# The implementation of "Kuznyechik" encryption algorithm using NVIDIA CUDA technology

**A N Borisov[1] and E V Myasnikov[1]**

[1]Samara National Research University, Moskovskoe Shosse 34A, Samara, Russia, 443086

e-mail: borisovalexey1996@gmail.com

**Abstract.** In this paper, we discuss various options for implementing the "Kuznyechik" block encryption algorithm using the NVIDIA CUDA technology. We use lookup tables as a basis for the implementation. In experiments, we study the influence of the size of the block of threads and the location of lookup tables on the encryption speed. We show that the best results are obtained when the lookup tables are stored in the global memory. The peak encryption speed reaches 30.83 Gbps on the NVIDIA GeForce GTX 1070 graphics processor .



## 1. Introduction

Cryptographic protection is an important part of a modern IT infrastructure. Nowadays, both the volume of information and computing power are continually increasing. Accordingly, there are growing demands on both the robustness and speed of cryptographic algorithms.

The idea of using graphics processors to speed up encryption algorithms appeared almost simultaneously with the idea of using them for general-purpose computing[1]. As known, the maximum profit from the use of graphics processors can be achieved only with massive parallel tasks. It is not surprising that the most noticeable results in this field were obtained for block encryption in the ECB (electronic code book) and CTR (gamming) modes, since the blocks of plain text are processed independently in this case.

At present, there is a lot of papers, which focuses on using graphics processors for encryption. Most of the papers are devoted to the AES encryption algorithm. In particular, in the paper [2], authors study the dependence of the encryption speed on the location of round keys. Experiments are carried out on NVIDIA GeForce GTX 780, NVIDIA GeForce GTX 1080 and NVIDIA GeForce Titan X graphics cards. In the paper [3], authors study popular block ciphers, namely, AES-128, CAST-128, Camelia, SEED, IDEA, Blowfish, and Threefish, using NVIDIA GeForce GTX 980. In both papers, the reported encryption speed exceeds 200 Gbit/s (25 GB/s).

The "Kuznyechik" cipher is a new symmetric encryption standard, which was introduced in Russia in 2015 [4]. By design, "Kuznyechik" is an SP-network with ten rounds of transformations, the size of a block equals to 128 bits, and the size of a key is 256 bits.

Because of a novelty and the limited use of chipher, fewer papers focus on the 'Kuznyechik" algorithm, most of them focuses on cryptoanalysis [5,6,7] rather than effective implementation of cipher. The paper [6] describes the optimization of the algorithm based on lookup tables.

The reported speed is 54 Mbps on a four-core CPU. The paper [7] is a direct continuation of the paper [6], which is devoted to the cryptanalysis of the "Magma" and "Kuznyechik" algorithms using CUDA to speed up a slide attack. The paper [8] describes the implementation of Kuznyechik cipher for FPGA using OpenCL, the reported troughput reaches 41 Gbit/s. In this paper, we study several possible implementations of the "Kuznyechik" cipher using the NVIDIA CUDA technology. The paper has the following structure. In Section 2, we givebrief theoretical foundations of the algorithm. In Section 3, we briefly discuss some features of the NVIDIA CUDA technology, which are necessary to understand this paper. Section 4 describes possible implementations of the cipher using the above technology. In Section 4, we provide the results of experiments. The paper ends up with the conclusion and reference list.

## 2. The description of the algorithm

As it was outlined earlier, the "Kuznyechik" algorithm is a ten-round cipher based on an SP network with the key length of 256 bits and the block length of 128 bits. An exhaustive description of the algorithm can be found in [4].

### *2.1. Key expansion algorithm*

Before the encryption, ten 128-bit round keys $K_1, ..., K_{10}$ are generated based on the main 256-bit key $K$. The expansion procedure is a Feistel network. Its transformation function is analogous to the round of the main cipher with a fixed key. The first two round keys $K_1, K_2$ are obtained from the halves of the main one. The rest of the keys are obtained by encrypting with the Feistel network. The keys $(K_{2i}, K_{2i-1})$ are given to each round of the network as an input, and the output is $(K_{2i+2}, K_{2i+1})$.

The expansion procedure is strictly sequential. Besides, it can be executed only once to obtain the array of round keys. Thus, we do not consider this procedure in the paper.

### *2.2. Basic transformations*

The cipher is based on two transformations:

(i) Nonlinear transformation $\pi : GF(2^8) \to GF(2^8)$ implemented through the lookup table.

(ii) Nonlinear transformation

$$\ell(a_{15}, ..., a_0) = 148 \cdot a_{15} + 32 \cdot a_{14} + 133 \cdot a_{13} + 16 \cdot a_{12} + 194 \cdot a_{11} + 192 \cdot a_{10} + 1 \cdot a_9 + \\ +251 \cdot a_8 + 1 \cdot a_7 + 192 \cdot a_6 + 194 \cdot a_5 + 16 \cdot a_4 + 133 \cdot a_3 + 32 \cdot a_2 + 148 \cdot a_1 + 1 \cdot a_0, \quad (1)$$

where $a_i \in GF(2^8)$, and operations take place in the field $GF(2^8)$. In particular, "+" is equivalent to the XOR operation).

The transformations that are used directly in the encryption are based on two previous transformations:

$$S(a) = S(a_{15}||...||a_0) = \pi(a_{15})||...||\pi(a_0), \quad (2)$$

$$R(a) = R(a15||...||a0) = \ell(a15, ..., a0)||a15||...||a1, \quad (3)$$

$$L(a) = R^{16}(a), \quad (4)$$

$$X[k](a) = k \oplus a, \quad (5)$$

where $||$ is the concatenation,
$a_i \in GF(2^8)$,
$k, a \in GF(2^{128})$,
$R^{16}(a)$ means that the $R$ function is applied for $a$ for 16 times.

Using the above notations, the "Kuznyechik" algorithm can be described as follows:

$$E_{K_1,...,K_{10}}(a) = X[K_{10}]LSX[K_9]...LSX[K_2]LSX[K_1](a), \quad (6)$$

where $LSX[K](a)$ is equivalent to $L(S(X[K](a)))$.

*2.3. Lookup tables*
It is easy to see that

$$\ell(a_{15}, ..., a_0) = \ell(a_{15}, 0, ..., 0) \oplus \ell(0, a_{14}, 0, ..., 0) \oplus ... \oplus \ell(0, 0, 0, ..., a_0), \tag{7}$$

$$\ell(a_{15} \oplus b_{15}, ..., a_0 \oplus b_0) = \ell(a_{15}, ..., a_0) \oplus \ell(b_{15}, ..., b_0). \tag{8}$$

Based on this, the $LS$ part of the round can be predicted in advance, with the result that we get 16 lookup tables. Each of the lookup tables contains 256 entries of size 128 bit. When using lookup tables, the entire encryption process comes down to 16 lookups across tables and 16 128-bit XOR operations (15 on the search results and 1 with a round key).

## 3. CUDA

CUDA is a proprietary API provided by NVIDIA that facilitates the use of video cards for general purpose computing. This programming model is required since the graphics processor can process hundreds of threads simultaneously according to the so-called SIMT model. A detailed description of CUDA technology can be found in [9]. We provide only the necessary information below.

The kernel is a function intended for the execution on the GPU. The unit of execution is a thread. Threads are combined into blocks, and blocks are combined into a grid. The grid and block configuration is specified when the kernel function is started. Resources for execution are allocated per block, and not on every single thread.

From a hardware point of view, the graphics processor is divided into multiprocessors, and multiprocessors are divided into warps. The warps, in turn, consist of 32 stream processors (terminal computing units). All stream processors inside one warp are synchronized. They either execute the same instruction or idle.

The memory organization on graphics processors has a number of features. A programmer has access to a global, shared, constant, and texture memory. The global memory is equivalent to RAM. The texture memory and constant memory are the areas of global memory with special access features. The shared memory has low volume (64KB), but very high speed.

Physically, it is divided into several independent areas, called banks (banks). An attempt to simultaneously access different memory cells within one bank will lead to the subsequent execution of requests. This situation is called a bank conflict. A large number of access conflicts reduce performance significantly.

## 4. The implementation of "Kuznyechik" encryption algorithm using CUDA

We use lookup tables as a base for the implementation of the algorithm. At first, we execute an initialization procedure, which copies the tables to the device. Before encryption, we allocate buffers on the device. We use the allocated buffers to copy round keys and data to be transformed. We copy data back and release the buffers after the encryption. Encryption keys are best to store in the shared memory. The memory consumption is small, and the access pattern allows to avoid access conflicts (all threads of a warp always read the same key).

The issue of locating the lookup tables does not have such a straightforward solution. The size of the $LS$ tables does not allow them to be entirely located in the shared memory since for the Pascal generation it is allowed to allocate only 48 KB per thread block. In addition, random access to the tables will result in numerous access conflicts, which significantly reduce performance. From this it follows that there are only three options for storing the tables, namely, the global, constant and texture memory. To provide an additional acceleration, we read the global memory using the $ldg()$ function.

In addition, we should also find out the optimal launch configuration of the kernel function. It does not make sense to use more than one thread per block of text. Although one XOR

operation for 128-bit values is translated into two 64-bit operations, the use of two threads to process one block of text will lead to undesirable consequences. First, an increase in the number of threads will lead to an overall decrease in simultaneously processed blocks of text, since the number of stream processors is limited. Second, since each thread needs to store its copies of local variables, the number of occupied registers will increase. This also becomes a limiting factor and reduces the amount of simultaneously processed blocks of text. However, it makes sense to study how the size of the block of threads affects the encryption speed.

In this paper, we consider the following sizes of a block: 32, 64, 128,192, 256, 384, 512, 768, and 1024 threads. There is no sense to consider less than 32 threads in a block since the minimum unit of execution on a graphics processor is a warp consisting of 32 stream processors.

## 5. Experimental results

We used the following configuration in our experiments:
CPU: Intel Core i5-6400
GPU: NVIDIA GeForce GTX 1070
RAM: 8 Gb DDR3

In general speed tests, the number of threads in a block was 512. The volume of encrypted data was equal to 256 MB in tests with changing block configuration. The CPU version of the "Kuznyechik" algorithm executed in 4 threads using SSE instructions. We did not take into account the time spent on copying data to the memory of the GPU and back.

The results of performance measurements are presented in Tables 1 and 2 and Figures 1 and 2 shows the results of the experiments.

**Table 1.** The dependence of the encryption speed on the data volume.

| Data volume | Encryption speed (Gbps) | | | |
|---|---|---|---|---|
| | CPU | Global memory | Constant memory | Texture memory |
| 1 KB | 0.09 | 0.43 | 0.03 | 0.45 |
| 32 KB | 2.46 | 8.55 | 0.34 | 8.36 |
| 128 KB | 2.50 | 24.75 | 0.59 | 20.83 |
| 1 MB | 3.70 | 24.75 | 0.80 | 16.07 |
| 16 MB | 4.73 | 24.74 | 0.66 | 14.29 |
| 64 MB | 4.74 | 25.09 | 0.65 | 14.01 |
| 128 MB | 4.79 | 27.26 | 0.64 | 13.86 |
| 256 MB | 4.79 | 30.83 | 0.64 | 13.97 |
| 512 MB | 4.81 | 30.82 | 0.66 | 13.93 |

Let us consider the results in more detail.

The CUDA SDK includes a profiler, which allows getting information on the kernel execution time, resources used by the kernel, multiprocessor loading, cache efficiency, and many other parameters. The profiler data is used for a more detailed analysis of the results.

The fastest option is the global memory. The correct access pattern allows us to achieve more than 90% hits fro L2 cache and about 20% hits for L1. However, even in this case, working with memory is still an essential factor limiting performance.

**Table 2.** The dependence of the encryption speed on the thread block size.

| Threads per block | Encryption speed (Gbps) | | |
|---|---|---|---|
| | Global memory | Constant memory | Texture memory |
| 32 | 30.86 | 0.80 | 16.64 |
| 64 | 30.94 | 0.66 | 13.59 |
| 128 | 30.75 | 0.65 | 13.28 |
| 192 | 30.66 | 0.67 | 13.85 |
| 256 | 30.81 | 0.63 | 13.55 |
| 384 | 30.70 | 0.66 | 13.81 |
| 512 | 30.67 | 0.69 | 14.07 |
| 768 | 30.81 | 0.77 | 15.03 |

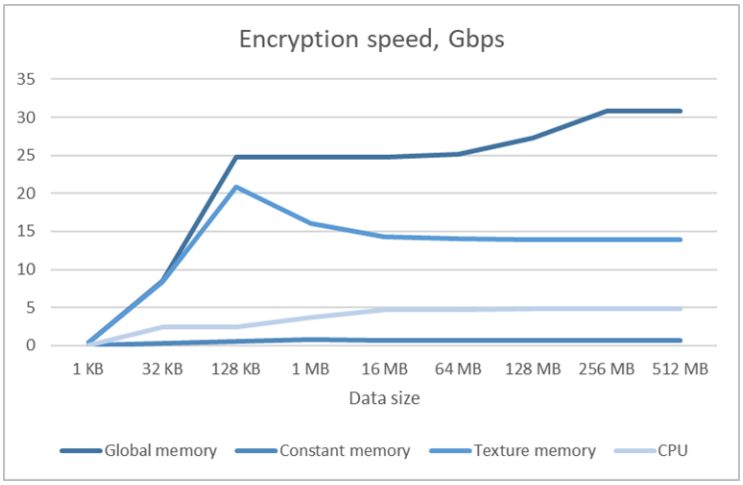


**Figure 1.** The dependence of the encryption speed on the data size.

The texture memory is slower due to the greater number of registers used and the greater memory load. When reading a single value, its neighbors are also loaded into the cache, so the memory subsystem has to serve the greater number of readings. So cost of each cache miss is high. In addition, in the Maxwell and Pascal architectures, the texture cache shared the memory with L1 cache and therefore does not give any gain compared to it.

For constant memory, the problem is a random access pattern, which causes frequent cache misses. Another problem is the read data size - 16 bytes per request. As a result, the performance degrades.

Let us turn to the size of the thread blocks.

No more than a certain number of threads, warps, and blocks can be executed on a multiprocessor simultaneously. In addition, there are limiting factors, namely, the total number of registers and shared memory. With small blocks, the overhead caused by the context switching is huge. At the beginning of execution, time is spent for each block to copy round keys to the shared memory. For large blocks, a multiprocessor simultaneously executes fewer blocks due to the presence of limiting factors. Besides, the location of the block cannot be taken by another one until the block is completed. In our case, the difference between thread block sizes for global memory is insignificant, since the main limiting factor is the global memory latency. Constant memory case reaches maximum troughput with 32 threads per thread block, whereas texture memory case have its maximum at 32 threads per threads block.

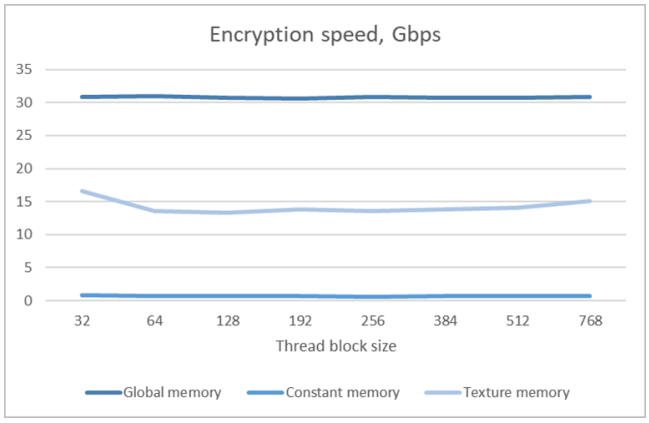


**Figure 2.** The dependence of the encryption speed on the thread block size.

## 6. Conclusion

In this paper, we considered various options for implementing the "Kuznyechik" encryption algorithm using CUDA technology. We used lookup tables as a basis for the implementation. We discussed the global, constant, and texture memory as an option for the location of the lookup tables. Besides, we considered the following sizes of thread blocks: 32, 64, 128, 192, 256, 384, 512. According to the experiments, the best results were obtained when search tables are stored in the global memory. The peak speed of the encryption was equal to 30.83 Gbps on the equipment used.

## 7. References

[1] Cook D L, Ioannidis J, Keromytis A D and Luck J 2005 CryptoGraphics: secret key cryptography using graphics cards CT-RSA *Lecture Notes in Computer Science* **3376** 334-350
[2] Abdelrahman A, Fouad M and Dashan H 2017 Analysis on the AES Implementation with Various Granularities on Different GPU Architectures *Advances in Electrical and Electronic Engineering* **15** 03
[3] Lee W K, Cheong H S, Phan R and Goi B M 2016 Fast implementation of block ciphers and PRNGs in Maxwell GPU architecture *Cluster Comput.* **19(01)** 335-347
[4] GOST R 34.12-2015 *Information technology. Cryptographic data security. Block ciphers* (Moskow: Standartinform) p 21
[5] Biryukov A, Perrin L and Udovenko A 2016 Reverse-engineering the S-Box of Streebog, Kuznyechik and STRIBOBr1 Advances in Cryptology *Lecture Notes in Computer Science* **3376** 372-402
[6] Ishchukova E, Babenko L and Koshucky R 2015 Implementation of high speed data encryption using "Kuznyechik" cipher *Auditorium* **04(8)**
[7] Ishchukova E, Babenko L and Anikeev M 2016 Fast Implementation and Cryptanalysis of GOST R 34.12-2015 Block Ciphers *Proc. 9th Int. Conf. on Security of Information and Networks* (New York: ACM) 104-111
[8] Korobeynikov A 2019 E˙ective Implementation of "Kuznyechik" block cipher on FPGA with OpenCL Platform *Proc. of IEEE Conf. of Russia Young Researchers in Electrical and Electronic Engineering* 1683-1686
[9] Sanders J and Kandrot E 2010 *CUDA by Example* (New York: Addison-Wesley) p 313

**Acknowledgments**
The work was funded by the Russian Federation Ministry of Science and Higher Education within a state contract with the "Crystallography and Photonics" Research Center of the RAS under agreement 007-GZ/Ch3363/26.