Efficient Implementation of AES-CTR and AES-ECB on GPUs With Applications for High-Speed FrodoKEM and Exhaustive Key Search

Wai-Kong Lee, *Member, IEEE*, Hwa Jeong Seo, *Member, IEEE*, Seog Chung Seo, *Member, IEEE*, and Seong Oun Hwang, *Senior Member, IEEE*

***Abstract*—The Advanced Encryption Standard (AES) is a standardized block cipher widely used to protect data confidentiality. Besides that, it can be used to generate pseudo-random numbers, which has many important applications. Recently, several works demonstrated the efficient implementations of AES electronics code book (ECB) and counter (CTR) mode on GPU platforms, achieving high throughput. In this brief, we set a speed record of AES implementation, which outperformed previous implementations. In particular, the proposed AES implementation achieved throughput 9% (CTR) and 7% (ECB) higher than the state-of-the-art, bit-sliced implementation. Moreover, the proposed technique does not require round keys to be embedded into the code during compilation, which is a serious limitation found in earlier work. The proposed technique also achieved up to 63% higher throughput compared to another technique presented recently. Two use cases are presented here to verify the efficiency of the proposed AES implementation. Firstly, AES is used to generate random samples in a NIST post-quantum key encapsulation mechanism (KEM), achieving 3,350, 1,503 and 7,716 key exchanges per second on V100, T4, and RTX3080 GPUs respectively. This allows the proposed FrodoKEM implementation to be 2.99× faster than the state-of-the-art performance. The proposed AES implementation was also used in an exhaustive key search application, achieving 11,428, 3,969, and 9,998 ×10^6 encryptions per second on V100, T4, and RTX3080 GPUs, respectively.**

***Index Terms*—Graphics processing unit, AES, counter mode, block cipher, key encapsulation mechanism.**

Manuscript received February 11, 2022; revised March 13, 2022; accepted March 28, 2022. Date of publication April 4, 2022; date of current version May 27, 2022. This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) under Grant 2021-0-00540, and in part by the National Research Foundation of Korea (NRF) funded by the Korea Government (MSIT) Grant 2019H1D3A1A01102607 and Grant 2020R1A2B5B01002145. This brief was recommended by Associate Editor B.-H. Gwee. *(Corresponding author: Seong Oun Hwang.)*

Wai-Kong Lee and Seong Oun Hwang are with the Department of Computer Engineering, Gachon University, Seongnam 13120, South Korea (e-mail: bardic@naver.com).

Hwa Jeong Seo is with the Department of IT Engineering, Hansung University, Seoul 04763, South Korea.

Seog Chung Seo is with the Department of Information Security, Kookmin University, Seoul 02707, South Korea.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSII.2022.3164089.

Digital Object Identifier 10.1109/TCSII.2022.3164089

I. Introduction

THE ADVANCED Encryption Standard (AES), standardized by NIST in 2001, is the most commonly used block cipher in the industry [1]. With the proliferation of emerging technologies like the Internet of Things (IoT), cloud computing, and big data, the need to attain high-speed encryption is becoming prevalent. To keep up with this emerging need, the use of encryption accelerators like the ASIC, the FPGA, and the GPU are common. Although AES and other block ciphers are traditionally used to encrypt digital content, AES can also generate pseudo-random numbers. One notable use case is the random samples generation in many lattice-based cryptography (LBC) schemes, including FrodoKEM [2], a Round 3 candidate in NIST post-quantum cryptography (PQC) standardization [3]. Besides that, password recovery applications typically generate many random keys [4] and perform an exhaustive key search to find the correct password. These two cases require a very high-throughput AES implementation to achieve a reasonable performance. A GPU-based AES implementation can provide high-throughput encryption, which has attracted a lot of research in the past decade.

Recently, Tezcan [5] improved the previous work [6] by removing some bank conflicts while reading the T-box, achieving 878.6 Gbps throughput for AES-128 on an RTX 2070 Super GPU. However, the bank conflicts still exist when writing to the T-box, which was not resolved. In another recent work, Hajihassani *et al.* [7] presented the fastest AES-128 implementation using a bit-sliced technique, achieving 1,478 Gbps on V100. However, it requires the round keys to be hard-coded during compilation, which limits its practicality.

In this brief, we propose a high-throughput counter mode (CTR) implementation of AES-128 on various advanced GPU platforms, which is faster than the state-of-the-art implementations [5], [7]. Our contributions are summarized below:

1) Performance of the AES-128-CTR implementation proposed by Tezcan [5] is seriously limited due to excessive bank conflicts when the T-box is written into the shared memory. To mitigate this issue, a warp-based

storage technique is proposed to completely remove bank conflicts in shared memory writes.

2) A pre-computation technique was proposed to speed-up the computation in Round 1. For each 2^{24} counter blocks in the AES-CTR implementation, the last four bytes in Round 1 are the same. A similar pattern is also observed on bytes 8 to 11, for each 2^{16} counter blocks. Based on these insightful observations, we proposed pre-computing the last eight bytes for Round 1, so that encryption of these eight bytes can be skipped, effectively cutting in half the computation in Round 1.
3) The bit-sliced AES-128 implementation by Hajihassani *et al.* [7] is considered the fastest to date, but their implementation requires the round key to be hard-coded during compilation. In contrast, our proposed techniques do not have this limitation. We are able to achieve 1,598 Gbps (AES-CTR) and 1,486 Gbps (AES-ECB) on V100 GPU, which is 9% faster than [7]. On the other hand, the proposed implementation is also 1.35×, 1.63×, and 1.50× faster than [5] on the V100, T4, and RTX 3080, respectively.
4) We evaluated the proposed AES implementations in two use cases to showcase its efficiency. By applying the proposed techniques, our FrodoKEM implementation is 2.99× faster than the one from Gupta *et al.* [8]. The throughput achieved by our AES exhaustive key search implementation are 11,428, 3,969, and 9,998 $\times 10^6$ enc/second on the V100, T4, and RTX3080 GPU, respectively. The source code of AES-CTR implementation is available at https://github.com/benlwk/AES-GPU.

II. Background

A. Overview of AES Implementation Techniques

AES is operating in a byte-wise manner, so it is challenging to implement AES efficiently on a processor architecture that operates on a larger bit size (e.g., 32-bit or 64-bit). A more efficient implementation can be achieved through the use of T-box, wherein all sub-bytes, shift-row, and mix-column are pre-computed into four look-up tables [1]. By using T-box, each AES round function can be computed with table lookup:

$$\begin{aligned}
g_0 &= T_0[g'_0 \gg 24] \oplus T_1[g'_1 \gg 16] \oplus T_2[g'_2 \gg 8] \oplus T_3[g'_3]\\
g_1 &= T_0[g'_1 \gg 24] \oplus T_1[g'_2 \gg 16] \oplus T_2[g'_3 \gg 8] \oplus T_3[g'_0]\\
g_2 &= T_0[g'_2 \gg 24] \oplus T_1[g'_3 \gg 16] \oplus T_2[g'_0 \gg 8] \oplus T_3[g'_1]\\
g_3 &= T_0[g'_3 \gg 24] \oplus T_1[g'_0 \gg 16] \oplus T_2[g'_1 \gg 8] \oplus T_3[g'_2]
\end{aligned} \quad (1)$$

Referring to the expressions in (1), g'_i is the input from the previous AES round and g_i is the output in each round. Assuming a 32-bit word is used, four words ($g_0, \ldots, g_3$) are required to represent AES internal states. T_0, T_1, T_2, and T_3 are the four pre-computed T-box. This technique is widely used to efficiently implement AES on a 32-bit CPU. Another popular approach to implementing AES is via bit-sliced representation [9]. An AES state is "rotated" to form a $K \times 128$-bit matrix, where K is the word size, which essentially encrypts K AES in a bit-wise manner. For a 32-bit architecture, we need

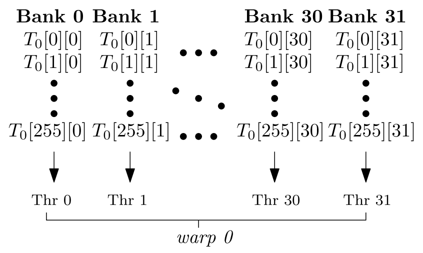

Fig. 1. T-box read without bank conflicts [5].

to form a 32×128-bit matrix; this approach can achieve high throughput, at the expense of using many registers. Hence, not all CPU architectures can adopt this technique efficiently.

B. GPU-Based AES Implementation

Hajihassani *et al.* [7] presented a very high-throughput bit-sliced AES implementation based on a new data representation scheme that exploits the parallelism in modern multi/many-core platforms. Each thread is responsible for computing 32 AES encryptions. This implementation achieved a throughput of 1,478 Gbps on the V100 GPU, which is the fastest record for a bit-sliced implementation. This approach consumed a lot of registers, resulting in insufficient registers to store the expanded round keys. To resolve this issue, they proposed to etch the expanded round keys into the code in a scripting manner using PyCUDA, before code compilation starts [7]. This implies that the high throughput achieved in this brief comes with a serious limitation—it cannot change the encryption key on the fly. This technique is useful if the encryption key used is not changed frequently. However, in many practical use cases, the encryption key may need to be updated frequently, due to the user preference or security concerns. Some examples are shown below.

1) User may want use a different key to encrypt each file.
2) A different seed (key) is required to the generated random samples for each key exchange session.
3) A new session key is used in each handshake in TLS protocol.
4) The exhaustive key search application needs to compute a different key for each key search process.

It could be impractical to recompile the code every time when the user wants to change the encryption key, as it is unlikely that the user has access to the source code. Hence, the technique proposed by [7] may have limited practical use and cannot be used in generic applications.

Another common approach to implementing AES on a GPU is through the use of pre-computed T-box. The conventional implementation on GPU [6] stores four T-boxes in shared memory to improve throughput, but it suffers from high bank conflicts. Tezcan [5] proposed using only one T-box in computing AES, because the other are just a byte-wise rotated version of the first T-box. Referring to Figure 1, they store 32 copies of the same T-box in shared memory, because a GPU groups 32 threads in an execution unit (also known as a *warp*). By doing so, each thread can access its own dedicated T-box. The shared memory read is guaranteed to be free from bank

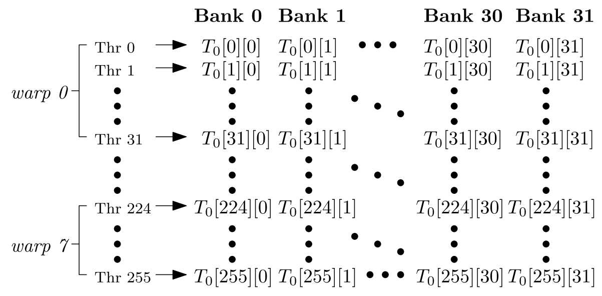
Bank 0
Bank 1
Bank 30
Bank 31
warp 0
warp 7
Thr 0
Thr 1
Thr 31
Thr 224
Thr 255
$T_0[0][0]$ $T_0[0][1]$ $T_0[0][30]$ $T_0[0][31]$
$T_0[1][0]$ $T_0[1][1]$ $T_0[1][30]$ $T_0[1][31]$
$T_0[31][0]$ $T_0[31][1]$ $T_0[31][30]$ $T_0[31][31]$
$T_0[224][0]$ $T_0[224][1]$ $T_0[224][30]$ $T_0[224][31]$
$T_0[255][0]$ $T_0[255][1]$ $T_0[255][30]$ $T_0[255][31]$

Fig. 2. T-box write with bank conflicts [5].

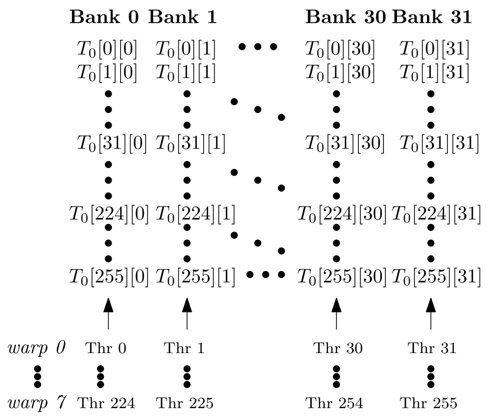
Bank 0
Bank 1
Bank 30
Bank 31
$T_0[0][0]$ $T_0[0][1]$ $T_0[0][30]$ $T_0[0][31]$
$T_0[1][0]$ $T_0[1][1]$ $T_0[1][30]$ $T_0[1][31]$
$T_0[31][0]$ $T_0[31][1]$ $T_0[31][30]$ $T_0[31][31]$
$T_0[224][0]$ $T_0[224][1]$ $T_0[224][30]$ $T_0[224][31]$
$T_0[255][0]$ $T_0[255][1]$ $T_0[255][30]$ $T_0[255][31]$
warp 0
Thr 0
Thr 1
Thr 30
Thr 31
warp 7
Thr 224
Thr 225
Thr 254
Thr 255

Fig. 3. Proposed technique to remove bank conflicts in T-box write.

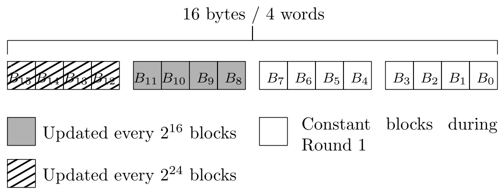
16 bytes / 4 words
B_{15} B_{14} B_{13} B_{12}
B_{11} B_{10} B_9 B_8
B_7 B_6 B_5 B_4
B_3 B_2 B_1 B_0
Updated every 2^{16} blocks
Constant blocks during Round 1
Updated every 2^{24} blocks

Fig. 4. The proposed pre-computation technique on the last eight bytes of AES state in Round 1.

conflicts. However, the shared memory write in this brief is still suffering from bank conflicts. This brief is also slower than the bit-sliced implementation [7].

C. Use Case 1: FrodoKEM

FrodoKEM [2] is a KEM used to encapsulate/decapsulate symmetric key in a secured communication. It is a Round 3 candidate in the NIST PQC standardization [3], and is considered secure even under attacks executed on a quantum computer. However, FrodoKEM suffers from slow execution speed compared to other lattice-based candidates. One of the main performance bottlenecks in FrodoKEM is the large amount of random samples generated, which is typically achieved by using a hash function (SHAKE) or a block cipher (AES). The second time-consuming part in FrodoKEM is matrix—matrix multiplication. Based on our analysis of FrodoKEM reference code, 90% of the total encapsulation time is spent on random sample generation using AES. This implies that optimizing implementation of AES can greatly improve the performance of FrodoKEM on a GPU. Gupta *et al.* [8] presented the first FrodoKEM implementation on GPU, which achieved 1,117 key exchanges per second on the V100.

D. Use Case 2: Exhaustive Key Search

Exhaustive key search is a widely used brute-force method to recover the encryption key used in block ciphers (e.g., AES). A massive amount of encryption is performed using distinct keys, and the generated ciphertexts are compared with the target ciphertext. If a generated ciphertext is found to be the same as the target, the encryption key is recovered. In password recovery applications, distinct keys are generated through password-based key derivation function (PBKDF) [4]. Note that this kind of applications can leverage the massively parallel architecture in GPU to achieve high-throughput AES encryption. Optimizing the AES implementation on a GPU can help to improve the key search rate in such applications.

III. Proposed AES Implementation on GPU

A. Removing Bank Conflicts in Shared Memory Write

Tezcan [5] open-sourced an implementation that requires storing one T-box (256 pre-computed values) 32 times, the process for which is illustrated in Figure 2. It can be seen that each thread within a warp writes to the same bank, causing many bank conflicts. Consider the case in warp 0, thread 0 writes to $T_0[0][1]$, and thread 1 writes to $T_0[1][1]$, and so on. Shared memory organizes a 2D array into 32 banks in a column-major manner. This implies that the row-major access pattern in Figure 2 is causing a 32-way bank conflicts. This process is repeated 32 times until the T-box is completely loaded into shared memory, which is a serious bottleneck. To mitigate this issue, a warp-based technique is proposed to write T-box into shared memory without introducing bank conflicts. In Figure 3, each warp is writing a T-box value into a different bank of shared memory. We use 32 warps (i.e., 8×32 threads) to load the four-T-box implementation (each of the four with 256 values) into shared memory, and we repeat this process eight times to completely load the T-box.

B. Pre-computing the Last Eight Bytes in Round 1

The AES-CTR implementation that uses the same encryption key for multiple counter blocks exhibits a special property. For each adjacent block, results after Round 1 encryption show only a very small difference. This is because each counter value passed into the AES round function differs by only one. This interesting property leads to the observation that for 2^{24} blocks, the last four bytes (byte 12 to 15) after Round 1 encryption are actually the same. Similarly, we observed that the results in byte 8 to 11 only update after every 2^{16} blocks. These observations are illustrated in Figure 4 and imply that the last eight bytes in Round 1 can be pre-computed, and therefore encryption of these eight bytes can be skipped. Based on this observation, we propose a pre-computation technique to further speed up AES-128-CTR implementation on a GPU.

In our implementation, we pre-compute Round 0 and Round 1 in AES-128-CTR on a CPU, and store the last eight bytes on the shared memory of a GPU, which are used for

TABLE I
EXPERIMENTAL PLATFORMS

Plat.	CPU	Clock Speed (GHz)	RAM (GB)	GPU	Clock Speed (GHz)	Mem. BW (GB/s)
A	i9-10900K	3.7	16	RTX3080	1.44	760.3
B	Xeon Gold 5120	2.2	16	V100	1.25	897.0
				T4	0.585	320.0

Round 1 encryption. Since the last eight bytes in Round 1 for all counter blocks are the same, all parallel threads can use the pre-computed values and skip the encryption of these eight bytes in Round 1. Note that the overhead for pre-computation is small, because we are only interested in the last four bytes for every 2^{24} counter blocks, and in bytes 8 to 11 for every 2^{16} counter blocks. Other values (bytes 0 to 7) are not calculated. Considering that a total of $64 \times 1024 \times 1024$ counter blocks are to be encrypted on the GPU, we only need to pre-compute four Round 1 results for the last four bytes, and 1,024 results for bytes 8 to 11. In total, only 1,028 32-bit words are computed and stored. Encrypting all these counter blocks on a V100 GPU takes 5ms, but the pre-computation only takes 59ns on a i9-10900K CPU, which is negligible.

C. Coalesced Ciphertext Storage

After the encryption, ciphertexts are stored in global memory before copying to the CPU's DRAM. To optimize the performance, the write processes need to be in a coalesced manner (burst mode). In AES implementation, each ciphertext size is four 32-bit words, causing a stride of four (25% efficiency) when it is written to global memory in uncoalesed manner. We store the ciphertext in a coalesced manner, where each thread writes to contiguous location in global memory, achieving a 100% efficiency.

IV. EXPERIMENTAL RESULTS AND DISCUSSION

The proposed algorithms are implemented in C under the CUDA 11.0 SDK. Experiments were carried out on two different platforms as described in Table I. Platform A is a workstation equipped with an Intel Core i9-10900K CPU and a RTX 3080 GPU. Platform B has four CPU cores (Xeon Gold), 16-GB RAM, and a GPU (V100 or T4). The three selected GPUs (V100, T4, and RTX3080) represent the state-of-the-art NVIDIA architectures (Volta, Turing, and Ampere).

A. High-Throughput AES Encryption on GPU

The experiment was carried out by encrypting 1GB of data with parallel blocks, each consisting of 1,024 threads. Within each thread, AES encryption was performed M times, where $M = 4, 16, 32, 64, 128$. A total of $\frac{262,144}{M}$ blocks were launched in the experiment. The same encryption key is expanded and stored in shared memory; it is used throughout the CTR and ECB encryption. Each thread encrypts a different counter in CTR mode. For ECB mode, each thread encrypts the plaintext copied from the CPU and stored in the global memory. Referring to Table II, the throughput achieved by the proposed implementation peaked at $M = 32$ for all GPU platforms. We have evaluated the work by Tezcan [5] on the same GPU platforms, where our implementation was 1.35×, 1.63×, and 1.50× faster on the V100, T4, and RTX 3080, respectively. Results proved that by solving the bank conflicts problem in shared memory writes, the performance from AES encryption is improved.

Our CTR mode result is also 9% higher than the throughput achieved by Hajihassani *et al.* [7] on the V100 GPU. Unlike their work, our implementation allows the use of different encryption keys without recompiling the code. This offers great flexibility that is not found in [7]. This high-throughput AES implementation can be used to efficiently encrypt a large file (several hundred megabytes to several gigabytes). For the ECB mode, our implementation is 7% faster than [7].

TABLE II
PERFORMANCE FROM AES-128 CTR AND ECB ENCRYPTION ON GPUs

M	Throughput (Gbps)					
	CTR			ECB		
	V100	T4	RTX 3080	V100	T4	RTX 3080
4	1,522	466	1,332	1,401	424	1,225
16	1,602	492	1,401	1,489	457	1,309
32	1,613	500	1,423	1,516	471	1,322
64	**1,598**	**490**	**1,489**	**1,486**	**453**	**1,370**
128	1,489	482	1,396	1,369	447	1,305
[5]	1,181	301	990	–	–	
[7]	1,478	–	–	1,385	–	–

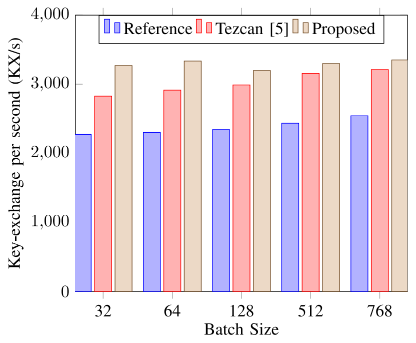

Fig. 5. Performance of FrodoKEM976-AES on a V100.

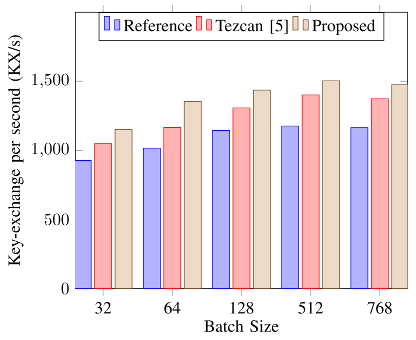

Fig. 6. Performance of FrodoKEM976-AES on a T4.

B. GPU-Based FrodoKEM-AES Implementation

We implemented FrodoKEM976-AES on different GPU platforms; the results are presented in Figures 5, 6 and 7. For all these implementations, we launched many parallel blocks

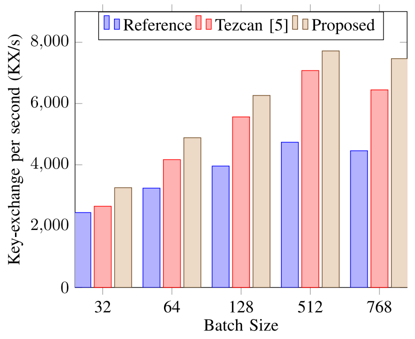

Fig. 7. Performance of FrodoKEM976-AES on a RTX 3080.

and performed one key encapsulation and decapsulation in each block. Random samples were generated through AES-128, which was implemented using the reference method, the technique from Tezcan [5] or the proposed solution. The reference code closely follows the FrodoKEM documentation [2] that implements AES-128 in bit-sliced mode. Each thread performs two AES encryptions, so the number of required registers is smaller than Hajihassani *et al.* [7].

Results show that the proposed solution consistently outperformed the reference code and Tezcan [5], across all GPU platforms. Our FrodoKEM implementation with the proposed AES techniques was able to achieve 3,350, 1,503 and 7,716 key exchanges per second (the best results among all the batch sizes) on V100, T4 and RTX 3080 GPUs, respectively. This result is also 2.99× faster than Gupta *et al.* [8] on V100 GPU.

C. Exhaustive Key Search on GPU

In this experiment, we evaluate the encryption rate of AES-128 on GPU platforms, using different symmetric keys. The plaintext-ciphertext pair used in this experiment is 16 bytes; each thread is responsible for searching one unique key. Hence, the proposed pre-computation technique for Round 1 is not applicable in this application, since there is no fixed pattern between each thread. A known plaintext-ciphertext pair is passed to the GPU, wherein the plaintext is encrypted by AES-128 with multiple unique keys. The resultant ciphertexts is compared with the known ciphertext to recover the key. Note that this is different from the AES-CTR mode encryption, because the round keys are unique and have to be generated on the fly. For this reason, round constants are used frequently in round key generation. For a fast access speed, we stored these round constants into registers. Table III shows the proposed solution can perform 11,428, 3,969, and 9,998 $\times 10^6$ encryptions per second on V100, T4, and RTX 3080 GPUs, respectively. These results are 1.13×, 1.34×, and 1.09× faster, respectively, than Tezcan [5]. This allows a faster password or key recovery to be carried out and reduces the time taken to execute a brute force attack on AES-128.

TABLE III
PERFORMANCE FROM THE AES-128 EXHAUSTIVE KEY SEARCH

	Throughput (10^6 enc/s)		
	V100	T4	RTX 3080
Proposed	11,428	3,969	9,998
Tezcan [5]	10,044	2,959	9,195
Speed-up	1.13	1.34	1.09

V. CONCLUSION

This brief sets a speed record on various GPU platforms for AES-128 CTR mode encryption, which is widely used in various cryptographic applications. Our performance on a V100 GPU was 9% faster than the previous record [7]. Two concrete use cases were presented in this brief, wherein the proposed AES implementation techniques were efficiently applied to a cryptographic scheme (FrodoKEM) and a key exhaustive search. With the proposed technique, the FrodoKEM implementation is 2.99× faster than the state-of-the-art implementation by Gupta *et al.* [8]. For AES encryption and FrodoKEM, the reported throughput takes into consideration the overhead of pre-computation. In other words, even if the key is renewed for every encryption, the proposed technique is still beneficial, since the pre-computation is very lightweight.

REFERENCES

[1] "Advanced encryption standard (AES)," *Federal Inf. Process. Stand. Publ. FIPS-197*, vol. 197, no. 441, p. 311, 2001.
[2] E. Alkim *et al.* "FrodoKEM Learning With Errors Key Encapsulation." 2020. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/FrodoKEM-Round3.zip
[3] NIST, Gaithersburg, MD, USA. "Post-Quantum Cryptography Standardization." 2017. [Online]. Available: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization (Accessed: Oct. 10, 2021).
[4] H. Choi and S. C. Seo, "Optimization of PBKDF2 using HMAC-SHA2 and HMAC-LSH families in CPU environment," *IEEE Access*, vol. 9, pp. 40165–40177, 2021.
[5] C. Tezcan, "Optimization of advanced encryption standard on graphics processing units," *IEEE Access*, vol. 9, pp. 67315–67326, 2021.
[6] W.-K. Lee, H.-S. Cheong, R. C.-W. Phan, and B.-M. Goi, "Fast implementation of block ciphers and PRNGs in maxwell GPU architecture," *Clust. Comput.*, vol. 19, no. 1, pp. 335–347, 2016.
[7] O. Hajihassani, S. K. Monfared, S. H. Khasteh, and S. Gorgin, "Fast AES implementation: A high-throughput bitsliced approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2211–2222, Oct. 2019.
[8] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: Frodokem, newhope, and kyber," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 575–586, Mar. 2021.
[9] E. Käsper and P. Schwabe, "Faster and timing-attack resistant AES-GCM," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Berlin, Germany: Springer, 2009, pp. 1–17.