphers in RTX3060 platforms. The throughput of the exhaustive search are 553.932 Gbps, 529.952 Gbps, 583.859 Gbps, and 214.284 Gbps for PRESENT-80, PRESENT-128, GIFT-64, and GIFT-128, respectively. For the case of data encryption, it achieved 24.264 Gbps, 24.522 Gbps, 85.283 Gbps, and 10.723 Gbps for PRESENT-80, PRESENT-128, GIFT-64, and GIFT-128, respectively. Specifically, the proposed implementation of a PRESENT block cipher is approximately 4× higher performance than the latest work that implements PRESENT block cipher. Lastly, the proposed implementation of a GIFT block cipher on GPU is the first implementation for the server environment.

**Keywords:** parallel processing; PRESENT; GIFT; lightweight block cipher; GPU implementation



**Citation:** Kim, H.; Eum, S.; Lee, W.-K.; Lee, S.; Seo, H. Secure and Robust Internet of Things with High-Speed Implementation of PRESENT and GIFT Block Ciphers on GPU. *Appl. Sci.* **2022**, *12*, 10192. https://doi.org/10.3390/app122010192

Academic Editors: Ryan Gibson and Hadi Larijani

Received: 6 September 2022
Accepted: 8 October 2022
Published: 11 October 2022



## 1. Introduction

The significant growth of Internet of Things (IoT) technologies in recent years has introduced many smart applications into our daily life. Due to this, the data exchanged between IoT devices increases on an unprecedented scale, and the need for data security [1] in mass data communication is also emerging. Block ciphers are being used to protect these data from malicious attacks. Massive computations introduced by bulk encryption using block ciphers can become a performance bottleneck of the server, requiring high throughput. To alleviate this problem, some researchers proposed to utilize the Graphics Processing Unit (GPU) as an accelerator for computing the cryptographic operations [2–9] can free up CPU for other tasks. Some research works also consider using the GPU for cryptanalysis purposes [10,11].

Many lightweight block ciphers have been introduced recently targeting resource-constrained devices. In order to improve the encryption and decryption throughput of these lightweight block ciphers in the server environment equipped with GPU, several research works are proposed. Lee et al. [3] presented optimized implementation techniques of LEA, Chaskey, SIMON, SPECK, and SIMECK on three different GPU architectures. In the case of LEA, the previous best performance was achieved by Seo et al. [2]. Li et al. [12] focus

on general software implementations of LED, Piccolo, and PRESENT on GPU architectures. This includes lookup table-based implementations that use explore various memory types to store the lookup tables. By studying parallel granularity and overlapping data transfer and processing, this work significantly improved the performance of the three selected block ciphers on various GPU platforms. An et al. [13] implemented the core part of each cipher with a manual inline pseudo-assembly code (i.e., Parallel Thread eXecution; PTX) and optimized each target cipher by considering the algorithmic characteristics of each cipher. This is the first GPU optimization study for CHAM, with throughput improvements of 10.7% and 67% compared to the previous best results.

Another interesting optimization technique is to bit-slicing the lightweight ciphers, which was covered in the previous studies. Bit-slicing was first used by Biham [14] instead of lookup tables to speed up the software implementation of DES. Bit-slicing refers to a technique of bit-wise operation by collecting each bit in several blocks. In this way, multiple blocks can be processed in parallel. Hajihassani et al. [4] achieved the highest AES throughput on GPU using the bit-slicing technique. Moreover, some other related works that implement PRESENT [15], GIFT [16] and AES [17] on embedded devices (e.g., microcontroller) also showed that a bit-slicing based technique can achieve the best performance. However, in embedded software, a bit-slicing technique only processes two blocks [17] due to high register utilization. The advanced bit-slicing technique that processes a large number of blocks (i.e., larger than two blocks) is not widely explored as there are insufficient registers to hold the intermediate results. However, GPU devices do not have such limitations as there are plenty of registers available, which is suitable for this advanced bit-slicing technique. This motivates us to improve the throughput of PRESENT and GIFT lightweight ciphers using the advanced bit-slicing technique on the GPU.

*Contributions*

We implemented PRESENT, the first lightweight block cipher, and GIFT, one of the most popular lightweight block ciphers. The bit-slicing technique was applied to improve the throughput of PRESENT and GIFT and optimize the algorithm to minimize overhead. The contributions of this paper are summarized below.

- Fast GPU implementation of PRESENT and GIFT Both PRESENT and GIFT block ciphers utilize S-box, which can be pre-computed and stored on various GPU memories to achieve high encryption performance. In this paper, we move away from this traditional technique and focus on the advanced bit-slicing technique to further improve the throughput performance. Existing bit-slice implementation on GPU suffers from excessive use of registers. To avoid this issue, the proposed implementation generates the round keys on the fly instead of pre-computing them.
- First GPU-based exhaustive key search and bulk data encryption for both PRESENT and GIFT block ciphers. We performed practical analysis by testing practical use cases (i.e., exhaustive key search and bulk data encryption). When implementing an exhaustive key search, the counter value is directly generated in the bit-sliced form on the GPU kernel. In this way, we do need to transfer the counter values to the GPU. This also reduces the memory copy delay and improves performance significantly.
- Speed record result on modern GPU architecture (RTX) The proposed implementation was evaluated on the RTX 3060 with NVIDIA Ampere architecture. We are able to achieve 553.932 Gbps PRESENT-80, 529.952 Gbps PRESENT-128, 583.859 Gbps GIFT-64, and 214.284 Gbps GIFT-128 throughput for encryption, respectively. For exhaustive search, 24.264 Gbps PRESENT-80, 24.522 Gbps PRESENT-128, 85.283 Gbps GIFT-64, and 10.723 Gbps GIFT-128 throughput are achieved, respectively. Note that our implementation of PRESENT is approximately 4× higher performance than the latest work that implements PRESENT [10].

## 2. Background

### *2.1. PRESENT Block Cipher*

PRESENT [18] is a lightweight block cipher based on the Substitution Permutation Network (SPN) structure standardized in ISO/IEC in 2012. PRESENT supports 64-bit input data blocks and key sizes of 80 and 128 bits. The input key is processed internally to generate a round key for each of a total of 31 rounds. The cipher is composed of a structure that repeats the following three basic operations for a state, as shown in Figure 1 The steps addroundkey, sboxLayer, and pLyaer are repeated for each round.

- AddRoundKey: Adding state to the 64-bit word of the round key using finite field arithmetic.
- sBoxLayer: Using an S-box (replacement box) with 16 values to replace 4-bits to 4-bits in the state.
- pLayer: Applying a bit-level shift to the state.

Based on the key schedule in the 80-bit version, the key schedule first processed the 80-bit key by rotating it to the left by 61-bits. Next, the leftmost 4-bits are passed through the S-box, and the round_counter value $i$ is XORed with the least significant bit of $K$.

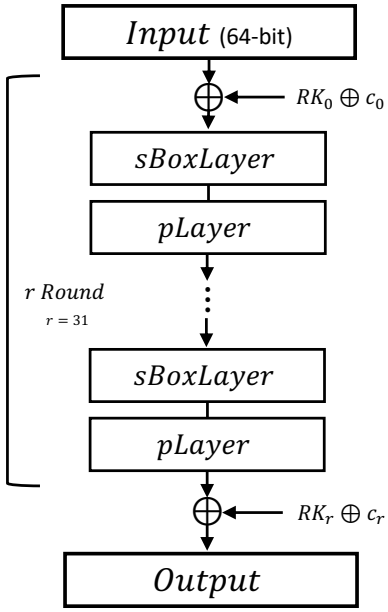


**Figure 1.** PRESENT structure.

### *2.2. GIFT Block Cipher*

GIFT [19] is a lightweight block cipher that corrects the well-known weakness of PRESENT and improves its efficiency. It is an SPN-based cipher with two variants: GIFT-64 and GIFT-128. 128-bit key GIFT-64 operates with 28 rounds and GIFT-128 with 40 rounds, and each round consists of three steps: SubCells, PermBits, and AddRoundKey as shown in Figure 2. GIFT is similar to PRESENT, but the half round key is computed and no addroundkey is performed at the start of encryption.

- sBoxLayer: S-box with 16 values (Substitute box) to replace from 4 bits to 4 bits in the state.
- pLayer: Applying a bit-level shift to the state.
- AddRoundKey: Adding state to the 64-bit word of the round key using finite field arithmetic.

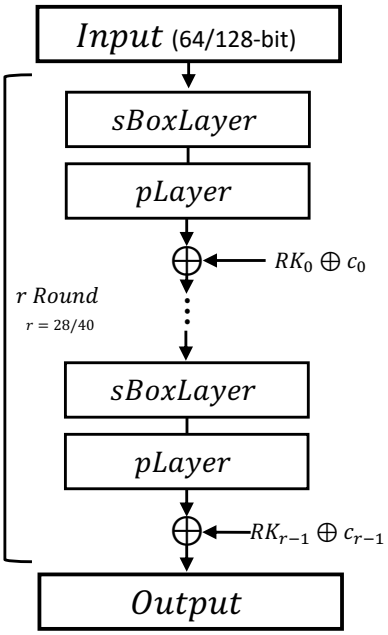


**Figure 2.** GIFT structure.

The key schedule and round constants are the same in both versions of GIFT. The difference is round key extraction. The round key is first extracted from the key state before the key state update. In the case of GIFT-64, two 16-bit words of the key state are extracted as round keys.

$$RK = U \parallel V,\ U \leftarrow W6,\ V \leftarrow W7$$

In the case of GIFT-128, four 16-bit words in the key state are extracted as a round key.

$$RK = U \parallel V,\ U \leftarrow W2 \parallel W3,\ V \leftarrow W6 \parallel W7$$

The key state is updated as follows:

$$\begin{gathered} W_0 \parallel W_1 \leftarrow W_0 \ggg 2 \parallel W_1 \ggg 12 \\ W_2 \parallel W_3 \leftarrow W_2 \parallel W_3 \\ W_4 \parallel W_5 \leftarrow W_4 \parallel W_5 \\ W_6 \parallel W_7 \leftarrow W_6 \parallel W_7 \end{gathered}$$

### *2.3. CUDA Framework*

CUDA (Compute Unified Device Architecture) is a GPGPU technology that enables parallel processing algorithms performed by GPUs to be written using high-level programming languages including C/C++ and FOTRAN. CUDA was developed by Nvidia, and this architecture requires an Nvidia GPU and special stream processing drivers. Under the CUDA framework, the GPU code (known as kernel) are executed in a single-instruction-multiple-data (SIMD) manner. CUDA organizes the parallel workload in grid, threads and blocks shown in Figure 3. The maximum size of a block is limited to 1024, and 32 threads are bundled as a warp. All 32 threads in a warp are executed simultaneously, and different warps are scheduled by the Streaming Multi-processor (SM).

To implement the exhausted key search, one password is decrypted in each thread, and many decryptions are performed in parallel using many threads. To achieve good performance, it is necessary to select appropriate values for the number of threads per block and the number of blocks per grid. It is also necessary to use sufficient threads and blocks to keep the GPU busy by fully utilizing all the computational resources available. Another key feature in GPU to achieve high-performance computing is the availability of a user-managed cache, which is known as shared memory. It can use shared data between

threads within the same block; this avoids writing back and forth to the slow global memory. Registers are the fastest memory in GPU, but the size is limited and it is only accessible within a thread.

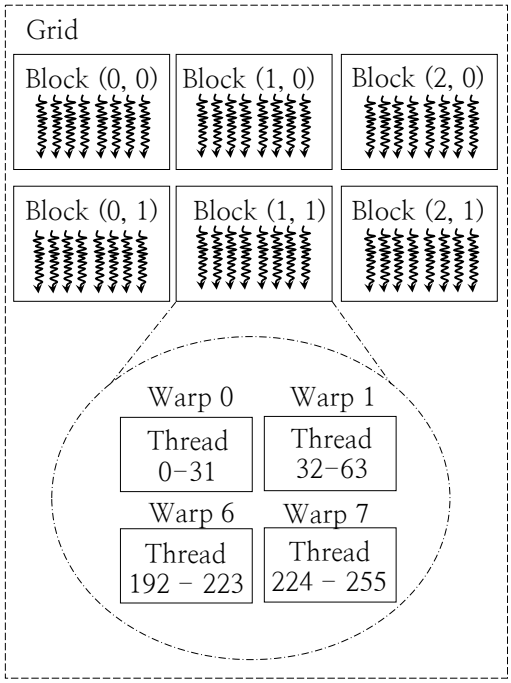


**Figure 3.** Grid, Thread, Block, and Warp configuration in CUDA.

## 3. Proposed Method

This section describes an optimized implementation technique based on bit-slicing for lightweight block ciphers on CUDA GPUs.

### *3.1. Bit-Slicing Techniques*

In order to apply the bit-slicing technique, the operation process of the cryptographic algorithm must be converted to a combination of simple logic gates such as AND, OR, and NOT. In addition, a packing process to convert the data into multi-block bit-sliced representation, and an unpacking process to return to the original representation, are required. Therefore, the overhead caused by these processes must be considered. On the other hand, the bit permutation in the linear layer of PRESENT and GIFT is complex and not efficient to implement in software, but it can be solved by applying the bit-slicing technique. Since the bit-slicing technique encrypts multiple blocks, it is suitable for platforms that require high throughput, such as GPU devices. A bit-slicing implementation that computes many blocks in parallel can save several computational domains but can incur overhead because it consumes many registers. The GPU environment provides relatively sufficient registers to efficiently perform this highly advanced bit-slice implementation. As shown in Table 1, 32 64-bit plaintext blocks are implemented in parallel since the GPU core is a 32-bit architecture.

Two types of overheads created by the bit-slicing technique are considered in the proposed implementation. When an implementation uses the same key, the key schedule is pre-launched to generate round keys to avoid duplicated work. When the bit-slicing technique is applied, there are 32 plaintexts encrypted in parallel, the round key is also increased by 32×. In this case, the kernel may have an additional delay in retrieving the pre-computed round keys from the global memory. On the other hand, these pre-computed round keys can be stored in the registers to achieve low latency, but it may not work if the number of registers are more than supported amount. For instance, Hajihassani et al. [4] hard-coded the AES round keys during compilation to avoid using an excessive number

of registers, but this seriously limits the usefulness of such software. To resolve this issue, we proposed to calculate the round keys on-the-fly instead of generating them in advance. Although this may introduce some additional computation overhead, it is still beneficial because the round key generation for lightweight ciphers is generally not complex.

On top of that, when the bit-slicing technique is applied, the additional overhead of packing and unpacking to change the state occurs. This overhead increases as the number and size of blocks to be converted increase. In lightweight ciphers using 64-bits, the block size is relatively small, so it is suitable for high-level bit-slicing on the GPU.

**Table 1.** Bit-slicing representation from using 64 32-bit registers $R_0, \ldots, R_5$ to process 32 blocks $b^0, \ldots, b^7$ in parallel where $b^i_j$ refers to the $j$-th bit of the $i$-th block.

| | ***Block*0** | ***Block*1** | ***Block*2** | ***Block*3** | $\cdots$ | $\cdots$ | ***Block*28** | ***Block*29** | ***Block*30** | ***Block*31** |
|---|---|---|---|---|---|---|---|---|---|---|
| $R_0$ | $b_0^0$ | $b_0^1$ | $b_0^2$ | $b_0^3$ | $\cdots$ | $\cdots$ | $b_0^{28}$ | $b_0^{29}$ | $b_0^{30}$ | $b_0^{31}$ |
| $R_1$ | $b_1^0$ | $b_1^1$ | $b_1^2$ | $b_1^3$ | $\cdots$ | $\cdots$ | $b_1^{28}$ | $b_1^{29}$ | $b_1^{30}$ | $b_1^{31}$ |
| $R_2$ | $b_2^0$ | $b_2^1$ | $b_2^2$ | $b_2^3$ | $\cdots$ | $\cdots$ | $b_2^{28}$ | $b_2^{29}$ | $b_2^{30}$ | $b_2^{31}$ |
| $R_3$ | $b_3^0$ | $b_3^1$ | $b_3^2$ | $b_3^3$ | $\cdots$ | $\cdots$ | $b_3^{28}$ | $b_3^{29}$ | $b_3^{30}$ | $b_3^{31}$ |
| $R_4$ | $b_4^0$ | $b_4^1$ | $b_4^2$ | $b_4^3$ | $\cdots$ | $\cdots$ | $b_4^{28}$ | $b_4^{29}$ | $b_4^{30}$ | $b_4^{31}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $R_{63}$ | $b_{63}^0$ | $b_{63}^1$ | $b_{63}^2$ | $b_{63}^3$ | $\cdots$ | $\cdots$ | $b_{63}^{28}$ | $b_{63}^{29}$ | $b_{63}^{30}$ | $b_{63}^{31}$ |

### *3.2. PRESENT Block Cipher*

#### 3.2.1. S-Box in Bit-Slicing Form

In order to apply the bit-slicing technique, the S-box must be converted to a combination of circuits using gates including AND, OR, and NOT. There could be several possible combinations to obtain these circuits, but only the one that performs the least bit operations is used. This is to ensure a minimum latency between the input and output of the circuits, which is optimized for performance. Due to this reason, we use the S-box implementation of [15], which performs 14-bit operations on four input bits, which is the most optimized one for the PRESENT S-box. This is detailed in the Algorithm 1.

**Algorithm 1** PRESENT S-box

**Input:** $x0, x1, x2, x3$
**Output:** $x0, x1, x2, x3$
1: $T1 = x2 \oplus x1$
2: $T2 = x1 \wedge T1$
3: $T3 = x0 \oplus T2$
4: $T5 = x3 \oplus T3$
5: $T2 = T1 \wedge T3$
6: $T1 = T1 \oplus T5$
7: $T2 = T2 \oplus x1$
8: $T4 = x3 \vee T2$
9: $x2 = T1 \oplus T4$
10: $x3 = \sim x3$
11: $T2 = T2 \oplus x3$
12: $x0 = x2 \oplus T2$
13: $T2 = T2 \vee T1$
14: $x1 = T3 \oplus T2$
15: $x3 = T5$

S-box receives 4 32-bit inputs, performs logical operations, and operates 16 times on 16 32-bit states expressed by bit slicing in one round.

#### 3.2.2. Permutation Layer (pLayer)

The efficiency of bit-slicing implementation is most visible in pLayer. Instructions are executed efficiently if the operand is given in the size of the computer word (e.g., 32-bit or 64-bit for modern processors). In this case, it can complete the given instructions at one time. Since the pLayer permutes every single bit, a lot of instructions are needed to implement it in software. Note that bit-wise permutation is especially cumbersome to implement in software, but it is not a problem in hardware, as it can be performed through simple wiring. Therefore, bit-wise permutation in software is more optimized if it is implemented using a lookup table. In bit-slicing, each register stores bits in blocks, so bit shifts can be treated as permuting values in a register, which is as efficient as in hardware implementation. Therefore, in bit-slicing, bit shifts in pLayer are calculated as simple value shifts between registers. The implementation of the proposed technique replaces permutation $P$ by performing $P0$ operation after $P1$ operation. The proposed technique is based on the idea of [15], in which the computation of the $P$ in two successive rounds by replacing the permutation $P$ with $P0$ and $P1$. In each round, instead of using $P$, $P0$ and $P1$ are used alternately. Executing $P0$ and $P1$ consumes fewer cycles than $P$. However, this process requires the round key to be computed beforehand. Since we do not perform the key schedule in advance, additional calculations are required. Therefore, $P$ was used as a combination of $P0$ and $P1$ in one round using only the advantage that $P0$ and $P1$ were faster.

Since several blocks are encrypted by bit-slicing, the key update operation is also modified to the form shown Algorithm 2. Note that the rotation step is now revised; it only performs shift operations and utilizes the S-boxes combinational circuit form. round_counter is changed and added in 32-bit form. To prevent overusing registers in the GPU, the proposed technique calculates the round keys on-the-fly. Therefore, the key update is performed every round.

**Algorithm 2** PRESENT RoundKey Update

**Input:** $Round\ r,\ X = \{x_0, x_1x_2, \cdots, x_{78}, x_{79}\}$
**Output:** $Y = \{y_0, y_1y_2, \cdots, y_{78}, y_{79}\}$
1: $For\ i = 1\ to\ 80\ do$
2: $\quad y_i = x_{i+61} mod\ 80$
3: $PRESENT\_SBOX(y_1,\ y_2,\ y_3,\ y_4)$
4: $y_{60} = y_{60} \oplus (((r \wedge 16) \gg 4)) * 0xFFFFFFFF)$
5: $y_{61} = y_{61} \oplus (((r \wedge 8) \gg 3)) * 0xFFFFFFFF)$
6: $y_{62} = y_{62} \oplus (((r \wedge 4) \gg 2)) * 0xFFFFFFFF)$
7: $y_{63} = y_{63} \oplus (((r \wedge 2) \gg 1)) * 0xFFFFFFFF)$
8: $y_{64} = y_{64} \oplus ((r \wedge 16) * 0xFFFFFFFF)$

### *3.3. GIFT Block Cipher*

#### 3.3.1. S-Box in Bit-Slicing Form

Similar to PRESENT, S-box in GIFT can be expressed also be expressed in a combinational circuit, which is described in the following logical operations:

$$
\begin{aligned}
S1 &\leftarrow S1 \oplus (S0 \wedge S2)\\
S0 &\leftarrow S0 \oplus (S1 \wedge S3)\\
S2 &\leftarrow S2 \oplus (S0 \vee S1)\\
S3 &\leftarrow S3 \oplus S2\\
S1 &\leftarrow S1 \oplus S3\\
S3 &\leftarrow S3\\
S2 &\leftarrow S2 \oplus (S0 \wedge S1)\\
S0, S1, S2, S3 &\leftarrow S3, S1, S2, S0
\end{aligned}
$$

The fact that bit permutation works efficiently in bit-slicing also applies to GIFT. In the proposed scheme, the last step of S-box $\{S0, S1, S2, S3\} \leftarrow \{S3, S1, S2, S0\}$ is not performed, but a part of sLayer is performed in pLayer. As S1 and S3 change, the value of the S-box is transferred to the pLayer. Since it has been converted to a bit-slicing representation, it is possible to move the values in consideration of the changes in the values of S1 and S3 without additional operation. Through this technique, the execution time of the S-box can be further reduced. Using an S-box composed of bit-wise operators for four input bits is shown in the Algorithm 3.

**Algorithm 3** GIFT S-box

**Input:** `x3, x2, x1, x0`
**Output:** `x3, x2, x1, x0`
1: $x1 = (x0 \wedge x2) \oplus x1$
2: $x0 = (x1 \wedge x3) \oplus x2$
3: $x3 = x3 \oplus x2$
4: $x1 = x1 \oplus x3$
5: $x3 = \sim x3$
6: $x2 = (x0 \wedge x1) \oplus x2$

#### 3.3.2. Permutation Layer (pLayer)

The last step of S-box $\{S0, S1, S2, S3\} \leftarrow \{S3, S1, S2, S0\}$ is processed in pLayer. Since S1 and S3 have changed, value permutation takes place in the pLayer. Since the movement path of bits is known in pLayer, if the existing S1 and S3 are moved appropriately, the operation is possible without additional operation. The process is similar to the pLayer of the PRESENT block cipher. Similarly, pLayer with bit-sliced implementation is computed as a simple value shift between registers. Due to this reason, we found that the efficiency of bit-slicing implementation is most visible in pLayer.

#### 3.3.3. Roundkey Update

Since several blocks are encrypted by bit-slicing, the picture key update operation is also changed. Rotation operations are changed to move values. In particular, round_counter is changed and added to a 32-bit form as shown in the CUDA C/C++ code as below:

```
__device__ void addRoundConstant(uint32_t* X, uint32_t r)
{
    uint32_t GIFT_RC[28] =
    {0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D,
    0x3B, 0x37, 0x2F, 0x1E, 0x3C, 0x39, 0x33,
    0x27, 0x0E, 0x1D, 0x3A, 0x35, 0x2B, 0x16,
    0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B };
    X[ 3] ^= (GIFT_RC[r] & 0x1)*0xFFFFFFFF;
    X[ 7] ^= ((GIFT_RC[r] >> 1) & 0x1)*0xFFFFFFFF;
    X[11] ^= ((GIFT_RC[r] >> 2) & 0x1)*0xFFFFFFFF;
    X[15] ^= ((GIFT_RC[r] >> 3) & 0x1)*0xFFFFFFFF;
    X[19] ^= ((GIFT_RC[r] >> 4) & 0x1)*0xFFFFFFFF;
    X[23] ^= ((GIFT_RC[r] >> 5) & 0x1)*0xFFFFFFFF;
    X[63] ^= 0xFFFFFFFF;
}
```

Similar to the implementation of PRESENT, the proposed technique calculates the round key in on-the-fly without generating the round key in advance.

### *3.4. Overhead of Data Transmission between CPU and GPU*

To perform encryption on a large amount of data, it is necessary to transfer the plaintext from the CPU to the GPU for encryption, which is very time-consuming. The problem is even more obvious in key exhaustive searches, wherein the guessed key values are transferred to the GPU and create a huge transmission delay. Tezcan [10] eliminates the process of transferring plaintext to the GPU by using the counter mode, in which the counter values are generated in the kernel as a key value, and increase by one in each thread. This idea was also adopted by us and applied to the bit-slicing technique. However, converting

the counter value in normal representation to the bit-slicing representation introduces overhead that cannot be ignored. Hence, we propose a technique to create a counter that is already in a bit-slicing representation, thus avoiding the expensive conversion process. This is shown in the CUDA C/C++ code as below:

```
uint32_t tid = blockIdx.x * blockDim.x + threadIdx.x;
uint32_t subkeys[80] = {0,};
subkeys[0] = 0x55555555;
subkeys[1] = 0x33333333;
subkeys[2] = 0x0F0F0F0F;
subkeys[3] = 0x00FF00FF;
subkeys[4] = 0x0000FFFF;

for (uint32_t i = 0; i < 31; i++) {
    subkeys[i+5] = ((tid & (1<<i)) >> i) * 0xFFFFFFFF;
}
```

The proposed method creates a counter in a packed state instead of packing it into a bit-slicing expression after generating a counter value. 32-bit blocks are processed simultaneously. Therefore, the same value is used for [0]∼[4] of subkeys, and the counter is incremented by 32. In the next step, the thread uses the counter value incremented by 32 times its own thread number as the key value.

## 4. Evaluation

In this Section, the performance of the proposed method is compared. Firstly, we measure the performance of exhaustive key search using PRESENT and GIFT block ciphers, which were optimized through the proposed techniques presented in Section 3. Next, we measure the performance of data encryption using PRESENT and GIFT applying the proposed method. Finally, we measure the overhead of packing and unpacking for GIFT-64 and GIFT-128 with different sizes of plaintext blocks. Starting from 1, the size of the block is doubled, and the size of the thread is increased by 32.

All the experiments are performed on an RTX 3060 GPU platform. The throughput per second was measured by considering data transmission between CPU and GPU, as well as the GPU kernel execution. In the implementation, 32 blocks per thread are calculated. When the kernel runs once, data of 32 × size of data block × number of threads per block (thread size) × number of blocks per grid (block size) is processed. For example, if the block size of data is 64-bits, the thread size is 256, and the block size is 1024, 536,870,912-bits of data are processed. The number of threads per block (thread size) and the number of blocks per grid (block size) affect performance. The right choice is necessary to achieve optimal performance. To confirm this, the performance was measured by changing the number of threads per block and the number of blocks per grid.

### *4.1. Exhaustive Search*

As shown in Figures 4–7 are the experimental results for each cipher. The X-axis is blocks per grid, the Y-axis is gigabits per second, the legend is threads per block. In general, the larger the blocks per grid, the higher the throughput. However, the threads per block are not high, the higher the throughput. According to blocks per grid, there are threads per block with the highest throughput. There are threads per block showing the highest throughput per block per grid. For example, in Figure 5, block 4096 per grid achieves the highest throughput with 256 threads per block. However, for blocks 8192 per grid, the highest throughput is achieved with 128 threads per block. All four ciphers implemented show similar characteristics. The achieved throughput of the proposed method is shown in Table 2. The peak throughput of PRESENT-80, PRESENT-128, GIFT-64, and GIFT-128 were 553.932 Gbps, 529.952 Gbps, 583.859 Gbps, and 214.284 Gbps, respectively. PRESENT-80 implemented by the proposed method as shown in Table 3 showed 4.39 times higher performance than 1.8852 Gblocks/s in RTX 3070 environment of [10], even in a GPU environment with lower performance (i.e., RTX 3060).

**Table 2.** The highest throughput results of the exhaustive search version of the proposed technique. The unit of throughput is Gigabit per second (Gbps).

| Cipher | Blocks per Gird | Threads per Block | Throughput |
|---|---|---|---|
| PRESENT-80 | 131,072 | 192 | 553.932 |
| PRESENT-128 | 131,072 | 128 | 529.952 |
| GIFT-64 | 131,072 | 64 | 583.859 |
| GIFT-128 | 16,384 | 160 | 214.284 |

**Table 3.** Comparison of implementation results for CPU and GPU.

| Cipher | Ref. | Model | Throughput |
|---|---|---|---|
| PRESENT-80 ECB | [20] | intel i7-9750H | 0.001 |
| PRESENT-80 ECB | [12] | Tesla V100 | 14.15 |
| PRESENT-80 ECB using multiple stream | [12] | Tesla V100 | 24.525 |
| PRESENT-80 ECB | this work | RTX 3060 | 24.264 |
| PRESENT-80 CTR | [10] | RTX 3070 | 115.73 |
| PRESENT-128 ECB | [12] | Tesla V100 | 14.15 |
| PRESENT-128 ECB using multiple stream | [12] | Tesla V100 | 24.525 |
| PRESENT-128 ECB | this work | RTX 3060 | 24.522 |
| GIFT-64 ECB | [21] | intel i7-9750H | 0.003 |
| GIFT-64 ECB | this work | RTX 3060 | 85.283 |
| GIFT-128 ECB | [21] | intel i7-9750H | 0.014 |
| GIFT-128 ECB | this work | RTX 3060 | 10.723 |
| PRESENT-80 EXHAUSTIVE | [10] | RTX 3070 | 120.64 |
| PRESENT-80 EXHAUSTIVE | this work | RTX 3060 | 553.932 |

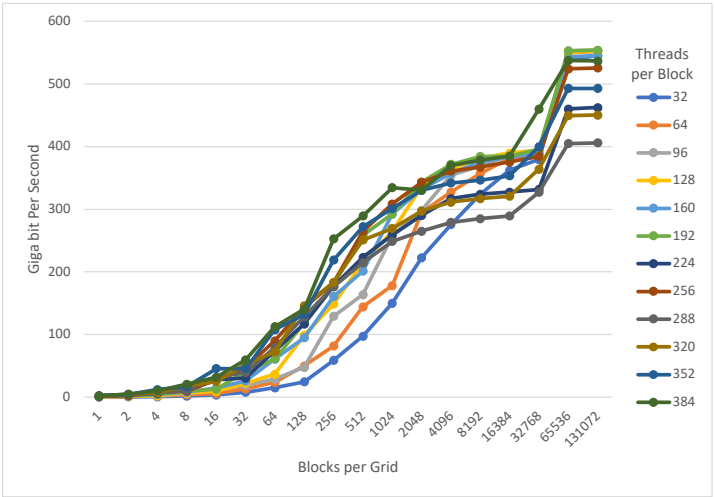


**Figure 4.** PRESENT-80 exhaustive key search: Comparison of the number of key-search per second.

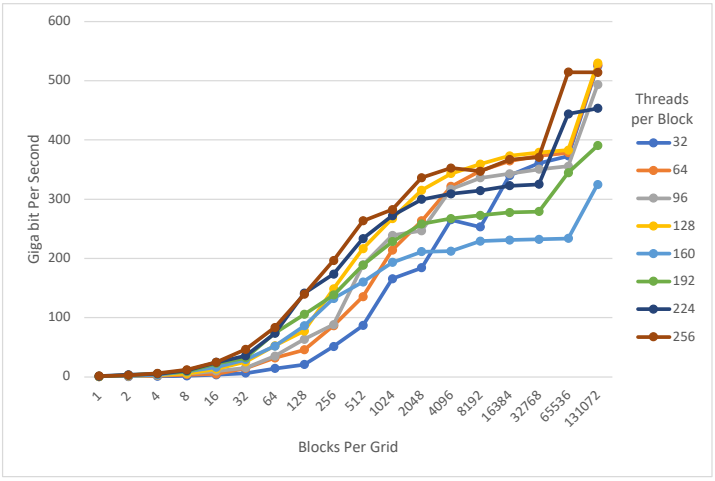


**Figure 5.** PRESENT-128 exhaustive key search: Comparison of the number of key-search per second.

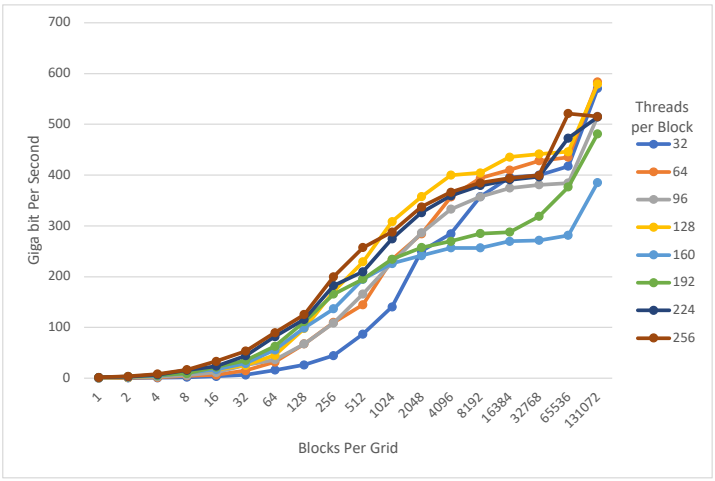


**Figure 6.** GIFT-64 exhaustive key search: Comparison of the number of key-search per second.

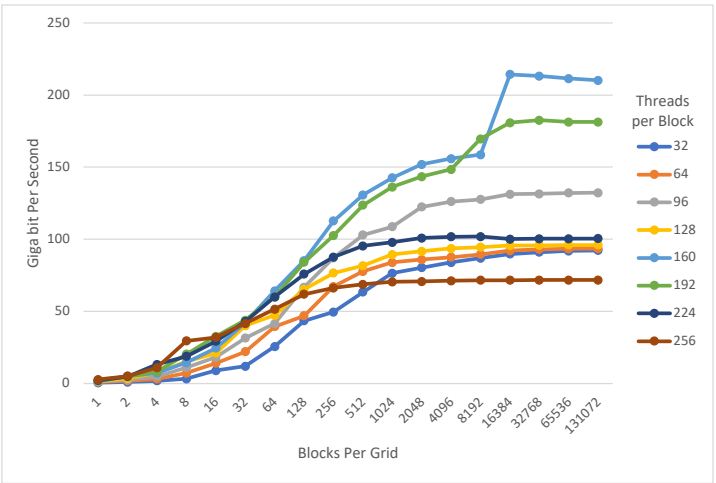


**Figure 7.** GIFT-128 exhaustive key search: Comparison of the number of key-search per second.

### *4.2. Data Encryption*

Figures 8–11, are the experimental results for each cipher. The X-axis is blocks per grid, Y-axis is gigabits per second, the legend is threads per block. Similarly to the experiment of Exhaustive Search, the larger the blocks per grid, the higher the throughput. According to

blocks per grid, there are threads per block with the highest throughput. All four ciphers implemented show similar characteristics. The achieved throughput of the proposed method is shown in Table 4. In the case of exhaustive search, high throughput was achieved because there is no such delay factor. However, in the case of ECB mode, a large amount of data is transferred between the CPU and GPU architectures, and there is a delay due to the large amount of data being called stored in the GPU's global memory.

**Table 4.** The highest throughput results of the data encryption version of the proposed technique. The unit of throughput is Gigabit per second (Gbps).

| Cipher | Blocks per Gird | Threads per Block | Throughput |
|---|---|---|---|
| PRESENT-80 | 32,768 | 128 | 24.264 |
| PRESENT-128 | 32,768 | 64 | 24.522 |
| GIFT-64 | 32,768 | 32 | 85.283 |
| GIFT-128 | 32,768 | 96 | 10.723 |

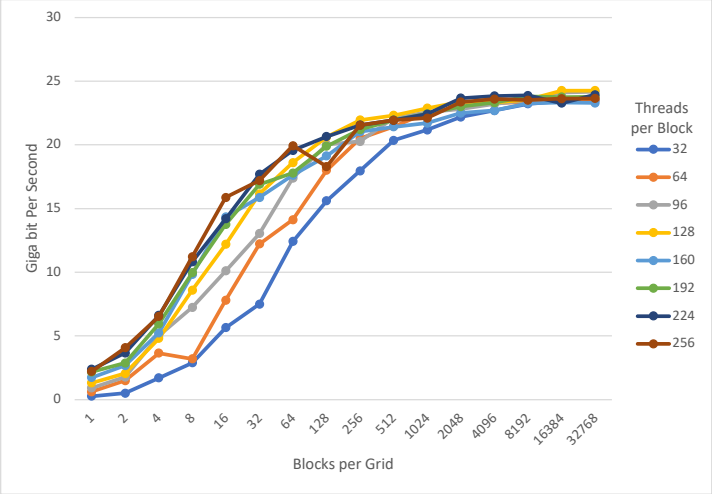


**Figure 8.** PRESENT-80 encryption: Comparison of the number of encryption per second.

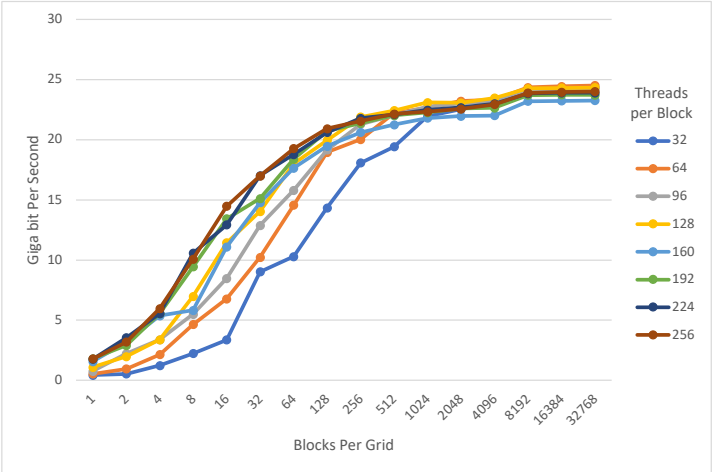


**Figure 9.** PRESENT-128 encryption: Comparison of the number of encryption per second.

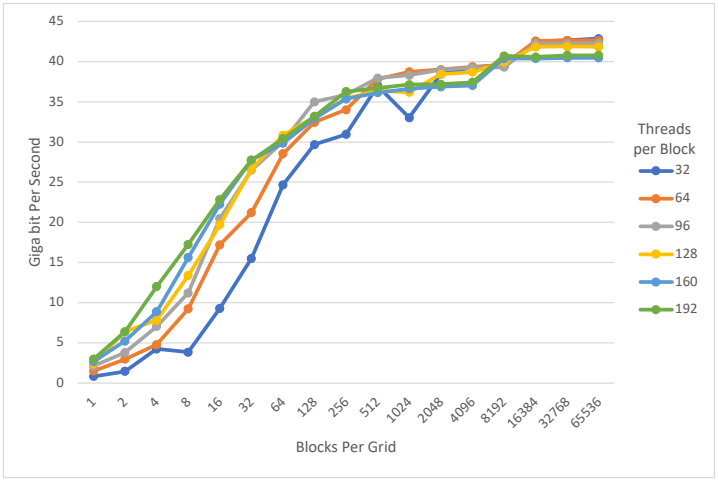


**Figure 10.** GIFT-64 encryption: Comparison of the number of encryption per second.

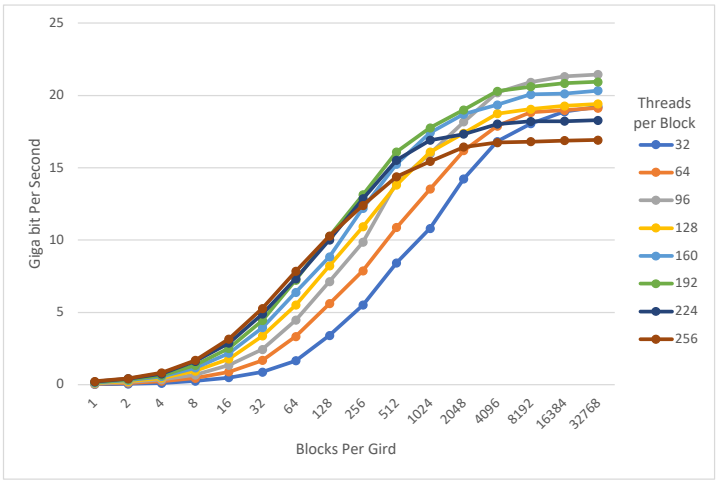


**Figure 11.** GIFT-128 encryption: Comparison of the number of encryption per second.

### *4.3. Shared Memory*

One of the main features of CUDA is that it provides access to fast shared memory, which is essentially a user-managed cache. It can be used to store lookup tables, providing fast access speed to constantly used data. Many prior GPU implementations utilized this feature by storing the round keys in the shared memory. In this paper, we compare the performance achieved by the proposed technique with one using shared memory.

In the proposed method, the key is updated during the round operation. Since the key update is performed with a round function, $80 \times 32$ bits for an 80-bit key and $128 \times 32$ bits for a 128-bit key are used for one algorithm operation. Since the size of the round keys that need to be stored is not large, they can be written to shared memory. We propose a technique for allocating shared memory as needed for each thread. However, the size of the shared memory used needs to be carefully considered. It does not exceed the maximum allowable range. Therefore, the number of threads per block must be selected so that the shared memory does not exceed the maximum allowable size. In addition, available shared memory has a different capacity depending on the GPU architecture in the GPU architecture. In this regard, the number of threads may vary depending on the shared memory available. It is implemented using dynamic allocation. Unfortunately, we did not see a huge performance difference. There was no significant difference in encryption per second. Figure 12 is a comparison of PRESENT-80 before and after shared memory use. Comparisons were made by increasing blocks per grid. When the blocks per grid were large enough, the performance difference between the two was small.

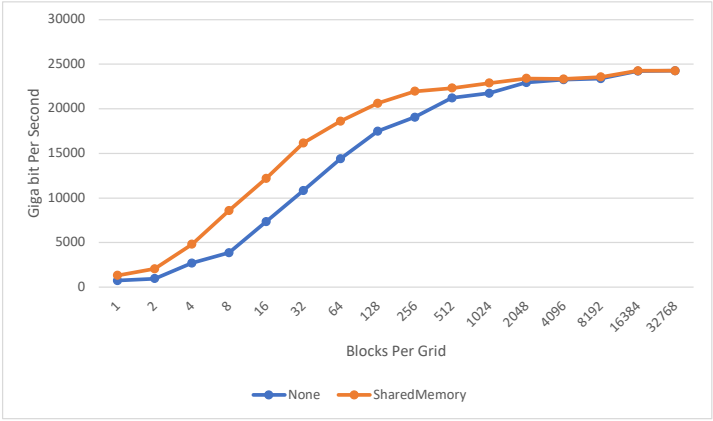


**Figure 12.** PRESENT-80 encryption: Comparison of using shared memory and not using shared memory.

## 5. Conclusions

We implemented both PRESENT and GIFT block ciphers with bit-slicing technology on CUDA GPU. We optimize the algorithm of cipher by applying the bit-slicing technique and minimize the overhead caused by applying the bit-slicing technique in normal format. Proposed implementation was evaluated on the RTX 3060 with NVIDIA Ampere architecture. We are able to achieve 553.932 Gbps PRESENT-80, 529.952 Gbps PRESENT-128, 583.859 Gbps GIFT-64, and 214.284 Gbps GIFT-128 throughput for encryption, respectively. For exhaustive search, 24.264 Gbps PRESENT-80, 24.522 Gbps PRESENT-128, 85.283 Gbps GIFT-64, and 10.723 Gbps GIFT-128 throughput are achieved, respectively. As a result of comparing the proposed PRESENT method with previous works, the bit-slicing method showed higher throughput than the existing table implementation method. The GIFT implementation is the first on GPU architectures. The implementation of PRESENT achieved approximately 4x higher throughput than the state-of-art work. As a result of our implementation, we were able to confirm that a lightweight cipher was suitable for a high degree of bit-slicing because of its simple key schedule and small blocks. In this paper, we dealt with PRESENT and GIFT of SPN structure. We believe that applying the bit-slicing technique to other lightweight ciphers is also effective. In the future, we plan to apply it to other design ciphers to which bit slicing is applicable.

**Author Contributions:** Conceptualization, S.L.; Software, H.K. and S.E.; Supervision, H.S.; Writing—original draft, H.K.; Writing—review & editing, W.-K.L. and 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. Singh, S.; Sharma, P.K.; Moon, S.Y.; Park, J.H. Advanced lightweight encryption algorithms for IoT devices: Survey, challenges and solutions. *J. Ambient. Intell. Humaniz. Comput.* **2017**, 1–18. [CrossRef]
2. Seo, H.; Park, T.; Heo, S.; Seo, G.; Bae, B.; Hu, Z.; Zhou, L.; Nogami, Y.; Zhu, Y.; Kim, H. Parallel implementations of LEA, revisited. In *International Workshop on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 318–330.
3. Lee, W.K.; Goi, B.M.; Phan, R.C.W. Terabit encryption in a second: Performance evaluation of block ciphers in GPU with Kepler, Maxwell, and Pascal architectures. *Concurr. Comput. Pract. Exp.* **2019**, *31*, e5048. [CrossRef]
4. Hajihassani, O.; Monfared, S.K.; Khasteh, S.H.; Gorgin, S. Fast AES implementation: A high-throughput bitsliced approach. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *30*, 2211–2222. [CrossRef]
5. Gupta, N.; Jati, A.; Chauhan, A.K.; Chattopadhyay, A. Pqc acceleration using gpus: Frodokem, newhope, and kyber. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *32*, 575–586. [CrossRef]
6. An, S.; Seo, S.C. Efficient parallel implementations of LWE-based post-quantum cryptosystems on graphics processing units. *Mathematics* **2020**, *8*, 1781. [CrossRef]
7. Lee, W.K.; Hwang, S.O. High throughput implementation of post-quantum key encapsulation and decapsulation on GPU for Internet of Things applications. *IEEE Trans. Serv. Comput.* **2021**. [CrossRef]

8. Jang, K.B.; Kim, H.J.; Lim, S.J.; Seo, H.J. Parallel Implementation of SPECK, SIMON and SIMECK by Using NVIDIA CUDA PTX. *J. Korea Inst. Inf. Secur. Cryptol.* **2021**, *31*, 423–431.
9. Han, K.; Lee, W.K.; Hwang, S.O. cuGimli: Optimized implementation of the Gimli authenticated encryption and hash function on GPU for IoT applications. *Clust. Comput.* **2022**, *25*, 433–450. [CrossRef]
10. Tezcan, C. Key lengths revisited: GPU-based brute force cryptanalysis of DES, 3DES, and PRESENT. *J. Syst. Archit.* **2022**, *124*, 102402. [CrossRef]
11. 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]
12. Li, P.; Zhou, S.; Ren, B.; Tang, S.; Li, T.; Xu, C.; Chen, J. Efficient implementation of lightweight block ciphers on volta and pascal architecture. *J. Inf. Secur. Appl.* **2019**, *47*, 235–245. [CrossRef]
13. An, S.; Seo, S.C. Highly efficient implementation of block ciphers on graphic processing units for massively large data. *Appl. Sci.* **2020**, *10*, 3711. [CrossRef]
14. Biham, E. A fast new DES implementation in software. In *International Workshop on Fast Software Encryption*; Springer: Berlin/Heidelberg, Germany, 1997; pp. 260–272.
15. Reis, T.; Aranha, D.F.; López, J. PRESENT runs fast. In *International Conference on Cryptographic Hardware and Embedded Systems*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 644–664.
16. Adomnicai, A.; Najm, Z.; Peyrin, T. Fixslicing: A new GIFT representation: Fast constant-time implementations of GIFT and GIFT-COFB on ARM Cortex-M. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**, *2020*, 402–427. [CrossRef]
17. Adomnicai, A.; Peyrin, T. Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**, *2021*, 402–425. [CrossRef]
18. Bogdanov, A.; Knudsen, L.R.; Leander, G.; Paar, C.; Poschmann, A.; Robshaw, M.J.; Seurin, Y.; Vikkelsoe, C. PRESENT: An ultra-lightweight block cipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 450–466.
19. Banik, S.; Pandey, S.K.; Peyrin, T.; Sasaki, Y.; Sim, S.M.; Todo, Y. GIFT: A small present. In *International Conference on Cryptographic Hardware and Embedded Systems*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 321–345.
20. Pepton21. PRESENT-cipher GitHub Repo. 2019. Available online: https://github.com/Pepton21/present-cipher (accessed on 7 October 2022).
21. giftcipher. GIFT GitHub Repo. 2020. Available online: https://github.com/giftcipher/gift (accessed on 7 October 2022).