*Sistemasi: Jurnal Sistem Informasi* ISSN:2302-8149
*Volume 15, Nomor 4, 2026: 1200-1213* e-ISSN:2540-9719

# Instruction-Efficient and Parallelized AES using CUDA and PTX for Data Encryption

**[1]Raditya Hakim Daniswara, [2] Yani Parti Astuti***
[1,2]Informatics Engineering, Faculty of Computer Science, Universitas Dian Nuswantoro
[1,2]Jl. Imam Bonjol 207 Semarang Jawa Tengah Indonesia
*e-mail: radityahd39@gmail.com, yanipartiastuti@dsn.dinus.ac.id

(***received*:** 10 December 2025, ***revised*:** 20 February 2026, ***accepted*:** 31 March 2026)

## Abstract

This research presents an instruction-efficient and parallelized implementation of the AES-256 encryption algorithm using NVIDIA CUDA with inline PTX to optimize instruction usage and execution performance on GPUs. Conventional AES implementation on CUDA often suffers from redundant instructions and high computational overhead, limiting encryption throughput. To address this, three implementation variants were developed: a baseline version, a parallelized version, and an inline PTX-optimized version. The proposed PTX-enhanced AES reduces instruction redundancy through vectorized memory access, byte permutation, and logical operation consolidation using PTX instructions. Instruction analysis using NVIDIA Nsight Compute and cuobjdump revealed a 66% reduction in executed instructions compared to the baseline, primarily due to optimized AES operations. Performance evaluation of 32 MB plaintext demonstrates a 16-fold improvement in throughput, increasing from 2.73 Gbps in the baseline to 43.9 Gbps in the PTX-optimized version, with cycles per byte decreasing from 2.63 to 0.164. These results confirm that integrating fine-grained PTX control within CUDA enables substantial gains in encryption efficiency while maintaining correctness and security. The proposed approach provides a scalable foundation for high-performance cryptographic operations in GPU-based computing environments.

**Keywords:** AES, CUDA, GPU acceleration, instruction reduction, PTX

## 1 Introduction

In this digital era, data exchange is a common activity among computer users. However, because a large amount of sensitive data is transmitted and received, cryptographic algorithms are essential for data protection. Cryptography involves two fundamental processes: encryption, which transforms readable data (plaintext) into an unreadable format (ciphertext) to prevent unauthorized access; and decryption, which converts the ciphertext back into its original form for authorized users. One of the most commonly used cryptographic algorithms is Advanced Encryption Standard (AES), a symmetric block cipher algorithm that has become a universal standard across various systems. Under the influence of the U.S. government and U.S. security strategies, the AES is the common encryption mechanism opted for data security [1]. Based on the Market Growth Report discussing the Document Encryption Software Market, Advanced Encryption Standard is the most widely used type, accounting for nearly 67% of the market. AES-256, in particular, is deployed by defense agencies and health systems for top-tier security. Over 145,000 global companies integrated AES-based document encryption in 2023, primarily due to its FIPS-140 compliance and superior performance under high load. [2].

The AES comes in three key sizes: AES-128, AES-192, and AES-256, where the suffix indicates the bit length of the key. Regardless of the key size, the block size for AES remains fixed at 128 bits [3]. The AES algorithm supports five standard modes of operation: ECB, CBC, CFB, OFB, and CTR. Among these, the ECB and CTR modes are distinguished by their ability to support parallel execution during both encryption and decryption. In ECB mode, each plaintext block is encrypted separately, enabling parallel processing. Similarly, CTR mode operates by generating a sequence of independent counter values, which can be encrypted concurrently and then combined with the plaintext, allowing for efficient parallelization [4], [5].

The strength of AES is its resistance to various cryptographic attacks, such as differential and linear cryptanalysis, which were major concerns for DES. The cryptographic community has

 ISSN:2302-8149
e-ISSN:2540-9719

examined AES, but no weakness has been found yet. It is also considered computationally secure against brute-force attacks [6]. Numerous software libraries, notably OpenSSL, implement the AES algorithm. Additionally, CPUs from Intel, AMD, and ARM architectures feature instruction sets like AES-NI to enhance AES encryption in hardware [7]. However, while the CPU has AES-NI, GPUs do not have a dedicated set of instructions for AES calculation, necessitating a custom implementation [8]. The standard AES implementation requires multiple processing cycles and numerous instructions, which leads to reduced performance and increased execution time [9], [10]. In this study, AES encryption is implemented using CUDA C++ with inline PTX assembly, enabling AES operations on GPUs with reduced instruction count and execution cycles compared to standard AES. This approach ensures AES encryption can be executed concurrently with other applications without degrading their performance or monopolizing computational resources.

The GPU is optimized for highly parallel tasks, prioritizing data processing over data caching and flow control by allocating more transistors to computation. To harness NVIDIA GPUs for processing, developers utilize the CUDA API, enabling effective communication between software and the GPU. CUDA comes with a software environment that allows developers to use C++ as a high-level programming language for GPU computing [11].

About AES implementation in CUDA, AES-CTR and AES-ECB are well-suited because they inherently support parallel processing, where each block can be encrypted independently, making them ideal for GPU computation [12]. The AES implementation of CUDA C++ has a different flow from a general CPU code. While the standard C++ implementation of AES can directly access and process the data from main memory, CUDA requires a specific flow: plaintext must first be copied from host memory to GPU global memory, then processed via a kernel, and finally copied back to the host memory [13].

Although GPUs are capable of processing data in parallel, it's essential to focus on optimizing the CUDA code to ensure efficient utilization of GPU resources. The general optimization strategies include coalesced memory access, parallel granularity, and precomputed data for repeated operations[14]. For performance-critical workloads, further optimization can be achieved through device-specific intrinsics or inline PTX, which enable fine-grained control and reduction of instruction overhead [15].

This research focuses on improving the efficiency of AES-256 encryption on GPU architectures through combination of CUDA-based parallelization and instruction-level optimization using inline PTX. The study examines how reducing computational overhead and minimizing redundant operations can enhance throughput and latency. The work includes analysis of execution instructions, performance benchmarking, and validation of encryption and decryption correctness across varying data sizes.

## 2 Literature Review

Several studies have explored AES CUDA implementations and instruction reduction for AES in different contexts. An and Seo, cited in [16], implements block encryption on GPU for large data with several GPU optimization methods, especially PTX implementation. Although the PTX implementation on AES achieves 20% higher speed compared to the baseline, there is no explicit showcase of how instructions are reduced for AES. Instead, the study just focused on the throughput of AES, CHAM, and LEA encryption. Assafli et al. [17] implements the AES algorithm for the GPU using MATLAB instead of CUDA. The study identifies MixColumns and ShiftRows as the slowest AES operations, but their execution time decreases by half when implemented on a GPU. Some studies implement a low-latency AES instruction extension on RISC-V, under the name AES-RV[18]. By utilizing custom instructions and buffer optimization, AES-RV reduces the execution time of AES on RISC-V from hundreds of thousands of cycles to just a few thousand cycles. Similarly, Marshall et al.[19] surveying and developing an Instruction Set Extension (ISE) for AES on RISC-V. This study shows that the hardware-assisted T-tables version is the best option for AES on 32-bit cores, while the 64-bit data path is better suited for 64-bit cores. Lastly, Yang and Jeong [20] implements an AES using a heterogeneous CPU-GPU workflow for acceleration. Although limited to AES-128, their study demonstrates that parallel bit-slicing enhances AES performance by about 35-40%.

 ISSN:2302-8149
e-ISSN:2540-9719

Although prior studies have demonstrated significant improvements in AES performance through GPU acceleration, PTX optimization, or hardware-assisted instruction sets, notable gaps remain in explicitly reducing the instruction count within CUDA-based AES implementations. Existing works primarily focus on throughput performance, instruction reduction on other hardware or hybrid acceleration strategies, yet they often overlook the fine-grained analysis and reduction of redundant instructions that directly impact execution efficiency in CUDA AES implementations. This research seeks to fill that gap by introducing a CUDA AES implementation with inline PTX that systematically minimizes AES instructions without compromising security or parallelism.

The objective of this research is to design, implement, and evaluate an optimized AES-256 encryption on GPUs using CUDA with embedded PTX instructions. By doing so, the study aims to reduce the number of executed instructions, minimize computational overhead, and achieve higher encryption efficiency compared to standard CUDA AES implementations. Furthermore, this implementation method is expected to allow AES encryption to run more efficiently in parallel computing environments, ensuring that data security processes can operate seamlessly alongside other high-performance applications.

## 3 Research Method

The steps conducted for this research are shown in Figure 1.

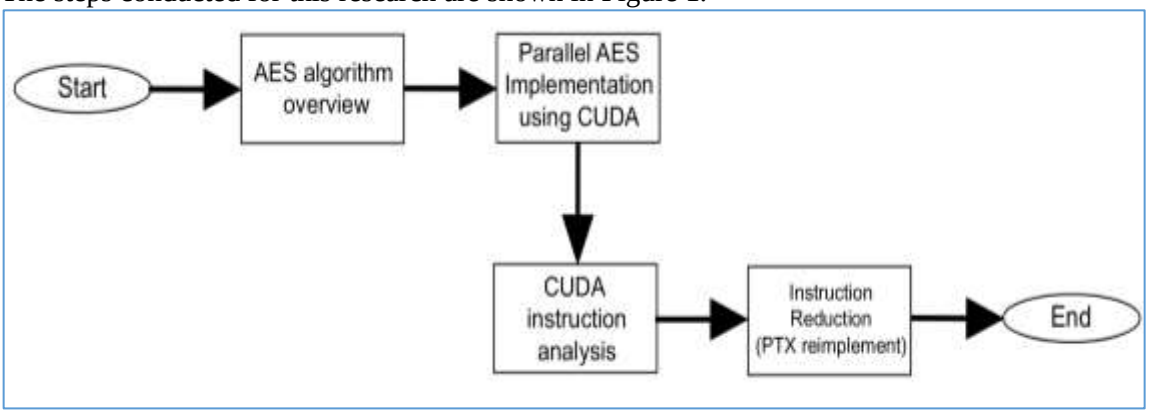


**Figure 1 Research stages**

### 3.1 AES overview

Before implementing the AES into the code, we're observing and understanding how the AES algorithm works. The operation principle depends on separating data into 128-bit blocks and encrypting each block with 128, 192, or 256-bit keys. The streaming blocks consist of 4x4 matrices, which are also called states. The core process for AES encryption involves these sub-processes:

1. AddRoundKey
   In this operation, each byte of the data is combined with a corresponding byte of the encryption key using a bitwise XOR method.
2. MixColumns
   The MixColumns operation replaces each byte with the result of mathematical field additions and multiplications of values in the bytes column.
3. ShiftRows
   A transposition step where the bytes in each row of the state are cyclically shifted by different offsets. This spreads the bytes across columns, improving diffusion
4. SubBytes
   A nonlinear substitution step where each byte in the state is replaced using a lookup table called an S-box. This introduces confusion into the cipher.

The AES encryption process begins with the AddRoundKey step. After the initial round key is added to the data block, the block undergoes a series of repeated operations, with the number of rounds depending on the length of the encryption key. Each round includes the AddRoundKey, SubBytes, ShiftRows, and MixColumns steps. Once all the rounds are completed, a final round is

 ISSN:2302-8149
e-ISSN:2540-9719

performed, which consists of SubBytes, ShiftRows, and AddRoundKey. The visual flow of AES encryption is shown in Figure 2.

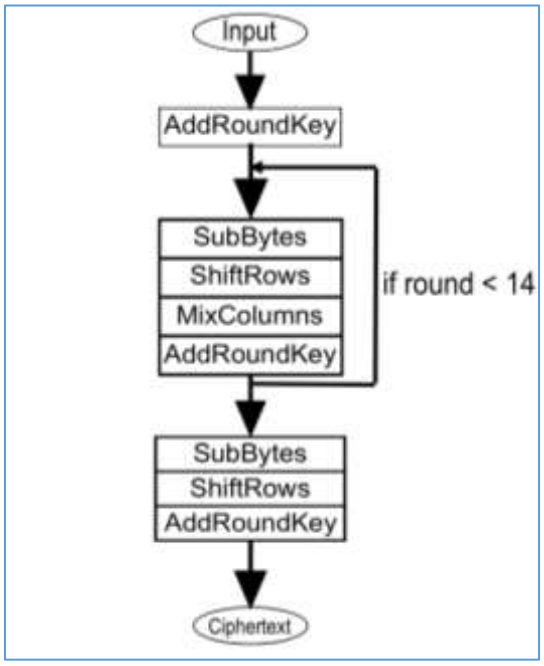


**Figure 2 AES encryption flow (AES-256)**

### 3.2 AES implementation on CUDA C++

In this study, AES-CTR is used for the implementation. The three versions of AES CUDA C++ implementations are evaluated: a baseline version, a parallel-optimized version, and a PTX-enhanced version. The baseline follows the standard AES specification, implementing each operation through iterative loops. While functionally correct, this approach introduces considerable performance overhead due to limited exploitation of GPU parallelism. The parallel-optimized version addresses this limitation by replacing loops with parallel execution strategies, thereby reducing overhead and improving utilization of GPU resources. Building upon this, the PTX-optimized version further refines performance by incorporating inline PTX instructions, which reduce computational cycles and instruction counts within the parallelized AES operations. The implementation follows the standard GPU programming model, consisting of host code (running on CPU) and device code (executed on GPU through CUDA kernels). The main steps of implementation are as follows:

1. Memory Allocation and Data Transfer
   The plaintext blocks and round keys are first prepared on the host. Using *cudaMalloc*, memory is allocated on the GPU global memory for plaintext, ciphertext, and round keys. The data is then transferred from the host to device memory using *cudaMemcpy*
2. Kernel Execution
   Each AES block (128 bits) is processed in parallel by CUDA threads. The grid and block dimensions are configured so that one thread handles one AES block. This parallelization enables concurrent encryption of multiple blocks, fully utilizing GPU resources.
3. AES Round Functions
   The AES encryption consists of multiple rounds (14 rounds for AES-256). Each round incorporates a sequence of four fundamental transformations. The SubBytes operation is executed through an S-box lookup table optimized in shared memory for fast access. The ShiftRows stage is realized by index manipulation within each processing thread. The MixColumns transformation employs finite field multiplication of *GF(2^8)* to achieve diffusion across the state matrix. Finally, the AddRoundKey step is performed using bitwise XOR operations with round keys that are preloaded into shared memory to ensure low-latency access.
4. Ciphertext Retrieval

 ISSN:2302-8149
e-ISSN:2540-9719

After kernel execution, the encrypted ciphertext blocks are copied back from device memory to host memory using *cudaMemcpy*. Finally, GPU memory cleanup is done using *cudaFree*.

### 3.3 Code Instruction Analysis

To evaluate instruction usage in each AES implementation, the analysis focused on identifying and counting the GPU operations generated during the execution of the four AES functions: AddRoundKey, SubBytes, ShiftRows, and MixColumns. The PTX/SASS instructions produced by the compiler are inspected using NVIDIA Nsight Compute and *cuobjdump*, allowing analysis of instruction types such as arithmetic operations, memory access, bitwise logic, and control flow. For the baseline version, each AES round is examined to determine the required instruction to complete the transformation steps. The parallel-optimized version is then examined to identify reductions in redundant operations achieved through thread-level parallelism and memory optimization. Finally, the PTX-enhanced implementation is evaluated by comparing inline PTX regions with the equivalent CUDA C++ code, with a focus on reductions in load/store instructions, elimination of loop overhead, and consolidation of bitwise operations. The results of this analysis provide a quantitative basis for assessing instruction efficiency and highlight which AES functions contribute the most to computational overhead before and after optimization.

### 3.4 Instruction Reduction

Instruction reduction is measured by comparing the total number of executed PTX instructions across the three AES implementation versions: baseline, parallelized, and PTX-optimized. For compiling the code, it uses CUDA version 12.6 (with *sm_86* capability) and standard compiler settings without any optimization flags. The evaluation focuses on minimizing instructions in the primary categories, namely memory operations (*ld* and *st*) and logical computations (such as *xor*, *prmt*, and *lop3*). For each AES operation (AddRoundKey, SubBytes, ShiftRows, and MixColumns), the instruction count is documented before and after optimization. In the parallelized version, reductions primarily stem from eliminating loop overhead and distributing operations across threads. The PTX-optimized implementation further decreases the instruction footprint by replacing compiler-generated sequences with hand-tuned inline PTX operations. The percentage reduction is calculated to quantify the efficiency gained in each stage.

## 4 Results and Discussion

This section of the paper reports on the experimental findings of a parallelized and instruction-efficient AES algorithm implemented using CUDA. The measurements were done on performance based on throughput, execution time, and instruction count for every AES operation to ensure the goal of this research.

### 4.1. Parallelized AES

In the parallelized implementation of AES on CUDA, each 128-bit plaintext block is assigned to an individual GPU thread, allowing concurrent processing of multiple blocks. This approach eliminates the sequential loop structure of AES operations found in the baseline version and instead distributes the workloads across threads. Two AES operations can be parallelized using threads:

1. SubBytes

   In the AES algorithm, the SubBytes operation is a non-linear byte substitution step that replaces each byte in the 4x4 state matrix with a corresponding value from a fixed substitution box (S-box). The S-box is designed to provide both non-linearity and resistance against known cryptanalytic attacks such as differential and linear cryptanalysis. In the baseline AES implementation, the SubBytes transformation substitutes each byte of the state with a corresponding value from the S-box table through iterative loops. This method, while straightforward, introduces sequential overhead since each of the 16 bytes of state must be processed using a loop and prevents the GPU from leveraging its massive thread-level parallelism. The visualization of the thread for baseline SubBytes on CUDA is shown in Figure 3.

 ISSN:2302-8149
e-ISSN:2540-9719

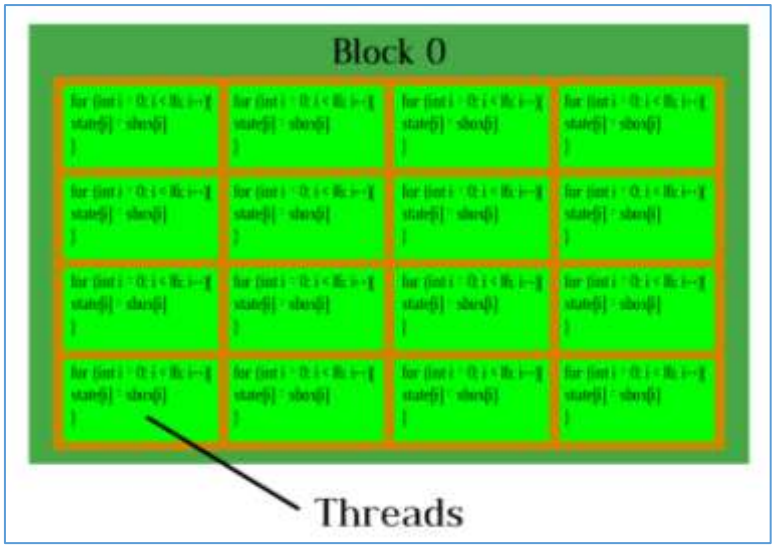


**Figure 3 Visualization of standard SubBytes on CUDA**

In the parallel implementation, each thread handles one byte of the state, so a total of 16 threads is sufficient to complete the SubBytes operations for a single AES block. This approach eliminates the need for explicit iteration as in the sequential version, since all state elements can be processed simultaneously. As shown in Figure 4, the use of thread allows direct access to the relevant byte index in the state. Thus, lookup in the S-box table can be performed efficiently by mapping byte values to replacement indexes.

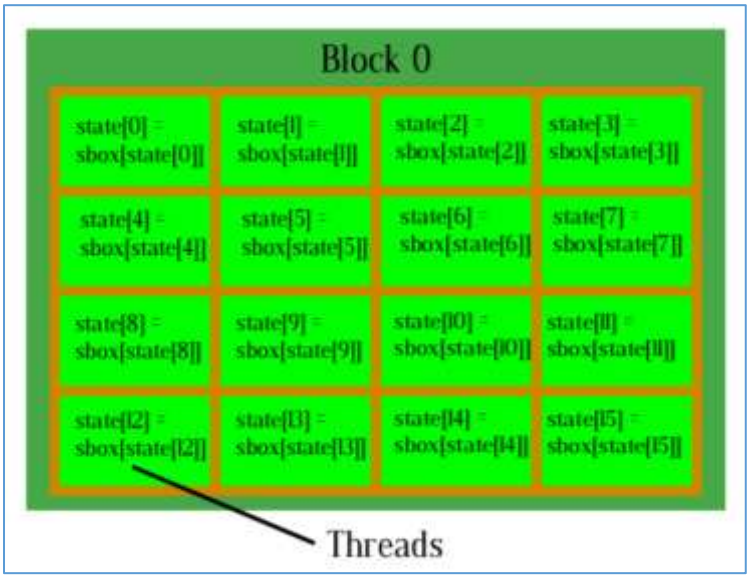


**Figure 4 Visualization of parallelized SubBytes on CUDA**

The primary advantage of this method lies in reduced latency, especially when multiple data blocks are encrypted simultaneously. Since each block can be mapped to a block thread, SubBytes operations can be executed massively on the GPU with overlap between blocks. In addition, memory access to the S-box table, which has been placed in shared memory, further speeds up the substitution process. This concept forms the foundation for the optimization of the subsequent stages in AES, as this non-linear transformation is one of the important components that determines the overall security and algorithm performance.

2. MixColumns

In the AES algorithm, the MixColumns transformation performs a linear mixing of each column in the 4x4 state matrix, ensuring diffusion across the block by combining bytes within a column using Galois Field arithmetic. In the baseline AES implementation, each column is processed iteratively

 ISSN:2302-8149
e-ISSN:2540-9719

using byte-wise XOR and multiplication. While this approach maintains correctness, it introduces multiple loop iterations and conditional operations on a thread, which are inefficient on GPUs due to serialization. As shown in Figure 5, the standard MixColumns uses a loop on each thread, which implies that a single thread processes four columns.

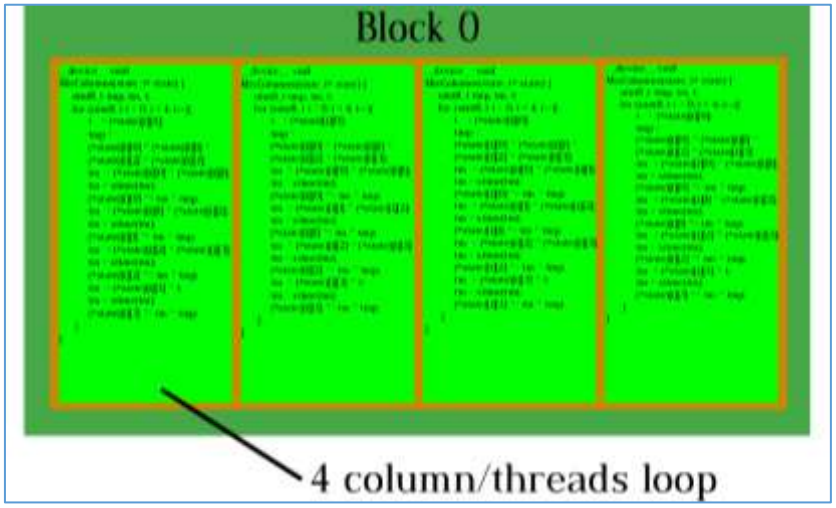


**Figure 5 Visualization of standard MixColumns on CUDA**

To improve performance, the parallelized version of MixColumns assigns one CUDA thread per column of the state matrix. As shown in Figure 6, each thread performs the entire column transformation concurrently, exploiting thread-level parallelism within a single AES block. Instead of repeated loops, all four bytes of a column are extracted and processed using 32-bit data registers, enabling fewer instructions and coalesced memory access.

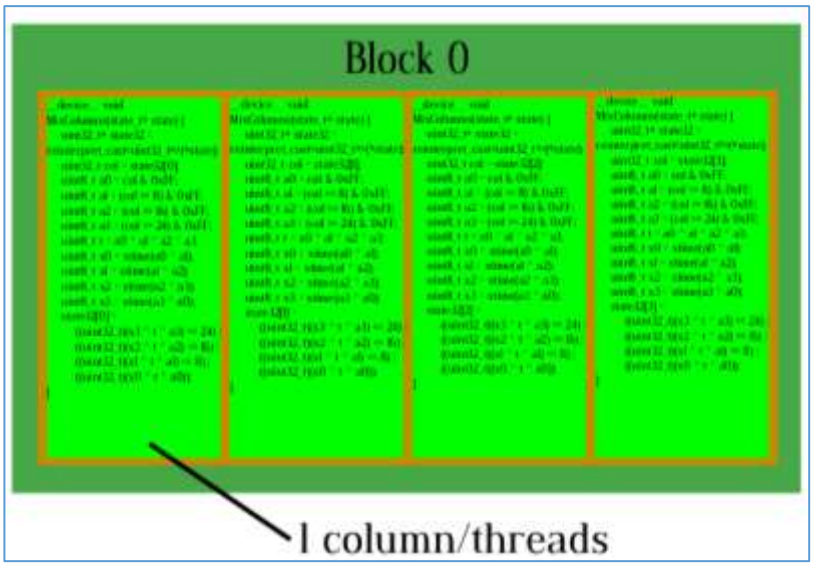


**Figure 6 Visualization of parallelized MixColumns on CUDA**

## 4.2. Instruction-efficient AES

The instruction-efficient AES implementation aims to minimize redundant GPU instructions by leveraging inline PTX (Parallel Thread Execution) to directly control arithmetic, logical, and memory operations. This approach bypasses compiler-generated instruction sequences that typically introduce inefficiencies such as unnecessary register moves, functional overheads, and conditional branching. Each AES operation is analyzed and optimized at the instruction level as follows:

1. AddRoundKey

   The AddRoundKey operation was optimized by replacing the compiler-generated XOR sequence with hand-written inline PTX to reduce the total number of executed instructions. In the AES

 ISSN:2302-8149
e-ISSN:2540-9719

specification, AddRoundKey combines the state matrix $S$ with the round key $K^{(r)}$ as shown in Equation (1).

$$S_{i,j} = S_{i,j} \oplus K_{i,j}^{(r)}, 0 \leq i,j < 4 \quad (1)$$

A baseline implementation performs 16 independent byte-wise XOR operations for each round. To minimize instruction count, the instruction-efficient version loads the 128-bit state using two 64-bit vector loads (*ld.v2.u64*) as shown in Equation (2) and (3):

$$V_0 = \left(S_{0,0}, S_{0,1}, S_{0,2}, S_{0,3}, S_{1,0}, S_{1,1}, S_{1,2}, S_{1,3}\right) \quad (2)$$

$$V_1 = \left(S_{2,0}, S_{2,1}, S_{2,2}, S_{2,3}, S_{3,0}, S_{3,1}, S_{3,2}, S_{3,3}\right) \quad (3)$$

The corresponding round keys ($K_0^{(R)}$ and $K_1^{(R}$) are packed into 64-bit vector loads as the state does. The XOR operation is then applied using only two 64-bit PTX instructions (*xor.b64*) with two packed round key vectors, effectively replacing 16 independent byte-wise XORs as shown in Equation (4).

$$V'_0 = V_0 \oplus K_0^{(r)},\ V'_1 = V_1 \oplus K_1^{(r)} \quad (4)$$

Finally, the updated state is stored back using a single 128-bit store (*st.v2.u64*). By consolidating loads, stores, and XOR logic into 64-bit PTX instructions, this version reduces the number of executed logical and memory operations. It also eliminates redundant register moves typically introduced by the CUDA compiler. As a result, AddRoundKey is executed with fewer instructions and lower latency.

2. MixColumns

In the baseline version, each thread processes one AES state (4 columns) using the conventional byte-wise MixColumns transformation. AES MixColumns is defined as the matrix multiplication over GF(2^8). Multiplication by 2 in GF(2^8) is implemented using the standard AES *"xtime"* operator and multiplication by 3. Although it has algorithmic correctness, the baseline version requires explicit byte extraction, per-byte *xtime* operations, and loop overhead. The instruction-efficient version rewrites both *xtime* and MixColumns using inline PTX to eliminate redundant logic, redundant branches, and leverage specialized PTX instructions:

a) Optimized *xtime()* using 3-input logic operation and funnel shifting

$$x' = shf.r.clamp.b32(x, x, 24); 2x = lop3.b32(x, x', 0x1B, 0xC8) \quad (5)$$

As shown in Equation (5), instead of computing *xtime*($a_i$) individually, the PTX implementation computes in a single sequence using a funnel shift (*shf)* operation to extract the MSB on each byte by shifting right to 24 bits (for $x >> 7$ calculation) and a single 3-input logic operation *(lop3)* which simultaneously performs the conditional reduction and the left shift (for $x << 1$ and 0x1B calculation). This removes all byte-wise shifting and bit-masking.

b) Column rotation using byte permutation

$$rot_n = (a_n \ll 24) + (a_{n+1} \ll 16) + \left(a_{(n+2)\%4} \ll 8\right) + a_{n-1} \quad (6)$$

The state is converted into 32-bit column vectors, portrayed as bit shifting in Equation (6). Byte permutation (*prmt.b32)* performs byte rotations of the 32-bit column in one instruction per rotation (*rot1*, *rot2*, *rot3*), which is equivalent to bit shifting.

c) XOR consolidation using *lop3.b32*

The intermediate value *t_rep* (equivalent to a0 ^ a1 ^ a2 ^ a3) is computed via a single *lop3* instead of multiple XOR chains. *lop3* allows three inputs to be combined without separate instructions. The AES *t_rep* value is reproduced across all four bytes as shown in Equation (7)

$$t_{rep} = lop3.b32(rot1, rot2, rot3, 0x96) \oplus col \quad (7)$$

d) Vectorized *xtime* and recombination

AES requires *xtime*($a_i$) and *xtime*($a_i$ ^ $a_i$+1). The PTX version avoids computing these mixed terms explicitly.

 ISSN:2302-8149
e-ISSN:2540-9719

$$out = lop3.b32(x, x_{rot1}, t_{rep}, 0x96) \oplus col \quad (8)$$

The PTX version uses the terms shown in Equation (8), which corresponds to [$2a_1$ || $2a_2$ || $2a_3$ || $2a_0$]. The combination $X \wedge X_1$ provides the same contribution as $xtime(a_i \wedge a_i+1)$ for all four bytes simultaneously using *lop3*, where it can produce the same result without computing four separate *xtime* operations.

The inline PTX version therefore reduces: loop overhead, multiple XORs, repeated 8-bit *xtime()* calculations, and shift-based data packing. By leveraging byte permute (*prmt*), logic operation (*lop3*), funnel shifting (*shf*), 32-bit state extraction, and packed XOR operations, the MixColumns step executes with significantly fewer logical and memory instructions while preserving functional correctness.

3. ShiftRows

The standard ShiftRows transformation is implemented using multiple byte-wise swaps across the 4x4 AES state matrix.

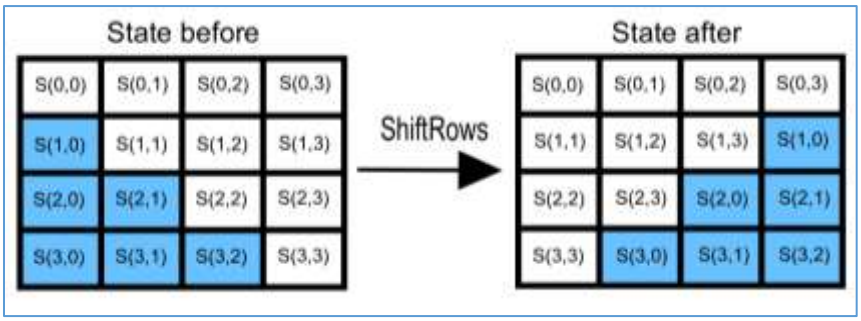


**Figure 7 Standard AES ShiftRows visualization**

As shown in Figure 7, each row is shifted to the left manually: the second row by one position, the third row by two positions, and the fourth row by three positions. While this implementation is correct, it performs numerous load, store, and temporary register operations, which increases the count and introduces unnecessary byte-level movement. The instruction-efficient version removes all per-byte assignments by treating each row as a 32-bit word and applying PTX-level permutation using the *prmt.b32* instruction. Instead of shifting individual bytes and storing intermediates, each row is loaded into a register and rotated in-place with a single instruction:

a) 32-bit row loading and storing
Each row is loaded into a 32-bit register, preserving the four bytes in their original order without unpacking. The rotated 32-bit rows are then stored back into the AES state without any additional shifts, masks, or temporary variables.

b) Byte-level rotation with byte permutation (*prmt.b32*)

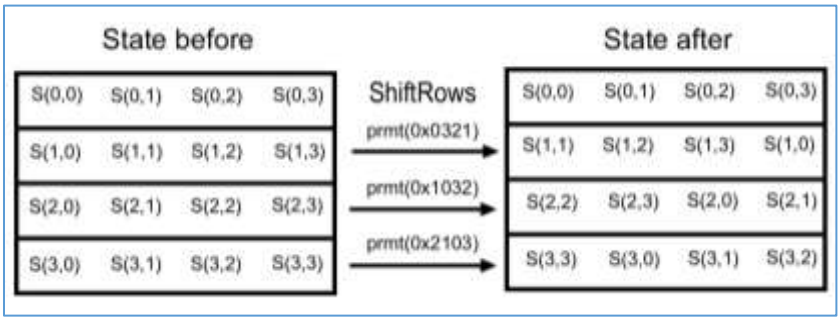


**Figure 8 Instruction-efficient AES ShiftRows visualization**

As shown in Figure 8, PTX permutation instruction (*prmt.b32*) rearranges the bytes of each 32-bit row based on an immediate control mask. This replaces a sequence of swaps and assignments with one instruction per row: the second row is rotated left by 1 byte, the third row is rotated left by 2 bytes, and the fourth row is rotated left by 3 bytes. The mask's 16 bits are

 ISSN:2302-8149
e-ISSN:2540-9719

divided into four 4-bit selectors (nibbles), from least to most significant: bits 3-0 (LSB), bits 7-4, bits 11-18, and bits 15-12 (MSB).

**Table 1 Rotation pattern of ShiftRows using CUDA byte permutation**

| Row | Rotation | Control mask | Output order | Meaning |
|---|---|---|---|---|
| Row 2 | Left 1 byte | 0x0321 | [1 2 3 0] | A1 (LSB) -> A2 -> A3 -> A0 (MSB) |
| Row 3 | Left 2 bytes | 0x1032 | [2 3 0 1] | A2 (LSB) -> A3 -> A0 -> A1 (MSB) |
| Row 4 | Left 3 bytes | 0x2103 | [3 0 1 2] | A3 (LSB) -> A0 -> A1 -> A2 (MSB) |

In Table 1, PTX mask for byte permutation (*prmt.b32*) is presented with the most significant byte first, following the standard convention that hexadecimal numbers are written with the most significant nibble on the left. Each nibble in the mask specifies one destination byte, with the leftmost nibble corresponding to the most significant byte and the rightmost nibble corresponding to the least significant byte.

4. SubBytes
   While other operations utilize PTX for instruction reduction, the SubBytes step continues to employ a parallelized implementation. However, in this case, the S-Box used in SubBytes is stored in shared memory rather than global memory. This design choice enhances access speed, as shared memory resides on the chip, providing much lower latency compared to global memory, which is located in off-chip DRAM and is therefore slower. Although shared memory offers faster access times, the capacity is smaller than that of global memory. Nevertheless, it is well-suited for storing the S-Box, which requires only 256 bytes of storage. As shown in Figure 9, shared memory is allocated per block, allowing all threads on the block to access the same shared memory.

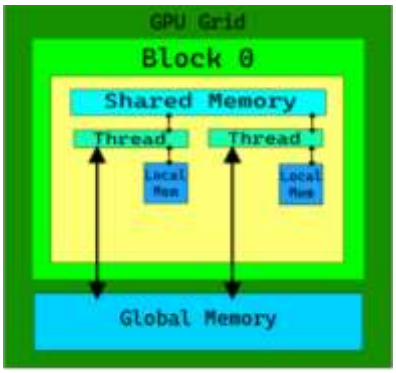


**Figure 9 Structure of GPU memory in NVIDIA CUDA architecture**

### 4.3. Instruction analysis

Instruction analysis was conducted to evaluate the efficiency of each AES implementation variant (baseline, parallelized, and PTX-optimized) based on the number of GPU instructions executed during encryption. Using NVIDIA Nsight Compute and *cuobjdump*, the PTX and SASS instructions were collected for all four AES transformation functions: AddRoundKey, SubBytes, ShiftRows, and MixColumns. The analysis focused on identifying a reduction in memory access, arithmetic, and logic operations resulting from CUDA-level and PTX optimizations. Table 2 summarizes the average instruction counts and reduction percentages across AES operations. The results indicate that inline

 ISSN:2302-8149
e-ISSN:2540-9719

PTX optimizations provide measurable efficiency gains, minimizing computational overhead while preserving encryption correctness.

**Table 2 AES CUDA instruction counts on different versions**

| AES Operation | Baseline | Parallelized | PTX-Optimized | Reduction (%) |
|---|---|---|---|---|
| AddRoundKey | 180 | 114 | 96 | 47% |
| SubBytes | 150 | 88 | 88 | 41% |
| ShiftRows | 210 | 210 | 64 | 70% |
| MixColumns | 438 | 192 | 88 | 80% |
| Total (per round) | 978 | 604 | 336 | 66% |

The reduction in instruction count directly correlates with improved execution efficiency, reduced latency per AES round, and higher throughput on GPU hardware. This analysis confirms that low-level PTX optimizations contribute substantially to instruction efficiency, making the proposed AES implementation more suitable for parallel encryption workloads.

### 4.4. Performance Analysis

Performance analysis was conducted to evaluate the execution efficiency of the three AES-256 CUDA implementations (baseline, parallelized, and PTX-optimized). The testing was performed on an Ampere architecture NVIDIA GPU (RTX 3060 with 900 MHz base clock, 1750 MHz memory clock, 6144 MB VRAM, and 3840 CUDA cores) using 32 megabytes of plaintext to measure scalability. For the kernel configuration, the AES CUDA utilizes 256 threads per block. The main performance metric evaluated is encryption throughput (Gbps) along with cycles per byte and time per AES round. Table 3 shows that the parallelized version achieves a significant speedup compared to the baseline, while the PTX-optimized version further improves throughput.

**Table 3 Performance analysis of AES-256 on CUDA**

| Implementation version | Kernel latency (ms) | AES Throughput (Gbps) | Cycles per byte (cpb) |
|---|---|---|---|
| Baseline | 98.21 | 2.73 | 2.63 |
| Parallelized | 12.51 | 21.46 | 0,335 |
| Parallelized + PTX optimized | **6.12** | **43.9** | **0.164** |

The PTX-optimized AES kernel achieves 16.04 times speedup on latency and 16.08 times on the throughput over baseline, confirming that fine-grained instruction control directly enhances GPU efficiency. The decrease in cycles per byte, from 2.63 to 0.164, indicates that most redundant operations and memory accesses were eliminated through inline PTX. This efficiency gain highlights how combining CUDA-level parallelism with instruction-level PTX tuning can deliver near-optimal utilization of GPU computational resources.

**Table 4 Performance comparison for previous works of AES on CUDA/GPU**

| Previous works | Kernel latency (ms) | AES Throughput (Gbps) |
|---|---|---|
| Inline PTX on AES (GTX 970) [16] | - | 37 |
| Hybrid CPU-GPU (Multiple GPUs) [20] | 37 | - |
| This works | 6.12 | 43.9 |

As shown in Table 4, the proposed AES implementation in this work achieves superior performance compared to previous CUDA/GPU-based studies. The inline PTX implementation on GTX 970 reported a throughput of 37 Gbps, while the hybrid CPU-GPU approach achieved a kernel latency of 37ms. In contrast, this work achieves a lower kernel latency of 6.12ms and a higher throughput of 43.9 Gbps.

 ISSN:2302-8149
e-ISSN:2540-9719

### 4.5. Encryption and Decryption test

The encryption and decryption test is conducted to ensure that the PTX version of AES still preserves the same results, where the decrypted file has the same results as the unencrypted one. The test is conducted using various file sizes to ensure compatibility. The results are shown in Table 5.

**Table 5 AES PTX encryption and decryption test result**

| File Size | Text Key | Results |
|---|---|---|
| 2.10 MB file | encryptionkey | Encryption successful. Decryption successful, no binary differences compared to the original file. |
| 128 MB file | secretdata12345 | Encryption successful. Decryption successful, no binary differences compared to the original file. |
| 1.14 GB file | thisismysecretdata67 | Encryption successful. Decryption successful, no binary differences compared to the original file. |

### 4.6. Discussion

The results demonstrate that integrating inline PTX within CUDA-based AES implementation substantially enhances instruction efficiency and overall encryption throughput. As shown in Table 2, the PTX-optimized version achieves a 66% of total instruction reduction compared to the baseline, primarily due to the elimination of redundant logic operations and loop overhead. This efficiency improvement directly correlates with the 16-time speedup in kernel latency and throughput increase to 43.9 Gbps, validating that low-level optimization of GPU instructions can significantly improve cryptographic performance.

The results also reveal that instruction-level optimization provides more substantial performance benefits than general parallelization alone. The transition from the baseline to the parallelized version already reduced the total instruction count by 38%, yet the inclusion of PTX optimization nearly doubled that gain. This outcome highlights that reducing the number of executed instructions per AES round has a direct and measurable impact on GPU efficiency, minimizing cycles per byte and overall kernel runtime.

The analysis of individual AES operations identifies MixColumns as the most instruction-intensive transformation in the standard AES implementation, but also the most improved through optimization, achieving an 80% instruction reduction. This was accomplished by applying PTX-level operations such as lookup logic operation (*lop3*), byte permutation (*prmt*), and funnel shift (*shf*), which replaced multiple shifts, masks, and XOR sequences with single combined instructions. The ShiftRows operation also benefited significantly, with a 70% reduction in instruction count due to the use of byte permutation instructions that replaced numerous swap and shift operations. Although SubBytes maintained a similar instruction count between the parallelized and PTX-enhanced versions, the use of shared memory for the S-box slightly reduced access latency and improved execution uniformity across threads.

Overall, these results confirm that fine-grained instruction control through inline PTX contributes substantially to improving AES efficiency on GPUs. The combination of PTX and reduced instruction complexity allows the encryption process to operate with minimal overhead while maintaining functional accuracy and security. This demonstrates that optimizing AES at both the parallelization and PTX levels can yield near-optimal utilization of GPU computational resources, making the implementation well-suited for high-performance encryption tasks and large-scale secure data processing.

 ISSN:2302-8149
e-ISSN:2540-9719

## 5 Conclusion

This research successfully demonstrated an instruction-efficient and parallelized implementation of the AES-256 encryption algorithm on NVIDIA GPUs using CUDA and inline PTX. By systematically reducing instruction redundancy and optimizing GPU resource utilization, the proposed approach achieved significant performance improvements over the baseline and parallelized implementations. The integration of inline PTX enabled fine-grained control over arithmetic and logical operations, reducing the total instruction count by 66% and improving encryption throughput from 2.73 Gbps to 43.9 Gbps. This 16-fold speedup in execution confirms that instruction-level optimization provides substantial benefits beyond general parallelization, allowing AES operation to execute more efficiently within GPU environments. In addition to improved performance, the PTX optimization maintained functional correctness across diverse file sizes and text keys, confirming the potential reliability for encryption tasks. Despite its strong performance, this study is limited to AES-256 encryption on NVIDIA GPUs and focuses primarily on instruction reduction and throughput performance. Future research could evaluate energy efficiency, side-channel attack test, using new architecture-specific intrinsics for further performance, and extending this approach to other cryptographic algorithms. Additionally, integrating dynamic workload balancing or hybrid CPU-GPU scheduling could further enhance performance for large-scale, real-time encryption tasks. Overall, this research provides a practical and scalable foundation for high-performance cryptographic processing in modern parallel computing systems.

## References

[1] J. Kaur, S. Lamba, and P. Saini, "*Advanced Encryption Standard: Attacks and Current Research Trends*," in *2021 International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE)*, Mar. 2021, pp. 112–116. DOI: https://doi.org/10.1109/ICACITE51222.2021.9404716.

[2] "*Document Encryption Software Market Size, Share, Growth, and Industry Analysis, by Type (IDEA Algorithm,RSA Algorithm,AES Algorithm), by Application (Confidential Document, Meeting minutes, Technical Information, Financial Statements, Others), Regional Insights and Forecast to 2033*," 106220, Aug. 2025. [Online]. Available: https://www.marketgrowthreports.com/market-reports/document-encryption-software-market-106220

[3] N. I. of S. and T. (NIST), M. J. Dworkin, M. S. Turan, and N. Mouha, "*Advanced Encryption Standard (AES)*," May 2023, DOI: https://doi.org/10.6028/NIST.FIPS.197-upd1.

[4] D. Sarangi, "*A Comparative Study of AES Encryption Modes and Hashing for Blockchain Applications*," 2024.

[5] J. Song and S. C. Seo, "*Efficient Parallel Implementation of CTR Mode of ARX-based Block Ciphers on ARMv8 Microcontrollers*," *Applied Sciences*, Vol. 11, No. 6, 2021, DOI: 10.3390/app11062548.

[6] V. V. N. Akwukwuma Chete F. O. ,. Oshioluamhe, M. N. and Okpako A. E. ,., "*Text Encryption using Advanced Encryption Standard (AES) Algorithm*," *NJSTR*, Vol. 6, No. 2, Jun. 2024, DOI: https://doi.org/10.5281/zenodo.12558923.

[7] V. Sanz, A. Pousa, M. Naiouf, and A. De Giusti, "*Performance Analysis of AES on CPU-GPU Heterogeneous Systems*," in *Cloud Computing, Big Data & Emerging Topics*, E. Rucci, M. Naiouf, F. Chichizola, L. De Giusti, and A. De Giusti, Eds., Cham: Springer International Publishing, 2022, pp. 31–42. DOI: https://doi.org/10.1007/978-3-031-14599-5_3.

[8] L. A. Pérez-Sarmiento and C. Mancillas-López, "*Parallel Implementation of OCB using VAES and GPUs*," *The Journal of Supercomputing*, Vol. 81, No. 5, p. 711, Apr. 2025, DOI: https://doi.org/10.1007/s11227-025-07179-w.

[9] G. Nişancı, P. G. Flikkema, and T. Yalçın, "*Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms*," *Cryptography*, Vol. 6, No. 3, p. 41, 2022, DOI: 10.3390/cryptography6030041.

[10] T. A. Kwame Assa-Agyei Funminiyi Olajide, "*Optimizing the Performance of the Advanced Encryption Standard Techniques for Secured Data Transmission*," *International Journal of Computer Applications*, Vol. 185, No. 21, pp. 31–36, Jul. 2023, DOI: 10.5120/ijca2023922941.

[11] D. Guide, "*Cuda C++ Programming Guide*," *NVIDIA, July*, 2025.
[12] 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 Transactions on Circuits and Systems II: Express Briefs*, Vol. 69, No. 6, pp. 2962–2966, 2022, DOI: 10.1109/TCSII.2022.3164089.
[13] 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, DOI: https://doi.org/10.1109/ACCESS.2022.3155810.
[14] P. Hijma, S. Heldens, A. Sclocco, B. van Werkhoven, and H. E. Bal, "*Optimization Techniques for GPU Programming*," *ACM Comput. Surv.*, Vol. 55, No. 11, Mar. 2023, DOI: https://doi.org/10.1145/3570638.
[15] O. Ozerk, C. Elgezen, A. C. Mert, E. Ozturk, and E. Savas, "*Efficient Number Theoretic Transform Implementation on GPU for Homomorphic Encryption*." 2021. [Online]. Available: https://eprint.iacr.org/2021/124
[16] H. Moussa and A. Fanfakh, "*Survey of Symmetric Encryption Techniques Implemented Over GPU Platforms*," *Journal of Intelligent Informatics, Networking, and Cybersecurity*, Vol. 1, No. 1, p. 2, 2025.
[17] H. T. Assafli, I. A. Hashim, and A. A. Naser, "*Advanced Encryption Standard (AES) Acceleration and Analysis using Graphical Processing Unit (GPU)*," *Applied Nanoscience*, Vol. 13, No. 2, pp. 1245–1250, Feb. 2023, DOI: https://doi.org/10.1007/s13204-021-01985-3.
[18] V. T. Nguyen, P. H. Pham, V. T. D. Le, H. L. Pham, T. H. Vu, and T. D. Tran, "*AES-RV: Hardware-Efficient RISC-V Accelerator with Low-Latency AES Instruction Extension for IoT Security*," *IEICE Electronics Express*, Vol. 22, No. 16, pp. 20250329–20250329, 2025, DOI: 10.1587/elex.22.20250329.
[19] B. Marshall, G. R. Newell, D. Page, M.-J. O. Saarinen, and C. Wolf, "*The Design of Scalar AES Instruction Set Extensions for RISC-V*," *TCHES*, Vol. 2021, No. 1, pp. 109–136, 03 2020, DOI: 10.46586/tches.v2021.i1.109-136.
[20] M. K. Yang and J.-S. Jeong, "*Optimized Hybrid Central Processing Unit–Graphics Processing Unit Workflow for Accelerating Advanced Encryption Standard Encryption: Performance Evaluation and Computational Modeling*," *Applied Sciences*, Vol. 15, No. 7, 2025, DOI: https://doi.org/10.3390/app15073863.