IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 11 July 2022, accepted 30 July 2022, date of publication 16 August 2022, date of current version 22 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3198707

RESEARCH ARTICLE

Efficient Parallel Implementations of PIPO Block Cipher on CPU and GPU

HOJIN CHOI, (Student Member, IEEE), AND SEOG CHUNG SEO, (Member, IEEE)
Department of Financial Information Security, Kookmin University, Seoul 02707, South Korea

Corresponding author: Seog Chung Seo (scseo@kookmin.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) by the Korea Government through the Ministry of Science and ICT (MSIT) (6G Autonomous Security Internalization-Based Technology Research to Ensure Security Quality at all Times, 50%) under Grant A2021-0270, and in part by the National Research Foundation of Korea (NRF) by the Korea Government through MSIT (50%) under Grant 2022R1C1C1013368.

ABSTRACT Data encryption is essential for securely managing clients' data in servers in data-centric ICT environment. Clients must encrypt the data before transmitting it to severs or other clients. Encrypting a large volumne of data requires a lot of time. Therefore, in order for servers and clients to not only secure but also smoothly communicate each other, the optimization of data encryption is necessary on both the server-side and the client-side. Especially, the server environment is responsible for managing/processing lots of data from clients. In this paper, we present two kinds of highly optimized PIPO cipher software in CPU and GPU environment, respectively. PIPO was proposed in ICISC'19 as a lightweight block cipher. For optimization, we take full advantage of two parallel processing technologies: AVX-related instructions in CPU and NVIDIA CUDA platform in GPU. Regarding the optimization in CPU environment, we process several plaintext blocks such as 32 and 64 blocks with the proper use of AVX2 and AVX-512 instruction sets and the proposed arithmetic techniques, respectively. Regarding the optimization on GPU environment, we propose a data alignment/data combining methods, and PTX inline assembly utilization method considering the characteristics of GPU architecture. In Intel Core i9-11900K (3.50GHz) architecture, our PIPO software utilizing AVX-2 has a performance improvement on 839.64% (resp. 985.46% [AVX-512]) compared to the existing reference code (Regarding AVX-512, this is the first PIPO software using AVX-512 instructions as far as we know). Finally, in RTX 2080Ti, our PIPO GPO software shows throughput of up to 1110.08 Gbps.

INDEX TERMS AVX-2, AVX-512, block cipher, CUDA, GPU, parallel processing, PIPO, SIMD.

I. INTRODUCTION

With the advent of the 4th industrial revolution, many data is transmitted through network communication. In particular, various services are provided as low-end devices and high-end devices communicate. Low-end devices mainly use IoT (Internet of Things), edge computing, and embedded devices, while high-performance devices use CPU (Central Processing Unit) and GPU (Graphics Processing Unit) architectures. If data are not encrypted in network communication, the data be exposed to network packets, leading to security incidents such as exposure to personal information and confidential information. Therefore, data encryption is essential for network communication. On low-end devices, cryptographic operations may be overloaded with the limited specifications of low-end devices. Thus, lightweight encryption algorithms are recommended for low-end devices. Mainly, the server communicates with many embedded devices and provides services. Therefore, the server must operate and manage data efficiently and quickly. Additionally, if the size of the data is large, the server should apply data encryption optimization to avoid the problem of communication delayed with the client.

PIPO (Plug-In and Plug Out) encryption algorithm is a lightweight block encryption algorithm that operates efficiently and quickly in an 8-bit AVR embedded device environment [1], [2]. Research on PIPO optimization in embedded devices (RISC-V, ARM, etc.) is ongoing [3], [4], [5]. However, research on PIPO optimization is insufficient

The associate editor coordinating the review of this manuscript and approving it for publication was Easter Selvan Suviseshamuthu.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

for high-end devices such as CPU/GPU. CPU/GPU architecture is mainly used in the server environment. And in network communication using PIPO cryptographic algorithm, since the server communicates with many clients, the server must efficiently operate the PIPO algorithm encryption and decryption. Therefore, in this paper, we proposed a parallel implementation method of PIPO using AVX (Advanced Vector eXtensions)-2 and AVX-512 of SIMD (Signle Instruction Multiple Data) instruction sets applicable in CPU environment. Additionally, we proposed a parallel implementation of PIPO using NVIDIA GPU CUDA C and Inline Assembly Parallel Thread Execution (PTX). The proposed PIPO optimization method enables efficient computational processing in an environment using CPU/GPU (eg, server, database encryption/decryption). Our implementation of PIPO using AVX-2 has a performance improvement of 876.72% compared with the naive ported GPU version of the CPU reference code. Finally, our PIPO implementation using GPU CUDA shows a throughput of up to 1,110.08 Gbps. To the best of our knowledge, our implementation of PIPO using GPU CUDA C and PTX is the first proposed approach. Additionally, the implementation of PIPO using our AVX instruction is the first proposed techniques.

A. CONTRIBUTIONS

The contribution of this paper is as follows:

1) **PIPO implementation methods using AVX-2 & AVX-512 instruction sets**
   AVX-2 and AVX-512 instruction registers support 256-bit and 512-bit, respectively. We considered the fact that the operation of the PIPO algorithm is a bitwise operation in 8-bit units and AVX-2/AVX-512 instructions that do not provide bitwise operation in 8-bit units. As a result, we proposed a 16-bit data combination and a PIPO algorithm bitwise operation processing method. Finally, we proposed a method that can effectively apply AVX-2/AVX-512 operation to the S-Layer and R-Layer processes of the PIPO cryptographic algorithm. As a result, in Intel Core i9-11900K (3.50GHz, 8 core and 16 processors) architecture, our implementation of PIPO using AVX-2 (AVX-512) has a performance improvement of 876.72% (resp. 985.46%) compared to the reference code.
2) **PIPO implementation methods using CUDA C language in GPU Architecture**
   GPU architecture uses many threads to enable massive computational parallelism. CUDA is a technology that allows the effective use of GPU architectures in computing languages. In this paper, we propose a PIPO implementation scheme and an optimization strategy using CUDA C and the inline assembly language PTX. In particular, we propose a 16-bit PIPO bit operation method that can effectively use PTX instructions.Finally, our PIPO implementation using our CUDA C achieved 275.60Gpbs (GTX 1650 architecture), 1,110.08Gbps (RTX 2080Ti architecture). Our PIPO implementation provides performance gains of 175.82% (GTX 1650 architecture) and 167.50% (RTX 2080Ti architecture) compared with the naive ported GPU version of the CPU reference code. In other words, our proposed implementation of PIPO on GPU architecture is effective regardless of the type of GPU architecture.

The remainder of this paper is organized as follows. In Section 2, we briefly review PIPO Block Cipher. In Section 3, we briefly provide an overview of AVX-2 & AVX-512 and GPU environment. In Section 4, we propose implementation and optimization methods for PIPO algorithm for each environment. In Section 5, we compare the performance. In Section 6, we conclude the paper.

II. OVERVIEW OF PIPO

At the ICISC (International Conference on Information Security and Cryptology) conference in 2021, Kim *et al.* presented PIPO, a 64-bit lightweight block cipher using a key of 128/256-bit [1]. PIPO offers efficient performance in 8-bit AVR software. Additionally, PIPO algorithm consists of 8-bit unit operations, and can be effectively implemented by applying bit-slicing method. Finally, PIPO cryptographic algorithm can apply an efficient higher-order masking implementation by providing minimal non-linear operations [1], [2]. Table 1 shows the definition of the block size and the number of key length rounds of each parameter of PIPO. When using a 128-bit key, 13 rounds is performed, and when using a 256-bit key, 15 rounds is performed. Each round consists of a non-linear operation stage expressed in S-Layer, a linear operation stage expressed in R-Layer, and a stage in which Round Key and eXclusive-OR (XOR) operation are processed.

TABLE 1. PIPO algorithm parameter [1], [2].

	Block Size(bit)	Key Size(bit)	Round
PIPO-64/128	64	128	13
PIPO-64/256	64	256	15

The internal structure of PIPO consists of S-Layer, R-Layer, and Addroundkey. The S-Layer consists of an 8-bit input and an 8-bit output process. The internal process of S-Layer consists of one 3-bit input/output S-box and two 5-bit input/output S-boxes. For a 3-bit S-box, the number of non-linear (linear) operations is limited to 3 (4). For a 5-bit S-box, the number of non-linear (linear) operations is limited to 4 (7). The PIPO S-Layer can be implemented by bit-slice method and Look Up table method [1], [2]. Algorithm 1 is a bit-slice implementation method for PIPO S-Layer [2]. In the PIPO S-Layer bit-slice implementation method, 22 XOR ($\oplus$) operations, 7 AND (&) operations, and 4 OR (|) operations are performed. R-Layer consists of bit rotations of bytes. The internal state of the PIPO algorithm consists of 64-bit, and 8 bit rotations are processed in R-Layer process. Figure 1 is a summary of the PIPO algorithm.

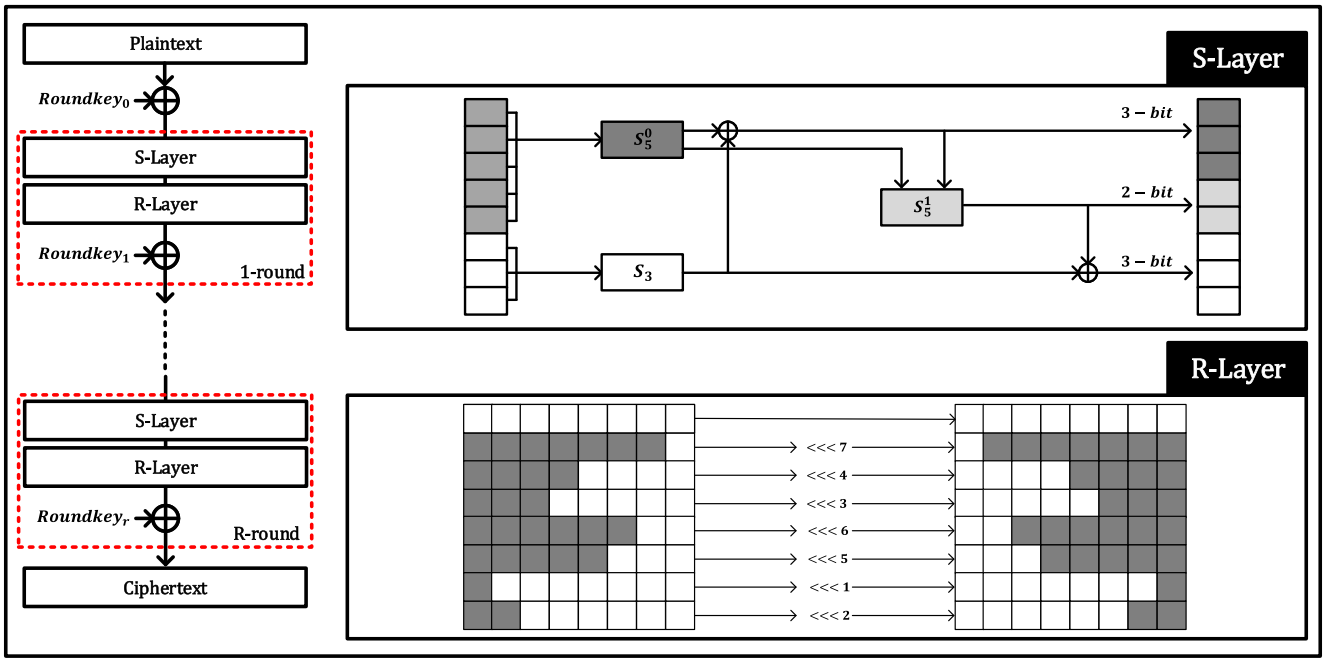


FIGURE 1. PIPO Algorithm Summary [2].

Algorithm 1 PIPO S-Layer Bit-Slice Implementation Method [2]

Require: 64-bit input$(x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0)$
Ensure: 64-bit output$(x'_7, x'_6, x'_5, x'_4, x'_3, x'_2, x'_1, x'_0)$

S_5^0 operation
$x_5 \leftarrow x_5 \oplus (x_7 \,\&\, x_6)$
$x_4 \leftarrow x_4 \oplus (x_3 \,\&\, x_5)$
$x_7 \leftarrow x_7 \oplus x_4$
$x_6 \leftarrow x_6 \oplus x_3$
$x_3 \leftarrow x_3 \oplus (x_4 \mid x_5)$
$x_5 \leftarrow x_5 \oplus x_7$
$x_4 \leftarrow x_4 \oplus (x_5 \,\&\, x_6)$
S_3 operation
$x_2 \leftarrow x_2 \oplus (x_1 \,\&\, x_0)$
$x_0 \leftarrow x_0 \oplus (x_2 \mid x_1)$
$x_1 \leftarrow x_1 \oplus (x_2 \,\&\, x_0)$
$x_2 \leftarrow \sim x_2$
Extend XOR
$x_7 \leftarrow x_7 \oplus x_1$
$x_3 \leftarrow x_3 \oplus x_2$
$x_4 \leftarrow x_5 \oplus x_0$
S_5^1 operation
$tmp_0 \leftarrow x_7$; $tmp_1 \leftarrow x_3$; $tmp_2 \leftarrow x_4$;
$x_6 \leftarrow x_6 \oplus (tmp_0 \,\&\, x_5)$
$tmp_0 \leftarrow tmp_0 \oplus x_6$
$x_6 \leftarrow x_6 \oplus (tmp_2 \mid tmp_1)$
$tmp_1 \leftarrow tmp_1 \oplus x_5$
$x_5 \leftarrow x_5 \oplus (tmp_6 \mid tmp_2)$
$tmp_2 \leftarrow tmp_2 \oplus (tmp_1 \,\&\, tmp_0)$
Truncate XOR and bit change
$x_2 \leftarrow x_2 \oplus tmp_0$
$tmp_0 \leftarrow x_1 \oplus tmp_2$
$x_1 \leftarrow x_0 \oplus tmp_1$
$x_0 \leftarrow x_7$; $x_7 \leftarrow tmp_0$
$tmp_1 \leftarrow x_3$; $x_3 \leftarrow x_6$
$x_6 \leftarrow tmp_1$; $tmp_2 \leftarrow x_4$
$x_4 \leftarrow x_5$; $x_5 \leftarrow tmp_2$
return $(x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0)$

1) RELATED WORKS

Since the publication of the PIPO block cipher algorithm, various studies on the PIPO cipher algorithm have been conducted. The results of implementation of PIPO encryption algorithm in various embedded devices such as 32/64-bit ARM processors, and RISC-V processors were published in [3], [4], and [5]. Song *et al.* proposed a method to use the NEON engine in the ARM-cortex-A environment and optimize the PIPO implementation register scheduling. As a result, Song *et al.*'s implementation of PIPO shows 15.1 Cycle Per Byte (CPB) (PIPO-64/128) and 19.6 CPB (PIPO-64/256) [3]. Kwak *et al.* proposed a register scheduling and logic scheme for the parallel implementation of PIPO in a RISC-V environment. As a result, the implementation by Kwak *et al.* shows a performance of 113.7 CPB (PIPO-64/128) [4]. Eum *et al.* proposed an internal operation optimization method and register scheduling method to implement PIPO optimization in a 64-bit ARM environment. As a result, the implementation of Eum *et al.* shows performance of 3.9 CPB (PIPO-64/128) and 4.8

(PIPO-64/256) [5]. In addition, many side-channel attack countermeasures research on PIPO block cipher had been published such as differential fault attack, masking methods etc [6], [7].

Recently, the results of the implementation of cryptographic algorithms in GPU architecture equipment have attracted attention. Research results on the implementation and optimization of block cipher algorithms in GPU architectures have been published in [8], [9], [10], [11], [12], [13] and [14]. Additionally, implementation results in cryptographic algorithms in GPU architectures such as PQC and public key algorithms have been published [15], [16], [17], [18].

Research on PIPO optimization in embedded devices (RISC-V, ARM, etc.) is ongoing. However, research on PIPO optimization is insufficient for high-end devices such as CPU/GPU. For servers that communicate with many clients, data processing speed needs to be accelerated to provide an efficient service. Therefore, research on optimization of the PIPO encryption algorithm in high-end devices for servers should be conducted for data processing speed in the PIPO algorithm-based communication environment.

III. TARGET PLATFORMS

A. SIMD: AVX-2 & AVX-512 IN CPU

SIMD is a parallel computing technology that processes multiple blocks of data as one instruction. SIMD provides higher computational performance than the existing SISD (Single Instruction Single Data) structure. SIMD technologies provided by Intel CPUs include MMX (MultiMedia eXtension), SSE (Streaming SIMD Extension), and AVX. Intel SIMD technology intrinsic function allows the use those instructions in a non-assembly C/C++ environment [19].

AVX-2 is a SIMD technology installed in Intel CPU Cores made in 2011, starting with Intel Sandy Bridge. Conventional SIMD SSE-2 technology uses a 128-bit register size and AVX-2 is configured with a 256-bit register size. The AVX-2 register allows one instruction to process 256-bit data. Recently, high-performance Intel CPUs are equipped with AVX-512 registers along with AVX-2 registers. AVX-512 SIMD technology was first installed in Intel Core X series (Skylake-X) CPU equipment released in 2017 (for server PCs, AVX-512 was first supported by Skylake-SP). The size of AVX-512 register consists of 512-bit. AVX-512 provides up with 7x performance improvement to traditional SSE technology [20].

B. GPU AND CUDA

A GPU architecture is a device created to process the graphics elements of a computer, and in its early development was primarily used for graphics processing for design purposes. The purpose of GPUs is to enhance parallelism and to quickly compute simple or specific operations through parallelism. Compared to CPUs with about 10 high-performance cores and auxiliary equipment, GPUs are designed with a simple combination of hundreds of cores. In other words, the GPU is a device that maximizes the operation speed through parallel operation of graphics processing and special operations.

With the development of GPU architecture technology, GPUs have begun to be useful for common tasks that CPUs are responsible for. Based on this, General Purpose Computing on GPU (GPGPU) technology was developed to process general-purpose computing operations through the GPU architecture. Consequently, in 2006, NVIDIA developed and launched CUDA, a GPGPU technology that can run on GPU architectures. CUDA is a technology that allows developers to easily write algorithms that can run on GPUs in various programming languages.

CUDA provides access to the GPU's instruction set and memory, and was developed to make effective use of the GPU. GPU architectures handle the same tasks in parallel with a single core, called a thread. Threads are configured in block/grid units and grouped into one configuration, and data can be shared through shared memory inside a block. GPUs also have global memory and constant memory.

IV. PROPOSED PIPO IMPLEMENTATIONS

In this section, we proposed PIPO implementation methods for each environment. We proposed implementation methodologies and optimization methods in AVX-2 & AVX-512 SIMD, and GPU environments that enable parallel environments.

A. OVERALL DESIGN PRINCIPLE FOR EFFICIENT PARALLEL COMPUTATION

Typically, when representing input messages in computing, data is organized into 8-bit data types (either unsigned char type or char type). Additionally, both S-Layer and R-Layer operations in the PIPO cryptographic algorithm consist of 8-bit unit datatype operations. Algorithms with these 8-bit arithmetic units have no problems in environments that provide 8-bit operators (AVR, C language, etc.). However, an environment that does not provide an 8-bit operator causes a problem. Environments that do not provide 8-bit operators should use 16/32/64-bit operators to handle data operations. In other words, if computing environment does not provide 8-bit data type operation, 8-bit data should be stored in other data type and data operation is processed. This process is very inefficient in a limited environment and in terms of computation.

The main operations used by S-Layer and R-Layer of the PIPO cryptographic algorithm are bitwise operators. AVX-2 & AVX-512 and GPU PTX instructions provide operators to handle bitwise operations. However, the minimum unit of bitwise operators of GPU PTX instruction consists of a 16-bit format. Additionally, the minimum data type of AVX-2 & AVX-512 register bit shift instruction is a 16-bit data format. Therefore, in GPU architecture, we proposed a method of processing bit-wise operators by storing two 8-bit plaintexts in a 16-bit data type. When handling PIPO internal

Algorithm 2 PIPO Implementation Using AVX-512

Require: 64 Plaintext Block
Require: Round key(RK)
Operation Define
XOR(A, B) ← _mm512_xor_si512(A,B)
AND(A, B) ← _mm512_and_si512(A,B)
OR(A, B) ← _mm512_or_si512(A,B)
Plaintext & Roundkey Register Setting
for $j = 0$ to 7 **do**
  __mm512i Reg[j] ← _mm512_setr_epi8($PT^i[j]$)
  __mm512i rk[j] ← _mm512_set1_epi8($RK[j]$)
  Reg[j] ← XOR(Reg[j], rk[j]) //Initial Round
end for
S-Layer operation
for $i = 1$ to 14 **do**
  S_5^0 operation
  Reg[5] ← XOR(Reg[5], AND(Reg[7], Reg[6]))
  Reg[4] ← XOR(Reg[4], AND(Reg[3], Reg[5]))
  Reg[7] ← XOR(Reg[7], Reg[4])
  Reg[6] ← XOR(Reg[6], Reg[3])
  Reg[3] ← XOR(Reg[3], OR(Reg[4], Reg[5]))
  Reg[5] ← XOR(Reg[5], Reg[7])
  Reg[4] ← XOR(Reg[4], AND(Reg[5], Reg[6]))
  S_3 operation
  Reg[2] ← XOR(Reg[2], AND(Reg[1], Reg[0]))
  Reg[0] ← XOR(Reg[0], AND(Reg[2], Reg[1]))
  Reg[1] ← XOR(Reg[1], AND(Reg[2], Reg[0]))
  Not(∼) operation
  tmp ← _mm512_set1_epi8(0xff)
  Reg[2] ← _mm512_andnot_si512(Reg[2], tmp)
  Extend XOR
  Reg[7] ← XOR(Reg[7], Reg[1])
  Reg[3] ← XOR(Reg[3], Reg[2])
  Reg[4] ← XOR(Reg[4], Reg[0])
  S_5^1 operation
  tmp_0 ← Reg[7]; tmp_1 ← Reg[3]; tmp_2 ← Reg[4]
  Reg[6] ← XOR(Reg[6], AND(tmp_0, Reg[5]))
  tmp_0 ← XOR(tmp_0, Reg[6])
  Reg[6] ← XOR(Reg[6], OR(tmp_2, tmp_1))
  tmp_1 ← XOR(tmp_1, Reg[5])
  Reg[5] ← XOR(Reg[5], OR(tmp_2, Reg[6]))
  tmp_2 ← XOR(tmp_2, OR(tmp_1, tmp_0))
  Truncate XOR and bit change
  Reg[2] ← XOR(Reg[2], tmp_0
  tmp_0 ← XOR(Reg[1], tmp_2
  Reg[1] ← XOR(Reg[0], tmp_1
  Reg[0] ← Reg[7]; Reg[7] ← tmp_0
  tmp_1 ← Reg[3]; Reg[3] ← Reg[6]
  Reg[6] ← tmp_1; tmp_2 ← Reg[4]
  Reg[4] ← Reg[5]; Reg[5] ← tmp_2
  R-Layer Operation processing[Algorithm 3]
  Addroundkey operation
  for $j = 0$ to 7 **do**
    rk[j] ← _mm512_set1_epi8($RK[8 * i + j]$)
    Reg[j] ← XOR(Reg[j], rk[j])
  end for
end for
return Reg

operations through this method, there are considerations for S-Layer and R-Layer, respectively.

Our implementation applies a bit-slicing method to handle the S-Layer of the PIPO cryptographic algorithm. The bit-slicing method is a method for processing the S-box through bitwise operations. The bit operation has an independent operation in a 1-bit unit. Thus, the bit slicing processing for the two plaintexts (16-bit data format) does not affect each other (bitwise operators operate independently on each bit). However, you should consider the following: A 16-bit datatype must contain plaintext with the same index. That is, the upper 8-bits of the first plaintext and the second plaintext must be contained in the same 16-bit datatype. The main operation of R-Layer is bit rotation. If two 8-bit data types are stored in a 16-bit data type and then moved as much as fixed offset, the correct value is not obtained. This means that if moved by offset, 2 plaintexts that should exist in different regions can be mixed. We handled the R-Layer bit rotation of bit-masking. Our PIPO implementation targets data parallelism. Task parallelism handles computational tasks by splitting them. The process of PIPO operation does not consist of many operations. Task parallelism is effective for computation-heavy and granular operations. Therefore, we implemented PIPO as a data parallelization method that effectively handles large amounts of data.

B. PROPOSED PIPO IMPLEMENTATION USING AVX-2 & AVX-512 INSTRUCTIONS

AVX-2 uses 256-bit registers to process data in parallel. Our implementation using AVX-2 computes PIPO encryption/decryption through parallel processing of 32 blocks of 64-bit plaintext. AVX-512 uses 512-bit registers to process data onto parallel. Our implementations using AVX-512 compute PIPO encryption/decryption by processing 64 blocks of 64-bit plaintext in parallel. Algorithm 2 is a pseudocode for our PIPO implementation using AVX-512.

1) AVX REGISTER SCHEDULING

Our implementation uses eight 256/512-bit registers. Each register contains plaintext data with the same index in each plaintext. That is, when plaintext is expressed as an 8-bit data type, each plaintext has plaintext data with 8 indexes. AVX-2 0-th (reg_0) 256-bit register stores plaintext data with the 0-th index of each plaintext. In the case of parallel processing of 32 plaintexts, data are allocated to all areas in the AVX-2 register. In this way, each of the AVX-2 256-bit 8 registers allocates plaintext data, and each register has data information from a different index. For AVX-512 registers, 8 registers are used. AVX-512 has a 512-bit register, thus 64 blocks of plaintext can be processed in parallel.

Algorithm 3 PIPO R-Layer Implementation Using AVX-512

Require: AVX-512 Registers(Reg[1], ..., Reg[7])
Operation Define
AND(A, B) ← _mm512_and_si512(A,B)
OR(A, B) ← _mm512_or_si512(A,B)
SHIFT_L ← _mm512_slli_epi16_(A, count)
SHIFT_R ← _mm512_srli_epi16_(A, count)
SET_16 ← _mm512_set1_epi16(count)
R-Layer operation
1: _mm512i tmp_0, tmp_1, tmp_2, tmp_3, bit_0, bit_1 ← 0
7-bit Left-rotation part
2: bit_0 ← SET_16(0 × 0101) bit-masking value
3: bit_1 ← SET_16(0xFEFE) bit-masking value
4: tmp_0 ← AND(bit_0, Reg[1])
5: tmp_1 ← AND(bit_1, Reg[1])
6: tmp_2 ← SHIFT_L(tmp_0, 7)
7: tmp_3 ← SHIFT_R(tmp_1, 1)
8: Reg[1] ← OR(tmp_2, tmp_3)
4-bit Left-rotation part
9: bit_0 ← SET_16(0 × 0F0F) bit-masking value
10: bit_1 ← SET_16(0xF0F0) bit-masking value
11: tmp_0 ← AND(bit_0, Reg[2])
12: tmp_1 ← AND(bit_1, Reg[2])
13: tmp_2 ← SHIFT_L(tmp_0, 4)
14: tmp_3 ← SHIFT_R(tmp_1, 4)
15: Reg[2] ← OR(tmp_2, tmp_3)
3-bit Left-rotation part
16: bit_0 ← SET_16(0 × 1F1F) bit-masking value
17: bit_1 ← SET_16(0xE0E0) bit-masking value
18: tmp_0 ← AND(bit_0, Reg[3])
19: tmp_1 ← AND(bit_1, Reg[3])
20: tmp_2 ← SHIFT_L(tmp_0, 3)
21: tmp_3 ← SHIFT_R(tmp_1, 5)
22: Reg[3] ← OR(tmp_2, tmp_3)
6-bit Left-rotation part
23: bit_0 ← SET_16(0 × 0303) bit-masking value
24: bit_1 ← SET_16(0xFCFC) bit-masking value
25: tmp_0 ← AND(bit_0, Reg[4])
26: tmp_1 ← AND(bit_1, Reg[4])
27: tmp_2 ← SHIFT_L(tmp_0, 6)
28: tmp_3 ← SHIFT_R(tmp_1, 2)
29: Reg[4] ← OR(tmp_2, tmp_3)
5-bit Left-rotation part
30: bit_0 ← SET_16(0 × 0707) bit-masking value
31: bit_1 ← SET_16(0xF8F8) bit-masking value
32: tmp_0 ← AND(bit_0, Reg[5])
33: tmp_1 ← AND(bit_1, Reg[5])
34: tmp_2 ← SHIFT_L(tmp_0, 5)
35: tmp_3 ← SHIFT_R(tmp_1, 3)
36: Reg[5] ← OR(tmp_2, tmp_3)
1-bit Left-rotation part
37: bit_0 ← SET_16(0 × 7F7F) bit-masking value
38: bit_1 ← SET_16(0 × 8080) bit-masking value
39: tmp_0 ← AND(bit_0, Reg[6])
40: tmp_1 ← AND(bit_1, Reg[6])
41: tmp_2 ← SHIFT_L(tmp_0, 1)
42: tmp_3 ← SHIFT_R(tmp_1, 7)
43: Reg[6] ← OR(tmp_2, tmp_3)
2-bit Left-rotation part
44: bit_0 ← SET_16(0 × 3F3F) bit-masking value
45: bit_1 ← SET_16(0xC0C0) bit-masking value
46: tmp_0 ← AND(bit_0, Reg[7])
47: tmp_1 ← AND(bit_1, Reg[7])
48: tmp_2 ← SHIFT_L(tmp_0, 2)
49: tmp_3 ← SHIFT_R(tmp_1, 6)
50: Reg[7] ← OR(tmp_2, tmp_3)
51: **return** Reg

TABLE 2. AVX register scheduling of plaintext (|*P*[*i*]| = 8-bit).

AVX Register	Data Information
Reg_0	$PT[0]$ of each plaintext (LSB 8-bit)
Reg_1	$PT[1]$ of each plaintext
Reg_2	$PT[2]$ of each plaintext
Reg_3	$PT[3]$ of each plaintext
Reg_4	$PT[4]$ of each plaintext
Reg_5	$PT[5]$ of each plaintext
Reg_6	$PT[6]$ of each plaintext
Reg_7	$PT[7]$ of each plaintext (MSB 8-bit)

Our implementation of PIPO using AVX-2 allocates 32 plaintexts in 8 256-bit registers. In our implementation, Reg_0 allocates an LSB 8-bit for each plaintext. The Reg_7 register allocates MSB 8-bit. Table 2 summarizes the allocation data for each register. When 64-bit plaintext is divided into 8-bit units, each plaintext is divided into 8 pieces of data. We specified the partitioned plaintext blocks as PT_0, PT_1, PT_2, ..., PT_7. We specified the i-th plaintext out of 32/64 plaintexts as PT^i. That is, the j-th 8-bit unit block of the i-th plaintext can be expressed as PT^i_j or $PT^i[j]$. In Algorithm 2, Reg[j] stores the plaintext message. Reg[0] contains an 8-bit LSB of each plaintext. That is, Reg[0] contains the LSB 8-bits of 64 plaintext messages.

2) S-LAYER PROCESS IN CPU

There are look-up table reference method and bit-slicing methods for calculating PIPO S-Layer. When using look-up table method in the PIPO S-Layer using AVX, plaintext information in AVX registers needs to be converted into 8-bit data format. The converted 8-bit data format is processed by the S-Layer through the look-up table method. After S-Layer operation processing, data are allocated to AVX registers. In this process, 2 data conversions and 1 memory access occur. Since the data conversion method incurs a lot of computational load, our implementation chose the bit-slicing method as the method for handling the S-Layer.

When processing S-Layer as a bit-slicing implementation method, the main operation of S-Layer consists of bit-operation operation. Bit operations (*XOR* (⊕), *AND* (&), *OR* (|), *NOT* (∼)) are performed in 1-bit units, and operations independent of other bits are possible. Therefore, parallel

processing was performed through the bit-wise operator instruction provided by AVX-2 & AVX-512.

Algorithm 2 (line 7 to 35) is a PIPO S-Layer bit-slice implementation method using AVX-512 instruction. In a general S-Layer bit-slice implementation, 1,408 XOR operations, 448 AND operations, and 256 OR operations are used to compute 64 message blocks. In the AVX-512-based S-Layer bit-slice implementation method that parallel operates on 64 message blocks, XOR instruction operator is used 22 times, AND instruction operator is used 7 times, and the OR instruction operation is used 4 times. Our PIPO S-Layer implementation based on AVX-512 minimizes operation calls through data parallel operation processing.

3) R-LAYER PROCESS IN CPU

R-Layer process updates the internal state through the left rotation operation as much as an offset. The main operation of PIPO is executed 8-bit units. Therefore, the offset of the left rotation was also set in 8-bit units.

We considered the following in our PIPO R-Layer implementation: First, in AVX-2 instruction, rotation operation does not exist. Therefore, we used AVX-2 shift operation and OR operation to handle the rotation operation of R-Layer. However, the minimum data unit of the AVX-2 shift operation instruction is a 16-bit data unit. Therefore, our PIPO implementation using AVX-2 uses 16-bit data combined to handle the R-Layer process. In the case of AVX-512, there is a rotation instruction (__mm512_ror(rol)_epi32(64)). However, the minimum operation unit of the AVX-512 rotation instruction is 32-bit. Therefore, in our R-Layer implementation using AVX-512, we applied R-Layer implementation method to AVX-2.

Second, each register contains plaintext data with the same index. Therefore, each register operates to perform a rotation operation as much as the offset. a general rotation operation in the 16-bit data format that combines two 8-bit data can invade each 8-bit data range. Therefore, to handle the R-Layer, our implementation added bit-masking operation. Algorithm 3 is R-Layer computation pseudocode using AVX-512. SHIFT_L instruction is a bitwise left-shift operator for data stored in AVX-512 registers in 16-bit units. For Reg[0] = (R[0] ‖ R[1] ‖ ……‖ R[31]) (R[i] is 16-bit data), the result for SHIFT_L(Reg[0], 3) is (R[0] $\ll$ 3 ‖ R[1] $\ll$ 3 ‖……‖ R[31] $\ll$ 3). The SET_16 instruction configures the AVX-512 registers in 16-bit data format. The result for SET_16(0 $\times$ 1F1F) is R = (0 $\times$ 1F1F‖0 $\times$ 1F1F‖……‖0 $\times$ 1F1F). The bit masking method is a method of processing rotation operation while preventing data invasion by a bit shift operation. In our PIPO implementation, two 8-bit data are composed of one 16-bit data to use 16-bit unit bitwise operators. A shift operation may result in a violation of each data. Therefore, the bit masking value preserves the data encroachment range for shifts and allows rotation operations to be handled.

C. PROPOSED PIPO IMPLEMENTATION IN GPU ENVIRONMENT

A GPU uses many threads, and the threads perform the same operation in parallel. All threads are divided into Grid/Block units. The performance of the GPU architecture operation differs depending on the division unit of Grid/Block. Additionally, GPU architecture has several memory areas. Commonly used areas of GPU memory consist of global memory, shared memory, and constant memory. GPU affects performance depending on the number/method of memory access. Therefore, to improve performance on the GPU, the memory area access method and number of times should be minimized.

PTX is an inline assembly language available in CUDA C. The assembly language is a low-level computer programming language that maps to machine language, and assembly language was developed for specific types of processors. Assembly language has the advantage of being able to directly correspond to the machine language through instructions and communicate directly with the machine (architecture). In other words, an assembly language has the fastest instruction execution speed among the programing languages.

In this section, we proposed a PIPO implementation method using the coalesced memory access method, which is an optimization method for GPU equipment, and CUDA PTX inline assembly. In our implementation of PIPO using GPU architecture, one thread computes encryption on plaintext. In other words, our PIPO implementation consisted of data parallelism. One thread consists of two-block parallel encryption considering the PTX bitwise operator. In other words, our PIPO implementation can encrypt the number of threads * 2 blocks of plaintext simultaneously.

TABLE 3. PTX bit-operation syntax.

Syntax	Operation	type
xor.type d, a, b	d = a $\oplus$ b	.pred, .b16, .b32, .b64
and.type d, a, b	d = a & b	.pred, .b16, .b32, .b64
or.type d, a, b	d = a | b	.pred, .b16, .b32, .b64
not.type d, a	d = $\sim$a	.pred, .b16, .b32, .b64
shl.type d, a, b	d = a <<b	.b16, .b32, .b64
shr.type d, a, b	d = a >>b	.b16, .b32, .b64

1) S-LAYER PROCESS ON GPU

Memory access on the GPU architectures causes performance degradation. Among the computational processing methods of S-Layer, look-up table frequently accesses memory. Therefore, our PIPO implementation on GPU has chosen the bit-slicing method. S-Layer operations with bit-slicing methods consist of bitwise operators. Table 3 is the bit-wise operator instruction used in the PTX implementation. The minimum data type for bit-wise operators of the PTX assembly is 16 bits. Therefore, as with AVX, two 8-bit plaintexts are combined into a 16-bit data type. Since bitwise operators are 1-bit operations, they have a bit-independent operation structure. Therefore, combining plaintexts does not affect each other. The combined plaintext is processed through PTX bit-wise operator instruction.

Algorithm 4 PIPO S-Layer Bit-Slicing Implementation Method Using PTX Inline Assembly

Require: 16-bit type 128-bit state (2 Plaintext)(state[0], ..., state[7])

```
asm("{\n\t"\\PTX inline assembly instruction
Register setting
".reg.b16 t0;"
".reg.b16 t1;"
".reg.b16 t2;"
".reg.b16 t3;"
".reg.b16 t4;"
".reg.b16 t5;"
".reg.b16 t6;"
".reg.b16 t7;"
".reg.b16 buf0;"
".reg.b16 buf1;"
".reg.b16 buf2;"
".reg.b16 temp;"
S5^0 operation
"and.b16 t5, %15, %14;"
"xor.b16 t5, t5, %13;"
"and.b16 t4, %11, t5;"
"xor.b16 t4, t4, %12;"
"xor.b16 t7, %15, t4;"
"xor.b16 t6, %14, %11;"
"or.b16 t3, t4, t5;"
"xor.b16 t3, t3, %11;"
"xor.b16 t5, t5, t7;"
"and.b16 t0, t5, t6;"
"xor.b16 t4, t4, t0;"
S3 operation
"and.b16 t2, %8, %9;"
"xor.b16 t2, t2, %10;"
"or.b16 t0, t2, %9;"
"xor.b16 t0, t2, %8;"
"not.b16 t2, t2;"
Extend XOR
"xor.b16 t7, t7, t1;"
"xor.b16 t3, t3, t2;"
"xor.b16 t4, t4, t0;"
S5^1 operation
"mov.b16 buf0, t7;"
"mov.b16 buf1, t3;"
"mov.b16 buf2, t4;"
"and.b16 temp, buf0, t5;"
"xor.b16 t6, t6, temp;"
"xor.b16 buf0, buf0, t6;"
"or.b16 temp, buf2, buf1;
"xor.b16 t6, t6, temp;"
"xor.b16 buf1, buf1, t5;"
"or.b16 temp, t6, buf2;"
"xor.b16 t5, t5, temp;"
"and.b16 temp, buf0, buf1;"
"xor.b16 buf2, buf2, temp;"
Truncate XOR and bit change
"xor.b16 t2, t2, buf0;"
"xor.b16 buf0, t1, buf2;"
"xor.b16 t1, t1, buf1;}"
Data Copy
"mov.b16 %0, t0;"  "mov.b16 %1, t1;"
"mov.b16 %2, t2;"  "mov.b16 %3, t3;"
"mov.b16 %4, t4;"  "mov.b16 %5, t5;"
"mov.b16 %6, t6;"  "mov.b16 %7, t7;"
Input Parameter setting
:"=h(state[0])", "=h(state[1])", "=h(state[2])",
"=h(state[3])""=h(state[4])","=h(state[5])","=h(state[6])",
"=h(state[7])"
:"h(state[0])", "h(state[1])", "h(state[2])", "h(state[3])",
"h(state[4])", "h(state[5])", "h(state[6])", "h(state[7])");
return state (state[0], state[1], ..., state[7])
```

Algorithm 4 is a PIPO S-Layer bit-slicing implementation method using CUDA PTX. We handle the PIPO S-Layer using 12 registers. Eight registers are registers that store input information. For GPU architectures, the latency of accessing memory areas is large. Therefore, we use 8 registers to store input plaintext information, and store intermediate operation values in registers. In our implementation, 3 registers are used to store T values used by the PIPO S-Layer standard code (Algorithm 1). One register is used to store the intermediate value. We encrypt two plaintexts in parallel, considering into account the fact that the bitwise operator of CUDA PTX is at least 16 bits. Compared to 8-bit plaintext encryption, the number of PTX instruction calls to process S-Layer is the same, but our implementation is efficient because it encrypts 2 blocks at the same time.

2) R-LAYER PROCESS ON GPU

R-Layer calculation method of the GPU is similar to the R-Layer processing method of AVX. The minimum data type of PTX bit-wise operator instruction is 16-bit. Therefore, in order to process the R-Layer through the PTX instruction, our implementation combines two 8-bit plaintext data onto a 16-bit data type just to the S-Layer. After that, the rotation operation of the R-Layer was processed through bit-masking. The bit-masking value was processed using the same value used in AVX.

Algorithm 5 is a PIPO R-Layer implementation method using PTX. We include Addroundkey process before storing the values while processing the R-Layer. Our R-Layer implementation uses *AND* (&) bitwise operator 2, shift bitwise operator 2, *OR* (|) bitwise operator 1. That is, the total number of operations used in R-Layer in one round is 14 bitwise operators *AND*, 14 bitwise shift operators, and *OR* 7 bitwise operators.

PIPO reference implementation does not consider bit masking to encrypt a single plaintext. PIPO reference implementation removed two *AND* operators compare to our implementation. However, when porting the PIPO reference implementation to PTX on GPU architectures, the

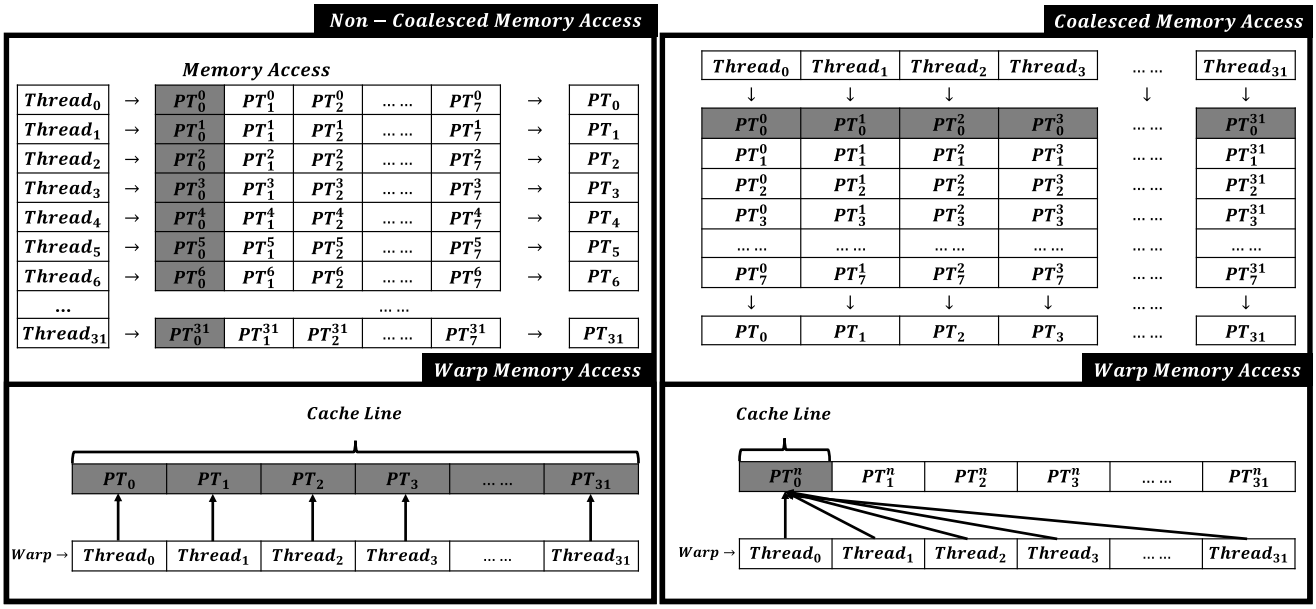


FIGURE 2. Coalesced Memory Access & Non-Coalesced Memory Access.

Algorithm 5 PIPO R-Layer & Key Addition Operation Using PTX Inline Assembly

Require: 16-bit plaintext(state)
Require: 16-bit Roundkey(rk)
Require: bit-masking values(bk_0, bk_1)
Require: rotation value(rv)

```
asm("{\n\t"
  Register setting
".reg.b16     t0;"
".reg.b16     t1;"
".reg.b16     t2;"
"mov.b16      t2, %1; %1 = copy state
"and.b16      t0, %3, t2;" %3 = bk0
"and.b16      t1, %4, t2;" %4 = bk1
"shl.b16      t0, t0, %5;" %5 = rv
"shl.b16      t1, t1, %6;" %6 = 8 − rv
"or.b16       t2, t1, t0;"
xor.b16     %0, t2, %2
  %0 = store state
  %2 = rk
"=h(state)
"h(state)", "h(rk)", "h(bk0)", "h(bk1)", "h(rv)", "h(8 − rv)");
return state
```

minimum unit for bitwise operators in PTX is 16 bits. Therefore, it is very inefficient to change the data type unit and process the operation to encrypt one plaintext. Additionally, in the PIPO reference implementation, the number of bitwise operations used by the R-Layer for 2 blocks is shift bitwise operator 28 times, *OR* bitwise operator 14 times. Therefore, we consider the smallest unit of the PTX bitwise operator. In other words, our PIPO R-Layer implementation reduces the number of bitwise operator calls while processing two plaintexts in parallel.

3) MEMORY ACCESS OPTIMIZATION WITH COALESCED MEMORY ACCESS

The implementation of GPGPU using CUDA is as follows: First, the Host (CPU) transmits the data to be processed to the Device (GPU). Second, the device stores the data received from the host. When data are used in the device operation, the device accesses the memory area where data is stored. The device handles assigned operations. Finally, the device transmits the calculated result value of the Host.

Before calculating cryptographic algorithms on a GPU architecture, CPU transmits plaintext data onto the GPU. Each thread accesses the memory area where plaintext data is stored When calculating the encryption algorithm in the GPU architecture. Unlike other architectures, the GPU architecture has a lot of performance delay when GPU architecture access to memory area. Therefore, an efficient memory access approach is required in GPU architectures.

When the GPU architecture accesses a memory area, memory access in warp units is performed. The warp consists of 32 threads. Before a thread performs encryption, thread accesses the memory area where the plaintext handled by the thread is stored. Memory access occurs inside the warp, rather than being accessed by each thread. If the memory area accessed by all threads belonging to the warp is contiguously configured, the warp can read data onto minimal cache line access. If the memory area accessed by each thread is configured non-contiguously, the Warp will access the cache line of to 32 times.

TABLE 4. AVX-2 & AVX-512 PIPO performance result [21].

AVX-2 PIPO Performance Result			
Version	Clock	Clock Per Byte(CPB)	Ratio
Reference Code(Bitsliced) [21] (C Implementation, 32 Block Encryption)	6,671	26.05	-
Reference Code(Bitsliced) [21] (C Implementation, 64 Block Encryption)	12,917	25.22	-
Our Works (AVX-2 Implementation, 32 Block Encryption)	683	2.67	+876.72%
Our Works (AVX-512 Implementation, 64 Block Encryption)	1,190	2.32	+985.46%

Algorithm 6 Plaintext Copy Using Coalesced Memory Access Method

Require: plaintext(pt)
Require: Roundkey(rk)
Ensure: Ciphertext(ct)
GPU Phase Function
1: uint16_t $index_0 \leftarrow (blockDim.x * blockIdx.x) + threadIdx.x$
2: uint16_t $index_1 \leftarrow (gridIdx.x * blockDim.x)$
3: uint16_t state[8]
GPU plaintext copy
4: state[0] $\leftarrow pt[index_0]$
5: state[1] $\leftarrow pt[index_0 + 1 * index_1]$
6: state[2] $\leftarrow pt[index_0 + 2 * index_1]$
7: state[3] $\leftarrow pt[index_0 + 3 * index_1]$
8: state[4] $\leftarrow pt[index_0 + 4 * index_1]$
9: state[5] $\leftarrow pt[index_0 + 5 * index_1]$
10: state[6] $\leftarrow pt[index_0 + 6 * index_1]$
11: state[7] $\leftarrow pt[index_0 + 7 * index_1]$
Thread operation
12: PIPO(pt, rk, ct) Algorithm 4 & 5
13: **return** ct

Therefore, for effective memory access of the GPU architecture, the data storage method accessed by each thread must be changed. Coalesced memory access means that memory areas are contiguously configured so that effective memory area access is possible. To use the coalesced memory access method, we changed the data storage method the row-wise storage method to the column-wise storage. Each thread accesses the 0-th index of the plaintext data. If the data are stored in column-wise storage methods, each plaintext 0-th index data is constructed consecutively. Therefore, coalesced memory access methods can be applied. Figure 2 shows coalesced memory access and non-coalesced memory access.

Algorithm 6 is our PIPO implementation with coalesced memory access. In our PIPO GPU internal function using CUDA C, we used the number of called threads and the number of blocks. *threadIdx.x* is the thread's unique number, which is the number of the threads belonging to the block. *blockIdx.x* is a unique number of the block, which is the number of the blocks belonging to the grid. *blockDim.x* is the total number of threads a block has. *gridIdx.x* is the total number of blocks called. We use the block/thread's unique number to sequentially configure the position of the plaintext read by each thread. Our memory approach allows for efficient handling of warp-wise message access.

V. PERFORMANCE ANALYSIS

In this section, we present the performance measurement results for our implementation. For our performance measurement, we performed performance measurement on PIPO-64/128. Information on the environments used is included in each section.

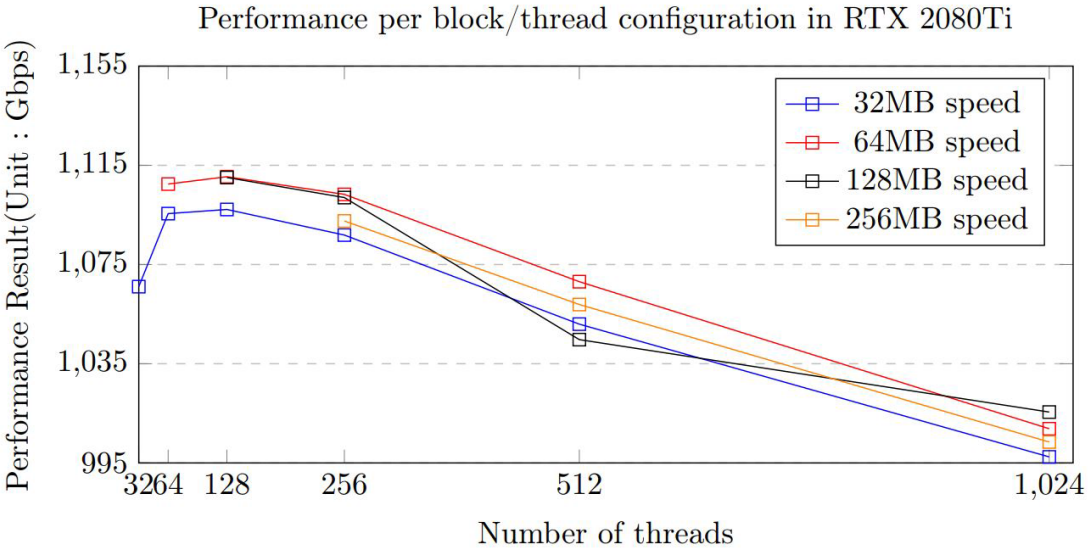
A. PERFORMANCE ANALYSIS ON CPU ENVIRONMENT

Table 4 shows the performance measurement results. Our PIPO implementation performance measurements were performed in an Intel Core i9-11900K (3.50GHz, 8 core and 16 processor) environment. Our experiment did not consider CPU multi-threading, and performed a performance measurement experiment on CPU single-thread. We used the open source code of the PIPO authors for performance measurements [21]. Our implementation utilizing AVX-2 encrypts 32 blocks of plaintext in parallel. In the case of AVX-2, a total of 10,000 parallel encryptions (320,000 plaintext blocks) were performed, and the average value was presented as the result. As a result of PIPO reference code performance measurement, the average value of the performance of 320,000 cryptographic operations was presented. Our implementation performance result using AVX-2 is 683 clocks (2.67 CPB), and there is a performance improvement of 876.72% compared to the reference code.

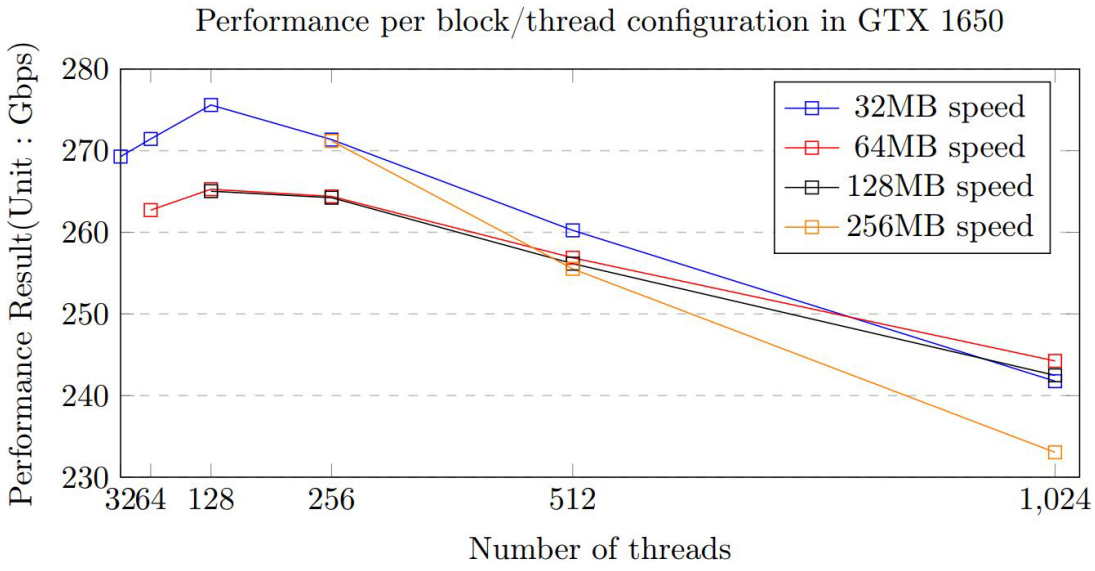
Our implementation utilizing AVX-512 encrypts 64 blocks of plaintext in parallel. For AVX-512, a total of 10,000 parallel encryptions (640,000 plaintext blocks) were performed. The resulting value is the average value. As a result of PIPO reference code performance measurement, the average value of the performance of 640,000 cryptographic operations was presented. Our implementation performance result using AVX-512 is 1,190 clocks (2.32 CPB), which is a performance improvement of 985.46% compared to the reference code.

B. PERFORMANCE ANALYSIS ON GPU ENVIRONMENT

In this section, we present the performance of the PIPO implementation on the GPU architecture. We measured PIPO performance on two GPU architectures (GTX 1650,



(a) GTX 1650 Block/Thread Performance Result

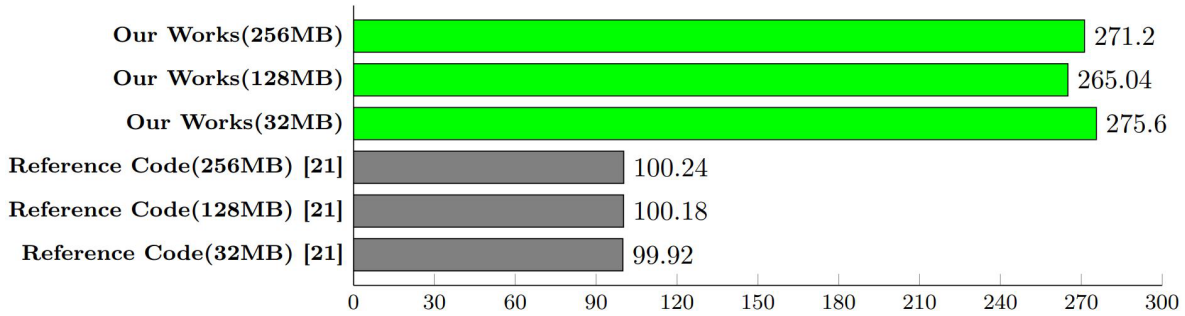


(b) RTX 2080Ti Block/Thread Performance Result

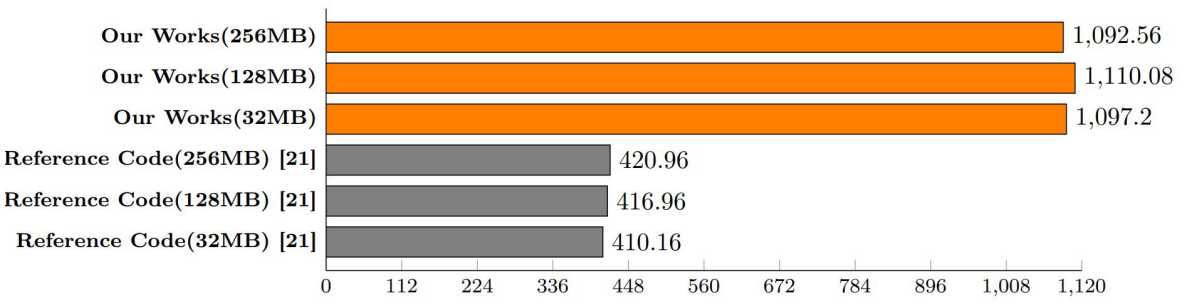
FIGURE 3. Measurement of encryption performance by block/thread configuration in GPU architecture (unit: Gbps(Gigabit per second)).

RTX 2080 Ti), using the Visual Studio CUDA Runtime version 10.2 compiler.

Algorithmic performance experiments on GPU architectures have several considerations. The first consideration is the compute share of the GPU architecture. That is, the algorithm can reach the highest performance for the performance that uses the maximum resources among the available resources of the GPU. In our PIPO experimental implementation, we set the GPU operation occupancy rate of all test cases to reach 100%, and then measured the performance. The second consideration is thread divergence. All the threads belonging to one warp do the same thing. In this case, if conditional and branch statements exist, threads can be configured in the sequential order of processing operations. Thus, if there is thread divergence, there can be a performance penalty. In this experiment, the PIPO call function is configured with Algorithm 6. Thread divergence situations have been eliminated by removing conditional and branching statements. When the GPU CUDA operation is called with Algorithm 6, all threads will proceed with PIPO encryption in parallel. The third consideration is that all threads must be guaranteed access to an area of memory. If a thread on the GPU accesses an undeclared memory area, an error occurs and the operation of the GPU ends immediately. In these cases, GPU performance is not measured correctly because the GPU does not perform any operation and terminates immediately (the GPU



(a) GTX 1650 Performance Result



(b) RTX 2080Ti Performance Result

FIGURE 4. PIPO performance results on GPU architecture(Unit: Gbps).

performance is measured faster than it actually is because the GPU is immediately shut down without performing any actual operations). In a PIPO operation, the memory area accessed by a thread is the memory for storing input messages and ciphertext. In our PIPO experiment, we set the input message memory area and ciphertext storage memory area to be (number of threads * number of blocks * 8). Additionally, the memory index has been changed to apply coalesced memory access. Our implementation of PIPO performs index control to access only the allocated memory area (see Algorithm 6).

Figure 3 shows the performance results of our PIPO implementation by block/thread on GPU architecture. We measured performance by varying the block/thread configuration of CUDA C for file size. The code used in this experiment is the proposed CUDA C PIPO implementation methods. As a result of our performance experiment measurements, our PIPO implementation shows the highest throughput at 128 threads for both the GTX 1650 and RTX 2080 Ti architectures. In addition, for all test cases regardless of file size, the case using 128 threads shows the highest throughput. Therefore, our PIPO implementation and the PIPO reference implementation were experimentally measured with a configuration using 128 threads.

Figure 4 shows our PIPO implementation performance results and PIPO reference code implementation performance results. The reference code is written in C language. Therefore, we ported the reference code to GPU CUDA C for experimental comparison. We measured PIPO performance in two architecture environments: GTX 1650 and RTX 2080 Ti. The PIPO performance implementation results measured in GTX 1650 are provided in Figure 4(a), and Figure 4(b) shows the PIPO performance results measured in the RTX 2080 Ti environment.

As a result of testing on the GTX 1650 architecture, the PIPO implementation performance of our proposed scheme is 275.6 Gbps (32 MB encryption performance), 265.04 Gbps (128 MB encryption performance), and 271.20 Gbps (256 MB encryption performance). Our proposed PIPO implementation performance provides performance improvements of 175.82%(32MB encryption performance), 164.56% (128MB encryption performance), and 170.55%(256MB encryption performance) compared with the PIPO reference code.

As a result of experiments on the RTX 2080Ti architecture, the PIPO implementation performance of our proposed scheme is 1,097.2Gbps (32MB encryption performance), 1,110.08Gbps (128MB encryption performance), and 1,092.56 Gbps (256MB encryption performance). Our proposed PIPO implementation performance provides performance gains of 167.50% (32MB encryption performance), 166.23% (128MB encryption performance), and 159.54% (256MB encryption performance) compared with the PIPO reference code.

VI. CONCLUDING REMARKS

In this paper, we proposed a method to speed up the PIPO encryption algorithm in a parallel processing architecture that can be used in a server environment. We proposed a 16-bit unit PIPO operation method considering the minimum unit of bitwise operator in AVX/PTX environment. In the CPU environment, we proposed a method to process

32/64 input messages in parallel through AVX instructions. In the GPU environment, we proposed a method for implementing PIPO operation and minimizing memory access by using PTX instructions. Our implementation of the PIPO cryptographic algorithm using AVX-2 (AVX-512) has a performance improvement of 876.72% (985.46%) compared to the reference code, and the maximum throughput of the PIPO cryptographic algorithm on GPU architecture is 1,110.08 Gbps (in RTX 2080Ti). In future works, we will to study PIPO cryptographic algorithm-based server communication environment establishment, PIPO cryptographic algorithm-based message authentication code in parallel processing environment, random number generator, etc.

REFERENCES

[1] H. Kim, Y. Jeon, G. Kim, J. Kim, B. Sim, D. Han, H. Seo, S. Kim, S. Hong, J. Sung, and D. Hong, "PIPO: A lightweight block cipher with efficient higher-order masking software implementations," in *Proc. 23rd Int. Conf. Inf. Secur. Cryptol. (ICISC)*, in Lecture Notes in Computer Science, Seoul, South Korea, vol. 12593, D. Hong, Ed. Cham, Switzerland: Springer, 2020, pp. 99–122, doi: 10.1007/978-3-030-68890-5_6.

[2] H. Kim, Y. Jeon, G. Kim, J. Kim, B. Sim, D. Han, H. Seo, S. Kim, S. Hong, J. Sung, and D. Hong, "A new method for designing lightweight S-boxes with high differential and linear branch numbers, and its application," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 1582, Nov. 2020. [Online]. Available: https://eprint.iacr.org/2020/1582

[3] J. Song, Y. Kim, and S. C. Seo, "High-speed fault attack resistant implementation of PIPO block cipher on ARM Cortex—A," *IEEE Access*, vol. 9, pp. 162893–162908, 2021.

[4] Y. Kwak, Y. Kim, and S. C. Seo, "Parallel implementation of PIPO block cipher on 32-bit RISC-V processor," in *Proc. Int. Conf. Inf. Secur. Appl.* Cham, Switzerland: Springer, 2021, pp. 183–193.

[5] S. Eum, H. Kwon, H. Kim, K. Jang, H. Kim, J. Park, G. Song, M. Sim, and H. Seo, "Optimized implementation of block cipher PIPO in parallel-way on 64-bit ARM processors," *KIPS Trans. Comput. Commun. Syst.*, vol. 10, no. 8, pp. 223–230, 2021.

[6] S. Lim, J. Han, T. Lee, and D. Han, "Differential fault attack on lightweight block cipher PIPO," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 1190, Feb. 2021. [Online]. Available: https://eprint.iacr.org/2021/1190

[7] H. Kim, M. Sim, S. Eum, K. Jang, G. Song, H. Kim, H. Kwon, W.-K. Lee, and H. Seo, "Masked implementation of pipo block cipher on 8-bit avr microcontrollers," in *Proc. Int. Conf. Inf. Secur. Appl.* Cham, Switzerland: Springer, 2021, pp. 171–182.

[8] W.-K. Lee, B.-M. Goi, R. C.-W. Phan, and G.-S. Poh, "High speed implementation of symmetric block cipher on GPU," in *Proc. Int. Symp. Intell. Signal Process. Commun. Syst. (ISPACS)*, Sarawak, Malaysia, Dec. 2014, pp. 102–107, doi: 10.1109/ISPACS.2014.7024434.

[9] S. An and S. C. Seo, "Highly efficient implementation of block ciphers on graphic processing units for massively large data," *Appl. Sci.*, vol. 10, no. 11, p. 3711, May 2020, doi: 10.3390/app10113711.

[10] S. An and S. C. Seo, "Efficient parallel implementations of LWE-based post-quantum cryptosystems on graphics processing units," *Mathematics*, vol. 8, no. 10, p. 1781, Oct. 2020, doi: 10.3390/math8101781.

[11] S. An and S. C. Seo, "Designing a new XTS-AES parallel optimization implementation technique for fast file encryption," *IEEE Access*, vol. 10, pp. 25349–25357, 2022.

[12] A. Fanfakh, H. Noura, and R. Couturier, "ORSCA-GPU: One round stream cipher algorithm for GPU implementation," *J. Supercomput.*, vol. 78, no. 9, pp. 11744–11767, 2022.

[13] G. Kim, Y. Jeon, and J. Kim, "Speeding up LAT: Generating a linear approximation table using a bitsliced implementation," *IEEE Access*, vol. 10, pp. 4919–4923, 2022.

[14] W.-K. Lee, H. J. Seo, S. C. Seo, and S. O. Hwang, "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, Exp. Briefs*, vol. 69, no. 6, pp. 2962–2966, Jun. 2022.

[15] W.-K. Lee, H. Seo, Z. Zhang, and S. O. Hwang, "TensorCrypto: High throughput acceleration of lattice-based cryptography using tensor core on GPU," *IEEE Access*, vol. 10, pp. 20616–20632, 2022.

[16] K. Han, W.-K. Lee, and S. O. Hwang, "CuGimli: Optimized implementation of the gimli authenticated encryption and hash function on GPU for IoT applications," *Cluster Comput.*, vol. 25, no. 1, pp. 433–450, Feb. 2022.

[17] M. Ceria, A. De Piccoli, M. Tiziani, and A. Visconti, "Optimizing the key-pair generation phase of McEliece cryptosystem," in *Proc. 4th Int. Conf. Wireless, Intell. Distrib. Environ. Commun.* Cham, Switzerland: Springer, 2022, pp. 111–122.

[18] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, "Efficient number theoretic transform implementation on GPU for homomorphic encryption," *J. Supercomput.*, vol. 78, no. 2, pp. 2840–2872, Feb. 2022.

[19] W. Muła, N. Kurz, and D. Lemire, "Faster population counts using AVX2 instructions," *Comput. J.*, vol. 61, no. 1, pp. 111–120, Jan. 2018, doi: 10.1093/comjnl/bxx046.

[20] Intel. (2017). *Intel Advanced Vector Extensions 512 Instructions*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html?wapkw=AVX-512

[21] (2020). *PIPO Blockcipher Github Open Source Code*. [Online]. Available: https://github.com/PIPO-Blockcipher/PIPO-Blockcipher

HOJIN CHOI (Student Member, IEEE) received the B.S. degree from the Department of Information Security, Cryptology, and Mathematics, Kookmin University. He is currently pursuing the master's degree in financial information security with Kookmin University. His research interest includes efficient implementation of cryptographic hash function in high-end-processes.

SEOG CHUNG SEO (Member, IEEE) received the B.S. degree in information and computer engineering from Ajou University, Suwon, South Korea, in 2005, and the M.S. degree in information and communications from the Gwangju Institute of Science and Technology (GIST), Gwangju, South Korea, in 2007, and the Ph.D. degree from Korea University, Seoul, South Korea, in 2011. He worked as a Research Staff Member at the Samsung Advanced Institute of Technology (SAIT) and the Samsung DMC Research and Development Center, from September 2011 to April 2014. He was a Senior Research Member at the Affiliated Institute of ETRI, South Korea, from 2014 to 2018. He is currently working as an Associate Professor with Kookmin University, South Korea. His research interests include public-key cryptography, its efficient implementations on various IT devices, cryptographic module validation program, network security, and data authentication algorithms.