

Received April 13, 2021, accepted April 28, 2021, date of publication May 4, 2021, date of current version May 12, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3077551

# Optimization of Advanced Encryption Standard on Graphics Processing Units

**CIHANGIR TEZCAN** 

CyDeS Laboratory, Department of Cyber Security, Informatics Institute, Middle East Technical University, 06800 Ankara, Turkey

e-mail: cihangir@metu.edu.tr


**ABSTRACT** Graphics processing units (GPUs) are specially designed for parallel applications and perform parallel operations much faster than central processing units (CPUs). In this work, we focus on the performance of the Advanced Encryption Standard (AES) on GPUs. We present optimizations which remove bank conflicts in shared memory accesses and provide 878.6 Gbps throughput for AES-128 encryption on an RTX 2070 Super, which is equivalent to 4.1 Gbps per Watt. Our optimizations provide more than 2.56x speed-up against the best GPU results in the literature. Our optimized AES implementations on GPUs even outperform any CPU using the hardware level AES New Instructions (AES-NI) and legacy FPGA-based cluster architectures like COPACOBANA and RIVYERA. Even on a low-end GPU like MX 250, we obtained 60.0 Gbps throughput for AES-256 which is generally faster than the read/write speeds of solid disks. Thus, transition from AES-128 to AES-256 when using GPUs would provide military grade security with no visible performance loss. With these breakthrough performances, GPUs can be used as a cryptographic co-processor for file or full disk encryption to remove performance loss coming from CPU encryption. With a single GPU as a co-processor, busy SSL servers can be free from the burden of encryption and use their whole CPU power for other operations. Moreover, these optimizations can help GPUs to practically verify theoretically obtained cryptanalysis results or their reduced versions in reasonable time.

**INDEX TERMS** Cryptography, encryption, cryptanalysis.

## I. INTRODUCTION

The Advanced Encryption Standard (AES) [10] still withstands all cryptanalytic attacks after 20 years of investigation and is arguably the most used encryption algorithm. Its usage in many platforms led to many optimized software implementations of AES for different CPU architectures. Although graphics processing units (GPUs) mainly target graphics application like video gaming, advances in technology made them a good candidate for general purpose computing. Their architecture has single instruction multiple data (SIMD) structure and they can perform the same type of operation on multiple data and obtain better performance than CPUs in parallelizable applications. Thus, GPUs can easily outperform CPUs when a parallelizable mode of operation is chosen for block cipher encryption.

CUDA optimizations for some lightweight ARX (Add-Rotate-XOR) based ciphers are provided in [3] and they obtained encryption throughput of 468 Gbps for HIGHT,

The associate editor coordinating the review of this manuscript and approving it for publication was Jun Wang .

2593 Gbps for LEA, and 3063 Gbps for revised CHAM on a single RTX 2070 Nvidia GPU. However, a general purpose block cipher like AES has more complicated round function compared to these ARX-based lightweight block ciphers and generally has lower throughput. There are many works focusing on the optimization of AES for GPUs, mostly on CUDA devices. CUDA implementations of these works reach high throughput and are more than an order of magnitude faster than software implementations on CPUs. Although the obtained throughputs are increased in years, this is mainly due to the fact that newer GPUs have more cores or higher clock rates. Since almost none of the papers in the literature provided source codes of their AES implementations, it is not possible to compare different results, let alone check the correctness of the provided experimental results.

Fast AES encryption is necessary for file or disk encryption, exhaustive search attacks, practically verifying theoretically obtained cryptanalysis results, and improving the mining performance of cryptocurrencies that use AES-based hashing algorithms in their proof of work consensus algorithms.

There are three main implementation techniques that can be used for GPU implementation of AES: naive, table based, and bitsliced. In the naive approach all of the operations in the round function of AES are implemented directly. Table based implementation divides the input into parts and for every possible input for each part, output of the round function is pre-computed and stored in a table. This way a single AES round can be turned into 16 table look-up and 16 XOR operations by storing four pre-computed tables of 4 KB. The third approach is the bitslicing technique where each of the 128-bit round input bits are stored in different 128 registers and operations are performed to each bit independently. Bitslicing technique provides SIMD level parallelism at the software level because if the registers are  $n$ -bit, we can perform  $n$  block encryption operations in parallel by storing  $n$  blocks in these 128 registers.

In early optimizations like [20], authors observed that the table based implementation was better than the naive approach on GPUs and they obtained 8.28 Gbps on an Nvidia 8800 GTX. Although there are many memory types in GPUs, they kept the tables in the constant memory. Moreover, they used four threads for the encryption of a single 128-bit block which means 4 bytes/thread granularity. However, 1 byte/thread and 16 bytes/thread granularities are also possible since AES operations are performed on bytes and block size of AES is 16 bytes.

The question of finding the best memory type for tables and granularity are answered in [16]. They showed that 16 bytes/thread provides better performance than 4 bytes/thread and 1 byte/thread since it does not require communication between threads. Moreover, it was shown in [16] that storing the tables in the shared memory instead of the constant memory increases the throughput from 5.0 Gbps to 35.2 Gbps on a GTX 285. The tables are copied to shared memory from the global memory at the beginning of each kernel. In order to reduce these memory operations, the granularity is increased to 32 bytes/thread in [1] so that a thread encrypts two blocks instead of one. They achieved 279.9 Gbps on a GTX 1080. Moreover, they used the new warp shuffle feature to speed-up the memory operations but they observed that warp shuffles reduced the performance instead. A similar table based implementation is recently performed in [4] and they obtained 123 Gbps for GTX 970, 214 Gbps for GTX 1070, and 310 Gbps for RTX 2070. The main bottleneck in these table based implementations is the shared memory bank conflicts.

One of the earliest bitsliced CUDA implementations is provided in [19] and they achieved 78.6 Gbps on a GTX 480. More recently, [23] achieved 605.9 Gbps on an Nvidia Tesla P100. The GPU architecture allows limited number of registers per thread block and 100% occupancy can be achieved on modern GPUs only when a thread uses 64 or less 32-bit registers. However, since a bitsliced implementation requires at least 128 register for every bit of the 128-bit block, a bitsliced implementation cannot fully

occupy the GPU. These bitsliced implementations can provide better results on future GPUs if they come with more registers per thread block.

A software implementation running on a CPU can never compete with these GPU results. However, modern CPUs are equipped with hardware instructions for performing AES encryption and this is always overlooked in publications about the performance of AES on GPUs. For fair comparison, GPU results should be compared with the hardware level implementation of AES on CPUs. In this work we obtained 134.7 Gbps AES Encryption on an Intel i7-10700F CPU which is 2.072 Gbps per Watt. This throughput is close to the results of [1] and [4]. Moreover, since CPUs do not consume as much as power as GPUs, throughput per Watt value of i7-10700F CPU is better than the best GPU results of [1] and [4] and close to the results of [23]. Thus, latest CPUs have better energy efficiency.

Our literature analysis showed that the best AES throughput on GPUs using the table based implementation were obtained by keeping the tables in the shared memory of the GPU. This is because shared memory is the fastest memory type in GPUs and AES tables are small enough to fit in the shared memory. However, modern GPUs have warps of 32 threads and shared memory consists of also 32 banks (note that the earlier devices had 16 banks). When two different threads try to read data from the same bank, bank conflict occurs and the accesses are serialized. Bank conflicts are the main bottleneck of the AES optimizations because intermediate values of AES are random and make each thread to access a random shared memory bank.

To obtain the best AES performance, in this work we aimed to remove the bottleneck in the previous best AES optimizations on GPUs by removing the bank conflicts in shared memory accesses. We achieved this aim by exploiting the architecture of modern GPUs and the relation between the tables of AES by analyzing the structure of the matrix of AES. Thus, we obtained the first table based AES CUDA implementation that keeps the tables in the shared memory and still has zero bank conflicts.

In order to show that our optimizations are applicable to all modern CUDA architectures, we used different mobile and desktop CUDA devices with different architectures and observed that our optimizations provide at least 2.56x speed-up against the previous best results on GPUs. Our optimizations also outperform CPUs with AES-NI support in terms of throughput per Watt and some FPGA-based clusters like COPACOBANA and RIVYERA S3-5000.

Finally, we considered the effects of improbable events like soft errors on our implementations. Nvidia provides Tesla GPUs with error correction code (ECC) memory for scientific computations and Geforce GPUs without ECC memory mainly for gaming. We observed that soft errors have negligible effects on our implementations and they can be used on GPUs without ECC memory.

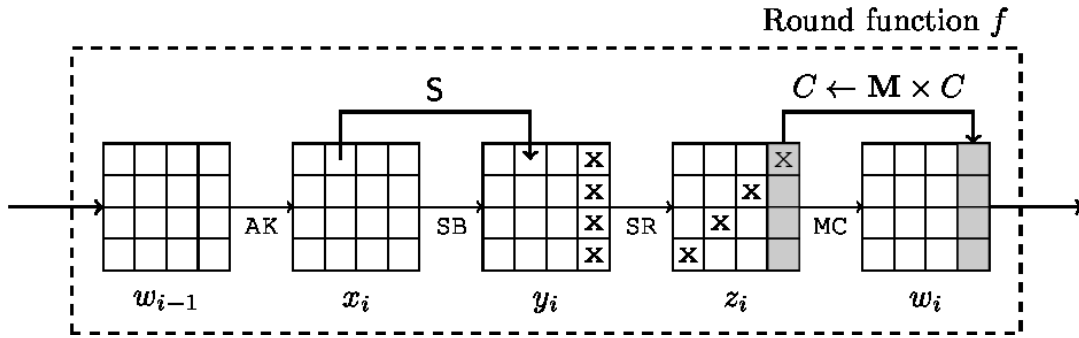


FIGURE 1. One round of AES. Figure is from [17].

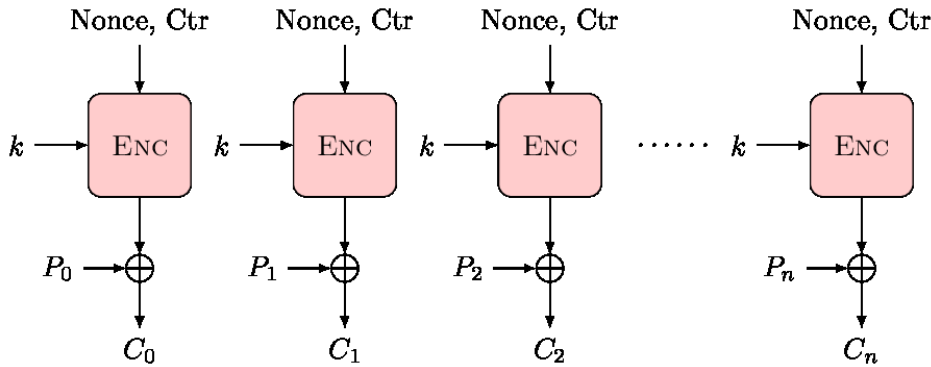


FIGURE 2. Counter (CTR) mode of operation for block ciphers. Figure is from [17].

We made our CUDA implementations publicly available on GitHub to allow further optimization and comparison between future optimizations.<sup>1</sup>

This paper is organized as follows: In Section II we describe AES. In Section III we provide the details of AES-NI and provide AES-NI performance of modern CPUs. We provide our GPU optimizations in Section IV and provide our experiment results in Section V. In Section VI we discuss the possible effects of soft errors on our implementation. We conclude the paper with Section VII. (1), as shown at the bottom of the next page.

## II. AES

The Advanced Encryption Standard [10] supports key sizes of 128, 192 and 256 bits and it is a *Substitution-Permutation network*. We represent AES with  $k$ -bit key as AES- $k$ . All three versions of AES have a block size of 128 bits which

<sup>1</sup>Our optimized CUDA codes that we used in this work are available at [https://www.github.com/cihangirtezcan/CUDA\\_AES](https://www.github.com/cihangirtezcan/CUDA_AES)

is represented as a  $4 \times 4$  matrix of bytes. The values of this internal state are in the finite field  $\mathbb{F}_{256}$  that is defined according to  $x^8 + x^4 + x^3 + x + 1$ , which is an irreducible polynomial.

The number of rounds  $N_r$  that is applied to the state depends on the key size of AES. AES-128 has  $N_r = 10$ , AES-192 has  $N_r = 12$ , and AES-256 has  $N_r = 14$ . One round of AES is shown in Fig. 1 and can be described as the application of SubBytes, ShiftRows, MixColumns, and AddRoundKey operations, respectively. These operations can be summarized as follows:  $f(x) = AK \circ MC \circ SR \circ SB(x)$ , where

- *SubBytes* (SB): the same 8-bit to 8-bit S-Box is applied to each byte of the state 16 times in parallel;
- *ShiftRows* (SR):  $i$ -th row is rotated  $i$  bytes to the left;
- *MixColumns* (MC): each column is multiplied by a constant  $4 \times 4$  invertible matrix  $M$ ;
- *AddRoundKey* (AK): The internal state is XORed with a 128-bit round key.

$$W[i] = \begin{cases} K[i] & \text{for } i < N_k \\ W[i - N_k] \oplus SB(SR(W[i - 1])) \oplus R[i/N_k] & \text{if } i \geq N_k \text{ and } i \equiv 0 \pmod{N_k} \\ W[i - N_k] \oplus SB(W[i - 1]) & \text{for } i \geq N_k, N_k > 6, i \equiv 4 \pmod{N_k} \\ W[i - N_k] \oplus W[i - 1] & \text{otherwise} \end{cases} \quad (1)$$

Note that the first round starts with an additional AddRoundKey operation and the MixColumns operation is omitted in the last round.

The key schedule of AES- $k$  transforms the  $k$ -bit master key into  $N_r + 1$  round keys which are 128 bits regardless of the value of  $k$ . We can denote the round key array by  $W[0, \dots, 4 \times N_r + 3]$  where each value is of 32 bits. The first  $N_k$  words of  $W[\cdot]$  are always the master key where  $N_{128} = 4$ ,  $N_{192} = 6$ , and  $N_{256} = 8$ . The rest of  $W[\cdot]$  are updated according to the Equation 1 where  $K[i]$  is the  $i$ -th column of the master key and  $R[\cdot]$  is an array of constants.

AES round operations MixColumns, SubBytes, ShiftRows, and AddRoundKey can be implemented independently in software. However, an implementation that uses T-tables is also provided in [10]. With the assumption that the architecture supports 32-bit instructions, for an 8-bit S-box input, we can pre-compute the 32-bit output corresponding to a column after the applications of sub-byte, shift-row, and mix-column operations. Thus, by pre-computing 4 tables of size 256 32-bit values, for each column we can reduce the one round encryption process to four table look-ups and four XOR operations. Fastest GPU implementations of AES use T-tables and keep them in the shared memory of the GPU which is the fastest memory type on a GPU. Our main improvements in this work come from removing bank conflicts in shared memory accesses. The four T-tables  $T_i$  can be calculated for every possible byte value of  $x$  as follows

$$T_0[x] = \begin{bmatrix} 2 \cdot S[x] \\ S[x] \\ S[x] \\ 3 \cdot S[x] \end{bmatrix} \quad T_1[x] = \begin{bmatrix} 3 \cdot S[x] \\ 2 \cdot S[x] \\ S[x] \\ S[x] \end{bmatrix}$$

$$T_2[x] = \begin{bmatrix} S[x] \\ 3 \cdot S[x] \\ 2 \cdot S[x] \\ S[x] \end{bmatrix} \quad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ 3 \cdot S[x] \\ 2 \cdot S[x] \end{bmatrix}$$

where “ $\cdot$ ” represents multiplication in the Galois field of  $2^8$ . Note that any T-table of AES differ from each other only by byte rotations. We use this observation in Section IV-B to store a single T-table in the shared memory, instead of four.

Although AES is a byte oriented cipher, using 32-bit unsigned integers in software implementations reduces the number of instructions and provide better performance. If we represent the columns of AES with  $c0$ ,  $c1$ ,  $c2$ , and  $c3$ , a round encryption of AES in C or CUDA language can be performed using T-tables as follows

```
t0 = T0[c0>>24] ^ T1[(c1>>16)&0xFF] ^ T2[(c2>>8)&0xFF] ^
    T3[c3&0xFF] ^ rk0;
t1 = T0[c1>>24] ^ T1[(c2>>16)&0xFF] ^ T2[(c3>>8)&0xFF] ^
    T3[c0&0xFF] ^ rk1;
t2 = T0[c2>>24] ^ T1[(c3>>16)&0xFF] ^ T2[(c0>>8)&0xFF] ^
    T3[c1&0xFF] ^ rk2;
t3 = T0[c3>>24] ^ T1[(c0>>16)&0xFF] ^ T2[(c1>>8)&0xFF] ^
    T3[c2&0xFF] ^ rk3;
```

where  $rk0$ ,  $rk1$ ,  $rk2$ , and  $rk3$  represent 32-bit words of the round key.

### III. AES NEW INSTRUCTIONS

In March 2008, CPU manufacturers AMD and Intel proposed an extension to x86 instruction set which is called Advanced Encryption Standard New Instructions (AES-NI). As described in Intel’s white paper [13], AES-NI contain four instructions AESDEC, AESDECLAST, AESENC, and AESENCLAST that perform one round AES decryption or encryption. Since the last round of AES does not contain the MixColumns operation, we need to use AESENCLAST and AESDECLAST only for the last round. AES-NI has two more instructions called AESIMC and AESKEYGENASSIST which are used in the key schedule. Basic C code examples of AES-NI for all key sizes of AES using CTR, CBC, and ECB modes of operations are also provided in [13].

CTR mode of operation for block ciphers encrypts a counter value and XORs the output with the plaintext block. Counter value is incremented for each block. This way it allows full parallelization. Note that this mode of operation turns the block cipher into a keystream generator. Therefore, a counter value should not be used more than once. For this reason, generally the counter is concatenated to a randomly generated nonce for each encryption as shown in Fig. 2.

CPUs with AES-NI provide users to perform AES encryption at the hardware level. Before the production of the first CPUs with these instructions, it was expected in [12] that these new instruction sets provide around 2-3x speed-up for non-parallelizable modes like CBC encrypt and the performance speed-up exceeds an order of magnitude over any software only AES implementations when using parallelizable modes of operations like CBC decrypt and CTR. Performance results for Intel i7-980X was later provided in Intel white paper [2] and the practical experiments met the theoretical expectations. They achieved 0.24 cycles/byte on 6 cores (12 threads) for AES-128 on parallel modes for large buffers. The Intel i7-980X CPU has a maximum turbo frequency of 3.60 GHz and a base processor frequency of 3.33 GHz. Since the turbo was disabled in [2], the base clock speed of 3.33 GHz means that they got 102.4 Gbps throughput.

Looking at single core/thread performance provides more information about the speed-up that comes with AES-NI. In [2], authors obtain 1.30 cycles/byte on a single core of Intel i7-980X and our implementation also achieved 1.30 cycles/byte on a single core of Intel i7-6700K. This shows that AES-NI performance has not changed after six generations. i7-6700K has a base processor frequency of 4.00 GHz and maximum turbo frequency of 4.20 GHz. Although, it has only 4 cores and 8 threads, due to its higher frequency than i7-980X, we obtained 90.6 Gbps on 8 threads.

Since modern CPUs have 8 or more cores, we also checked AES-NI performance on Intel i7-10700F. i7-10700F has 8 cores and 16 threads with a base processor frequency of 2.90 GHz and maximum turbo frequency of 4.80 GHz. We obtained 134.7 Gbps on i7-10700F using all 16 threads. These results are summarized in Table 1.

**TABLE 1.** AES-128 throughput results of CPUs with AES-NI in counter mode of operation. Thermal Design Power (TDP) represents the maximum power consumption under full load when CPU works at the base clock speed.

Gbps/W	Gbps	Device	Architecture	Cores	Base Clock	Launch Year	TDP	Reference
0.788	102.4	Intel i7-980X	Gulftown	6	3.33 GHz	2010	130 W	[2]
0.996	90.6	Intel i7-6700K	Skylake	4	4.00 GHz	2013	91 W	Section III
2.072	134.7	Intel i7-10700F	Comet Lake	8	2.90 GHz	2020	65 W	Section III

**TABLE 2.** Some architectural differences between compute capabilities of CUDA devices.

	5.0	5.2	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6
Max. number of resident blocks per multiprocessor	32	32	32	32	32	32	32	16	32	16
Max. number of resident warps per multiprocessor	64	64	64	64	64	64	64	32	64	48
Max. number of resident threads per multiprocessor	2048	2048	2048	2048	2048	2048	2048	1024	2048	1536
Max. number of 32-bit registers per thread block	64K	64K	64K	64K	32K	64K	64K	64K	64K	64K
Max. amount of shared memory per multiprocessor	64KB	96KB	64KB	96KB	64KB	96KB	96KB	64KB	164KB	100KB
Max. amount of shared memory per thread block	48KB	48KB	48KB	48KB	48KB	96KB	48KB	64KB	163KB	99KB

Intel and AMD have recently announced that their future CPUs will have vectorized AES-NI instructions, namely VAESENC, VAESENCLAST, VAESDEC, and VAESDECLAST which can work on wide registers like the ones that are available on the AVX512 architectures. In [11], based on simulations and theoretical analysis, it was claimed that these new instructions can provide a 4 times speed-up on AES encryptions that use parallelizable mode of operation, like CTR which is shown in Fig. 2. Although these vectorized instructions would reduce the total instruction count, such a speed-up would not be possible unless the CPUs contain 4 times more pipes that can perform AES encryption. These new CPUs are on the market for some time now and we have not seen any practical verification of the claims of [11]. For this reason, we are not including these theoretical results in Table 1. However, even if these claims are correct, the theoretical value of 0.24 cycle/byte that was obtained in [11] means that even an 8 core CPU at 4.00 GHz would not surpass our optimizations on modern GPUs. On the other hand, we believe that these new vectorized AES instructions can be used to speed up the key schedule. Such an improvement can be useful when performing exhaustive search or similar tasks but the key schedule is performed only once in a CTR encryption. Thus, it would not affect the throughput.

#### IV. AES GPU OPTIMIZATIONS

Our breakthrough AES performance on CUDA devices comes from the following achievements:

- 1) Replacing the two SHIFT and one AND instructions with `__byte_perm()` CUDA instruction for byte rotations of T-tables
- 2) Removing the shared bank memory conflicts when accessing the T-tables
- 3) Removing the shared bank memory conflicts when accessing the S-box in the last round and during the key schedule

To show that our optimizations do not focus on a specific architecture or compute capability (CC), we performed all of our experiments in this work on different mobile and desktop GPUs. Some architectural differences between

compute capabilities that might affect the performance of our optimizations are provided in Table 2. However, many details about the GPU memory systems remain unknown to the public because Nvidia provides very limited information. It is shown in [22] that in many different architectures the throughput and latency of the shared and global memory accesses differ significantly. Similarly, it is shown in [22] that the access latency varies significantly on different architectures when there are shared memory bank conflicts. We provide the specifications of the GPUs that we used in this work in Table 3.

**TABLE 3.** The specifications of the GPUs that are used in this work. Note that clock rates might change depending on the GPU manufacturer.

GPU	Cores	Clock Rate	CC	Architecture
MX 250	384	1582 MHz	6.1	Pascal
GTX 860M	640	1020 MHz	5.0	Maxwell
GTX 970	1664	1253 MHz	5.2	Maxwell
RTX 2070 Super	2560	1770 MHz	7.5	Turing

#### A. `__byte_perm()` CUDA INSTRUCTION

Although many processors have the *SHIFT* instruction, they rarely have a single instruction for rotation. For this reason, bit rotations are generally implemented as two *SHIFT* operations and an *AND* operation. CUDA devices do not have an instruction for bit rotations as well but we observed that `__byte_perm()` CUDA instruction can be used for byte permutations. This way we can perform a rotation using a single instruction, instead of three.

`__byte_perm(x, y, n)` instruction of CUDA returns four selected bytes from two 32-bit unsigned integers  $x$  and  $y$ . It has three inputs where  $x$  and  $y$  are the two 32-bit values whose bytes are going to be selected and  $n$  is the selector which represents the bytes that are going to be selected. The ShiftRows operation of AES consists of 8, 16, and 24-bit left rotations of 32-bit values. These rotation numbers are multiples of 8, so `__byte_perm()` CUDA instruction can be used for these permutations. Since our implementations use T-tables we do not perform the ShiftRows in our codes. However, the T-tables of AES are just byte rotations of each other and in our optimizations we are going to keep multiple



copies of only one T-table  $T_0$  in the shared memory. Remaining T-tables are going to be obtained by the `__byte_perm()` CUDA instruction. T-table  $T_i$  is just  $i$  byte right rotation of  $T_0$ .

Since we need to rotate a 32-bit value, we chose  $x = y$  in our implementations. By analyzing the `__byte_perm()` CUDA instruction, we observed that for 8, 16, 24-bit right rotations, the values of  $n$  should be 25923, 21554, and 17185, respectively. These integer values correspond to  $0 \times 00006543$ ,  $0 \times 00005432$ , and  $0 \times 00004321$  in hexadecimal. Thus, we defined our `Rot()` device function that rotates a 32-bit value to right as follows:

```
__device__ u32 Rot(u32 x, u32 n) {
    return __byte_perm(x, x, n);
}
```

We implemented rotation operations both with the `__byte_perm()` instruction and traditional arithmetic operations. We observed around %2 speed-up in all tested GPUs when the rotations are performed via the `__byte_perm()` instruction.

## B. REMOVING SHARED MEMORY BANK CONFLICTS IN T-TABLES ACCESSES

Almost every CUDA implementation we observed so far kept the four T-tables using 256 32-bit values in shared memory which results in 4 KBs of shared memory. Our experiments showed that keeping these tables in the shared memory instead of any other type of GPU memory provides at least 10.23x speed-up. Since the GPU kernel is run with more than 256 threads, the first 256 threads write these tables  $T_0$ ,  $T_1$ ,  $T_2$ , and  $T_3$  to shared memory by copying them from the global memory as follows:

```
__shared__ u32 T3[256], T2[256], T1[256], T0[256];
if (threadIdx.x < 256) {
    T0[threadIdx.x] = T0G[threadIdx.x];
    T1[threadIdx.x] = T1G[threadIdx.x];
    T2[threadIdx.x] = T2G[threadIdx.x];
    T3[threadIdx.x] = T3G[threadIdx.x];
}
```

where  $T0G$ ,  $T1G$ ,  $T2G$ , and  $T3G$  are the T-tables that are kept in the global memory.

Shared memory is divided into 32-bit memory modules and these banks can be accessed simultaneously. Note that the bandwidth of each bank is 32 bits per clock cycle and successive 32-bit values are assigned to successive banks. Thus, when we declare the table  $T_0$  as an array of 256 32-bit words,  $T_0[0]$ ,  $T_0[31]$ ,  $T_0[63]$ ,  $T_0[95]$ ,  $T_0[127]$ ,  $T_0[159]$ ,  $T_0[191]$ ,  $T_0[223]$  are stored in bank 0. Similarly,  $T_0[1]$ ,  $T_0[32]$ ,  $T_0[64]$ ,  $T_0[96]$ ,  $T_0[128]$ ,  $T_0[160]$ ,  $T_0[192]$ ,  $T_0[224]$  are stored in bank 1, and so on.

Each warp of CUDA consists of 32 threads and if each thread in a warp accesses different banks then no bank conflicts occur. An exception is when all threads in a warp try to access the same shared memory value in which case the data is broadcast and no bank conflict occurs. But if two or more threads in a warp try to read data from the same shared memory bank, these requests become serialized and this situation is referred as a bank conflict.

T-table accesses during the encryption are randomized because during encryption intermediate round values become random. This results in many bank conflicts each round. We remove these bank conflicts by duplicating the T-table 32 times for each bank and reserve banks to individual threads in a warp. In CUDA we can store a copy of the T-table  $T_0$  in each 32 bank as follows

```
__shared__ u32 T0[256][32];
if (threadIdx.x < 256)
    for (u8 bankIndex = 0; bankIndex < 32; bankIndex++)
        T0[threadIdx.x][bankIndex] = T0G[threadIdx.x];
```

and each thread in a warp can access its own bank to avoid bank conflicts as follows

```
int wTI = threadIdx.x & 31;
T0[c0 & 0xFF][wTI];
```

This way we spend 32 KBs of shared memory instead of 1 KB. Duplicating remaining three tables would sum up to 128 KBs but the GPUs on the market have 64 KBs of shared memory. Although CUDA devices with compute capabilities 7.0, 8.0, and 8.6 have shared memory of sizes 96 KB, 163 KB, and 99 KB, only 64 KB of them are accessible by users. This is why current GPUs do not allow us to store a second T-table in each bank. However, since all of the T-tables of AES can be obtained from a single T-table by performing byte rotations, in this work we only kept  $T_0$  in the shared memory and obtained other table values by rotating  $T_0$ . This way one round of AES encryption turns into the following CUDA codes

```
t0 = T0[c0 >> 24][wTI] ^
    Rot(T0[(c1 >> 16) & 0xFF][wTI], 17185) ^
    Rot(T0[(c2 >> 8) & 0xFF][wTI], 21554) ^
    Rot(T0[c3 & 0xFF][wTI], 25923) ^ rk0;
t1 = T0[c1 >> 24][wTI] ^
    Rot(T0[(c2 >> 16) & 0xFF][wTI], 17185) ^
    Rot(T0[(c3 >> 8) & 0xFF][wTI], 21554) ^
    Rot(T0[c0 & 0xFF][wTI], 25923) ^ rk1;
t2 = T0[c2 >> 24][wTI] ^
    Rot(T0[(c3 >> 16) & 0xFF][wTI], 17185) ^
    Rot(T0[(c0 >> 8) & 0xFF][wTI], 21554) ^
    Rot(T0[c1 & 0xFF][wTI], 25923) ^ rk2;
t3 = T0[c3 >> 24][wTI] ^
    Rot(T0[(c0 >> 16) & 0xFF][wTI], 17185) ^
    Rot(T0[(c1 >> 8) & 0xFF][wTI], 21554) ^
    Rot(T0[c2 & 0xFF][wTI], 25923) ^ rk3;
```

Note that if we can get GPUs with more shared memory in the future, we can keep more than one T-table in the shared memory and get rid of the rotation instructions. This would provide significant speed-ups.

## C. REMOVING SHARED MEMORY BANK CONFLICTS IN S-BOX ACCESSES

We cannot use the T-tables in the last round of AES because the last round does not have the MixColumns operation. For this reason, we keep the AES S-box in the shared memory and use it in the last round. Moreover, the S-box is also used in the key schedule. Although we perform the key schedule once on CPU and transfer round keys from CPU to GPU for encryption, we need to perform the key schedule repeatedly on the GPU cores for other use cases like exhaustive search attack.

Removing the shared memory bank conflicts in T-tables provided huge speed-up in experiments on CUDA devices with Kepler, Maxwell, and Pascal architectures. However, the speed-up in our experiments with RTX 2070 Super which has Turing architecture was less than what we expected. We used Nvidia Nsight Compute to detect the bottleneck and observed that the bank conflicts of the S-box accesses caused the bottleneck for this GPU. We do not have another 32 KBs of shared memory so we cannot keep 32 copies of the S-box in the way we just did for the T-table  $T_0$ . However, since the S-box produces an 8-bit output, we can store every four values of the S-box in a single bank since bank size is 32 bits. This way the total table size reduces to 8 KBs from 32 KBs. We achieved this by storing the S-box in a three dimensional array  $S[64][32][4]$  where the thread  $i$  can access the output of the S-box with input  $j$  as  $S[j/4][i][j\%4]$ . The following CUDA code copies the S-box values  $SG[256]$  from the global memory and writes them to our three dimensional array  $S[64][32][4]$  in the shared memory:

```
__shared__ u8 S[64][32][4];
if (threadIdx.x < 256)
for (u8 bank = 0; bank < 32; bank++)
S[threadIdx.x/4][bank][threadIdx.x%4]=SG[threadIdx.x];
```

Thus, the last round of AES-128 that does not have any shared memory bank conflicts turns into the following CUDA code:

```
s0 = Rot(S[((c0>>24)/4][wTI][((c0>>24)%4),17185)
^ Rot(S[((c1>>16)&0xff)/4][wTI][((c1>>16)%4),21554)
^ Rot(S[((c2>>8)&0xff)/4][wTI][((c2>>8)%4),25923)
^ (S[((c3&0xFF)/4)][wTI][((c3&0xFF)%4)]) ^ rkS[40];
s1 = Rot(S[((c1>>24)/4][wTI][((c1>>24)%4),17185)
^ Rot(S[((c2>>16)&0xff)/4][wTI][((c2>>16)%4),21554)
^ Rot(S[((c3>>8)&0xff)/4][wTI][((c3>>8)%4),25923)
^ (S[((c0&0xFF)/4)][wTI][((c0&0xFF)%4)]) ^ rkS[41];
s2 = Rot(S[((c2>>24)/4][wTI][((c2>>24)%4),17185)
^ Rot(S[((c3>>16)&0xff)/4][wTI][((c3>>16)%4),21554)
^ Rot(S[((c0>>8)&0xFF)/4][wTI][((c0>>8)%4),25923)
^ (S[((c1&0xFF)/4)][wTI][((c1&0xFF)%4)]) ^ rkS[41];
s3 = Rot(S[((c3>>24)/4][wTI][((c3>>24)%4),17185)
^ Rot(S[((c0>>16)&0xff)/4][wTI][((c0>>16)%4),21554)
^ Rot(S[((c1>>8)&0xFF)/4][wTI][((c1>>8)%4),25923)
^ (S[((c2&0xFF)/4)][wTI][((c2&0xFF)%4)]) ^ rkS[43];
```

Our motivation to remove the bank conflicts in the S-box was due to the fact that unlike previous architectures, RTX 2070 Super with Turing architecture did not achieve the expected speed-up when we removed the bank conflicts in the T-tables. We overcame this problem when we removed the bank conflicts in the S-box and obtained an implementation that has zero bank conflicts.

This optimization provided speed-ups for other GPUs that we used in this work as well, with the exception of GTX 860M, which is the oldest GPU that we used. For example, we can verify  $2^{29.18}$  keys per second on GTX 860M without this optimization when performing an exhaustive search attack. However, this number reduces to  $2^{28.97}$  keys per second when we remove the bank conflicts in the S-box. Apparently the bank conflicts in the S-box accesses are not the cause of the bottleneck for this GPU and using more shared memory by having 32 copies of the S-box and performing byte rotations for the output of S-box reduce the occupancy and cause a bottleneck.

Therefore, this optimization might cause a negative effect on legacy GPUs.

Also note that GTX 860M and GTX 970 GPUs both have the Maxwell architecture and this optimization provided significant speed-ups on GTX 970. Aside from the architecture, the capabilities of GPUs are determined by their compute capability. GTX 860M and GTX 970 have compute capabilities 5.0 and 5.2, respectively. Therefore, the performance loss of GTX 860M might be coming from its compute capability.

The performance change for RTX 2070 Super and GTX 860M with this optimization shows that the bottleneck might be different for different GPU architectures or models and further optimizations might be necessary for future architectures. One reason for these differences might be explained by the performance of different compute capabilities under shared memory bank conflicts. In [22], authors observed that Maxwell has superiority against Fermi and Kepler in performance under shared memory bank conflicts. Our results show that this performance might also depend on the compute capability.

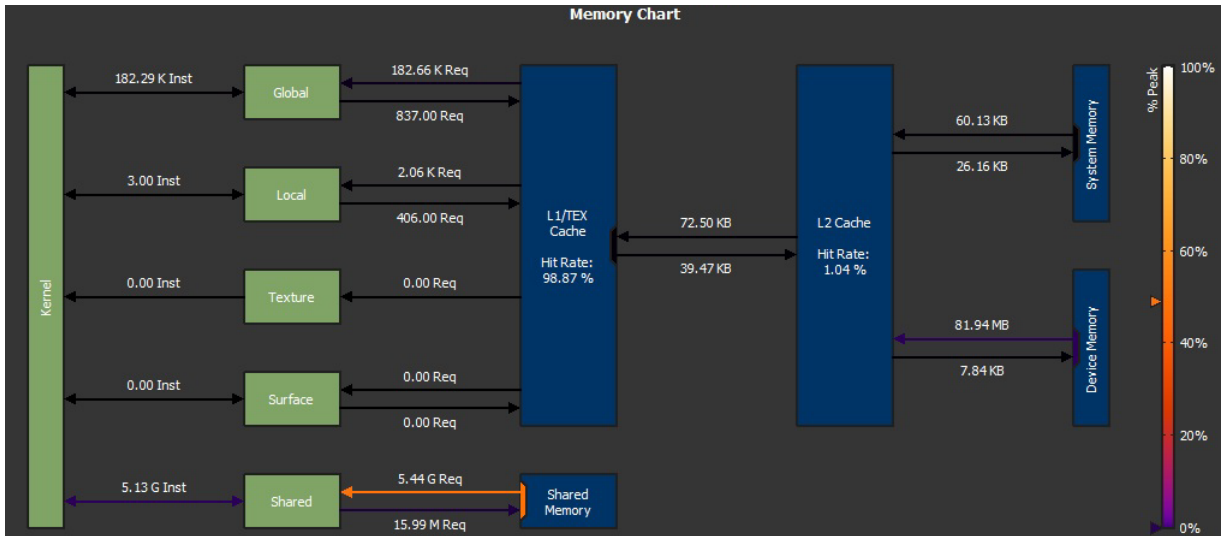
## V. POTENTIAL APPLICATIONS

Our AES optimizations can be used for file or disk encryption, exhaustive search attacks, practically verifying theoretically obtained cryptanalysis results, and improving the mining performance of cryptocurrencies that use AES-based hashing algorithms in their proof of work consensus algorithms. These potential applications either use the key schedule only once as in encryption or multiple times as in the exhaustive search attack. Thus, we provide our experimental results for these two cases in the following subsections.

A CUDA kernel has two launch parameters: block and thread counts. After many benchmarking, in our implementations we defined thread count as 1024 to achieve maximum performance and upper bounded the block count again as 1024 to fully occupy the GPU. Thus, a thread performs more than one encryption when the total number of encryptions exceeds  $2^{20}$ . For example, an experiment that performs  $2^{35}$  encryptions use  $2^{10}$  CUDA blocks with  $2^{10}$  threads and each thread performs  $2^{15}$  encryptions. Thus, for  $2^n$  block encryptions our implementations have 16 bytes/thread granularity when  $n < 20$  and  $2^{n-20} \cdot 16$  bytes/thread granularity when  $n \geq 20$ .

## A. ENCRYPTION

We performed all our experiments on various CUDA SDKs like versions 6.5, 7.5, 8, 9, 10, and 11.1 and many compute capabilities like 3.7, 5.0, 5.2, 6.1, and 7.5. Although, we observed negligible differences, compute capability 5.2 generally provided the best results. Note that a GPU is backward compatible in terms of compute capability. For example, RTX 2070 Super has CC 7.5 and can run codes that are compiled for earlier CC like 6.1 or 5.2. Our throughput results for AES-256, AES-192, and AES-128 on many different Nvidia GPUs are provided in Table 4.



**FIGURE 3.** Memory chart for  $2^{30}$  CTR block encryptions (16 GBs) on RTX 2070 Super using CUDA SDK 11.1. Compiled for compute capability 7.5.

Since plaintext is not used during the encryption in CTR mode, we do not need to copy the plaintext to GPU memory. Thus, AES encryption can be performed even before the plaintext is ready. Note that we can copy the keystream from the GPU global memory to RAM asynchronously while GPU kernel keeps encrypting other counters. Thus, an optimized implementation would not suffer from memory copy operations. For instance, the RTX 2070 Super has a memory bandwidth of 448 GB/second.

Memory chart for  $2^{30}$  CTR block encryptions (16 GBs) on RTX 2070 Super is provided in Fig. 3 in order to show the intensity of shared memory accesses compared to other memory operations.

File encryption and full disk encryption are other applications that could benefit from these GPU performances. Although we reached 878.6 Gbps on an RTX 2070 Super, current disk drives have lower read and write speeds. Thus, the speed of these applications would be limited by the disk speed and the encryption operation would not cause any delay. We even obtained 102.7 Gbps AES-128 throughput on MX 250 which is a basic GPU that is used in ultrabooks. Thus, encryption would cause no delay even when a low-end GPU is used as a cryptographic co-processor.

Table 4 shows that even the low-end GPUs have AES-256 encryption performance which is better than the current read and write speed of solid state disks. Thus, for file or full disk encryption, transition from AES-128 to AES-256 would not have any visible performance loss.

GPUs are more efficient than CPUs in parallel operations due to their single instruction multiple data level parallelism. GPUs can outperform CPUs when encrypting huge files using block ciphers with a mode of operation that is parallelizable. The literature has many AES GPU performance results that are significantly faster than software implementations on CPU. The most recent and the best results using

**TABLE 4.** Throughput results of AES with counter mode of operation.

GPU	AES-128	AES-192	AES-256
MX 250	102.7 Gbps	72.3 Gbps	60.0 Gbps
GTX 860M	95.9 Gbps	77.1 Gbps	65.3 Gbps
GTX 970	315.2 Gbps	254.8 Gbps	214.7 Gbps
RTX 2070 Super	878.6 Gbps	718.3 Gbps	606.9 Gbps

the table based implementation in the literature belong to [4], which is published in March 2020. The best throughputs obtained in [4] are 123 Gbps for GTX 970, 214 Gbps for GTX 1070, and 310 Gbps for RTX 2070. Although these results are impressive, comparing them to software implementations on CPUs would be misleading because almost every Intel and AMD CPU come with AES-NI since 2009. For instance, we achieved 90.6 Gbps on a 4 core Intel CPU i7-6700K. This shows that modern 8-10 core CPUs can easily exceed 200 Gbps which make them comparable to the GPU results of [4] and better in terms of energy efficiency.

Our encryption results are summarized in Table 5 and compared with the results of the most recent and the best GPU results of the related works.

Our optimizations allowed us to reach 315.2 Gbps on GTX 970. This result allows us to compare our optimizations with the previous GPU results because [4] reached 123 Gbps on the same GPU. Thus, our throughput is at least 2.56 times faster. Our optimizations also outperform bitsliced implementations of [19] and [23] as it is shown in Table 5. Moreover, we obtained 878.6 Gbps on RTX 2070 Super which shows that GPUs outperform every consumer grade CPU that come with AES-NI. This also shows that better GPUs like RTX 2080 Super or the new RTX 3000 series have throughputs at terabits per second level. To the best of our knowledge, these are the best AES throughput results. These results show that servers can use a GPU as a cryptographic co-processor



**TABLE 5.** AES-128 encryption performance on different CUDA devices and Intel CPUs with AES-NI. For fair comparison, the table is sorted with respect to Gbps per Watt. CPU results are given in italic.

Gbps/W	Gbps	Device	Architecture	Launch Year	Max Power	Reference
0.053	8.3	Nvidia 8800 GTX	Tesla	2006	155 W	[20]
0.173	35.2	Nvidia GTX 285	Tesla 2.0	2008	204 W	[16]
0.236	50.8	Nvidia RTX 2070 Super	Turing	2019	215 W	[5]
0.314	78.6	Nvidia GTX 480	Fermi	2010	250 W	[19]
0.322	80.5	Nvidia GTX 780	Kepler	2013	250 W	[1]
<i>0.788</i>	<i>102.4</i>	<i>Intel i7-980X</i>	<i>Gulftown</i>	<i>2010</i>	<i>130 W</i>	[2]
0.829	207.3	Nvidia GTX TITAN X	Maxwell	2015	250 W	[1]
0.848	123.0	Nvidia GTX 970	Maxwell	2014	145 W	[4]
<i>0.996</i>	<i>90.6</i>	<i>Intel i7-6700K</i>	<i>Skylake</i>	<i>2013</i>	<i>91 W</i>	Section III
1.427	214.0	Nvidia GTX 1070	Pascal	2016	150 W	[4]
1.555	279.9	Nvidia GTX 1080	Pascal	2016	180 W	[1]
1.771	310.0	Nvidia RTX 2070	Turing	2018	175 W	[4]
<i>2.072</i>	<i>134.7</i>	<i>Intel i7-10700F</i>	<i>Comet Lake</i>	<i>2020</i>	<i>65 W</i>	Section III
2.174	315.2	Nvidia GTX 970	Maxwell	2014	145 W	Section V-A
2.423	605.9	Nvidia Tesla P100	Pascal	2016	250 W	[23]
4.087	878.6	Nvidia RTX 2070 Super	Turing	2019	215 W	Section V-A

and perform encryption and decryption operations on the fly without consuming any CPU power. In this way, busy SSL servers would be free from the burden of encryption.

## B. EXHAUSTIVE SEARCH

The security of block ciphers is upper bounded by exhaustive search. When an attacker captures a single plaintext-ciphertext pair, they can encrypt the plaintext block with every possible key and check if the expected ciphertext block is observed. This is a generic attack and any block cipher with a  $k$ -bit key can be broken by performing  $2^k$  encryptions. Thus, the key length  $k$  is decided depending on the current technology. For instance, NIST predicts it is secure to use keys of length 112 bits or longer until 2030 (see NIST SP 800-57, Recommendation for Key Management).

We used the key schedule algorithm only once when we were measuring the throughput of our optimized CUDA codes since we use a single key for encryption. However, for exhaustive search we use a different key for every block encryption. Thus, the key schedule algorithm should also be run on the GPU. Removing the shared bank conflicts in the S-box accesses provides a huge speed-up for the exhaustive search because the S-box is used four times in every round of the key schedule. Our best optimizations allowed us to check  $2^{32.433}$  keys per second on an RTX 2070 Super when performing an exhaustive search attack on AES-128. Since a year has around  $2^{24.910}$  seconds, one would need  $2^{70.657}$  RTX 2070 Super GPUs to break AES-128 in a year.

Our exhaustive search results for AES-256, AES-192, and AES-128 on many different Nvidia GPUs are provided in Table 6.

In order to observe the performance gain of our optimizations, we used Nvidia Nsight Compute version 2020.2.1 (Build 29181059). We first implemented AES using the four T-tables  $T_0$ ,  $T_1$ ,  $T_2$ , and  $T_3$  as it is done in [1], [5], [16], [20], and [4]. Although the previous studies did not publish their CUDA codes, the performance of this implementation matches the throughput results of [4]. Thus, comparison of our optimizations with this implementation can be regarded

**TABLE 6.** Number of key searches per second for the exhaustive search attack on AES.

GPU	AES-128	AES-192	AES-256
MX 250	$2^{29.19}$ keys/s	$2^{28.75}$ keys/s	$2^{28.42}$ keys/s
GTX 860M	$2^{29.18}$ keys/s	$2^{28.72}$ keys/s	$2^{28.56}$ keys/s
GTX 970	$2^{30.74}$ keys/s	$2^{30.51}$ keys/s	$2^{30.25}$ keys/s
RTX 2070 Super	$2^{32.43}$ keys/s	$2^{32.01}$ keys/s	$2^{31.66}$ keys/s

as the comparison of our best results with the best results of the previous studies. Then we removed the bank conflicts in T-tables in our second kernel, and in the third kernel we removed the bank conflicts in S-boxes. We performed experiments on these kernels by searching  $2^{30}$  AES-128 keys on an RTX 2070 Super and we provide Nvidia Nsight Compute results in Table 7. For comparison, we also performed CTR mode AES-128 encryption of  $2^{30}$  blocks (16 GBs) using our best optimization and we provide Nvidia Nsight Compute results also in Table 7.

CUDA Nsight Compute results that are provided in Table 7 confirm that our optimizations provide bank conflict free implementations of AES for the first time. It also shows a 2.61x speed-up when performing exhaustive search compared to the naive T-table based CUDA implementation which was the best GPU implementation before this work. It can be seen that the achieved occupancy and active warps are close to the theoretical values in our optimizations. Although registers per thread are reported as 56, 60, 60, and 47, these large numbers do not affect the occupancy because compiling the same codes for CC 5.2 instead of 7.5 reduces the register count to 32 and does not result in any observable speed-up.

Although our main speed-ups came from removing shared memory bank conflicts, we also benefited from using the `__byte_perm()` CUDA instruction. We observed around %2 performance gain when we replaced our rotations with the `__byte_perm()` CUDA instruction which is provided in Table 8. Bit rotations are common in cryptographic algorithms. Thus, when the rotation number is a multiple of 8, we recommend the use of the `__byte_perm()` CUDA instruction instead of the common practice of using two `SHIFT`

**TABLE 7.** Nvidia Nsight Compute version 2020.2.1 (Build 29181059) results of optimizations for  $2^{30}$  key searches or  $2^{30}$  CTR block encryptions (16 GBs) on RTX 2070 Super using CUDA SDK 11.1. Compiled for compute capability 7.5.

Speed of Light (SOL)	CTR Encryption	Exhaustive 3	Exhaustive 2	Exhaustive 1
Duration (msecond)	171.31	220.40	290.86	577.39
SOL SM	92.61%	93.67%	70.87%	35.78%
SOL Memory	92.61%	93.67%	70.87%	49.24%
SOL L1 Cache	98.23%	95.02%	98.46%	98.49%
SOL L2 Cache	0.03%	0.03%	0.03%	0.03%
Elapsed Cycles	277,516,269	357,050,134	472,132,240	935,278,015
SM Active Cycles	273,187,856.45	351,422,359.27	463,892,406.73	919,141,871.05
SM Frequency (cycle/nsecond)	1.62	1.62	1.62	1.62
L1 Hit Rate	99.46%	99.23%	99.23%	99.23%
<b>Shared Memory</b>				
Load Instructions	5,134,417,920	6,677,331,968	6,677,331,968	6,677,331,968
Wavefronts	5,436,473,344	6,677,331,968	9,274,214,116	18,382,330,945
Peak	49.02%	46.83%	49.21%	49.24%
Bank Conflicts	0	0	2,596,882,148	11,704,998,977
<b>Instruction Statistics</b>				
Executed Instructions	27,923,829,032	41,811,083,264	37,415,329,792	31,039,627,264
Issued Instructions	27,923,833,520	41,811,087,734	37,415,333,492	31,039,630,790
<b>Launch Statistics</b>				
Grid Size	1,024	1,024	1,024	1,024
Block Size	1,024	1,024	1,024	1,024
Threads	1,048,576	1,048,576	1,048,576	1,048,576
Registers per Thread	56	60	60	47
Static Shared Memory	41.14 KB/block	41.04 KB/block	41.04 KB/block	2.12 KB/block
<b>Occupancy</b>				
Theoretical Occupancy	100%	100%	100%	100%
Achieved Occupancy	94.07%	97.59%	97.62%	97.98%
Theoretical Active Warps	32 warp/SM	32 warp/SM	32 warp/SM	32 warp/SM
Achieved Active Warps	30.10 warp/SM	31.23 warp/SM	31.24 warp/SM	31.35 warp/SM

**TABLE 8.** Latency comparison between `__byte_perm()` CUDA instruction and arithmetic rotation when performing  $2^{35}$  key searches during an exhaustive search attack. Arithmetic rotations use two *SHIFT* operations and an *AND* operation.

GPU	<code>__byte_perm()</code>	Arithmetic rotation
MX 250	56.070 seconds	57.083 seconds
GTX 860M	56.667 seconds	57.418 seconds
GTX 970	19.209 seconds	19.246 seconds
RTX 2070 Super	5.924 seconds	6.059 seconds

operations and an *AND* operation. Depending on the architecture and the code, `__byte_perm()` CUDA instruction might provide significant performance gain.

We also compared our results with FPGAs and Biryukov's imaginary GPU-based supercomputer [7]. Another hardware that can be used for cryptanalysis is FPGA-based cluster architectures like COPACOBANA or RIVYERA [14]. In 2006, the architecture of a special-purpose hardware system called COPACOBANA (Cost-Optimized Parallel Code Breaker), which provides a cost-performance that is significantly better than that of recent PCs for the exhaustive key search on DES, is introduced in CHES workshop [18]. COPACOBANA uses 120 Xilinx Spartan-3 FPGAs, costs around 10,000 Euros and checks approximately 65 billion DES keys per second. RIVYERA, the commercial successor of the COPACOBANA, uses 128 Xilinx Spartan-6 LX150 FPGAs in its newer version and achieves around 691 billion DES keys/s. However, since the cost of RIVYERA

is around 100,000 Euros, it looks like the cost-performance have stayed the same.<sup>2</sup>

It was noted that the 2013 version of RIVYERA which is known as RIVYERA S3-5000 can check 500 million AES keys in a second. The official webpage<sup>3</sup> of the product later announced that a newer version of RIVYERA can verify 119 billion keys per second for AES which equals to 20.52 times of keys that we get from a single RTX 2070 Super GPU. Thus, 21 RTX 2070 Super GPUs which costs less than \$10,500 can outperform that version of RIVYERA which costs around 100,000 Euros. Thus, our GPU optimizations outperform FPGA-based systems like COPACOBANA and the old versions of RIVYERA for AES key search attacks. However, for fair comparison between modern GPUs and FPGAs it should be noted that:

- 21 GPUs would require other parts like CPUs, motherboards, and power supplies which would at least double the cost.
- RIVYERA results come from an earlier version which uses old FPGAs. Performance of the current RIVYERA models should be significantly better. We could not compare the latest RIVYERA products with GPUs because their performances are not provided by the manufacturer.
- RIVYERA is a commercial product. Thus, aside from its building cost, its price also contains recuperation of the R&D effort, management and sales overhead, logis-

<sup>2</sup>Performance and price values are obtained via personal communication.

<sup>3</sup>www.sciengines.com

tics, quality control and planning, and at least one-year warranty.

- Aside from the price, number of operations per Watt should also be compared.

With the advent of fast GPUs, Biryukov [7] tried to visualize the performance of a GPU-like special purpose hardware created just to apply exhaustive search attack on AES in 2012. They considered the related-key attack on AES-256 [8] introduced at ASIACRYPT 2009 and the time-memory-key trade-off attacks on AES-128 [9] introduced at SAC 2005 on the full AES which require  $2^{99.5}$  and  $2^{80}$  AES operations, respectively. They concluded that an organization that has one trillion US\$ for building and designing a supercomputer based on GPUs like Nvidia GT200b could theoretically perform the related-key attack in a year and the time-memory-key attack in a month.

Since we can perform around  $2^{54}$  AES operations in a month using a single RTX 2070 Super, we can perform the time-memory-key attack in a month with  $2^{26}$  RTX 2070 Super GPUs which would cost 33 billion US\$. Although this is just the cost of the GPUs, a distributed system designed with this many GPUs would cost significantly cheaper than the one trillion US\$ estimate of Biryukov. This result shows how the GPU technology is evolved since 2012. Also note that we assumed that an RTX 2070 Super costs 500 US\$ but recently RTX 3070 is launched which has more cores than RTX 2070 Super and one can get an RTX 3070 for the same price.

## VI. ON SOFT ERRORS

One concern for GPU computation is soft errors. Soft errors are flip of bits in the memory mainly because of environmental reasons. This is why RAMs with error correcting codes (ECC) are used in workstations for scientific computations. However, regular PC RAMs do not have ECC. In the case of GPUs, Nvidia provides Tesla GPUs with ECC for scientific computations and Geforce GPUs without ECC mainly for gaming. A soft error in GPU memory would not be detectable in gaming because a wrong pixel in one frame would be almost impossible for a human eye to detect. In [24], on 60 Geforce 8800 GTS 512 GPUs 72-hour run of their Memtest for CUDA provided no errors on 52 GPUs, less than 10 DRAM errors on 7 GPUs, and 57 DRAM errors on 1 GPU. All of these errors were silent. Namely, there were bit flips but no other visible abnormal behavior. However, in [15], authors conducted a large-scale assessment of GPU error rate on over 50,000 hosts on a distributed computing network by running MemtestG80 and observed no soft errors. In our experiments we also did not observe any soft errors.

Thus, there is no hard evidence for soft errors in GPUs. Yet, for the case of exhaustive search, soft errors happening with a very low probability would not cause any problem. However, a bit flip during encryption in CTR mode would cause a single bit error in decryption which might be hard to detect. Thus, some form of software level ECC can be used for these cases like the one proposed in [21]. Software level ECC should add

just a small performance drop. It would not affect encryption speed because the bottleneck would still be the hard drive, not the GPU.

Overclocking GPUs should be avoided when they are going to be used for encryption since overheating can cause soft errors. Note that soft errors cannot be a reason to avoid Geforce GPUs for encryption because CPU encryption would rely on RAM and regular PC RAMs do not support ECC either. Thus, our optimized codes can be safely used on GPUs with non-ECC memory.

## VII. CONCLUSION

In this work we provided a bank conflict free GPU implementation of AES for the first time and reached 878.6 Gbps throughput for AES-128 encryption. Our optimizations on GPUs are faster than any CPU with hardware level AES instructions AES-NI, are 2.56 times faster than the previous best table based GPU optimizations, and provide better performance-cost ratio compared to legacy FPGA-based cluster architectures like COPACOBANA and RIVYERA. Even on a low-end GPU like MX 250, we obtained 60.0 Gbps throughput for AES-256 which is currently faster than the read/write speeds of solid disks that are commercially available. Thus, transition from AES-128 to AES-256 when using GPUs would provide military grade security with no visible performance loss.

We conclude that with our optimizations GPUs can be used as a cryptographic co-processor for file or full disk encryption to reduce performance loss that comes from CPU encryption. With a single GPU as a co-processor, busy SSL servers would be free from the burden of encryption and use the whole CPU power for other operations. Moreover, these optimizations would help GPUs to practically verify the theoretically obtained cryptanalysis results or their reduced versions in reasonable time.

Finally, many hash functions like SHA-3 Competition candidates Arirang, Cheetah, Echo, Lane, Lesamnta, Lux, Shavite-3, or Vortex use AES rounds or its variants in their internal operations. Moreover, how much these algorithms can benefit from AES-NI was shown in [6]. Some of these AES-based hash functions are used by cryptocurrencies in their proof of work consensus algorithms. Thus, our optimizations can be used to improve GPU based mining for some cryptocurrencies using AES-based hash functions.

## REFERENCES

- [1] A. A. Abdelrahman, M. M. Fouad, H. Dahshan, and A. M. Mousa, "High performance CUDA AES implementation: A quantitative performance analysis approach," in *Proc. Comput. Conf.*, Jul. 2017, pp. 1077–1085.
- [2] K. D. Akdemir, M. Dixon, W. K. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, "Breakthrough AES performance with intel AES new instructions," Intel White Paper, 2010. [Online]. Available: <http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/aes-breakthrough-performance-paper.pdf>
- [3] S. An, Y. Kim, H. Kwon, H. Seo, and S. C. Seo, "Parallel implementations of ARX-based block ciphers on graphic processing units," *Mathematics*, vol. 8, no. 11, p. 1894, Oct. 2020.

- [4] S. An and S. C. Seo, "Highly efficient implementation of block ciphers on graphic processing units for massively large data," *Appl. Sci.*, vol. 10, no. 11, p. 3711, May 2020.
- [5] S. W. An and S. C. Seo, "Study on optimizing block ciphers (AES, CHAM) on graphic processing units," in *Proc. IEEE Int. Conf. Consum. Electron. Asia (ICCE-Asia)*, Nov. 2020, pp. 1–4.
- [6] R. Benadjila, O. Billet, S. Gueron, and M. J. B. Robshaw, "The intel AES instructions set and the SHA-3 candidates," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.* in Lecture Notes in Computer Science, M. Matsui, Ed. Tokyo, Japan: Springer, Dec. 2009, pp. 162–178.
- [7] A. Biryukov and J. Großschädl, "Cryptanalysis of the full AES using GPU-like special-purpose hardware," *Fundamenta Informaticae*, vol. 114, nos. 3–4, pp. 221–237, 2012.
- [8] A. Biryukov and D. Khovratovich, "Related-key cryptanalysis of the full AES-192 and AES-256," in *Proc. 15th Int. Conf. Theory Appl. Cryptology Inf. Secur.* in Lecture Notes in Computer Science, vol. 5912, M. Matsui, Ed. Tokyo, Japan: Springer, 2009, pp. 1–18.
- [9] A. Biryukov, S. Mukhopadhyay, and P. Sarkar, "Improved time-memory trade-offs with multiple data," in *Proc. Int. Workshop Sel. Areas Cryptogr. (SAC)* in Lecture Notes in Computer Science, vol. 3897, B. Preneel and S. E. Tavares, Eds. Kingston, ON, Canada: Springer, Aug. 2005, pp. 110–127.
- [10] J. Daemen and V. Rijmen, "The design Rijndael: AES—The advanced encryption standard," in *Information Security and Cryptography*. Berlin, Germany: Springer-Verlag, 2002. [Online]. Available: <https://www.springer.com/gp/book/9783540425809>
- [11] N. Drucker, S. Gueron, and V. Krasnov, "Making AES great again: The forthcoming vectorized AES instruction," in *Proc. 16th Int. Conf. Inf. Technol.-New Generat. (ITNG)*, S. Latifi, Ed. Cham, Switzerland: Springer, 2019, pp. 37–41.
- [12] S. Gueron, "Intel's new AES instructions for enhanced performance and security," in *Proc. 16th Int. Workshop Fast Softw. Encryption (FSE)* in Lecture Notes in Computer Science, vol. 5665, O. Dunkelman, Ed. Leuven, Belgium: Springer, Feb. 2009, pp. 51–66.
- [13] S. Gueron, "Intel advanced encryption standard (AES) new instructions," Intel White Paper 323641-001 Revision 3.0, 2010. [Online]. Available: <https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf>
- [14] T. Güneysu, T. Kasper, M. Novotný, C. Paar, L. Wienbrandt, and R. Zimmermann, *High-Performance Cryptanalysis on RIVYERA and COPACOBANA Computing Systems*. New York, NY, USA: Springer, 2013, pp. 335–366.
- [15] I. S. Haque and V. S. Pande, "Hard data on soft errors: A large-scale assessment of real-world error rates in GPGPU," in *Proc. 10th IEEE/ACM Int. Conf. Cluster, Cloud Grid Comput.*, May 2010, pp. 691–696.
- [16] K. Iwai, N. Nishikawa, and T. Kurokawa, "Acceleration of AES encryption on CUDA GPU," *Int. J. Netw. Comput.*, vol. 2, no. 1, pp. 131–145, 2012.
- [17] J. Jean. (2016). *TikZ for Cryptographers*. [Online]. Available: <https://www.iacr.org/authors/tikz/>
- [18] S. S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler, "Breaking ciphers with COPACOBANA—A cost-optimized parallel code breaker," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* in Lecture Notes in Computer Science, vol. 4249, L. Goubin and M. Matsui, Eds. Yokohama, Japan: Springer, Oct. 2006, pp. 101–118.
- [19] R. K. Lim, L. R. Petzold, and Ç. K. Koç, "Bitsliced high-performance AES-ECB on GPUs," in *The New Codebreakers* (Lecture Notes in Computer Science), vol. 9100, P. Ryan, D. Naccache, and J. J. Quisquater, Eds. Berlin, Germany: Springer, 2016, doi: [10.1007/978-3-662-49301-4\\_8](https://doi.org/10.1007/978-3-662-49301-4_8).
- [20] S. A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," in *Proc. IEEE Int. Conf. Signal Process. Commun.*, Nov. 2007, pp. 65–68.
- [21] N. Maruyama, A. Nukada, and S. Matsuoka, "Software-based ECC for GPUs," White Paper, Jul. 2009. [Online]. Available: [https://www.researchgate.net/publication/255641637\\_Software-Based\\_ECC\\_for\\_GPUs](https://www.researchgate.net/publication/255641637_Software-Based_ECC_for_GPUs)
- [22] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 72–86, Jan. 2017.
- [23] N. Nishikawa, H. Amano, and K. Iwai, "Implementation of bitsliced AES encryption on CUDA-enabled GPU," in *Proc. Int. Conf. Netw. System Secur.* in Lecture Notes in Computer Science, vol. 10394, Z. Yan, R. Molva, W. Mazurczyk, and R. Kantola, Eds. Helsinki, Finland: Springer, Aug. 2017, pp. 273–287.
- [24] G. Shi, J. Enos, M. Showerman, and V. Kindratenko, "On testing GPU memory for hard and soft errors," in *Proc. Symp. Appl. Accel. High-Perform. Comput.*, 2009, pp. 1–3.



**CIHANGIR TEZCAN** received the B.Sc. degree in mathematics and the M.Sc. and Ph.D. degrees in cryptography from Middle East Technical University, in 2007, 2009, and 2014, respectively. He is currently the Head of the Department of Cyber Security and the Director of the Cyber Defense and Security Research Center, Middle East Technical University. Before joining METU, he was a Teaching Assistant at the École Polytechnique Fédérale de Lausanne and a Postdoctoral Researcher with Ruhr-Universität Bochum.

• • •