

Article

Advanced Vector Extensions 512 Acceleration of LSH and LEA-GCM

Seung-Won Lee , Min-Ho Song , Ha-Gyeong Kim , Ui-Jae Kim , Si-Woo Eum  and Hwa-Jeong Seo *

Department of Information Computer Engineering, Hansung University, Seoul 02876, Republic of Korea; lsw@hansung.ac.kr (S.-W.L.); smino732@hansung.ac.kr (M.-H.S.); khg4420@hansung.ac.kr (H.-G.K.); chrisvt424@hansung.ac.kr (U.-J.K.); smile267@hansung.ac.kr (S.-W.E.)

* Correspondence: hwajeong@hansung.ac.kr

Abstract

This paper presents high-performance Advanced Vector Extensions 512 (AVX-512) implementations of two Korean standard cryptographic algorithms: the Lightweight Secure Hash (LSH) function and Lightweight Encryption Algorithm–Galois/Counter Mode (LEA-GCM) authenticated encryption. For LSH, we apply three optimization strategies: single-message processing using AVX-512 512-bit vector registers, dual-message parallel processing through register interleaving, and multi-core parallelization using a dynamic queue-based pthread thread pool. For LEA-GCM, we propose an end-to-end optimization that replaces scalar counter-mode (CTR) encryption with 16-block AVX-512 parallel processing and Streaming SIMD Extensions (SSE)-based Galois Hash (GHASH) authentication with VPCLMULQDQ-based four-block parallel processing. Performance evaluation on an Intel Core i7-1165G7 (Tiger Lake) processor shows that LSH-256 achieves an average of $1.16\times$ throughput improvement and LSH-512 achieves an average of $1.61\times$ improvement over the Korea Internet and Security Agency (KISA) AVX2 reference implementation. Dual-message interleaving achieves an average superlinear speedup of $2.28\times$, driven by instruction-level parallelism (ILP), and the thread pool delivers speedups of $3.73\times$ to $5.13\times$ across eight logical cores (four physical cores with hyperthreading). The optimized LEA-GCM implementation achieves a $3.26\times$ throughput improvement over the KISA SSE-based reference and a $12.1\times$ improvement over the pure software implementation for 4096-byte inputs, with correctness verified against KISA official test vectors.

Keywords: Advanced Vector Extensions 512; lightweight secure hash; lightweight encryption algorithm; Galois/counter mode; single instruction–multiple data; Korean cryptography; performance optimization



Academic Editor: Christos Bouras

Received: 29 May 2026

Revised: 3 July 2026

Accepted: 6 July 2026

Published: 8 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Cryptographic hash functions support the security of modern digital systems by providing verification of data integrity, digital signatures, and message authentication across network protocols, cloud services, and embedded devices [1]. As the volume of data requiring cryptographic processing grows, performance limits in hash-function implementations impose measurable costs on system throughput and latency. The Korea Internet and Security Agency (KISA) LSH algorithm was developed in 2014 and is the Korean national standard hash function. Intel’s recent processors support the Advanced Vector Extensions 512 (AVX-512) [2] instruction set, which provides 512-bit Single Instruction–Multiple Data (SIMD) registers—twice the width of the previous Advanced Vector Extensions 2 (AVX2)

standard. Prior work has demonstrated the effectiveness of AVX-512 on Korean block ciphers such as LEA, HIGHT, and CHAM [3–5].

Authenticated encryption, which simultaneously provides confidentiality and integrity, is a core component of modern secure communication protocols. LEA-GCM [6], which combines the Korean standard block-cipher LEA with Galois/Counter Mode (GCM), is an authenticated encryption scheme approved by KISA's Korea Cryptographic Module Validation Program (KCMVP) [7] and is used in domestic security products. However, the KISA LEA-GCM reference implementation [8] does not reach AVX-512-level performance. It processes counter-mode (CTR) encryption one block at a time with scalar operations, and it computes Galois Hash (GHASH) authentication with 128-bit Streaming SIMD Extensions (SSE) PCLMULQDQ instructions. The VPCLMULQDQ (512-bit carry-less multiplication) instruction provided by AVX-512 can process GCM GHASH operations in groups of four blocks, significantly improving GHASH throughput compared to existing SSE-based implementations.

Unlike AES-GCM [9], which benefits from dedicated AES-NI hardware and whose GCM optimizations therefore focus mainly on carry-less multiplication-based GHASH acceleration [10,11], LEA is an ARX cipher without a hardware instruction, so its CTR encryption must be parallelized purely in software by packing multiple blocks into SIMD registers. This block-packing strategy is not specific to LEA and applies to other S-box-free ARX ciphers such as those accelerated in prior AVX-512 and parallel work [3,12].

The goal of this study is to design, implement, and evaluate AVX-512-based optimizations for the Korean standard LSH hash function and LEA-GCM authenticated encryption. To this end, we propose three LSH implementation strategies—single-message SIMD processing, dual-message register interleaving, and dynamic thread-pool parallelization—alongside an optimized LEA-GCM pipeline combining 16-block parallel CTR mode with 4-block parallel VPCLMULQDQ GHASH. Specifically, this comprehensive evaluation and structural analysis are guided by four central research questions: First, how effectively does the AVX-512 intrinsics-based implementation improve the single-message throughput and cycles per byte (CPB) across all six LSH variants compared to the AVX2 reference? Second, to what extent does dual-message simultaneous processing using an interleaved register layout for LSH-256 exploit instruction-level parallelism (ILP) relative to the $2\times$ data-parallel bound? Third, to what extent does the dynamic queue-based thread-pool parallelization scale total throughput and multi-core efficiency across various LSH variants and input sizes? Fourth, how much does combining AVX-512 CTR 16-block parallel encryption with VPCLMULQDQ GHASH 4-block parallel processing quantitatively improve the authenticated encryption performance of LEA-GCM over the KISA references?

By systematically addressing and answering these research questions, this study delivers the following primary contributions:

1. The first systematic AVX-512 implementation of LSH covering all six hash variants, with correctness verified against the KISA reference implementation;
2. Demonstration that a dual-message parallel kernel based on 512-bit register interleaving achieves an average speedup of $2.28\times$ over the AVX2 reference (about $2\times$ over an AVX-512 single-message baseline), with performance-counter measurements showing a 32% higher instructions-per-cycle rate that confirms the instruction-level parallelism (ILP) it exploits;
3. A dynamic queue thread-pool implementation that achieves near-linear multi-core scalability for large input sizes, along with a quantitative analysis of scaling limits caused by the physical core count and memory-bandwidth contention.
4. An optimized LEA-GCM implementation combining AVX-512 CTR (16-block parallel) and VPCLMULQDQ GHASH (four-block parallel), achieving $3.26\times$ authenticated

encryption throughput improvement over the KISA SSE PCLMULQDQ reference and a $12.1\times$ improvement over the software GHASH reference for 4096-byte inputs, with correctness verified against KISA reference outputs.

The remainder of this paper is organized as follows. Section 2 provides background on the LSH algorithm, LEA block cipher, GCM authenticated encryption, AVX-512 instruction set, and related work that establishes the context for the research questions. Section 3 describes the proposed implementation methods, covering the design decisions for four strategies corresponding to single-message, dual-message, thread-pool, and LEA-GCM AVX-512 optimizations. Section 4 presents the experimental results and statistical analysis that quantitatively answer the research questions. Section 5 concludes with a summary of findings, a discussion of limitations encountered during the study, and directions for future work.

2. Background

This section provides background knowledge to help readers understand the context of this study. We first introduce the Korean standard cryptographic algorithms that are the targets of optimization: the LSH hash function, the LEA block cipher, and LEA-GCM authenticated encryption. We then summarize the key features of the AVX-512 SIMD instruction set used for optimization and conclude with a review of related prior work.

2.1. LSH Algorithm

The LSH algorithm is a hash-function family standardized by KISA in 2014 and designed for fast software implementation on general-purpose processors. The family is divided into two subfamilies [13]: LSH-256, which operates on 32-bit words and uses a 256-bit internal state, and LSH-512, which operates on 64-bit words and uses a 512-bit internal state. Output lengths of 224, 256, 384, and 512 bits are supported in each subfamily, giving a total of six variants.

The internal state of LSH is shown in Figure 1. It consists of 16 words divided into eight even-chain words (cv_e) and eight odd-chain words (cv_o). The compression function repeats two steps. In the *Mix* step, Addition–XOR–Rotation (AXR) operations are applied to corresponding pairs of words in the even and odd chains. In the *Word Permutation* step, the positions of the 16 words are rearranged according to a predefined permutation. The step constants used at each step are precomputed values specific to each variant. LSH-512 performs 28 rounds, and LSH-256 performs 26 rounds.

LSH is well suited to AVX-512 because of its internal-state size. Each chain of LSH-512 (8 words $\times$ 64 bits = 512 bits) maps exactly to one ZMM register, so the full state fits in two ZMM registers, and each Mix operation is applied to both registers at once. In contrast, the LSH-256 reference implementation is already efficiently optimized with AVX2, leaving less room for additional gains from moving to AVX-512.

2.2. LEA

LEA is a Korean national standard block cipher developed by KISA in 2013 with the goal of fast encryption in big data, cloud, and mobile environments [6]. LEA adopts an ARX structure consisting only of addition, rotation, and XOR, without an S-box, enabling efficient implementation on general-purpose processors without hardware lookup tables.

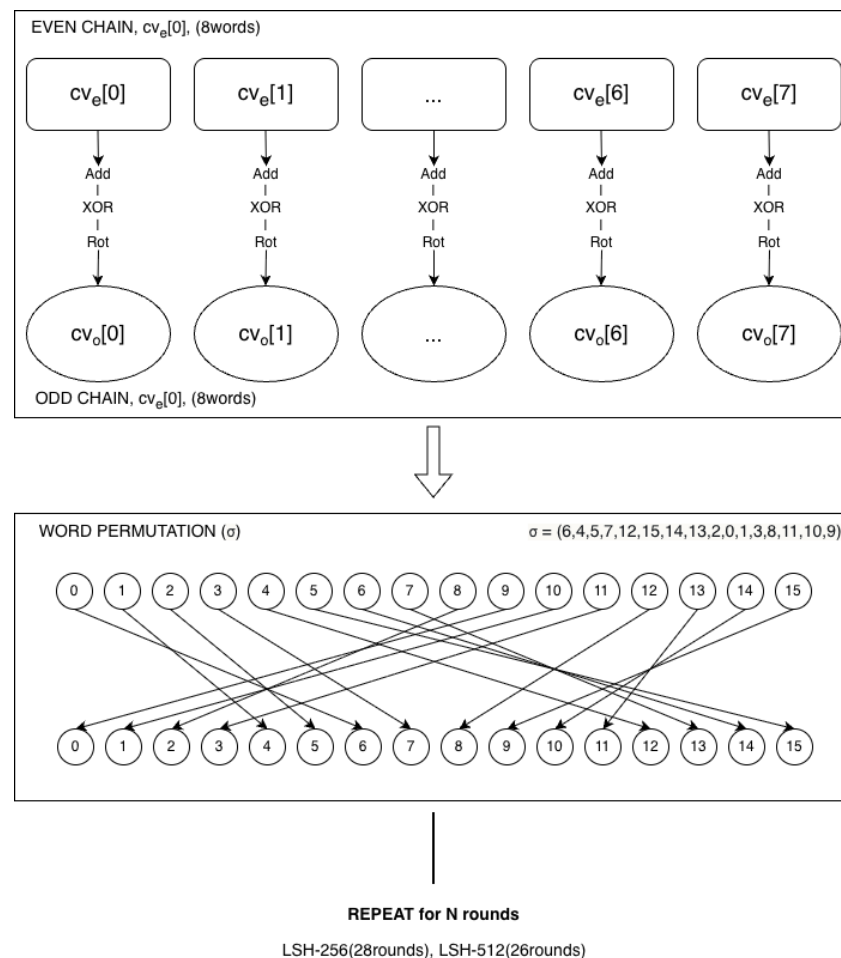


Figure 1. LSH compression-function structure.

Round Function

LEA splits a 128-bit plaintext file into four 32-bit words (X_0, X_1, X_2, X_3) and applies the following XAR (XOR+Add+Rotate) operations repeatedly over 24 rounds:

$$\begin{aligned} X'_2 &= \text{ROR}((X_2 \oplus RK_4) \boxplus (X_3 \oplus RK_5), 3) \\ X'_1 &= \text{ROR}((X_1 \oplus RK_2) \boxplus (X_2 \oplus RK_3), 5) \\ X'_0 &= \text{ROL}((X_0 \oplus RK_0) \boxplus (X_1 \oplus RK_1), 9) \\ X'_3 &= X_0 \end{aligned}$$

where $\boxplus$ denotes modular 32-bit addition, ROR/ROL denotes right/left rotation, and RK_0 – RK_5 are six round keys. Since all operations are completed at the word level, without data dependencies between words, the structure is well suited for SIMD parallelization.

The key schedule generates six 32-bit round keys per round from a 128-bit master key, using a total of $24 \times 6 = 144$ round keys. The key schedule uses predefined delta constants combined with addition and rotation operations.

For AVX-512 parallelization, since the encryption of each block is independent, data parallelism can be exploited. An AVX-512 ZMM register (512 bits) holds sixteen 32-bit elements, so packing the same-position words from 16 blocks into a single ZMM register allows a single XAR operation to process 16 blocks simultaneously [3]. In this study, this 16-block parallel implementation is used as the CTR encryption core for LEA-GCM.

2.3. GCM Authenticated Encryption

Galois/Counter Mode (GCM) is an Authenticated Encryption with Associated Data (AEAD) mode of operation specified in NIST SP 800-38D[14]. GCM provides confidentiality and integrity in a single pass and is widely used in modern security protocols such as TLS 1.3 [15], IPsec, and SSH [14]. LEA-GCM uses LEA as its block cipher and is an authenticated encryption scheme approved by KISA KCMVP [7].

GCM consists of two internal operations [16]. In CTR encryption [17], a counter block composed of a 96-bit nonce and a 32-bit counter is encrypted by LEA to produce a keystream, which is then XORed with the plaintext to generate the ciphertext. In GHASH authentication, polynomial multiplication over $GF(2^{128})$ is used to accumulate the additional authentication data (AAD), ciphertext, and length information to produce an authentication tag. Details are shown in the figure below.

The $GF(2^{128})$ multiplication in GHASH is expressed as carry-less multiplication, which can be accelerated on x86 architectures using the PCLMULQDQ instruction from SSE4.1 [10]. However, PCLMULQDQ processes only one pair of 64-bit operands in a 128-bit XMM register. The VPCLMULQDQ instruction in AVX-512 uses 512-bit ZMM registers to process four pairs of 64-bit carry-less multiplications in a single instruction, improving GHASH throughput. The KISA LEA-GCM reference implements CTR encryption one block at a time (scalar) and GHASH with PCLMULQDQ (128-bit) and does not provide AVX-512-level acceleration.

2.4. Advanced Vector Extensions 512 Instruction Set

AVX-512 extends Intel's SIMD architecture to 512-bit vector registers (ZMM0–ZMM31), providing 32 registers that are twice the width of AVX2's 256-bit YMM registers [18]. A single AVX-512 instruction can process sixteen 32-bit integers or eight 64-bit integers simultaneously. It also introduces mask registers that enable element-wise conditional operations and new gather/scatter instructions for non-contiguous memory access. AVX-512 is available on Intel Skylake-X, Ice Lake, Tiger Lake, Rocket Lake, and other microarchitectures [19]. A notable characteristic is frequency down-clocking: executing 512-bit operations can trigger an AVX-512 license transition that lowers the clock. This can offset throughput gains, especially when several cores run AVX-512 instructions at once.

2.5. Related Work

This section reviews prior work on the optimization of cryptographic algorithms using the AVX-512 instruction set, organized into three categories. We first examine research most directly related to this study on the optimization of Korean standard block ciphers, followed by work on ARX-based stream ciphers and authenticated encryption and, finally, examples of AVX-512 use in post-quantum cryptography.

2.5.1. AVX-512 Optimization of Korean Standard Block Ciphers

Seo et al. [20] proposed SIMD-based parallel implementations of LEA on ARM processors and demonstrated performance improvements on various embedded platforms. This work provides early analysis of the data parallelism in LEA. Choi et al. [3] proposed AVX-512 parallel implementations of Korean block ciphers LEA, HIGHT, and CHAM, demonstrating up to 1325.81% performance improvement over the KISA reference implementations. Their study adopted a data-parallel approach that packs same-position words from multiple blocks into AVX-512 512-bit registers, processing 16 blocks for LEA, 64 blocks for HIGHT, and 32 blocks for CHAM-64 simultaneously. However, this work was limited to pure block-cipher mode and did not address performance in authenticated encryption modes such as GCM. Our study builds on their LEA AVX-512 parallel encryption

routine as the CTR encryption core to optimize the end-to-end performance of LEA-GCM. Choi et al. [21] proposed parallel implementations of the LSH hash function on both CPUs and GPUs. This work combined thread-level parallelism on multi-core CPUs with large-scale parallel processing on GPUs to improve LSH throughput and experimentally demonstrated that LSH has a structure well suited for data parallelism.

2.5.2. AVX-512 Optimization of Stream Ciphers and Authenticated Encryption

Käsper and Schwabe [11] proposed a timing-attack-resistant AES-GCM software implementation, laying the groundwork for GHASH optimization. This work presents a method for implementing GCM without secret-dependent memory accesses and serves as a predecessor to subsequent PCLMULQDQ-based GHASH acceleration research. Wang et al. [12] implemented AVX-512 acceleration of the ChaCha20 stream cipher in a CPU–GPU hybrid computing environment, achieving a throughput of 10.05 Gb/s. ChaCha20, like LEA, uses an ARX structure consisting only of addition, XOR, and rotation, and the simultaneous generation of multiple keystream blocks using AVX-512 is the key to its performance improvement.

Bellizia et al. [22] implemented sponge-based authenticated encryption scheme Spook with AVX-512, achieving 271 cycles per 128-byte block. Like GCM, Spook is an AEAD scheme that provides both confidentiality and integrity in a single pass, and this work shows that SIMD optimization can be applied to various authenticated encryption structures.

2.5.3. AVX-512 Optimization of Post-Quantum Cryptography

Takahashi [23] proposed a parallel Number-Theoretic Transform (NTT) implementation using Intel AVX-512 instructions, accelerating a key operation in lattice-based post-quantum cryptography. This work demonstrates that AVX-512's 512-bit registers are effective for parallel processing of polynomial coefficients. Cheng et al. [24] applied AVX-512 to finite-field arithmetic operations in the NIST post-quantum cryptography candidate SIKE, achieving up to $4.61\times$ performance improvement. This work shows that AVX-512 can be broadly applied not only to symmetric-key cryptography but also to public-key operations. More broadly, the transition toward quantum-resistant cryptography is advancing on two fronts—algorithmic post-quantum schemes [25] and quantum key distribution [26]—and efficient implementations of today's standard primitives, such as the one studied here, remain complementary to this transition.

This study complements the prior work described above in two directions. First, we provide the first systematic AVX-512 implementation of the LSH function. Second, we present an end-to-end LEA-GCM optimization that combines CTR parallelization with VPCLMULQDQ-based GHASH acceleration—a combination not addressed in prior work.

3. Proposed Implementation Methods

This section describes four implementation strategies, each corresponding to one of the research questions. Sections 3.1–3.3 target the LSH hash function. Section 3.1 covers single-message throughput improvement using wide AVX-512 registers, Section 3.2 describes simultaneous processing of two messages through register interleaving, and Section 3.3 explains multi-core parallelization using a pthread thread pool. The final section (Section 3.4) targets LEA-GCM authenticated encryption and presents an end-to-end optimization that combines AVX-512 16-block parallel CTR encryption with VPCLMULQDQ 4-block parallel GHASH.

3.1. Single-Message Implementation Using Advanced Vector Extensions 512

The single-message AVX-512 implementation replaces the AVX2-based scalar and 256-bit vector operations of the KISA reference implementation with 512-bit operations, making use of the wider register file. For LSH-512, the internal state consists of 16 64-bit

words, split into an even chain and an odd chain of 8 words each. Each chain fills one 512-bit ZMM register, so the full state is held in two ZMM registers. The Mix step applies its add–rotate–XOR operations between the two registers, processing all eight word pairs at once, and the word permutation rearranges the 16 words across the two registers. For LSH-256, the two 128-bit lanes of 32-bit words are packed into a YMM register, and the AVX-512 extension provides additional instruction variants that reduce the number of operations compared to AVX2. Correctness is established by comparing the output of the proposed AVX-512 implementation against the KISA reference implementation for all six variants and multiple input lengths, including an empty message, short strings, and multi-block inputs up to 4096 bytes.

3.2. Dual-Message Parallel Implementation

The dual-message parallel implementation processes two independent messages simultaneously by interleaving their data within AVX-512 registers. Each LSH-256 message has a 256-bit even chain and a 256-bit odd chain, so the two messages are packed into two ZMM registers: one register holds the even chains of both messages (one message in each 256-bit half), and the other holds the odd chains. All arithmetic and logical operations of the LSH-256 compression function are then applied to both registers, updating the same chain of both messages with a single instruction, effectively cutting the number of instructions per message in half compared to two sequential single-message calls.

Concretely, two messages (A and B) are packed so that the low 256 bits of each register hold message A and the high 256 bits hold message B :

$$\begin{aligned} \text{ZMM}_e &= [A.cv_e[0..7] \mid B.cv_e[0..7]], \\ \text{ZMM}_o &= [A.cv_o[0..7] \mid B.cv_o[0..7]]. \end{aligned}$$

This interleaving exploits two sources of parallelism. The first is SIMD data parallelism across the two messages. The second is reduced data dependency: because the two streams are independent, the processor overlaps their instructions and gains additional ILP. The expected speedup over two sequential reference calls is $2\times$, and any observed speedup that exceeds this bound reflects additional ILP gains.

3.3. Multi-Core Parallelization Using a Thread Pool

Algorithm 1 shows the pseudocode for dynamic-queue thread-pool processing.

Algorithm 1 Dynamic-Queue Thread-Pool Processing

Require: Message set $\mathcal{M} = \{m_1, \dots, m_N\}$, batch size $B = 128$

Ensure: Hash value set $\mathcal{H} = \{h_1, \dots, h_N\}$

```

1: Divide  $\mathcal{M}$  into  $\lceil N/B \rceil$  batches of size  $B$ 
2: for each batch  $\mathcal{B}_k$  do
3:   Insert  $\mathcal{B}_k$  into the queue and signal worker threads
4: end for
   // Worker threads (running in parallel)
5: while queue is not empty do
6:   Acquire mutex, dequeue batch  $\mathcal{B}$ , release mutex
7:   for each message  $m_i$  in  $\mathcal{B}$  do
8:      $h_i \leftarrow \text{AVX512\_LSH}(m_i)$ 
9:   end for
10:  Decrement completion counter; signal completion when it reaches 0
11: end while
12: Wait for completion signal
13: return  $\mathcal{H}$ 

```

The thread-pool implementation uses the pthread library to distribute hash computations across all available logical cores. The design consists of three components:

3.3.1. Dynamic Linked-List Queue

Work items are managed as a singly linked list protected by a mutex. Worker threads enter a waiting state when there are no items to process and wake up immediately via a condition variable when new items are added. A separate completion-condition variable notifies the caller when all work items have been processed. To minimize contention, a worker holds the mutex only long enough to detach one batch from the queue and releases it before computing the hashes, so the compression itself runs entirely outside the critical section. Completion is tracked by a shared counter guarded by the same mutex: it is incremented when a batch is enqueued and decremented when a worker finishes one, and when it reaches zero, the worker signals the completion-condition variable, on the basis of which the caller blocks without busy waiting.

3.3.2. Batch Units

Instead of submitting individual messages one at a time, messages are grouped into batches of size $B = 128$ and inserted into the queue. Each batch is handled as a single work item, reducing synchronization overhead from N to $\lceil N/B \rceil$ for N message submissions. For $N = 10,000$ messages, this produces 79 work items, spreading synchronization costs by a factor of 128. The value of $B = 128$ was chosen based on experiments: the throughput difference between $B = 128$ and $B = 256$ is negligible, while smaller batch sizes cause disproportionate overhead for short messages.

3.3.3. Dynamic Load Balancing

Since worker threads independently dequeue items, faster threads automatically process more work items, eliminating the idle time that would arise from static partitioning. The number of worker threads is set automatically based on the number of logical cores in the system, which is eight on the test platform.

3.4. LEA-GCM AVX-512-Optimized Implementation

This section describes the AVX-512-optimized implementation of LEA-GCM authenticated encryption. The optimization targets the two bottlenecks of GCM: CTR encryption and GHASH authentication. The KISA LEA-GCM reference processes CTR sequentially one block at a time, handles GHASH one block at a time using SSE instructions, and therefore does not exploit AVX-512-level parallelism. The proposed implementation replaces both components with AVX-512 instructions to improve end-to-end performance.

3.4.1. CTR Encryption: 16-Block Parallelization

As described in Figure 2, GCM's CTR encryption works by encrypting sequentially incrementing counter blocks with a block cipher to produce a keystream, which is then XORed with the plaintext. A counter block is a 128-bit value composed of a 96-bit nonce and a 32-bit counter.

The KISA reference implementation generates counter blocks one at a time and encrypts them with scalar LEA, achieving approximately 8 CPB. The proposed implementation processes them in parallel through three steps.

(1) Generate 16 counter blocks sequentially: The current counter value is expanded into 16 blocks with values of $c, c + 1, \dots, c + 15$. Counter increments are performed as big-endian addition on the lower 32 bits.

(2) Encrypt 16 blocks simultaneously with AVX-512: The 16 counter blocks are processed at once using the AVX-512 LEA parallel encryption routine from Choi et al. [3].

This routine packs same-position words from each block into 512-bit registers and applies XAR operations to 16 blocks simultaneously using data parallelism.

(3) XOR keystream with plaintext: The keystream is extracted from the 16 encrypted counter blocks and XORed with the plaintext to produce the ciphertext. Remaining bytes below 256 bytes are handled by a single-block path, including a final partial block of 1–15 bytes that is encrypted with one more counter block and XORed byte-wise.

The overall procedure of the proposed AVX-512 CTR mode encryption is summarized in Algorithm 2.

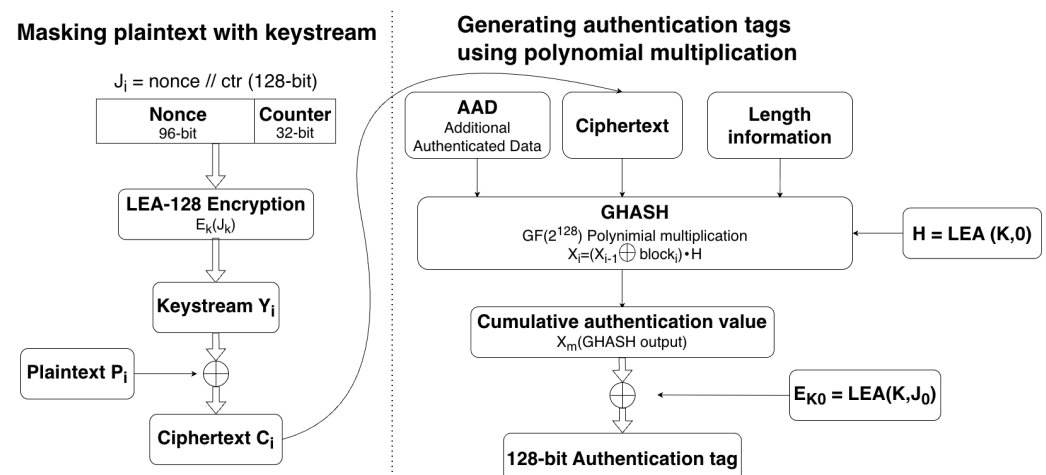


Figure 2. LEA-GCM internal operations. (Left) CTR encryption: counter-block encryption followed by plaintext XOR. (Right) GHASH authentication: accumulation of AAD and ciphertext via $GF(2^{128})$ multiplication, followed by tag generation.

Algorithm 2 AVX-512 CTR Mode Encryption

Require: Plaintext P , initial counter c , round keys RK , length ℓ

Ensure: Ciphertext C

```

1: while  $\ell \geq 256$  do                                     ▷ Process 16 blocks at a time
2:   for  $i = 0$  to 15 do
3:      $CTR[i] \leftarrow c + i$                              ▷ Big-endian increment
4:   end for
5:    $KS \leftarrow \text{AVX512\_LEA128}(CTR, RK)$              ▷ Encrypt 16 blocks simultaneously
6:   for  $i = 0$  to 15 do
7:      $C[i] \leftarrow P[i] \oplus KS[i]$ 
8:   end for
9:    $c \leftarrow c + 16$ ;  $\ell \leftarrow \ell - 256$ 
10: end while
11: while  $\ell \geq 16$  do                                     ▷ Process remaining full blocks
12:    $KS \leftarrow \text{LEA128}(c, RK)$ 
13:    $C \leftarrow P \oplus KS$ 
14:    $c \leftarrow c + 1$ ;  $\ell \leftarrow \ell - 16$ 
15: end while
16: if  $\ell > 0$  then                                         ▷ Final partial block (1–15 bytes)
17:    $KS \leftarrow \text{LEA128}(c, RK)$ 
18:   for  $i = 0$  to  $\ell - 1$  do
19:      $C[i] \leftarrow P[i] \oplus KS[i]$ 
20:   end for
21: end if
22: return  $C$ 

```

3.4.2. GHASH Authentication: VPCLMULQDQ Four-Block Parallelization

GHASH generates an authentication tag by accumulating $GF(2^{128})$ multiplications over the ciphertext blocks. Sequential GHASH over n blocks is expressed as follows:

$$S_n = H^n(S_0 \oplus b_0) \oplus H^{n-1}b_1 \oplus \dots \oplus Hb_{n-1} \quad (1)$$

$H = \text{LEA}(K, 0^{128})$ is the hash subkey. By processing four blocks as a group Equation (1), we can pack H^1, H^2, H^3 , a H^4 into a single 512-bit ZMM register, perform four $GF(2^{128})$ multiplications simultaneously with VPCLMULQDQ, and XOR the results to obtain the new state. Since Karatsuba-based 128-bit GF multiplication requires four CLMUL operations [10], processing four blocks with PCLMULQDQ (128-bit) requires 16 CLMUL instructions, whereas VPCLMULQDQ (512-bit) performs the same computation with just four instructions, as illustrated in Figure 3.

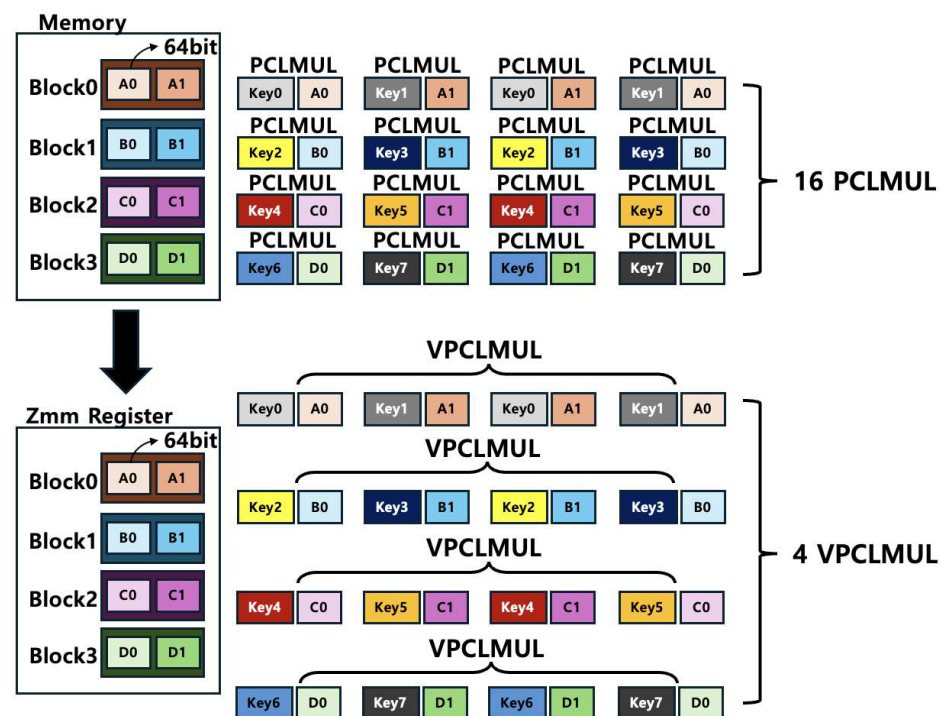


Figure 3. Comparison of instruction counts between PCLMULQDQ (128-bit) and VPCLMULQDQ (512-bit): 16 instructions reduced to 4 for 4-block GHASH.

The VPCLMULQDQ GHASH implementation consists of three steps.

(1) Precompute powers of H at initialization: At GCM context initialization, H^1, H^2, H^3 , and H^4 are computed once and stored in the GCM context. This eliminates the overhead of recomputing powers on every GHASH call and prevents performance degradation on short messages.

(2) VPCLMULQDQ four-block parallel multiplication: Four input blocks and $[H^4, H^3, H^2, H^1]$ are packed into 512-bit ZMM registers, respectively, and four VPCLMULQDQ operations are performed according to the Karatsuba decomposition:

$$\begin{aligned} \text{lo} &= \text{VPCLMULQDQ}(\mathbf{b}, \mathbf{H}, 0x00) \\ \text{hi} &= \text{VPCLMULQDQ}(\mathbf{b}, \mathbf{H}, 0x11) \\ \text{mid} &= \text{VPCLMULQDQ}(\mathbf{b}, \mathbf{H}, 0x01) \oplus \text{VPCLMULQDQ}(\mathbf{b}, \mathbf{H}, 0x10) \end{aligned}$$

$\mathbf{b}$ represents the four input blocks packed into a single 512-bit register, and $\mathbf{H} = [H^4 | H^3 | H^2 | H^1]$ is a 512-bit register holding the precomputed powers of H. lo and hi

hold the lower and upper 64-bit multiplication results of each 128-bit lane, **mid** holds the cross terms, and each contains four 128-bit partial products.

The middle terms are distributed across the upper and lower 128-bit lanes to form four 256-bit products.

(3) XOR reduction and GF(2¹²⁸) reduction: The four 256-bit products are XORed into a single 256-bit value lane by lane, viewed as high and low 128-bit halves (D_{hi}, D_{lo}). This value is the reduced modulo of the GCM polynomial ($x^{128} + x^7 + x^2 + x + 1$). Since $x^{128} \equiv x^7 + x^2 + x + 1$, the high half is folded into the low half as $D_{lo} \oplus D_{hi} \cdot (x^7 + x^2 + x + 1)$; this multiplication reduces to left shifts of D_{hi} by 1, 2, and 7 bits combined by XOR, together with a small precomputed table for the highest-order term, so the reduction requires no further carry-less multiplication. The result is the updated 128-bit accumulated state. Remaining blocks below 64 bytes are processed by the SSE PCLMULQDQ single-block path.

3.4.3. Correctness Verification

The correctness of the proposed implementation is verified in two steps. First, we confirm that both the LEA scalar implementation and the AVX-512 parallel implementation produce outputs matching the KISA official test vectors for the same key and plaintext. Second, for five plaintext lengths ranging from 16 to 4096 bytes, we verify that the ciphertext and authentication tag produced by the proposed VPCLMULQDQ implementation fully match those of the KISA SSE-based GCM reference.

4. Performance Evaluation

This section quantitatively evaluates the performance of the implementation strategies described in Section 3. Section 4.1 describes the measurement environment and methodology common to all experiments. Sections 4.2–4.4 present the single-message, dual-message, and thread-pool results for LSH, respectively. Section 4.5 compares the LEA-GCM optimization results against the KISA reference, and Section 4.6 provides an overall discussion of the observed performance-limiting factors.

4.1. Experimental Setup

Table 1 summarizes the experimental environment.

Table 1. Experimental environment.

Item	Value
Processor	Intel Core i7-1165G7 (Tiger Lake, 11th Gen)
Logical cores	8
Base frequency	2.8 GHz (Turbo Boost disabled)
Operating system	Ubuntu Linux
Compiler	GCC 13.3.0
Compiler flags	-O3 -mavx512f -mavx512vl -mavx512dq -mavx512bw
Measurement tools	clock_gettime(CLOCK_MONOTONIC), rdtsc(lfence-serialized)
Trials per configuration	30
Confidence interval	95%
CPU affinity	Pinned to core 0 (single-core benchmarks)
Frequency governor	Performance mode

Two performance metrics are reported. *Throughput* (MiB/s) is computed as $(S \times N) / (T \times 2^{20})$, where S is the input size in bytes, N is the number of repetitions, and T is the time in seconds. *Cycles per byte* (CPB) is computed as $C / (S \times N)$, where C is the cycle count. The cycle count was measured with the `rdtsc` instruction. Since modern CPUs can reorder instructions, an `lfence` was placed right before each `rdtsc` read so that

all earlier work finishes first; this keeps the cycle count accurate, even for small inputs, such as 64 bytes. The same timing method was used for all LSH and LEA-GCM benchmarks. Thirty independent trials were performed for each configuration, and the mean, standard deviation (SD), and 95% confidence interval (CI) are reported. Warm-up iterations were run before all measurements to ensure thermal and cache stability.

4.2. Single-Message Performance

Table 2 presents representative single-message results (LSH-256-256 and LSH-512-512); the complete results for all six variants are given in Appendix A (Table A1). The baseline is the KISA AVX2-based reference implementation, and the proposed implementation uses AVX-512.

Table 2. Representative single-message performance: AVX2 reference vs. AVX-512 (runs = 30, repetitions = 200,000). Values are means $\pm$ 95% CIs. (* = ours). Full results for all six variants presented are in Table A1.

Mode	Input (B)	AVX2 (Reference)		* AVX-512		Speedup
		(MiB/s)	(CPB)	(MiB/s)	(CPB)	
LSH-256-256	64	336.7 $\pm$ 0.6	7.94 $\pm$ 0.01	401.7 $\pm$ 0.2	6.66 $\pm$ 0.00	1.19 $\times$
	256	505.9 $\pm$ 0.3	5.28 $\pm$ 0.00	593.4 $\pm$ 0.5	4.51 $\pm$ 0.01	1.17 $\times$
	1024	699.2 $\pm$ 0.6	3.82 $\pm$ 0.01	798.6 $\pm$ 3.3	3.35 $\pm$ 0.04	1.14 $\times$
	4096	772.7 $\pm$ 0.8	3.46 $\pm$ 0.00	876.0 $\pm$ 1.0	3.05 $\pm$ 0.01	1.13 $\times$
LSH-512-512	64	242.4 $\pm$ 0.5	11.03 $\pm$ 0.02	389.0 $\pm$ 1.8	6.87 $\pm$ 0.09	1.60 $\times$
	256	538.0 $\pm$ 0.4	4.97 $\pm$ 0.00	899.0 $\pm$ 0.9	2.97 $\pm$ 0.01	1.67 $\times$
	1024	918.2 $\pm$ 0.5	2.91 $\pm$ 0.00	1452.0 $\pm$ 11.1	1.84 $\pm$ 0.04	1.58 $\times$
	4096	1114.5 $\pm$ 0.4	2.40 $\pm$ 0.00	1721.8 $\pm$ 4.2	1.55 $\pm$ 0.01	1.54 $\times$

The AVX-512 implementation achieves an average speedup of 1.16 $\times$ for the LSH-256 family and a higher average speedup of 1.61 $\times$ for the LSH-512 family.

The performance difference between the two subfamilies is explained by their structural differences. The AVX2 reference requires two 256-bit operations to process the same state, so AVX-512 directly provides a 2 $\times$ reduction in the number of vector instructions. In contrast, the LSH-256 reference is already efficiently vectorized with AVX2, leaving less room for AVX-512 to improve. The CPB values support this analysis: the LSH-512 CPB drops from approximately 5.1 (reference) to 3.2 (AVX-512), while the LSH-256 CPB decreases more gradually from approximately 5.0 to 4.4.

4.3. Dual-Message Performance

Table 3 presents the dual-message results for LSH-256 variants, comparing two sequential reference calls against a single AVX-512 dual-message call.

All eight configurations exceed 2.00 $\times$ relative to the two AVX2 reference calls, with speedups ranging from 2.23 $\times$ to 2.31 $\times$. A performance-counter measurement explains this result. Processing one million LSH-256 messages, two sequential single-message calls execute 16.2 G instructions at 1.30 instructions per cycle (IPC), whereas the dual-message kernel executes 10.5 G instructions at 1.72 IPC. The 32% higher IPC confirms the improved instruction-level parallelism (ILP) obtained by interleaving two independent messages; combined with the lower instruction count, it yields a 2.03 $\times$ cycle-level speedup over two AVX-512 single-message calls, i.e., essentially the 2 $\times$ data-parallel bound. The larger 2.28 $\times$ figure in Table 3 is measured against the AVX2 reference and additionally reflects AVX-512's per-message advantage over AVX2. The 1024 B and 4096 B inputs show slightly

higher speedups than the smaller 64 B inputs, consistent with the ILP benefit growing as it is spread over more computation rounds.

Table 3. Dual-message performance: $2 \times$ AVX2 reference vs. AVX-512 dual-message (runs = 30, repetitions = 200,000). Throughput reflects $2 \times$ data; values are means $\pm 95\%$ CIs (MiB/s). (* = ours).

Mode	Input (B)	$2 \times$ Reference	* AVX-512 Dual	Speedup
LSH-256-224	64	331.6 ± 0.1	737.9 ± 1.2	$2.23 \times$
	256	505.1 ± 0.2	1142.0 ± 13.1	$2.26 \times$
	1024	698.3 ± 1.5	1602.8 ± 1.0	$2.30 \times$
	4096	772.7 ± 0.5	1774.8 ± 3.9	$2.30 \times$
LSH-256-256	64	336.0 ± 0.8	760.2 ± 0.4	$2.26 \times$
	256	505.4 ± 0.2	1160.6 ± 2.2	$2.30 \times$
	1024	698.4 ± 1.3	1608.8 ± 0.8	$2.30 \times$
	4096	772.3 ± 0.8	1780.2 ± 0.4	$2.31 \times$

4.4. Thread-Pool Performance

Table 4 presents the thread-pool results for one, two, four, and eight threads, using $N_{\text{msg}} = 10,000$ messages per measurement; two representative variants are shown here, and the full six-variant results appear in Appendix A (Table A2).

Table 4. Representative thread-pool performance of AVX-512 LSH at 1, 2, 4, and 8 worker threads on the Intel i7-1165G7 (4 physical cores/8 logical threads). Runs = 30, $N_{\text{msg}} = 10,000$, batch = 128, Turbo off, and governor = performance. Values are means $\pm 95\%$ CIs (MiB/s). Speedup is 8 threads relative to 1 thread. (* = ours). Full results for all six variants are presented in Table A2.

Mode	Input (B)	1 Thread	2 Threads	4 Threads	* 8 Threads	Speedup
LSH-256-256	64	400.4 ± 0.6	631.8 ± 8.8	1281.4 ± 17.5	1494.4 ± 49.9	$3.73 \times$
	256	593.5 ± 0.3	1069.7 ± 4.1	2080.5 ± 24.2	2558.4 ± 37.2	$4.31 \times$
	1024	799.1 ± 2.0	1538.8 ± 3.3	2937.3 ± 50.5	3819.1 ± 48.4	$4.78 \times$
	4096	876.0 ± 0.6	1730.4 ± 2.4	3192.9 ± 83.8	4386.1 ± 83.8	$5.01 \times$
LSH-512-512	64	386.3 ± 3.6	639.5 ± 14.7	1122.1 ± 45.9	1478.2 ± 54.0	$3.83 \times$
	256	895.7 ± 0.9	1495.2 ± 13.8	2576.7 ± 39.1	3545.2 ± 77.2	$3.96 \times$
	1024	1450.8 ± 4.8	2670.6 ± 12.6	5223.2 ± 49.8	6776.6 ± 86.4	$4.67 \times$
	4096	1722.9 ± 0.7	3307.0 ± 69.0	5790.7 ± 133.3	8773.9 ± 129.8	$5.09 \times$

The test processor has four physical cores and eight logical threads, so the results show both the gain from real cores and the smaller gain from hyperthreading. Performance almost doubles when going from one to two threads (about $1.95 \times$) and again from two to four threads (about $1.89 \times$), since every thread runs on its own physical core; for 4096 B inputs, four threads are about $3.7 \times$ faster than one thread. From four to eight threads, however, the speedup grows by only about $1.38 \times$ because the extra four threads share the four physical cores rather than adding new ones. With eight threads, the thread pool achieves speedups of $3.73 \times$ to $5.13 \times$, with an average of $4.43 \times$. Two further trends are observed. First, the speedup grows with input size: 64 B inputs reach 3.73 – $3.98 \times$, while 4096 B inputs reach 5.01 – $5.13 \times$. Larger inputs spend more time on computation, so the fixed cost of thread synchronization matters less. Second, the thread-pool results vary more than the single-thread results because multi-core scheduling is less predictable.

4.5. LEA-GCM Performance

The performance evaluation was conducted using the 128-bit key of LEA. Table 5 compares LEA-GCM authenticated encryption performance across three implementations:

a pure software GHASH implementation (SW), an SSE PCLMULQDQ-based implementation (SSE), and the proposed implementation (AVX) combining AVX-512 CTR (16-block) and VPCLMULQDQ GHASH (4-block). The performance metric is CPB (cycles per byte) measured on plaintext bytes; the AAD (13 bytes) and authentication tag (16 bytes) are not included in the denominator. Speedup is computed as SSE CPB divided by AVX CPB.

Table 5. LEA-GCM authenticated encryption performance comparison (runs = 30, repetitions = 200,000). Values are means $\pm$ 95% CIs. (* = ours).

Input (B)	SW (CPB)	SSE (CPB)	* AVX (CPB)	Speedup
64	53.83 $\pm$ 0.26	14.28 $\pm$ 0.03	13.23 $\pm$ 0.02	1.08 $\times$
256	40.61 $\pm$ 0.02	10.78 $\pm$ 0.01	4.40 $\pm$ 0.00	2.45 $\times$
1024	37.40 $\pm$ 0.03	10.00 $\pm$ 0.02	3.30 $\pm$ 0.00	3.03 $\times$
4096	36.65 $\pm$ 0.01	9.89 $\pm$ 0.01	3.03 $\pm$ 0.00	3.26 $\times$

The proposed implementation achieves a 3.26 $\times$ improvement over the SSE-based implementation and a 12.1 $\times$ improvement over the software implementation for 4096-byte inputs. Figure 4 visually compares CPB by message size.

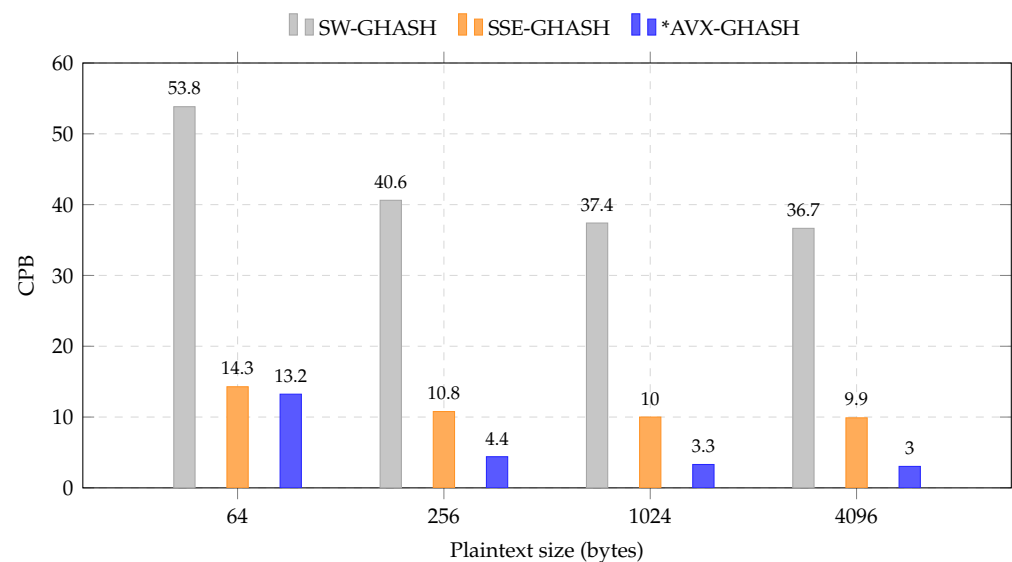


Figure 4. LEA-128-GCM authenticated encryption performance comparison (CPB; lower is better) (* = ours).

The performance improvement is explained by the complementary contributions of two optimizations.

CTR AVX-512 contribution: AVX-512 16-block parallel processing replaces scalar CTR encryption and removes the main CTR bottleneck. In the SSE baseline, CTR alone accounted for roughly 8 CPB of the total 9.89 CPB. This is the primary contributor to the performance improvement.

VPCLMULQDQ GHASH contribution: Replacing SSE-based GHASH (approximately 2 CPB) with VPCLMULQDQ four-block parallel processing further reduces the GHASH overhead. Replacing only CTR with AVX-512 yields 4.02 CPB; additionally, replacing GHASH with VPCLMULQDQ yields 3.03 CPB, giving an additional 1.33 $\times$ improvement.

Message-length dependency: The lower speedup at 64 bytes (1.08 $\times$) compared to larger inputs is due to the relatively large GCM initialization overhead for short messages. For inputs of 256 bytes and above, the four-block processing benefit of VPCLMULQDQ consistently exceeds the overhead, and the speedup shows a stable increasing trend.

4.6. Discussion

The results collectively answer the research questions and examine the factors that could limit multi-core scaling: AVX-512 frequency throttling, the number of physical cores, and memory bandwidth contention.

Intel's AVX-512 license transition can lower the processor clock when 512-bit operations are active. However, with Turbo Boost disabled, turbostat measurements showed that the busy core frequency remained essentially constant, at about 2.76 GHz, under both single-core and eight-thread execution (a change of less than 0.5%). Frequency throttling is therefore not a significant factor in our configuration; instead, the thread-pool speedup is bounded mainly by the four physical cores, with the additional gain from four to eight threads contributed by hyperthreading.

In addition, when eight cores access memory simultaneously, cache and bus bandwidth become shared resources. Hash computation is a streaming workload: each message is read once, processed, and discarded, creating continuous memory pressure. The highest speedups observed for large messages (4096 B) suggest that larger inputs more effectively spread both synchronization overhead and memory latency.

Because the proposed kernels use only ISA-level intrinsics, they also run on other AVX-512-capable processors, such as the AMD Zen 4 and Intel Sapphire Rapids, without modification. The observed speedups are expected to transfer to these platforms, although the exact figures will depend on three machine-specific factors: cache size, the width of the vector execution units, and frequency-reduction behavior. Zen 4 executes AVX-512 on double-pumped 256-bit units and applies no license-based frequency penalty, whereas Sapphire Rapids provides native 512-bit units, larger caches, and higher memory bandwidth. Therefore, both are expected to ease the limits identified above, and a full cross-platform evaluation is left to future work.

Because this work concerns cryptographic software, we summarize the side-channel properties of the optimized code. LSH and the LEA CTR core are ARX constructions (addition, rotation, and XOR) with no S-boxes or data-dependent table lookups, and the VPCLMULQDQ-based GHASH multiplication is a constant-time, carry-less multiply; therefore, these kernels contain no secret-dependent branches or memory-access patterns. The only data-dependent memory access is the small 256-byte table used for the highest-order term of the GHASH reduction; replacing it with an equivalent shift-based reduction would remove this access and yield a fully constant-time implementation, which we leave as a straightforward hardening step. For reproducibility, the source code and benchmarking scripts are publicly available at <https://github.com/Lee-Seungwon1215/LSH-LEA-on-AVX512> (accessed on 5 July 2026).

5. Conclusions

This paper proposed AVX-512-based optimized implementations of two Korean standard cryptographic algorithms—the LSH hash function and LEA-GCM authenticated encryption—and demonstrated that the proposed strategies lead to measurable performance improvements.

For LSH optimization, three strategies were applied: single-message processing using AVX-512 ZMM registers, dual-message simultaneous processing based on register interleaving, and multi-core scaling through a dynamic queue-thread pool. The single-message implementation achieved $1.16\times$ throughput improvement for LSH-256 and $1.61\times$ for LSH-512 over the KISA AVX2 reference. Dual-message interleaving achieved an average speedup of $2.28\times$ over the AVX2 reference (about $2\times$ over an AVX-512 single-message baseline); a measured 32% increase in instructions per cycle confirms the instruction-level parallelism (ILP) it exploits. The thread pool achieved scalability of $3.73\text{--}5.13\times$ on a processor with four physical cores (eight logical cores).

For LEA-GCM optimization, 16-block AVX-512 parallelization of CTR encryption was combined with VPCLMULQDQ 4-block parallelization of GHASH authentication. This strategy achieved a $3.26\times$ authenticated encryption throughput improvement over the KISA SSE-based implementation and a $12.1\times$ improvement over the pure software implementation for 4096-byte inputs. Correctness was verified by matching the official KISA test vectors.

These results provide a systematic basis for the AVX-512 optimization potential of Korean standard cryptographic algorithms. By demonstrating that performance improvements using the AVX-512 instruction set are achievable for both the LSH function and LEA-GCM authenticated encryption, this work establishes a practical performance baseline for the adoption of Korean standard cryptography.

Several directions remain for future work. First, since the experiments were conducted on a single Intel Tiger Lake platform, repeating the experiments on other AVX-512-capable platforms would establish the generalizability of the speedup results. Second, applying VPCLMULQDQ parallelization of GHASH beyond eight blocks and CTR-GHASH pipelining in LEA-GCM is expected to yield further improvements for large messages. Third, combining LSH and LEA-GCM into a single AVX-512 authenticated-hash construction could form a broader study of the Korean standard cryptographic suite.

Author Contributions: Conceptualization, S.-W.L. and H.-J.S.; methodology, S.-W.L.; software, S.-W.L.; validation, S.-W.L. and M.-H.S.; formal analysis, S.-W.L.; investigation, S.-W.L.; data curation, S.-W.L.; writing—original draft preparation, S.-W.L.; writing—review and editing, M.-H.S., H.-G.K., U.-J.K., S.-W.E. and H.-J.S.; visualization, S.-W.L.; supervision, H.-J.S.; project administration, H.-J.S.; funding acquisition, H.-J.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was financially supported by Hansung University.

Data Availability Statement: The source code is publicly available at <https://github.com/Lee-Seungwon1215/LSH-LEA-on-AVX512> (accessed on 5 July 2026).

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

Appendix A. Full Performance Results

The following tables report the complete results for all six LSH variants, which are summarized by representative variants in the main text.

Table A1. Full single-message performance for all six variants: AVX2 reference vs. AVX-512 (runs = 30, repetitions = 200,000). Values are means $\pm$ 95% CIs. (* = ours).

Mode	Input (B)	AVX2 (Reference)		* AVX-512		Speedup
		(MiB/s)	(CPB)	(MiB/s)	(CPB)	
LSH-256-224	64	331.5 $\pm$ 0.4	8.06 $\pm$ 0.01	386.9 $\pm$ 0.7	6.91 $\pm$ 0.01	1.17 $\times$
	256	503.4 $\pm$ 5.5	5.32 $\pm$ 0.07	592.1 $\pm$ 0.6	4.51 $\pm$ 0.00	1.18 $\times$
	1024	699.4 $\pm$ 0.3	3.82 $\pm$ 0.00	799.8 $\pm$ 0.6	3.34 $\pm$ 0.00	1.14 $\times$
	4096	772.0 $\pm$ 1.5	3.46 $\pm$ 0.01	876.3 $\pm$ 0.3	3.05 $\pm$ 0.00	1.14 $\times$
LSH-256-256	64	336.7 $\pm$ 0.6	7.94 $\pm$ 0.01	401.7 $\pm$ 0.2	6.66 $\pm$ 0.00	1.19 $\times$
	256	505.9 $\pm$ 0.3	5.28 $\pm$ 0.00	593.4 $\pm$ 0.5	4.51 $\pm$ 0.01	1.17 $\times$
	1024	699.2 $\pm$ 0.6	3.82 $\pm$ 0.01	798.6 $\pm$ 3.3	3.35 $\pm$ 0.04	1.14 $\times$
	4096	772.7 $\pm$ 0.8	3.46 $\pm$ 0.00	876.0 $\pm$ 1.0	3.05 $\pm$ 0.01	1.13 $\times$

Table A1. Cont.

Mode	Input (B)	AVX2 (Reference)		* AVX-512		Speedup
		(MiB/s)	(CPB)	(MiB/s)	(CPB)	
LSH-512-224	64	236.5 ± 0.3	11.30 ± 0.01	377.9 ± 0.2	7.07 ± 0.00	1.60×
	256	530.5 ± 0.5	5.04 ± 0.00	899.6 ± 0.3	2.97 ± 0.00	1.70×
	1024	910.6 ± 4.6	2.94 ± 0.02	1456.9 ± 1.1	1.84 ± 0.00	1.60×
	4096	1111.9 ± 1.7	2.40 ± 0.00	1723.4 ± 1.0	1.55 ± 0.00	1.55×
LSH-512-256	64	240.3 ± 4.5	11.16 ± 0.28	389.9 ± 0.2	6.86 ± 0.00	1.62×
	256	538.3 ± 0.4	4.97 ± 0.01	896.8 ± 1.4	2.98 ± 0.01	1.67×
	1024	917.9 ± 0.9	2.91 ± 0.00	1456.0 ± 0.8	1.84 ± 0.00	1.59×
	4096	1114.9 ± 0.3	2.40 ± 0.00	1719.2 ± 5.7	1.56 ± 0.01	1.54×
LSH-512-384	64	236.7 ± 0.1	11.29 ± 0.01	376.9 ± 0.6	7.09 ± 0.01	1.59×
	256	530.9 ± 0.3	5.04 ± 0.00	898.9 ± 1.4	2.97 ± 0.01	1.69×
	1024	912.6 ± 0.7	2.93 ± 0.01	1457.2 ± 1.0	1.83 ± 0.00	1.60×
	4096	1112.8 ± 0.3	2.40 ± 0.00	1719.4 ± 6.0	1.55 ± 0.01	1.55×
LSH-512-512	64	242.4 ± 0.5	11.03 ± 0.02	389.0 ± 1.8	6.87 ± 0.09	1.60×
	256	538.0 ± 0.4	4.97 ± 0.00	899.0 ± 0.9	2.97 ± 0.01	1.67×
	1024	918.2 ± 0.5	2.91 ± 0.00	1452.0 ± 11.1	1.84 ± 0.04	1.58×
	4096	1114.5 ± 0.4	2.40 ± 0.00	1721.8 ± 4.2	1.55 ± 0.01	1.54×

Table A2. Full thread-pool performance for all six variants of AVX-512 LSH at 1, 2, 4, and 8 worker threads on the Intel i7-1165G7 (4 physical cores/8 logical threads). Runs = 30, $N_{\text{msg}} = 10,000$, batch = 128, Turbo off, and governor = performance. Values are means ± 95% CIs (MiB/s). Speedup is 8 threads relative to 1 thread. (* = ours).

Mode	Input (B)	1 Thread	2 Threads	4 Threads	* 8 Threads	Speedup
LSH-256-224	64	385.6 ± 0.5	631.7 ± 13.1	1139.8 ± 44.4	1470.0 ± 48.9	3.81×
	256	591.0 ± 1.2	1099.4 ± 12.8	2084.9 ± 23.4	2540.8 ± 42.2	4.30×
	1024	798.4 ± 1.6	1546.9 ± 5.0	2847.2 ± 69.5	3834.1 ± 58.3	4.80×
	4096	876.2 ± 0.3	1730.5 ± 0.9	3372.1 ± 4.8	4470.1 ± 69.9	5.10×
LSH-256-256	64	400.4 ± 0.6	631.8 ± 8.8	1281.4 ± 17.5	1494.4 ± 49.9	3.73×
	256	593.5 ± 0.3	1069.7 ± 4.1	2080.5 ± 24.2	2558.4 ± 37.2	4.31×
	1024	799.1 ± 2.0	1538.8 ± 3.3	2937.3 ± 50.5	3819.1 ± 48.4	4.78×
	4096	876.0 ± 0.6	1730.4 ± 2.4	3192.9 ± 83.8	4386.1 ± 83.8	5.01×
LSH-512-224	64	374.9 ± 2.1	600.5 ± 8.6	1155.4 ± 29.3	1490.4 ± 39.4	3.98×
	256	895.6 ± 0.8	1478.3 ± 10.2	2491.8 ± 89.4	3625.5 ± 80.2	4.05×
	1024	1454.2 ± 1.4	2659.1 ± 4.7	5027.2 ± 133.1	6652.4 ± 91.4	4.57×
	4096	1722.2 ± 0.7	3346.0 ± 4.4	6496.3 ± 8.8	8737.5 ± 161.6	5.07×
LSH-512-256	64	386.4 ± 3.2	610.4 ± 8.2	1207.4 ± 33.0	1496.7 ± 58.8	3.87×
	256	895.4 ± 0.6	1484.9 ± 10.9	2797.1 ± 72.6	3560.3 ± 72.0	3.98×
	1024	1454.0 ± 1.6	2661.3 ± 4.6	5231.5 ± 20.9	6671.2 ± 77.4	4.59×
	4096	1721.4 ± 2.0	3345.7 ± 4.3	6491.6 ± 10.6	8774.8 ± 140.5	5.10×
LSH-512-384	64	375.4 ± 0.3	595.2 ± 8.2	1029.6 ± 24.1	1454.9 ± 30.9	3.88×
	256	896.3 ± 0.6	1485.9 ± 13.0	2331.3 ± 73.0	3611.6 ± 82.5	4.03×
	1024	1453.6 ± 2.4	2662.2 ± 12.3	4780.1 ± 167.9	6634.2 ± 92.9	4.56×
	4096	1722.0 ± 1.0	3368.7 ± 6.1	6521.8 ± 16.1	8841.9 ± 135.3	5.13×
LSH-512-512	64	386.3 ± 3.6	639.5 ± 14.7	1122.1 ± 45.9	1478.2 ± 54.0	3.83×
	256	895.7 ± 0.9	1495.2 ± 13.8	2576.7 ± 39.1	3545.2 ± 77.2	3.96×
	1024	1450.8 ± 4.8	2670.6 ± 12.6	5223.2 ± 49.8	6776.6 ± 86.4	4.67×
	4096	1722.9 ± 0.7	3307.0 ± 69.0	5790.7 ± 133.3	8773.9 ± 129.8	5.09×

References

- Kim, D.-C.; Hong, D.; Lee, J.-K.; Kim, W.-H.; Kwon, D. LSH: A new fast secure hash function family. In *Proceedings of the International Conference on Information Security and Cryptology*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 286–313.
- Cebrian, J.M.; Natvig, L.; Jahre, M. Scalability analysis of AVX-512 extensions: JM Cebrian et al. *J. Supercomput.* **2020**, *76*, 2082–2097.
- Choi, Y.; Choi, H.; Seo, S.C. AVX512crypto: Parallel implementations of Korean block ciphers using AVX-512. *IEEE Access* **2023**, *11*, 55094–55106. [\[CrossRef\]](#)
- Koo, B.; Roh, D.; Kim, H.; Jung, Y.; Lee, D.G.; Kwon, D. CHAM: A family of lightweight block ciphers for resource-constrained devices. In *Proceedings of the International Conference on Information Security and Cryptology*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 3–25.
- Hong, D.; Sung, J.; Hong, S.; Lim, J.; Lee, S.; Koo, B.S.; Lee, C.; Chang, D.; Lee, J.; Jeong, K.; et al. HIGHT: A new block cipher suitable for low-resource device. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 46–59.
- Hong, D.; Lee, J.-K.; Kim, D.-C.; Kwon, D.; Ryu, K.H.; Lee, D.-G. LEA: A 128-bit block cipher for fast encryption on common processors. In *Proceedings of the International Workshop on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 3–27.
- Chan-Hee, P.; Hae-Yong, K.; Jang-Hyun, J.; Jin-Jae, L.; Ho-Won, K. To Design and Work with Cryptographic Modules for KCMVP Authentication. In *Proceedings of the Korea Institute of Communication Sciences Conference*, Seoul, Korea, 19–21 June 2019.
- Korea Internet & Security Agency (KISA). Source Code and Manual for GCM/CCM Operation Modes of Domestic Block Ciphers (LEA, ARIA, SEED). 2021. Available online: <https://seed.kisa.or.kr/kisa/Board/20/detailView.do> (accessed on 11 May 2026).
- Chuang, C.C.; Su, C.K.; Huang, Y.S.; Lee, D.H.; Chen, P.Y. High-Performance AES-GCM Hardware via Circuit and Architecture Co-Design of AES and GHASH. In *Proceedings of the 2026 IEEE International Symposium on Circuits and Systems (ISCAS)*; IEEE: New York, NY, USA, 2026.
- Gueron, S.; Kounavis, M.E. Intel® Carry-Less Multiplication Instruction and Its Usage for Computing the GCM Mode. In *Intel White Paper*; Revision 2.0; Intel Corporation: Santa Clara, CA, USA, 2010.
- Käsper, E.; Schwabe, P. Faster and timing-attack resistant AES-GCM. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 1–17.
- Wang, Z.; Chen, H.; Cai, W. A hybrid CPU/GPU scheme for optimizing ChaCha20 stream cipher. In *Proceedings of the 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*; IEEE: New York, NY, USA, 2021; pp. 1171–1178.
- National Security Research Institute. *Specification of Lightweight Secure Hash Algorithm LSH*; Technical Report; National Security Research Institute (NSR): Daejeon, Republic of Korea, 2019.
- Dworkin, M.J. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2007.
- Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Internet Engineering Task Force (IETF). 2018. Available online: <https://www.rfc-editor.org/rfc/rfc8446> (accessed on 5 July 2026).
- McGrew, D.; Viega, J. The Galois/counter mode of operation (GCM). *Submiss. NIST Modes Oper. Process* **2004**, *20*, 0278–0070.
- Lipmaa, H.; Rogaway, P.; Wagner, D. CTR-mode encryption. In *Proceedings of the First NIST Workshop on Modes of Operation*, Baltimore, MD, USA, 20 October 2000; Citeseer: University Park, PA, USA, 2000; Volume 39.
- Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer’s Manuals. 2026. Available online: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html> (accessed on 11 May 2026).
- Intel Corporation. *Intel® Architecture Instruction Set Extensions and Future Features Programming Reference*; Technical Report 319433-047; Intel Corporation: Santa Clara, CA, USA, 2022.
- 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 *Proceedings of the International Workshop on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 318–330.
- Choi, H.; Choi, S.; Seo, S. Parallel implementation of lightweight secure hash algorithm on CPU and GPU environments. *Electronics* **2024**, *13*, 896. [\[CrossRef\]](#)
- Bellizia, D.; Berti, F.; Bronchain, O.; Cassiers, G.; Duval, S.; Guo, C.; Leander, G.; Leurent, G.; Levi, I.; Momin, C.; et al. Spook: Sponge-based leakage-resistant authenticated encryption with a masked tweakable block cipher. *IACR Trans. Symmetric Cryptol.* **2020**, *2020*, 295–349.
- Takahashi, D. An implementation of parallel number-theoretic transform using intel AVX-512 instructions. In *Proceedings of the International Workshop on Computer Algebra in Scientific Computing*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 318–332.
- Cheng, H.; Fotiadis, G.; Großschädl, J.; Ryan, P.Y. Highly vectorized SIKE for AVX-512. *Iacr Trans. Cryptogr. Hardw. Embed. Syst.* **2022**, *2022*, 41–68.

25. Singh, M.; Sood, S.K.; Bhatia, M. Post-quantum cryptography: A review on cryptographic solutions for the era of quantum computing. *Arch. Comput. Methods Eng.* **2026**, *33*, 3747–3788.
26. Gandelman, S.P.; Rozenman, G.G. Emulation of the Six-State Quantum Key Distribution Protocol with Pulsed Lasers. *Eur. Phys. J. Spec. Top.* **2026**, 1–9. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.