Received: 20 February 2021 | Revised: 8 June 2021 | Accepted: 28 June 2021

DOI: 10.1002/cpe.6507

SPECIAL ISSUE PAPER

WILEY

GPU accelerated 3DES encryption

Kaan Furkan Altınok[1] | Afşin Peker[1] | Cihangir Tezcan[2] | Alptekin Temizel[2]

[1]Electrical and Electronics Engineering, Middle East Technical University, Ankara, Turkey

[2]Graduate School of Informatics, Middle East Technical University, Ankara, Turkey

Correspondence
Alptekin Temizel, Graduate School of Informatics, Middle East Technical University, Ankara, Turkey.
Email: atemizel@metu.edu.tr

Abstract

Triple DES (3DES) is a NIST and ISO/IEC standard block cipher that is also used in some web browsers and several electronic payment applications. We propose an optimized bit-level parallelization of 3DES for GPU accelerated encryption to allow processing high volumes of data. Since the block size of 3DES is 64 bits, our approach considers a kernel block as a 64-bit 3DES block. Each kernel block performs XOR, permutation, and S-box operations of this cipher in parallel and memory accesses are optimized by the use of constant and shared memory. Although table based and bitsliced implementations of block ciphers on GPUs outperform naive implementations, their performance vary significantly on different GPU models and architectures. Lack of publicly available source codes prohibit a fair comparison of the performance results for different implementations. In this work, we provide performance results on various GPU models and make our implementation publicly available for reproducibility and further comparisons. When compared against the baseline multi-threaded CPU implementation, our optimization achieves an average of 15.95× speed-up when encrypting large files using an RTX 2070 Super GPU. Moreover, when modified into a key search attack, more than 94.4 million 3DES key searches per second can be conducted on an RTX 2070 Super GPU.

KEYWORDS

3DES, CUDA, DES, encryption, GPU, parallelization

1 | INTRODUCTION

As digital transactions and communications become more prevalent, sensitive information is at greater risk of being compromised. Encryption algorithms are used to protect such confidential information by encrypting the data that travel through an insecure channel. Since increasing number of people are using such digital services, the amount of data that has to be encrypted and protected is increasing significantly. Hence, even though the most important property of an encryption algorithm is considered to be its toughness, encryption speed is also becoming important to cope with such high volumes. While using custom hardware is helpful in that respect, it is a costly solution and not feasible in many use-cases. On the other hand, GPUs are a viable alternative due to their prevalence, lower costs and they can be used to exploit the parallel nature of encryption algorithms. Since most algorithms work on plaintext blocks, multiple blocks can be encrypted in parallel using GPUs, drastically increasing the amount of data that can be encrypted at a unit of time.

Although GPU cores are not as fast and powerful as CPU cores, GPUs outperform CPUs in parallel algorithms for GPUs are equipped high-number of cores. Thus, it is commonplace to obtain 2–3× speed-ups encrypting plaintext blocks in parallel on a GPU. In addition, using table based or bitsliced implementations instead of the naive approach can provide further speed-ups on GPU architectures.

In table based implementations, input of a round function of a block cipher is partitioned into small parts and outputs of every possible input are pre-computed and stored in tables called T-tables. GPUs have different memory types such as global, constant, and shared memory. The speed of

table based implementations depends on which memory type is used for storing the tables. Shared memory is the fastest type of GPU memory and storing T-tables in the shared memory provides the fastest encryption on GPUs.[1] However, bank-conflicts may negatively affect shared memory performance. GPU kernel calls are grouped in warps having 32-threads and modern GPUs have 32 shared memory banks, attempts to use of the same shard memory bank at the same time (bank conflict) results in serialization.

Although many block ciphers have bit-level operations, programming languages are byte-oriented. Therefore, bit operations require extra instructions in the code. These extra operations can be prevented in bitsliced implementations. In this technique, bits of a block of the cipher are stored in different variables so that operations on bits can be easily performed. Bitslicing can be performed in two different ways on GPUs: thread-level parallelization and variable-level parallelization. GPU kernels are run for blocks of threads. In thread-level parallelization, each bit of a block is assigned to a thread in a GPU block. Unlike the table based implementations, in this approach the threads need to communicate with each other. Thus, the optimizations in this technique focus on the communication of the threads and the memory type usage.

In the variable-level bitslicing method, each bit of the block is stored in a different variable. If one uses n-bit variables, in this approach n plaintext blocks can be encrypted in parallel. However, this approach requires use of at least b registers for a block cipher of b-bit block size and using more than 64 registers in modern GPUs prevent full occupancy of the GPU, in effect, resulting in degrading performance.

Many cryptographic algorithms are optimized for GPUs in the literature. For instance, breakthrough performance is reported by An et al.[2] for some lightweight Add-Rotate-XOR based ciphers like HIGHT, LEA, and CHAM. Encryption at terabit per second level for LEA, SIMON, Chaskey, SIMECK, and SPECK on GPUs is reported by Lee et al.[3]

Many CUDA optimizations are provided in the literature for the Advanced Encryption Standard (AES),[4] which is arguably the most used encryption algorithm. One of the most recent of these by An and Seo[5] provided highly efficient CUDA implementation of AES on GPUs to cope with massive amounts of data using an optimized table based implementation. They achieved 310.0 Gbps throughput for AES-128 on an RTX 2070 Super. The main performance bottleneck of this implementation is the shared memory bank conflicts. Nishikawa et al.[6] achieved 605.9 Gbps throughput for AES-128 on a Tesla P100 using an optimized bitsliced implementation. Although bitsliced implementation of Nishikawa et al.[6] outperformed previous table based optimizations of AES, recently Tezcan[7] achieved 878.6 Gbps throughput for AES-128 on an RTX 2070 Super by eliminating all shared memory bank conflicts in an optimized table based implementation. The main optimization of that work is the duplication of one out of four T-tables of AES 32 times for shared each memory bank and other three T-tables are obtained from the stored one by permutation. However, this optimization is not possible for most of the other block ciphers since T-tables are generally not permutations of each other in a block cipher round function. Thus, multiple copies of T-tables of a block cipher generally would not fit into the shared memory which is 64 KB for modern GPUs.

Although there are many GPU optimizations for AES, there are a limited number of previous studies on the parallelization of its predecessor Data Encryption Standard (DES) or its triple application Triple Data Encryption Standard (3DES), on CPUs and GPUs. Swierczewski[8] implemented the 3DES algorithm on GPUs using CUDA with considerable improvements over the CPU versions. Guler et al.[9] implemented the original DES algorithm using CUDA for lightweight systems. They also observed a significant performance increase over the conventional implementation. Fadhil[10] implemented 3DES using CUDA for the purpose of watermarking images with concealed information. His results show that the parallel implementation is about 6× faster than its sequential counterpart. Finally, Beletskyy and Burak's work[11] parallelizes the 3DES algorithm using OpenMP on CPUs. They analyze the data dependencies during the operations and parallelize the loops accordingly, which in turn results in about 1.95× speed-up.

Most of these implementations parallelize the operation at a text-block scale, with the exception of Beletskyy and Burak's work.[11] In our proposed approach, the parallelization is done at bit-level. Since most of the operations during the encryption/decryption are shuffling and shifting the bits around, these operations can be done for all bits concurrently, considerably improving the performance of the algorithm.

In this article, we propose a parallel GPU implementation of the Triple-DES (3DES) algorithm using CUDA. Unlike the current solutions which parallelize the operations on a block by block basis, the operations are done on a bit-by-bit basis. Namely, we provide optimizations for 3DES using the thread-level bitsliced implementation technique, instead of the common tables based implementations. In our earlier work,[12] we performed our experiments on 3DES using the electronic codebook (ECB) mode of operation to have a fair comparison between CPUs and GPUs. However, ECB mode does not provide security because it maps the same plaintext blocks to the same ciphertext blocks. Since ECB mode is never used for cryptographic purposes in practice, in this work we also provide 3DES encryption performance using Counter (CTR) mode of operation. CTR has comparatively better performance than ECB on GPUs because CTR does not require the plaintext to be copied to GPU memory since counters, instead of plaintext blocks are encrypted in this mode. Finally, we convert our key generation and encryption kernels into a key search kernel and obtain performance results for applying an exhaustive search attack on DES or 3DES for different GPUs.

Although our results outperform previous GPU optimizations of 3DES, a fair comparison with the ones in the literature is improbable due to unavailability of publicly available source codes and results being reported for a single GPU model. Lack of publicly available source codes prevents the reproducibility of the experiments and the comparison of new GPUs with the earlier models used in the original research. Moreover, only having the results for a single GPU model prohibits arriving at robust conclusions on the generalizability of the results for other GPUs. Although GPU manufacturers provide some information about the differences between the architectures, most of the differences are not known to users and their effect on the performance may not be the same under all running conditions. For instance, Mei and Chu[13] observed that Maxwell has superiority against Fermi and Kepler in performance under shared memory bank conflicts. This shows that the performance might change drastically on different

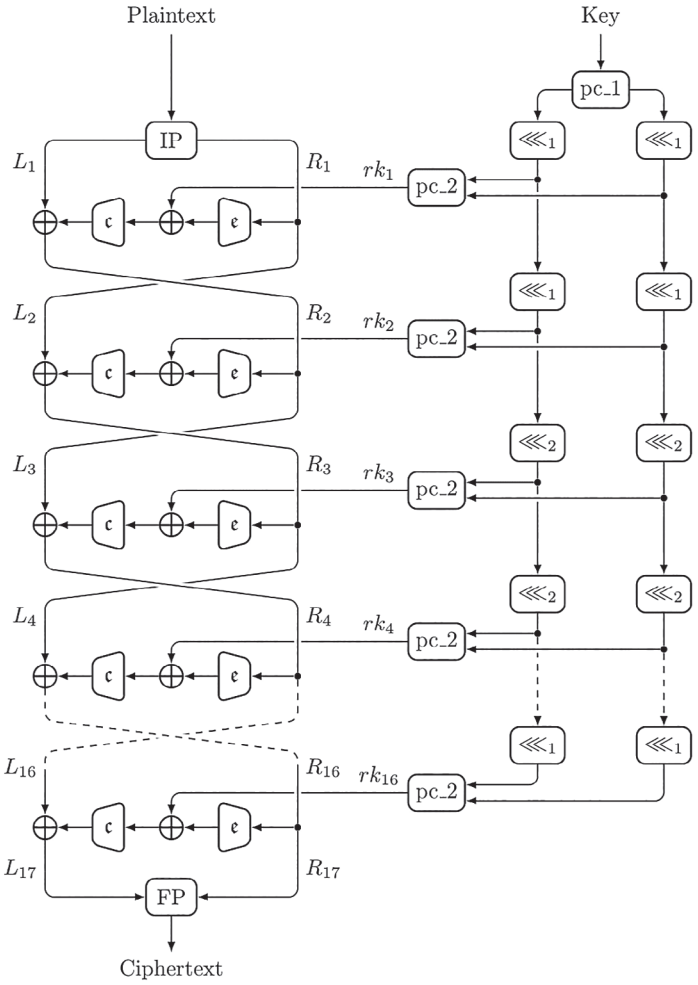

FIGURE 1 Data Encryption Standard (DES) and its key schedule.[15] Three successive repetition of this figure provides Triple Data Encryption Standard (3DES)

architectures since different implementation techniques have different bottlenecks. For instance, the bottleneck in table based implementations is the shared memory bank conflicts. Thus, the results of Mei and Chu[13] shows that GPUs with Maxwell architecture outperform similar GPUs with Fermi and Kepler. Hence it can be said that it is not possible to reliably predict the performance of these implementations on future GPU architectures.

For these reasons, we provide our results on a variety of different desktop and mobile Nvidia GPUs with Maxwell, Pascal, Turing, and Ampere architectures with different compute capabilities (i.e., compute capabilities 5.0, 5.2, 6.1, 7.5, and 7.6). Moreover, we make our source codes publicly available*.

2 | TRIPLE DES

DES (Data Encryption Standard) is a block cipher that is mainly used for the encryption of digital data. It was designed by IBM in the 1970s and became a worldwide standard until the 1990s. Although DES is almost 50 years old, it still appears in contemporary work and applications.[14] For this algorithm, the block size is 64 bits, key length is 56 bits, and there are 16 rounds using the 16 48-bit subkeys generated from the original 56-bit key. The algorithm starts with an initial permutation of the 64-bit plaintext. Then, the data is split into two halves, and for 16 rounds, they undergo certain operations that scramble the data. After 16 rounds, a final permutation is performed, and the ciphertext is obtained. The general structure of the algorithm is shown in Figure 1. The decryption of the ciphertext is nearly identical; only the round key order must be reversed.

*The implementation is available at https://github.com/kaanfurkan35/3DES_GPU

However, due to its short key length, DES was vulnerable to brute force attacks and was subsequently removed from the standards. For this reason, it is also referred to as DEA (data encryption algorithm) for clarity. Efforts to alleviate this vulnerability gave rise to the 3DES, also known as 3DEA or TDEA (triple data encryption algorithm). In 3DES, the DES encryption-decryption-encryption is applied to the plaintext sequentially with different keys, increasing the key length to 168. While, a trivial meet-in-the-middle attack reduces the effective security to 112 bits, with the currently available technology, 112-bit security is considered to be secure against brute force attacks. In Section 4.3 we provide brute force/exhaustive search attack performance of our optimizations on GPUs and show that an exhaustive search attack on 3DES is not feasible using current GPU technology.

3DES is an ISO/IEC standard[16] and is a NIST standard until 2024.[17] Although the ISO/IEC standard that includes 3DES was published in 2010, it was last reviewed and confirmed in 2020. We analyzed many credit cards in Turkey that support contactless payments and observed that more than half of them use Java Card OpenPlatform (JCOP) operating system which uses 3DES. During our analysis we observed that popular NFC applications like NFC Tools mistakes these smart cards with Mifare Classic because they have Mifare Classic emulation. However, Mifare Classic smart cards cannot be used for credit card payments because the CRYPTO1 stream cipher inside Mifare Classic cards can be broken in 10 minutes via online attacks[18] and GPUs can capture secret keys of these cards in just a few hours via offline attacks.[19] Devices like Proxmark3 correctly identifies these credit cards as JCOP31 or JCOP41.

In the following sections, steps of the 3DES key generation and encryption/decryption algorithms will be presented. The constant arrays mentioned in the following sections can be found in Appendix A.1.

2.1 | Triple-DES key generation

Key generation algorithm in DES, which is also known as key schedule algorithm, produces 16 48-bit round keys from a single 56-bit secret key. 3DES key schedule algorithm uses the key generation algorithm of DES three times in parallel. Hence it generates 48 48-bit round keys from three 56-bit secret keys. Key generation of DES consists of two permutations pc_1 and pc_2 and left rotations of 1 or 2 bits as it is depicted in the right side of Figure 1.

Although DES algorithm uses a 64-bit secret key, rightmost bit of each byte is used as a parity bit. Thus, effective secret key of DES is 56 bits and these parity bits are removed by the application of pc_1 permutation. It has to be noted that 57th bit of the original key will be the 1st bit of the permuted key and so on. The same logic applies for every permutation function (also in encryption).

In order to generate a 48-bit round key, the 56-bit key is divided into two arrays, each consisting 28-bit part of it. Each array is circularly left-shifted using the εshift_keysε array at each of the 16 rounds consecutively. A combination of the two arrays, (56 bits), is then permuted to 48-bit to finalize the ith round key rk_i using the pc_2 array. This process is repeated 16 times to generate all round keys of DES.

2.2 | Triple DES encryption/decryption

An initial permutation IP is applied to DES/3DES input in order to make the bit ordering suitable for the DES chip. Similarly, at the end of the block encryption a final permutation FP, which is the inverse of IP, is applied. However, IP and FP do not provide any security and have no cryptographic importance.

DES is a Feistel block cipher and therefore the input is divided into left and right halves. The right half becomes left in the following round. A round function is applied to the right half and the output is XORed with the left part. The result becomes the right half in the following round. This round function is applied 16 times in DES and 48 times in 3DES.

The round function of DES/3DES consists of expansion, key addition, substitution, and permutation layers and it is provided in Figure 2.

These layers can be described as follows:

1. Expansion layer: 16 bits of the 32-bit input is duplicated via the exp_d array in order to obtain 48 bits so that it matches the size of round keys.
2. Key addition layer: Expanded 48 bits are XORed with the 48-bit ith round key rk_i.
3. Substitution layer: Eight 6×4 S-boxes are applied in parallel to 48-bit input to provide confusion. Each 6-bit block from the 48-bit array is supplied to S-boxes to reduce it to 4-bits, which finally makes a 32-bit array. Operation is as follows; Six bits are represented as, for example, 010111. Middle 4 bits are 1011, which is 11 in decimal. First and last bits are 01, which makes 2 in decimal. S-boxes are accessed with the block number, then row 2 is used, and column 11 is used to access it. The number there will be the 4-bit char array for our algorithm. There are a total of 8 S-boxes for each 6-bit block as shown in the Appendix.
4. Permutation layer: Places of bits of the 32-bit value are shuffled according to *per* permutation as shown in the Appendix.

Decryption operation also consists of these steps with the only difference that the round key orders are reversed. That is, the 1st round key for encryption becomes the 16th round key for decryption and so on.

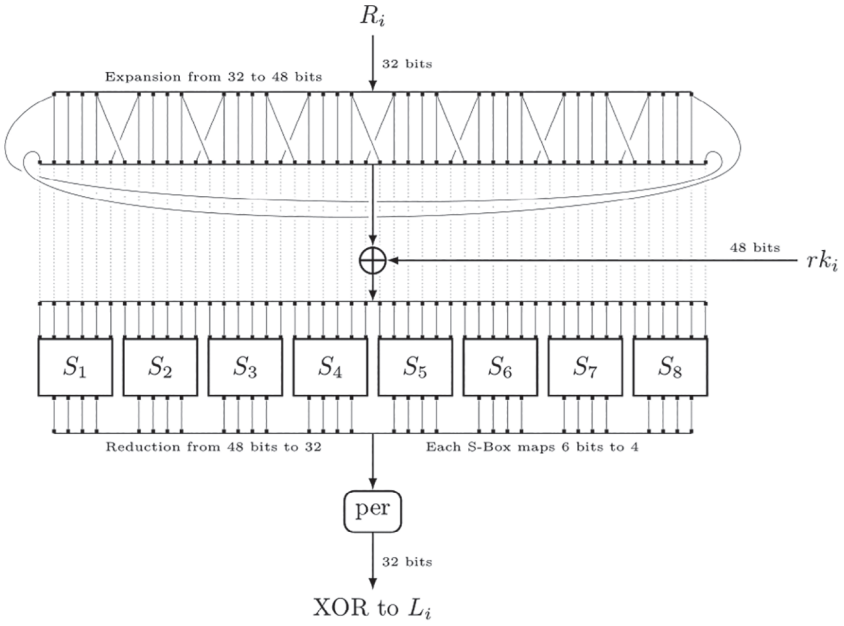

FIGURE 2 A single round function of DES and 3DES consisting of expansion, key addition, substitution, and permutation layers[15]

3DES encryption and decryption are implemented as follows, where *D* represents decryption operation, *E* represents encryption operation and the subscripts *Ki* represent the *i*th 56-bit key used for these operation:

$$ciphertext = E_{K3}(D_{K2}(E_{K1}(plaintext)))$$
$$plaintext = D_{K1}(E_{K2}(D_{K3}(ciphertext)))$$

This method also allows backward compatibility: the system behaves like the original DES when all three base keys are the same, that is, $K1 = K2 = K3$.

3 | THE PROPOSED APPROACH

In this section, we introduce the proposed bit-wise parallelization scheme and explain the memory design for optimized access. Two separate kernels, described in separate sections below, were implemented for key generation and encryption/decryption. For performance evaluation, in this work we used a number of GPUs with different specifications and architecture and their specifications are provided in Table 1.

TABLE 1 The specifications of the GPUs used in the experiments

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
GTX 1660 Super Gaming X	1408	1830 MHz	7.5	Turing
RTX 2060 Super	2176	1650 MHz	7.5	Turing
RTX 3060 Laptop	3840	1702 MHz	8.6	Ampere
RTX 2070 Super	2560	1770 MHz	7.5	Turing

Note: CC refers to Compute Capability of the device. Note that clock rates might vary depending on the GPU manufacturer.

3.1 | Key generation kernel

Key generation kernel (Figure 1) kernel runs with 3 blocks for each base key, and each block consists of 56 threads. Even though the keys are 64 bits, in the first step, they are permuted to 56 bits and 56 threads, corresponding to each bit that are used. The kernel will take an array of 3 base keys (64 bits) as input and will output an array of 48 round keys (48 bits), 16 for each base key.

The algorithm is parallelized as follows:

1. Permutation is performed bitwise; each thread reads an element from pc_1 and uses that value to index the base key bits accordingly. The read bit is written to the permuted 56-bit array for the next steps.
2. A thread in each block splits the permuted key into two.
3. The final step of the algorithm cannot be fully parallelized as each round has to use the result of the previous round. So, each block runs a for loop for 16 times for each subkey.
4. In the for loop, shifts are handled by two threads in each block, for left and right half of the 56-bit permuted key. Then, another permutation is performed using pc_2, in the same fashion as in the first step. The output of this permutation is one of the subkeys, it is written to the output, and this operation is repeated 16 times.

3.2 | Encryption/decryption kernel

Encryption/decryption kernel (Figure 1) is run subsequent to the key generation kernel. This kernel has as many blocks as the plaintext, and each block consists of 64 threads since the blocks in DES are 64 bits. In other words, each kernel block is responsible for a block of plaintext, and each thread is responsible for a bit in the plaintext block. It will take the plaintext, subkeys, and mode as the input. Depending on the mode (encryption or decryption), relevant subkeys are used and the output is the encrypted/decrypted text.

Parallelization of the algorithm is as follows:

1. The first permutation using the initial_perm array is performed in parallel, like in the key generation kernel.
2. A thread in each block splits the permuted text into two.
3. This part of the algorithm cannot be parallelized fully, as in the case of key generation. So, the following operations are executed in a loop for 16 rounds. In each iteration of the loop, the first 48 threads in each block expands the right half to 48 bits. The expanded half is XORed with the round key, each bit is processed in parallel.
4. Next, 6-bit groups of the expanded and XORed right half use the S-boxes in parallel. There are eight such groups, and this operation is handled by four threads for each group. One of these four threads accesses the S-box and gets the substitution value, and four threads write each bit of this value to the respective indices of the substituted array.
5. This array is permuted in parallel using the per array.
6. The output of this permutation is XORed with the left half, each bit in parallel.
7. Finally, the left half and the output of the previous step are swapped. These operations are repeated 16 times.
8. After 16 rounds, two 32-bit arrays are combined to get a 64-bit array.
9. The final_perm array is used for a final permutation of this 64-bit array in parallel, and the output is the encrypted/decrypted text block, depending on the mode.

This kernel needs to be called three times consecutively for 3DES. For decryption, the same kernel is used by first reversing the supplied key order on the CPU before the kernel launch.

3.3 | Memory management

GPUs have different memory spaces like global, shared, constant, texture, or register memory. They all have different size constraints and access speeds. Register and shared memory provide the fastest performance and they actually reside on the GPU chip. Table based implementations of block ciphers keep the T-tables in the shared memory for fast access but generally suffer from shared memory bank conflicts. On the other hand, variable-level bitsliced implementations of ciphers keep the variables in the registers but they generally suffer from low-occupancy problems due to the limited number of available registers.

In our thread-level bitsliced implementation, a text file is fed into the main function, which includes a plaintext in any size consisting of 64-bit blocks, three different 64-bit keys as characters. Since CPU and GPU registers cannot fetch at bit granularity, the smallest available data type (char) is used to represent bits. CPU sends the plaintext and keys as char arrays to the global memory of the GPU.

Constant tables like permutation tables, expansions tables, S-box tables are stored in the read-only cache. Constant memory is not suitable for these tables since every thread accesses a different index of those tables. On the other hand, shift table is stored in constant memory as every thread accesses its same index at the same time. Plaintext is divided into 64-bit blocks for each kernel block, and in the kernel, plaintext blocks are copied to the shared memory for processing.

From three different keys, 48 round keys are used in each 3DES round. Each thread accesses a different bit of the key, so again, the read-only cache is the preferred memory type to store the keys.

4 | EXPERIMENTAL RESULTS

Other than encryption, fast implementation of a block cipher has other use cases like verifying theoretically obtained cryptanalysis results or checking the key length security of the cipher by performing an exhaustive search attack. Use cases like encryption uses a single secret key. In these scenarios, performing the key schedule algorithm on the CPU instead of the GPU provides better performance. However, a different key is used in an exhaustive search or similar scenarios. Thus, having a key schedule GPU kernel is necessary for these cases.

We obtained 3DES electronic codebook encryption (ECB) performance on two GPUs[12] and we provide them in Section 4.1. In order to show that our optimizations were not focused on specific GPUs or architectures, we performed the same experiments on many different desktop and mobile GPUs and provide the results in the same section. Moreover, we modified our kernels for counter (CTR) mode of operation since ECB is not a cryptographically secure mode of operation and CTR encryption has the advantage of not requiring the transfer of plaintext to GPU memory. We provide those results in Section 4.2. Finally, we modified our kernels for exhaustive search and provide these results in Section 4.3.

4.1 | Electronic codebook (ECB) mode experiments

To evaluate the performance of the proposed algorithm[†], we used text files with varying sizes as input and measured the encryption time on two different GPUs and CPUs: GPU-1: NVIDIA GeForce RTX 2060 Super, GPU-2: NVIDIA GeForce GTX 1660 Super Gaming X, CPU-1: AMD Ryzen 5 3600 and CPU-2: Intel i7 9700K. In all cases, encryption function was used in electronic codebook (ECB) mode, that is, each block is encrypted independently from each other. The block sizes were increased exponentially, starting from 2^2 to 2^{17}.

For GPU measurements, NSight Profiler was used. Encryption kernel average running time was multiplied by three since 3DES consists of 3 kernel launches. For CPU measurements, the standard CPU implementation was partly parallelized using OpenMP, for a fair comparison. More specifically, the `omp parallel for` pragma was used on the for loop that iterates over each block of plaintext. Hence, several blocks get encrypted in parallel.

Table 2 shows the encryption times for different block sizes for all devices. It can be seen that the processing time increases linearly with the increasing block size. While GPUs do not offer any advantage for smaller block sizes, the speed-up advantage becomes more pronounced with larger block sizes. This effect is also shown in Figure 3.

Table 3 shows the speed-up of each GPU with respect to each CPU for different block sizes and average and Table 4 shows average speed-ups. The results show that the proposed GPU implementation provides a speed-up of 9.31 to 10.70 × on GPU-1 and 7.01 to 8.03 × on GPU-2 for files bigger than 8 KB. The best speed-up rate of 20.25 × is obtained for 64 KB files.

These results show that our GPU implementation speeds up the 3DES algorithm considerably, compared to the conventional CPU implementation. The expected speed-up in real-life cases is expected to be higher than these averages when bigger file sizes are used.

An analysis with NSight Profiler shows that compute and memory are well-balanced in encryption kernel. Considering memory workload analysis, L1 Cache Hit is 98.69%, which indicates that it was effective to put constant arrays to the read-only cache. Maximum bandwidth utilization, reported as 71.16%, indicates that the implementation was effectively using the bandwidth and there was no bottleneck. The achieved occupancy reported by the profiler, 98.25%, and achieved active warps per SM, 31.44, were close to these theoretical upper limits of 100% and 32 respectively.

In our earlier work,[12] we performed our experiments on 3DES using only two GPU models, in this work we extend the results to all 6 GPUs listed in Table 1. Table 5 summarizes the speed-up results for 1 MB ECB encryption using all these 6 GPUs.

Beletskyy and Burak[11] achieved 19.6 Mbps on a two core CPU. When memory copy and other operations are not included, Swierczewski[8] achieved 180.60 Mbps on Tesla C2050. Guler et al.[9] achieved 2.84 Mbps but they did not specify the GPU used and it is not clear whether the result includes memory copy operations. Although Fadhil[10] reported a 6× speed-up against CPU implementations, their results show that it takes 40.32 seconds to encrypt a plaintext of 4069 bits, which corresponds to 100.92 bits per second. This suggests that their results are likely to include image processing times which adversely affect their performance. As it can be calculated from Table 5, we achieved 972.64 Mbps throughput on an RTX 2070 Super. Although our results outperform the previous optimizations, for a fair comparison one needs the source codes to perform the

[†]The implementation is available at https://github.com/kaanfurkan35/3DES_GPU

TABLE 2 Encryption time (ms) for different processors where GPU-1 is NVIDIA GeForce RTX 2060 Super, GPU-2 is NVIDIA GeForce GTX 1660 Super Gaming X, CPU-1 is AMD Ryzen 5 3600, and CPU-2 is Intel i7 9700K

Block size	File size	GPU-1	GPU-2	CPU-1	CPU-2
4	32B	0.032	0.029	0.012	0.017
8	64B	0.033	0.030	0.014	0.016
16	128B	0.033	0.030	0.034	0.028
32	256B	0.032	0.030	0.055	0.050
64	512B	0.033	0.031	0.107	0.096
128	1KB	0.032	0.031	0.209	0.184
256	2KB	0.035	0.037	0.406	0.367
512	4KB	0.050	0.064	0.718	0.653
1024	8KB	0.089	0.120	1.387	1.237
2048	16KB	0.161	0.225	2.401	2.463
4096	32KB	0.305	0.438	5.498	4.977
8192	64KB	0.592	0.862	11.986	10.144
16384	128KB	1.184	1.725	20.354	16.701
32768	256KB	2.324	3.396	35.319	29.843
65536	512KB	4.511	6.261	67.626	52.587
131072	1MB	8.399	12.441	131.577	100.715

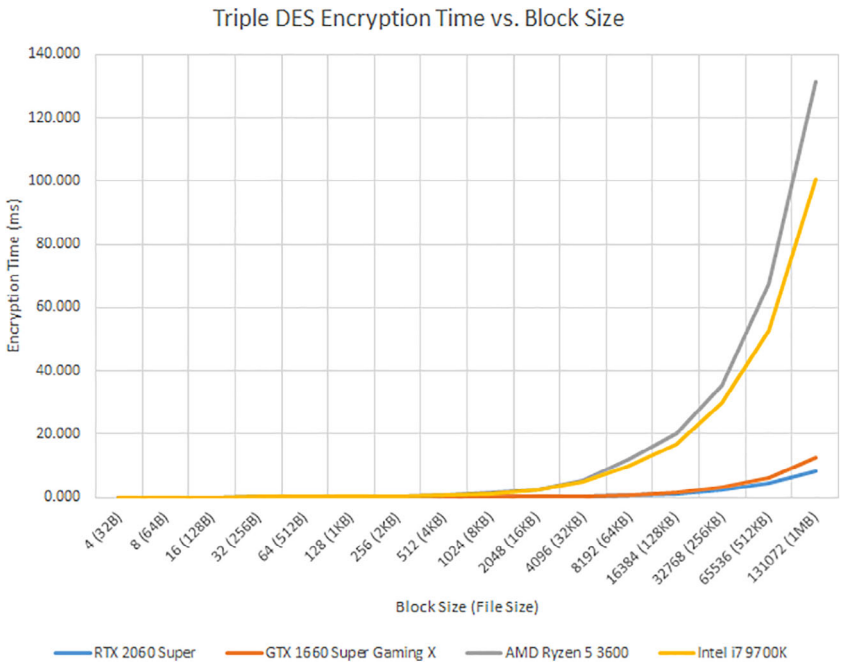

FIGURE 3 3DES encryption time versus block size graph for different processors

experiments on the same or similar GPUs. However, this was not possible since previous results were reported on older hardware setups and it was not possible to access their source codes.

4.2 | Counter (CTR) mode experiments

Although ECB mode of operation is good for benchmarking and provides a fair ground for comparing CPU and GPU performance, it is not indicative of real-life performance. Because, it cannot be used for encryption in practice as the same plaintexts blocks are encrypted to the same ciphertext

TABLE 3 GPU versus CPU Speed-ups where GPU-1 is NVIDIA GeForce RTX 2060 Super, GPU-2 is NVIDIA GeForce GTX 1660 Super Gaming X, CPU-1 is AMD Ryzen 5 3600, and CPU-2 is Intel i7 9700K

Block size	File size	GPU-1 vs CPU-1	GPU-2 vs CPU-1	GPU-1 vs CPU-2	GPU-2 vs CPU-2
4	(32B)	0.38	0.41	0.53	0.58
8	(64B)	0.42	0.47	0.49	0.53
16	(128B)	1.03	1.13	0.85	0.93
32	(256B)	1.72	1.83	1.56	1.67
64	(512B)	3.24	3.45	2.91	3.10
128	(1KB)	6.53	6.74	5.75	5.94
256	(2KB)	11.60	10.97	10.48	9.92
512	(4KB)	14.36	11.22	13.06	10.20
1024	(8KB)	15.59	11.56	13.89	10.30
2048	(16KB)	14.91	10.67	15.29	10.94
4096	(32KB)	18.03	12.55	16.31	11.36
8192	(64KB)	20.25	13.91	17.13	11.76
16384	(128KB)	17.20	11.80	14.10	9.68
32768	(256KB)	15.20	10.40	12.84	8.79
65536	(512KB)	15.00	10.80	11.65	8.40
131072	(1MB)	15.67	10.58	12.00	8.10

TABLE 4 Average speed-ups of the GPUs with respect to CPUs where CPU-1 is AMD Ryzen 5 3600, and CPU-2 is Intel i7 9700K

	CPU-1	CPU-2
RTX 2060 Super	10.70	9.31
GTX 1660 Super Gaming X	8.03	7.01

TABLE 5 Speed-ups of the GPUs for 1 MB file with respect to CPUs where CPU-1 is AMD Ryzen 5 3600, and CPU-2 is Intel i7 9700K

	CPU-1	CPU-2
GTX 860M	1.46	1.12
MX 250	1.88	1.44
GTX 970	5.72	4.38
GTX 1660 Super Gaming X	10.58	8.10
RTX 3060 Laptop	14.90	11.40
RTX 2060 Super	15.67	11.99
RTX 2070 Super	15.95	12.21

blocks in the ECB mode. Thus, the redundancy in the plaintext causes a redundancy in the ciphertext which would allow an attacker to recover some or all of the plaintext. This weakness of the ECB mode can be illustrated best with encryption of bitmap images. Figure 4 shows a bitmap image together with its encrypted version via Advanced Encryption Standard (AES)[4] using ECB and Counter (CTR) mode of operations. Plaintext can easily be guessed just by looking at the ECB encryption, implying that even AES does not provide security when the mode of operation is ECB. Thus, the weakness is not in the block cipher but rather, in the mode of operation and ECB is not recommended for cryptographic purposes.

Aside from ECB and CTR, NIST also recommends[20] the cipher block chaining (CBC), the output feedback, and the cipher feedback (CFB) modes. However, these modes are not parallelizable (except CBC decryption) and for this reason we do not focus on them in this work.

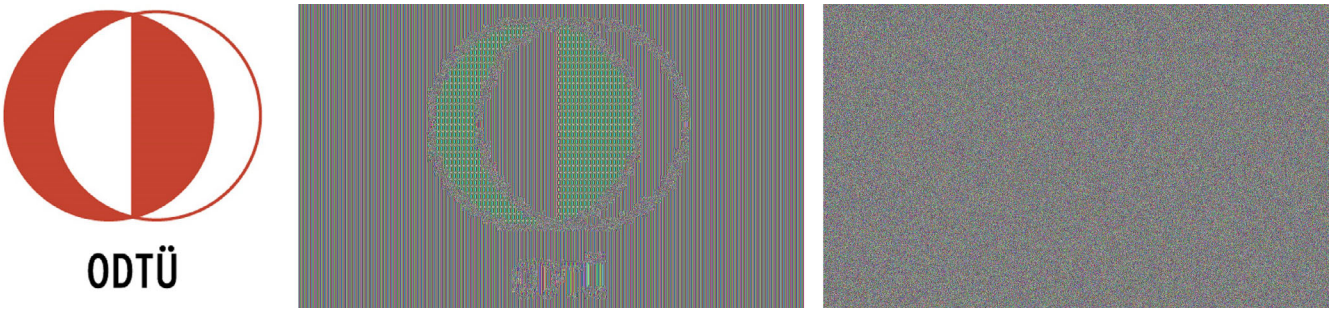

FIGURE 4 A bitmap image (left) and its encryption using AES-ECB (middle) and AES-CTR (right). ECB mode does not provide security since the original image can be easily guessed from the encrypted image

CTR is a fully parallelizable mode of encryption where any block can be encrypted independently. Instead of the plaintext blocks, a counter value which is optionally appended to a nonce is taken as the first block and the counter is incremented for the successive blocks. Output blocks of the encryption are XORed with the plaintext blocks to obtain ciphertext blocks. CTR encryption is shown in Figure 5.

CTR mode does not require the plaintext to be copied to GPU memory since counters, instead of plaintext blocks are encrypted in this mode. Thus, we can perform the final XOR of the output and the plaintext in RAM. Moreover, since the plaintext is used at the end, the encryption of the counters can be performed even before the plaintext is available and the output can be kept at the global memory of the GPU. Thus, when the plaintext is ready, the XOR of the output and the plaintext would provide the ciphertext. This reduction in memory copy operations provides an extra speed-up on GPUs compared to the ECB mode. Decryption is similar to encryption except the plaintext is obtained by XORing the output blocks with the ciphertext blocks.

3DES encryption time measurements on various GPUs using ECB and CTR encryption with and without key schedule are provided in Table 6.

In Table 6 we included the time cost of memory copy operations to show that in real world applications CTR implementation benefits from almost 2× speed-up against the ECB implementation since we do not need to copy plaintext to GPU memory in CTR mode encryption.

4.3 | Exhaustive search

Exhaustive search attack is a generic attack that can be applied to any symmetric key cryptographic algorithm. For a block cipher, if an attacker captures a plaintext and a ciphertext block pair, they can encrypt the plaintext with every possible key and check if the result is equivalent to the

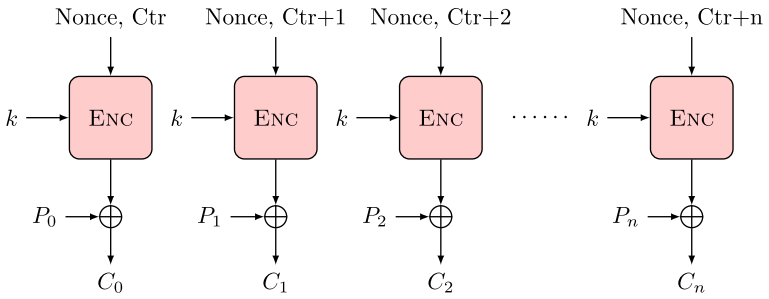

FIGURE 5 Counter (CTR) mode of operation[15]

TABLE 6 3DES Encryption time (ms) of a 4 MB file using CTR and ECB mode of operations on different GPUs

Mode of operation	GTX 860M	MX 250	GTX 970	RTX 3060 Laptop	RTX 2060 S	RTX 2070 S
ECB with key schedule	459.08	373.98	145.82	82.82	69.15	63.68
CTR with key schedule	398.54	295.77	95.68	39.74	39.16	33.98
ECB without key schedule	455.08	370.87	143.09	81.80	67.76	62.80
CTR without key schedule	396.58	292.55	94.04	38.98	38.35	33.20

Note: Time spent for the memory copy operations are included.

TABLE 7 3DES exhaustive search time (seconds) of trying 2^{30} keys on different GPUs

GPU	Time	Keys per second
GTX 860M	238.96	4,493,396
MX 250	180.94	5,934,167
GTX 970	54.69	19,631,803
RTX 2060 Super	13.71	78,306,726
RTX 3060 Laptop	11.56	92,884,240
RTX 2070 Super	11.37	94,403,185

captured ciphertext. The key that provides the captured ciphertext block is a candidate for the correct key. Note that when the key length is larger than the block length, more than one key might provide the captured ciphertext block. However, the correct key can easily be found among these candidates by decrypting other ciphertext blocks.

For a symmetric key algorithm with a k-bit secret key, the time complexity of the exhaustive search is 2^k encryptions or decryptions. Thus, an exhaustive search attack on 3DES requires 2^{168} 3DES encryptions. But it is a common knowledge that 3DES actually provides 112-bit security instead of 168 bits. This is achieved by dividing 3DES into two parts: since 3DES is simply the application of the DES algorithm thrice, we can partition 3DES into DES and 2DES. Moreover, we can perform an exhaustive search attack on DES as a precomputation by encrypting a plaintext block with every possible 2^{56} DES keys. Then we keep the obtained 2^{56} ciphertext blocks in a sorted table and perform exhaustive search attack on 2DES which has a time complexity of 2^{112} 2DES encryptions. Note that the sorted table requires 2^{56} blocks of data storage.

Unlike encryption, in exhaustive search attack we need to perform the key schedule algorithm for every block encryption since we are searching for the correct key. For this reason, we combined our key generation kernel and encryption kernel to perform an exhaustive search attack on 3DES. We performed our experiments on different GPUs with different architectures. The results are provided in Table 7.

For the exhaustive search attack, we perform the initial and final permutations only once on CPU and do not perform these operations on GPU. As shown in Table 7, we achieved more than 94 million key searches per second on a single RTX 2070 Super GPU. Although these results are for 3DES, a small modification to our kernel can convert it to an exhaustive search attack on DES or 2DES. Thus, an exhaustive search attack on DES with 512 RTX 2070 Super can capture the secret key in a month. However, an exhaustive search on 3DES would require 2^{64} RTX 2070 Super GPUs to capture the secret key in a month.

Note that RTX 3060 Laptop outperforms RTX 2060 Super in Table 7 during exhaustive search. However, the exact opposite happens in Table 6, RTX 2060 Super outperforms RTX 3060 Laptop during ECB and CTR mode encryptions. This shows that the bottleneck introduced by the key generation kernel in the exhaustive search is better handled in Ampere architecture than Turing architecture. This observation confirms the importance of experimental evaluation of different implementations on different GPUs for a fair comparison.

5 | CONCLUSION

Our research showed that, especially in big data chunks, modern GPUs can provide significant speed-up against multi-threaded CPU implementation for parallelizable cryptographic algorithms. Our bit-level parallelization of 3DES decryption and encryption on GPU obtained an average 15.95× speed-up against the baseline multi-threaded CPU implementation when encrypting large files using an RTX 2070 Super GPU. While it was not possible to experimentally compare the proposed implementation with those in the literature, we made our source code publicly available to make our results reproducible and facilitate future comparisons. We also observed that multi-GPU exhaustive search attack on DES can be performed in days whereas an exhaustive search attack on 3DES using GPUs is not possible even if with an implausible scenario involving the use of every GPU that has ever been manufactured.

ACKNOWLEDGMENT

We would like to thank Roberto Avanzi for providing the drawings used in Figures 1 and 2, Diana Maimut for providing the drawing used in Figure 5 and allowing free use of them as part of Tikz repository.[15]

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Github at https://github.com/kaanfurkan35/3DES_GPU

ORCID

Cihangir Tezcan https://orcid.org/0000-0002-9041-1932

Alptekin Temizel https://orcid.org/0000-0001-6082-2573

REFERENCES

1. Iwai K, Nishikawa N, Kurokawa T. Acceleration of AES encryption on CUDA GPU. *Int J Netw Comput*. 2012;2(1):131-145.
2. An S, Kim Y, Kwon H, Seo H, Seo SC. Parallel implementations of ARX-based block ciphers on graphic processing units. *Mathematics*. 2020;8(11). https://doi.org/10.3390/math8111894
3. Lee WK, Goi BM, Phan RCW. Terabit encryption in a second: performance evaluation of block ciphers in GPU with Kepler, Maxwell, and Pascal architectures. *Concurr Comput Pract Exper*. 2019;31(11):e5048. https://doi.org/10.1002/cpe.5048
4. Daemen J, Rijmen V. *The Design of Rijndael - The Advanced Encryption Standard (AES)*. Information Security and Cryptography. 2nd ed. New York, NY: Springer; 2020.
5. An S, Seo SC. Highly efficient implementation of block ciphers on graphic processing units for massively large data. *Appl Sci*. 2020;10(11). https://doi.org/10.3390/app10113711
6. Nishikawa N, Amano H, Iwai K. Implementation of bitsliced AES encryption on CUDA-enabled GPU. In: Yan Z, Molva R, Mazurczyk W, Kantola R, eds. *Proceedings of the 11th International Conference on Network and System Security, 10394 of Lecture Notes in Computer Science*. Helsinki, Finland: Springer; 2017:273-287.
7. Tezcan C. Optimization of advanced encryption standard on graphics processing units. *IEEE Access*. 2021;9:67315-67326. https://doi.org/10.1109/ACCESS.2021.3077551
8. Swierczewski L. 3DES ECB optimized for massively parallel CUDA GPU architecture. *CoRR*. 2013;abs/1305.4376.
9. Güler Z, Özkaynak F, Çınar A. CUDA Implementation of DES Algorithm for Lightweight Platforms. *ICBEB 2017*. New York, NY: Association for Computing Machinery; 2017:49-52.
10. Fadhil H. Accelerating Concealed ISB Steganography and triple-DES encryption using massive parallel GPU. *J Appl Sci Res* 2017; 13: 17-26.
11. Beletskyy V, Burak D. Parallelization of the data encryption standard (DES) algorithm. In: Pejaś J, Piegat A, eds. *Enhanced Methods in Computer Security, Biometric and Artificial Intelligence Systems*. Boston, MA: Springer; 2005:23-33.
12. Altınok KF, Peker A, Temizel A. Bit-level parallelization of 3DES encryption on GPU. Paper presented at: Proceedings of the 6th High Performance Computing Conference, Ankara, Turkey; 2020.
13. Mei X, Chu X. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans Parallel Distrib Syst*. 2017;28(1):72-86. https://doi.org/10.1109/TPDS.2016.2549523
14. Zheng L, Wang Z, Tian S. Comparative study on electrocardiogram encryption using elliptic curves cryptography and data encryption standard for applications in internet of medical things. *Concurr Comput Pract Exper*. 2020;e5776. https://doi.org/10.1002/cpe.5776
15. Jean J. TikZ for cryptographers; 2016. https://www.iacr.org/authors/tikz/.
16. ISO/IEC 18033-3:2010 Information technology –– security techniques –– encryption algorithms –– Part 3: block ciphers; 2010. https://www.iso.org/standard/54531.html.
17. Barker E, Roginsky A. *Transitioning the Use of Cryptographic Algorithms and Key Lengths. NIST SP 800-131A Rev. 2*; Gaithersburg, Maryland: National Institute of Standards and Technology; 2019. https://doi.org/10.6028/NIST.SP.800-131Ar2
18. Meijer C, Verdult R. Ciphertext-only cryptanalysis on hardened Mifare classic cards. In: Ray I, Li N, Kruegel C, eds. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, October 12-16*. Denver, CO: ACM; 2015:18-30.
19. Tezcan C. Brute force cryptanalysis of MIFARE classic cards on GPU. In: Mori P, Furnell S, Camp O, eds. *Proceedings of the 3rd International Conference on Information Systems Security and Privacy, ICISSP 2017, February 19-21*. Porto, Portugal: SciTePress; 2017:524-528.
20. Dworkin M. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques. NIST SP 800-38A*; Gaithersburg, Maryland: National Institute of Standards and Technology; 2001. https://doi.org/10.6028/NIST.SP.800-38A

How to cite this article: Furkan Altınok K, Peker A, Tezcan C, Temizel A. GPU accelerated 3DES encryption. *Concurrency Computat Pract Exper*. 2022;34(9):e6507. doi: 10.1002/cpe.6507

APPENDIX. A CONSTANT ARRAYS

```
BYTE pc_1[56] = {
  57,49,41,33,25,17,9,1,58,50,42,34,26,18,10,2,59,51,43,35,27,19,11,3,60,52,44,36,
  63,55,47,39,31,23,15,7,62,54,46,38,30,22,14,6,61,53,45,37,29,21,13,5,28,20,12,4 };
```

Listing 1: Permuted choice table

```
int shift_keys[16] = {1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1};
```

Listing 2: Shift keys

```
BYTE pc_2[48] = {
  14,17,11,24,1,5,3,28,15,6,21,10,23,19,12,4,26,8,16,7,27,20,13,2,
  41,52,31,37,47,55,30,40,51,45,33,48,44,49,39,56,34,53,46,42,50,36,29,32 };
```

Listing 3: Key-Compression Table

```
BYTE initial_perm[64] = {
  58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,
  62,54,46,38,30,22,14,6,64,56,48,40,32,24,16,8,
  57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,3,
  61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7 };
```

Listing 4: Initial Permutation

```
BYTE s[8][4][16] = { {
                14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7, //0
                0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
                4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,
                15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13 },
        {
                15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10, //1
                3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5,
                0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15,
                13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9 },
        {
                10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8, //2
                13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1,
                13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,
                1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12 },
        {
                7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15, //3
                13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9,
                10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,
                3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14 },
        {
                2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9, //4
                14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6,
                4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,
                11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3 },
        {
                12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11, //5
                10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8,
                9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,
                4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13 },
        {
                4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1, //6
                13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6,
                1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,
                6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12 },
        {
```

```
13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7, //7
1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2,
7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,
2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11 } };
```

Listing 5: S-box Table, total 8 s-boxes

```
BYTE per[32] =  {
  16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,
  2,8,24,14,32,27,3,9,19,13,30,6,22,11,4,25 };
```

Listing 6: Straight Permutation Table