Methods for Implementation of Pseudo-Random Number Generator Based on GOST R 34.12-2015 in Hybrid CPU/GPU/FPGA High-Performance Systems

Andrey. A. Skitev[1], Mikhail M. Rovnyagin[2], Ekaterina N. Martynova[3], Marina I. Zvyagina[4], Kirill D. Shelopugin[5], Anastasiia A. Chernova[6]
Department of Computer Systems and Technologies
National Research Nuclear University MEPhI (Moscow Engineering Physics Institute)
Moscow, Russia
[1]suharev_p@mail.ru, [2]m.rovnyagin.2015@ieee.org, [3]martynova@netup.ru,
[4]zvyagina.marina@gmail.com, [5]kshelopugin@gmail.com, [6]chernovaaa94@gmail.com

Abstract — **The architecture of high-performance data storage and processing systems has changed considerably. Modern cloud computing systems are often not just a hybrid but also supports hardware acceleration. The paper describes the scope of information security protocols based on PRNG in industrial systems. The work provides a method for implementing GOST R 34.12-2015 Based Pseudo-Random Number Generator in hybrid systems. The description and results of studies parallel CUDA and FPGA versions of the algorithm for use in hybrid data centers are presented.**

Keywords — High performance computing, NVIDIA CUDA, Encryption, hybrid architecture, FPGA

I. Introduction

Currently, pseudo-random number generators (PRNG) are used to solve a wide range of tasks. Depending on the features, PRNGs [1] can be used: in computational mathematics to solve tasks by Monte-Carlo method, in the tasks of simulation modeling [2], for queuing systems (QS), in cryptography, in verification tasks and many more. However, the strictest requirements are imposed on the generators that are used to build data security algorithms, as in order to ensure considerable encryption strength, PRNGs shall be unpredictable, statistically safe, as well as generate a sequence with a big period. At the same time, cryptographic algorithms impose high requirements to the computing power of computers.

There are a large number of tailored industrial solutions to be used in data processing centers (DPC), automated industrial process control systems (AIPCS), and systems with increased requirements of confidential and personal data security (PD). In industrial systems, for example SCADA [3], reliable equipment, which is basically hardware solutions, resistant to the impact of external factors, having compact sizes and standard mounting types (for example, on a DIN-rail) are used in controlled facilities. In DPCs performing the processing of data coming from objects and the calculation of controlling actions, standard computing solutions with less strict requirements to the operational conditions are applied. At the same time, it is necessary not only to encrypt data transferred through a network, but also to ensure the security of data uploaded to external memory for long-term storage. Specialized hardware/software solutions starting with the support of encryption and assignment of privileges at the level of virtualization subsystem hypervisor, ending with the use of hard disks that are certified according to FIPS are also used for such purposes. Until recently, clusters with CPU-architecture could process the incoming data, and, where necessary, hardware solutions were used. However, today, due to a considerable increase in the volumes of processed data, more DPCs switch from classic to hybrid CPU/GPU-architecture for computing.

Recently, the architecture of high-performance data storage and processing systems has changed considerably. Hybrid technologies have become the most widespread in the field of supercomputing. The availability of computing accelerators of different sense in the system's node is to be understood as hybridism. Both CUDA [4] or Xeon Phi [5] computing machines, and sets of reprogrammable FPGA-coprocessors can be used as such accelerators. The abovementioned technologies have a high degree of parallelism and solve many tasks much more effectively than classic CPU-devices. Generally, hybrid technologies are applied to perform physical calculations, chemical modeling, etc. However, research into the possibility of applying GPGPU/FPGA-devices as cryptographic coprocessors, compression/decompression modules, subsystems of graphical data background processing, etc. is carried out. There is a real need of high-performance versions of cryptographic algorithms for their application in hybrid supercomputer data capturing and processing systems. In 2015, Russia adopted a new standard of block encryption [6]. It included description

978-1-5090-4865-6/17/$31.00 ©2017 IEEE

of two ciphers – *Magma* (GOST 28147-89 version with the table of fixed replacements), and *Kuznechik* (*Grasshopper*) – a new 128-bit round block cipher based on the SP network. Due to its architecture, *Grasshopper* is an excellent choice for hybrid systems (CPU/GPU/FPGA).

II. Related Works

Emergence of the development technologies for hybrid systems (such as NVIDIA CUDA and AMD APP) has generated considerable interest and attention in the scientific environment. There are lots of works on how to improve performance of the block cryptographic algorithms with the use of CUDA technology. In the work [7], the authors analyze advantages of CUDA technology as compared to the previously used OpenGL. According to the work carried out by the researchers of AES algorithm parallel version, the input information got stored to the global memory of CUDA device, while the replacement blocks were stored to the cached constant memory in order to speed up the process of reading the S-box data. In the article [8], the authors provide an overview of all AES encryption modes and put emphasis on those suitable for parallelization: ECB (Electronic codebook), CTR (Counter). For the cipher-block chaining (CBC) mode, parallelization is enabled only through decryption operations. For these modes, the authors provide graphs of bandwidth capacity and some tips to improve performance.

In addition to the articles on implementation of the block algorithms on CUDA, there are also some other works [9-12] describing implementation of parallel versions of block encryption algorithms for FPGA and GPGPU co-processors.

III. Method for Implementing the GOST Algorithm on GPU

When it comes to developing an algorithm with the use of parallel computing, the granularity of data access is the factor to be considered. If parallelization of the process data is implemented with a step size equal to N bytes, each thread will have its own N-byte key-space, so there is no dependency between threads. This approach helps eliminate unnecessary stages of synchronization, in providing the shared access of threads to the memory containing source data.

According to the research [7], it was decided to partition data with the 16-byte granularity, in order to achieve maximum bandwidth capacity. In view of this feature, it is worth noting that the size of the file space allocated to the file in the memory should be rounded up to 128 bits. For the purposes of economical use of memory, all the key information (for example, the table of replacements) is stored to the constant memory, since such data are not modified during the algorithm operation. Each GPU thread computes 16 bytes of the original sequence, then it copies the obtained result from the global GPU memory into the resulting array on the CPU.

For correct processing of files with the size exceeding GPU memory, it is necessary to determine the number of bytes able to process all the threads available for allocation on GPU. The number of bytes can be calculated according to the formula (1):

$$\mathrm{N}_{max} = \mathrm{B}_{max} * \mathrm{T} * 16 \quad (1)$$

where B_{max} is the maximum number of blocks per grid; T is the fixed number of threads in a block, 16 is the number of bytes processed by a single thread (*Grasshopper* block size). In this study, the number of threads in the block equals 128. If the number of bytes to be processed exceeds the allowable value calculated by the formula (1), it is necessary to increase the number of threads per block by 128 (2) and recalculate the value using the formula (1).

$$\mathrm{T} = \mathrm{T} + 128 \quad (2)$$

If necessary, the original file can be split into parts not exceeding the free GPU memory size.

In addition to calculating the required memory size, one should also determine the number of blocks and CUDA threads necessary for the most optimal solution of the task. As mentioned above, the number of the threads in block is accepted to be 128. Based on that, the number of blocks can be calculated by the formula (3):

$$B = \frac{threads * 16}{N} \quad (3)$$

where B is the number of blocks, threads refer to the previously set number of *threads* allocated per block (equal to 128), N is the total number of bytes of the transmitted data, 16 is the number of bytes in a single block. If number of threads is insufficient, it is necessary to increase threads to 128 and repeat calculations as in (2).

Launching kernels on GPU, each thread, currently running in the block, performs the key procedure of encryption/decryption. The entire processing mechanism can be described by the following algorithm:

1. Definition of the allocated thread, which is currently running. For this purpose, it is necessary to use the formula (4):
$$j = (\mathrm{blockIdx.x} * \mathrm{blockDim.} + \mathrm{threadIdx.x}) + \\ +(\mathrm{blockIdx.y} * \mathrm{blockDim.y} + \mathrm{threadIdx.y}) + \\ + (\mathrm{blockIdx.z} * \mathrm{blockDim.z} + \mathrm{threadIdx.z}), \quad (4)$$
where j is the desired number of the thread, *blockIdx.x*, *blockIdx.y*, *blockIdx.z* – number of the currently active block, *blockDim.x*, *blockDim.y*, *blockDim.z*, *threadIdx.x*, *threadIdx.y*, *threadIdx.z* – number of the thread running in the current block.
2. The offset from the starting address of the array, in which a thread will be running, can be calculated according to the formula (5):
$$\mathrm{offset} = \mathrm{j} * 16 \quad (5)$$
where *j* is the number of the currently running thread, 16 is a *Grasshopper* block size.
3. Implementation of the conversion procedure.

IV. Method for Implementing the GOST Algorithm on FPGA

As the basic linear operation, the *Grasshopper* algorithm uses operations of multiple constant multiplication and addition in Galois Field. In GOST this operation is denoted as R-conversion. Each round includes 16 repetitions of this conversion, which in GOST is identified as *L-conversion*.

Grasshopper cypher includes nine rounds, so it is easy to calculate that R operation is repeated 16·9 = 144 times.

Underlying implementation of the linear conversion is time-consuming. To improve performance of the algorithm, it is possible to speed up this conversion completing it 16 clock cycles earlier.

In its underlying implementation, the linear conversion block can be represented as an eight-bit shift register and a circuit implementing the *R-conversion* (Fig. 1.a). The *R-conversion* can be implemented as a set of multiplication-by-constants diagrams and the modulo 2 eight-bit adder (Fig. 1.b).

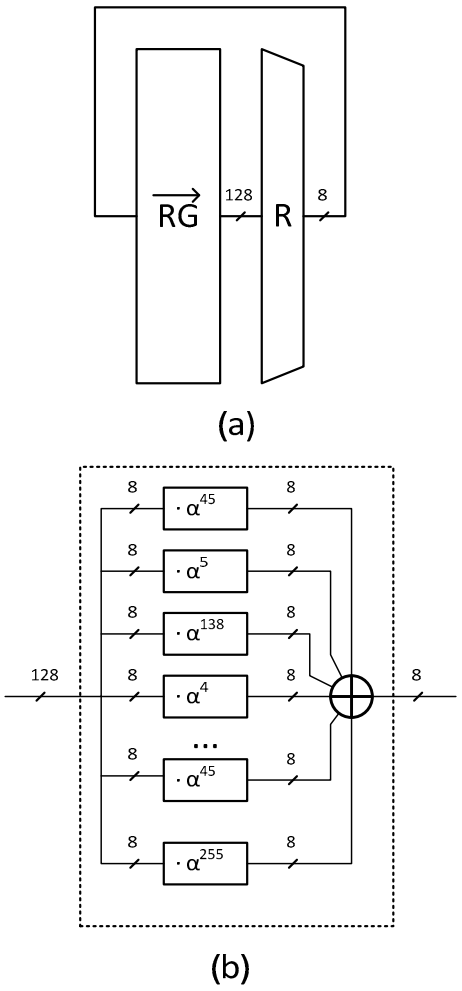

Fig. 1 The structure of L-conversion block (a) and R-conversion block (b)

The minimum period of this scheme operation T_{min} is determined as the sum of operation time of the multiplying circuit t_m and adding circuit t_S. In that case, the time to perform linear operation will be equal to $t_L = 16\cdot(t_m + t_S)$.

Let us consider the mathematical representation of linear operations. Basic linear conversion can be represented as the product of a row and a column (hereinafter, for the sake of convenience, the elements of the field are provided in the form of α degrees):

$$\ell(a_{15},\dots,a_0) = (a_{15} \quad a_{14}\cdots a_1 \quad a_0)\cdot\begin{pmatrix}\alpha^{45}\\ \alpha^{5}\\ \vdots\\ \alpha^{45}\\ \alpha^{255}\end{pmatrix} = b \qquad (6)$$

In this case, the operation $R(a)$ can be viewed as the product of a row and a matrix:

$$R(a) = (a_{15} \quad a_{14}\cdots a_1 \quad a_0)\cdot \begin{pmatrix}\alpha^{45} & 1 & 0 & \cdots & 0 & 0\\ \alpha^{5} & 0 & 1 & \cdots & 0 & 0\\ \alpha^{138} & 0 & 0 & \cdots & 0 & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots\\ \alpha^{5} & 0 & 0 & \cdots & 1 & 0\\ \alpha^{45} & 0 & 0 & \cdots & 0 & 1\\ \alpha^{255} & 0 & 0 & \cdots & 0 & 0\end{pmatrix} = \vec{a}\cdot C, \qquad (7)$$

where $\vec{a}$ is a string from the elements $a_{15}...a_0$ and C is a constant coefficient matrix. $R^2(a)$, subsequently, is a double multiplication by the matrix:

$$R^2(a) = R\big(R(a)\big) = \left((a_{15} \quad \cdots \quad a_0)\cdot \begin{pmatrix}\alpha^{45} & 1 & 0 & \cdots & 0 & 0\\ \alpha^{5} & 0 & 1 & \cdots & 0 & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots\\ \alpha^{45} & 0 & 0 & \cdots & 0 & 1\\ \alpha^{255} & 0 & 0 & \cdots & 0 & 0\end{pmatrix}\right)\cdot$$
$$\begin{pmatrix}\alpha^{45} & 1 & 0 & \cdots & 0 & 0\\ \alpha^{5} & 0 & 1 & \cdots & 0 & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots\\ \alpha^{45} & 0 & 0 & \cdots & 0 & 1\\ \alpha^{255} & 0 & 0 & \cdots & 0 & 0\end{pmatrix} = (a_{15} \quad \cdots \quad a_0)\cdot$$
$$\begin{pmatrix}\alpha^{45}\cdot\alpha^{45}+\alpha^{5} & \alpha^{45} & 1 & \cdots & 0 & 0\\ \alpha^{5}\cdot\alpha^{45}+\alpha^{138} & \alpha^{5} & 0 & \cdots & 0 & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots\\ \alpha^{45}\cdot\alpha^{45}+\alpha^{255} & \alpha^{45} & 0 & \cdots & 0 & 0\\ \alpha^{255}\cdot\alpha^{45} & \alpha^{255} & 0 & \cdots & 0 & 0\end{pmatrix} = \qquad (8)$$
$$= \vec{a}\cdot C^2.$$

As far as $R^n(a) = R\big(R^{n-1}(a)\big)$, the conversion $L(a)$ can be put in the following form:

$$L(a) = R^{16}(a) = (a_{15} \quad \cdots \quad a_0) \cdot \begin{pmatrix} c_{1,1} & \cdots & c_{16,1} \\ \vdots & \ddots & \vdots \\ c_{1,16} & \cdots & c_{16,16} \end{pmatrix} =$$

$$= \vec{a} \cdot C^{16}. \qquad (9)$$

Thus, *L-conversion* can be implemented in a single clock cycle, in which case the minimum period of such a scheme (as in the case of the underlying implementation) will be equal to the amount of operation of the multiplying circuit t_m and the adding circuit t_S. Diagram of a single cycle *L-conversion* block is shown in Fig. 2.

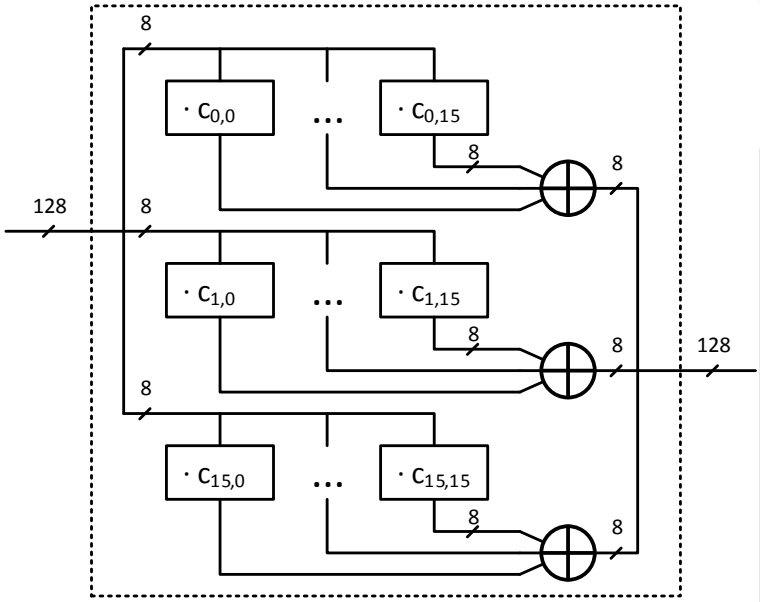

Fig. 2 R^{16}-conversion block

It easy to notice that instead of the C^{16} matrix it is possible to use the half-size C^8 matrix and to perform conversion in two clock cycles. The same applies to the C^4 and C^2 matrices.

This approach allows for the necessary balance between the required capacity and consumed resources.

V. Experimental Results

Table 1 shows the experimentally obtained values of the consumed resources and performance, when allocating *L-conversion* blocks on FPGA (Xilinx Spartan 6 XC6SLX45-3fgg484 [13]). XILINX ISEF CAD was used for allocation.

TABLE 1 L-CONVERSION ALLOCATION RESULTS

	LUT, pcs	FD, pcs	Tmin, nanoseconds	N, Gbit/s	K, Mbit/s·pcs
R	395	160	4.95	1.62	4.09
R^2	563	154	4.95	3.23	5.74
R^4	885	150	5.5	5.82	6.57
R^8	1429	138	5.75	11.13	7.79
R^{16}	2165	3	4.85	26.39	12.19

The first two columns contain the consumed FPGA resources – LUTs and triggers. The third column shows a minimum clock period, which enables circuit allocation without violating timing constraints. It is logical that the circuit with no shift register has got the shortest clock period. The last but one column contains the estimated capacity of *L-conversion* block. Finally, the last column defines the efficiency of use of logical resources (LUT) as the ratio of bandwidth capacity to the amount of the used resources.

According to the data given in Table 1, it is possible to construct a graph (Fig. 3) demonstrating the effectiveness of the proposed method.

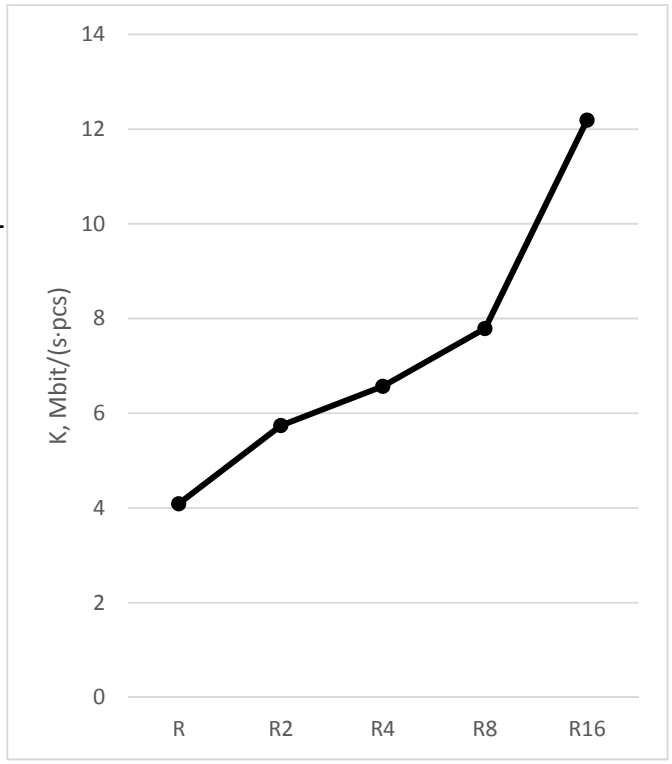

Fig. 3 The diagram of the efficient use of resources for various L-conversion block implementations

The diagram shows that the increased number of *R-conversions* adds to the efficient use of resources, which reaches its maximum when using *R^{16}-conversion*. Such results can be achieved due to the fact that the implemented circuits of multiplying the same numbers by various constants have common areas or even duplicate constants. It should also be noted that implementation with the use of *R^{16}-conversion* does not require any shift register, which influences logical resources and requires few or no triggers.

For testing and debugging the *Grasshopper* GPGPU version, there was introduced an additional program version using a single CPU core for calculations. Timing of program execution with the use of both CPU and GPU has been also enabled in the cypher. The testing wad held based on the Intel Core i7 3.4 GHz and 4 Gb RAM and display card – NVIDIA GeForce GTX 550 Ti.

Table 2 illustrates the testing results for the mode of simple replacement.

TABLE 2 *THE RESULTS OF PERFORMANCE COMPARISON*

File sizes	**Simple replacement**		
	CPU ms	***GPU ms***	***Performance improvement, times***
0.42 KB	0	20.93	none
1.25 KB	15	20.92749	none
1.56 KB	25	21.088448	1.18
1022.36 KB	14652	827.24	17.71
5.77 MB	83676	4429.25	18.89
34.35 MB	492549	26334.69	18.70
70.08 MB	1007925	56142.96	17.95
152.49 MB	2220453	124413.90	17.85
484.93 MB	7075060	388047.22	18.23
0.98 GB	14794350	819572.75	18.05
2.23 GB	33526563	1862586.75	18.01

Based on the obtained results, we can conclude that the use of hybrid computing for implementation of GOST cryptoalgorithm enables a several-fold accelerated computing due to the efficient algorithm parallelization and its implementation on GPU. It should be noted that, when using GPU for the purpose of encryption/decryption of small amounts of data (up to 1 MB), the performance increase can't be observed in view of the time-consuming GPU initialization and building of a three-dimensional grid for thread execution.

VI. CONCLUSION

The findings presented in this article speak for the feasibility of using multidimensional algorithms of stochastic conversion in high-performance industrial GPGPU systems. Due to the high degree of parallelism, the *Grasshopper* algorithm offers high bandwidth capacity in multithreaded systems, which is particularly relevant due to the massive reorientation of the main DPC equipment manufacturers in favor of hybrid solutions. For the special purpose systems used in EUC (Equipment Under Control), the hardware approach to the implementation of the *Grasshopper* GOST R 34.12-2015 is still an efficient solution.

REFERENCES

[1] A. Mitra, A. Kundu and C. Das, "Cost effective PRNG using ELCA: A BIST application," *2014 First International Conference on Automation, Control, Energy and Systems (ACES)*, Hooghy, 2014, pp. 1-6.

[2] L.I. Skiteva, A.G.Trofimov,V.L. Ushakov, D.G. Malakhov, B.M. Velichkovsky. MEG data analysis using the Empirical Mode Decomposition method. Biologically Inspired Cognitive Architectures (BICA) for Young Scientists.Volume 449 of the series Advances in Intelligent Systems and Computing pp 135-140.

[3] SCADA – Portal of Information Security. SecurityLab [Online]. Available: http://www.securitylab.ru/news/tags/SCADA/

[4] A.V. Boreskov et al .: Parallel computing on the GPU. Architecture and programming model of CUDA – Moscow: MSU publishing house, 2013. – 335 p.

[5] Intel Developer Zone: Intel Xeon Phi Coprocessor [Online]. Available: http://software.intel.com/mic-developer

[6] National Standard of the Russian Federation GOST R 34.12–2015 [Online]. Available: http://tc26.ru/en/standard/gost/GOST_R_34_12_2015_ENG.pdf

[7] S. A. Manavski CUDA compatible GPU as an efficient hardware accelerator for AES cryptography // ICSPC 2007 IEEE Los Alamitos, pp. 65-68, November 2007.

[8] Ortega, J. Trefftz, H. Trefftz, Parallelizing AES on multicores and GPUs, IEEE International Conference on Electro/Information Technology (EIT), 2011, vol., no., pp.1-5, 15-17 May 2011.

[9] I. V. Chugunkov *et al.*, "Three-dimensional data stochastic transformation algorithms for hybrid supercomputer implementation," *MELECON 2014 - 2014 17th IEEE Mediterranean Electrotechnical Conference*, Beirut, 2014, pp. 451-457

[10] I. V. Chugunkov, O. Y. Novikova, V. A. Perevozchikov and S. S. Troitskiy, "The development and researching of lightweight pseudorandom number generators," *2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW)*, St. Petersburg, 2016, pp. 185-189.

[11] Kris Gaj, Pawel Chodowiec. FPGA and ASIC Implementations of AES //Cryptographic Engineering - 2009, pp 235-294, 2009.

[12] Xinmiao Zhang, Keshab K. Parhi, High-Speed VLSI Architectures for the AES Algorithm, // IEEE transactions on very large scale integration (VLSI) systems, VOL. 12, NO. 9, 2004

[13] Spartan-6 FPGA Family – Xilinx official site [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/spartan-6/