NFV Chaining Technology in Hybrid Computing Clouds

Mikhail M. Rovnyagin[1], Sergey S. Varykhanov[2], Yury V. Maslov [3], Iuliia S. Riakhovskaia [4], Oleg V. Myltsyn[5]
National Research Nuclear University MEPhI (Moscow Engineering Physics Institute)
Moscow, Russia
[1]m.rovnyagin.2015@ieee.org, [2]masmx64@gmail.com, [3]ocean.ave@ya.ru, [4]rebzenok@yandex.ru, [5]oleg.myltsyn@yandex.ru

***Abstract* — Currently, the network functions virtualization technology (NFV) is becoming more widespread. Hardware devices are replaced with software-based cloud solutions. One way to increase the performance of virtual network functions (VNF) is to use hybrid computing technologies. In this paper, we propose ways to implement the most popular networking functions. The architecture of the orchestration system for hybrid systems and related features is explained. Also, the article presents the experimental results.**

Keywords — High performance computing; NVIDIA CUDA; service chaining; hybrid architecture; NFV; IDS.

I. INTRODUCTION

Today, telecommunications networks contain a large number of branded devices, often narrowly specialized: one device performs only one specific network function. This model requires unnecessary hardware costs, has little flexibility and dynamism. The network administrator is forced to support a variety of equipment produced by various manufacturers. This problem can be resolved with the concept of virtualization of network functions. First, virtualization allows you to run several network functions on one server, which will lead to more compact system, reduce power consumption and the number of network connections. Secondly, this approach simplifies network management, since the dependence on the manufacturer of the network device will disappear and it will be possible to configure the system in a programmatic manner.

It is known that virtualization carries with it certain performance losses compared to specialized hardware solutions, so it is logical to try to compensate or even overcompensate these losses, for example, by parallelizing the GPU.

This article discusses one of the approaches to implementing some common network functions designed for use in hybrid computing systems and presents an example of an orchestra that manages hybrid VNFs.

II. RELATED WORKS

Over the past few years, network virtualization (NFV) has quickly become a widely accepted approach to simplifying the organization of network applications. Managing virtualized network functions is easier than physical functions, because of their great flexibility.

In this article, to demonstrate the capabilities of NFV, combining virtual network functions (VNF) in combination with algorithms for encryption (AES), information compression (LZSS), video stream processing, and their application in intrusion detection systems are presented.

There is a vast amount of literature on the development of common network functions designed for use in hybrid computing systems. In the course of this study, papers [1], [2] have been studied in which the development of orchestrators for 5G networks has been presented and official documentation [3] on the development of orchestrators in NFV has been developed. In the authors' papers [4], [6], an overview of all the AES encryption modes is presented and the most optimal encryption option is selected when processing of AES blocks occurs in shared memory. In [5], the authors describe a method for implementing another symmetric block ciphering algorithm on the GPU, the Russian standard GOST R 34.12-2015.

In addition to the articles on the implementation of encryption algorithms on CUDA, there are other works, for example, on the dictionary compression algorithms, the document [7] was studied, which describes the technical specification of the lossless algorithm Deflate, using a combination of LZ77 and Huffman algorithms. When studying video stream processing, the article [13], whose authors describe the encoding and decoding of video in NFV using the T-NOVA framework, has become very useful.

III. ORCHESTRATOR

The orchestrator has the following functions:

- Connects to hosts and deploys virtual machines with VNF to them. A different number of VNFs may be deployed on the same host depending on the number of CPU cores and the availability of a GPU, so the concept of a slot is introduced. In fact, a slot is a cell in which a virtual network function can be placed. In this work, 2 types of slots are used: CPU-only slot (2 CPU cores and 0 GPU-devices) and CPU-GPU slot (2 CPU cores and 1 GPU-device). In a CPU-only slot, one virtual network function that uses the processing power of the CPU is allowed. In the CPU-GPU slot, it is allowed to place two VNFs: one CPU-only and one VNF GPU. For example, if there are 8 CPU cores and 2 GPUs on the host, then there will be 2 CPU-GPU slots and 2 CPU-only slots on this host;
- Take control of existing VNF, which could remain from past "chains";
- Performs diagnostics of VNF. In the event of a failure, the orchestra repairs the damaged VNF;
- Collects statistics of the work of VNF.

Below is a diagram of the DB of the orchestrator.

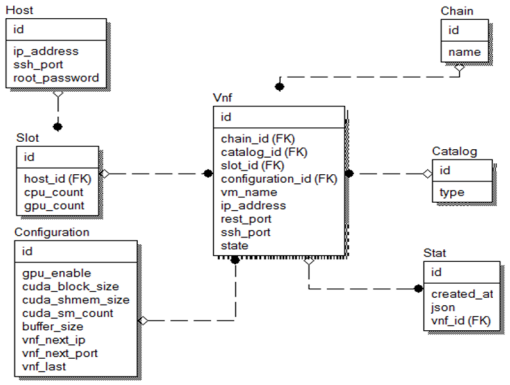

Fig. 1. DB scheme of the orchestrator

Purpose of tables:

- Host. The table contains information about the hosts on which the virtual network function can be deployed, namely the IP address, the user's root password and the ssh protocol port;
- Slot. The table contains information about the slots that the host has, namely the number of CPUs and GPUs of the virtual network functions that can be located in this slot;
- Catalog. The table contains information about the types of virtual network functions (compression, encryption, etc.) and which jar file should be run in the virtual machine of the functional node;
- Chain. Contains information about the "chains" of virtual network functions;
- Configuration. The table contains information on the settings of virtual network functions. The table is similar to the Configuration table from the database of the functional node, except for the activity flag (it does not exist in this database);
- Vnf. The table contains information about the virtual network functions themselves, namely: the name of the virtual machine, the IP address and port of the ssh protocol, the port of the REST interface, the slot number, the directory, the configuration, and the VNF state (active, fail,) The state is coded as follows: 0 - VNF is working normally, 1 - VNF has not responded to the check request once, 2 - twice, 3 - three times, 5 - VNF has just been activated or restored and has not yet been picked up by the controlling node;
- Stat. In this table, the statistics of the functioning of virtual network functions in the format of the statistics collection time are accumulated - a statistical record.

IV. AES NFV

The AES algorithm is a symmetric block cipher algorithm. The block size is 128 bits. Each block is a two-dimensional 4x4 byte array, called a state. The algorithm consists of four main functions: AddRoundKey, MixColums, ShiftRows and SubBytes. The basic idea when parallelizing the AES algorithm is that each block is encrypted by its computational flow. The first step is to load the text to be encrypted into the global memory of the GPU. After that, the optimal number of CUDA-blocks is calculated, depending on the GPU used, and the CUDA-kernel call. The incoming text is represented in the GPU as a one-dimensional array.

In the kernel, the data is divided into pieces equal to THREADS_PER_BLOCK (the number of threads in the CUDA block) * AES_SQUARE_SIZE (AES block size in bytes) bytes, and in parts are loaded into shared memory, and after processing they are returned back to the global one. Access to shared memory is much faster than to global memory, so this approach speeds up the execution of encryption. Each thread is allocated 1 thread for processing. According to [6], this is the most optimal option. In the case where there are more blocks than streams, the streams with the number i after processing the block with the number i go to the processing of the block with the number (i + total number of threads), and so on. Below are graphs of the dependence of the bandwidth of VNF and the encryption time on the size of the source text.

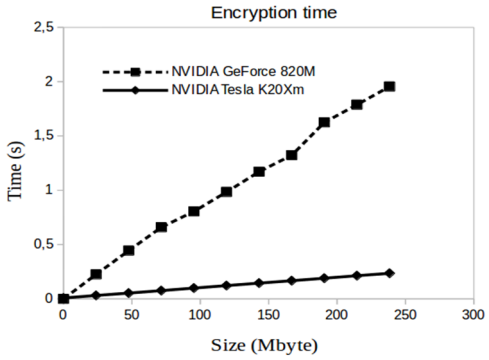

Fig. 2. The dependence of the encryption time on the size of the source text

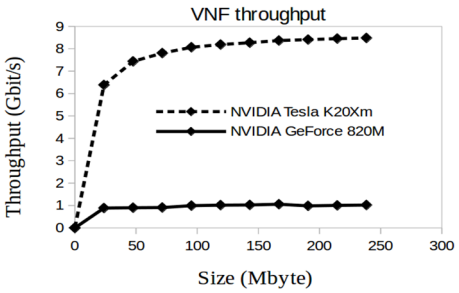

Fig. 3. The dependence of the bandwidth on the size of the source text

V. The Dictionary Compression Algorithms Implementation Features in Hybrid Systems

As the basis for the implementation was taken one of the most popular dictionary algorithms for lossless compression - LZSS. The previous version of this algorithm LZ77 is used for data compression in the DEFLATE method [7], which is the main one in the widely used gzip utility [8].

An arbitrary sequence of symbols of some alphabet is fed to the input of the algorithm. The algorithm works in such a way

that it replaces the sequences of characters that were already encountered earlier in the input data sequence, by reference to previously encountered sequences.

LZSS has a dictionary and a buffer of specified sizes, which are implemented as ring buffers. The algorithm traverses the characters of the incoming sequence. Each character is added to the end of the buffer, after which a sequence is searched for from the buffer in the dictionary. If the dictionary has managed to find a match of sufficient length, then a link to the dictionary is written to the output sequence. Otherwise, the symbol itself is output. After that, the encoded character or sequence is written to the end of the dictionary.

It should be noted that the size of the link depends on the size of the buffer and the dictionary. The size of the link is [*log2 (n) + log2 (m)*], where m is the size of the buffer, n is the size of the dictionary. It follows that an increase in the size of the buffer or dictionary will not always lead to an increase in the compression ratio, due to the increase in the size of the link. Before the symbol or the reference, a bit, "0", for the character and "1" for the reference is written.

The decoding process is as follows: First, the first bit is read, which determines whether it is a symbol or a link to the dictionary. Then, the symbol or sequence from the dictionary is read by reference. After that, the read data is written into the dictionary and the output sequence. The most difficult task for compression is to find the desired sequence in the dictionary. To solve this problem on CUDA, the most effective solution was the hash table of the lists. The key to the list hash table is the first few characters of the sequence. The space for the table is highlighted immediately. This imposes serious restrictions on the number of characters in the key. In the CPU implementation, you can use up to three characters, in the GPU it's better to limit to two.

Each entry in the table refers to the beginning and end of a doubly linked list of references to the dictionary. Each sequence begins with the symbols from the key. The maximum length of the list is limited by a constant. This avoids problems with too many similar sequences that fall into one list. New nodes are added to the end of the list. The same words are ignored. Due to the pointer to the end of the list, this operation takes O(1) of the time. The deletion also takes O(1) of the time, because the element to delete is the oldest, and therefore the very first in the list for its key. The outline of the hash table is shown in Fig. 3.

The implementation of the algorithm should be able to encode and decode files of any size. It follows that the file should be read in parts. When using CUDA technology, the encoding process is as follows: The source file is divided into fragments of a fixed length. First, a number of fragments from the file are read.

Each fragment has its own CUDA thread. For each thread, their own buffer and dictionary are highlighted. Due to this, each fragment is processed independently of the others. After the processing is complete, the resulting output sequences are glued together into a single file. And the process is repeated again, until the entire input file is processed. The number of simultaneously processed fragments and their size is limited by the amount of video memory.

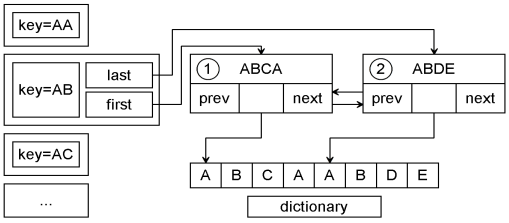

Fig. 4. Hash Table Structure

Unfortunately, parallelization at the operation level of the algorithm itself is inefficient. Most of the operations must be performed sequentially, and you cannot allocate sufficiently large sequences of operations for their parallel processing.

VI. NFV IDS Snort with Intel Xeon Phi

Intrusion Detection Systems (IDS) is a network software designed to detect suspicious traffic in a network, defined by a set of rules created by the network administrator. OpenMP is a set of environment variables needed to manage the characteristics of executable code, and compiler directives designed to control areas of code that use shared memory.

Compilation of heterogeneous applications for the Intel Xeon Phi architecture is done using the offload mode. During the execution of the program, the binary executable code is loaded onto the coprocessor and the libraries are initialized, after which the offload code is called.

As the used intrusion detection system, IDS Snort was selected. When profiling the application, it was found that the greatest time in processing the package (about 49%) was taken by searching patterns in it. Moreover, the detection module captures about 80% of the CPU resource, unlike the other modules of the program, in which the CPU is practically not loaded. The cycle of processing the packet with the percentage of time spent for each module is shown in Figure 5.

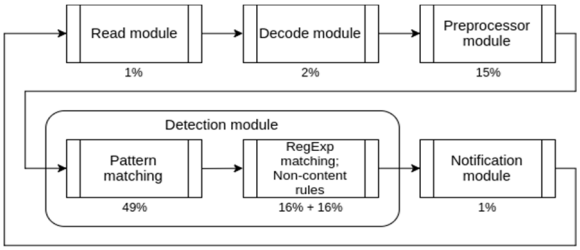

Fig. 5. The processing cycle of the packet with the percentage of time spent per module.

For parallelization, the algorithm for searching for Akho-Korasik patterns was chosen, which is one of the most popular search algorithms. As a result of finalizing the source code of the program for the purpose of executing parallelized sections on the Intel Xeon Phi coprocessor, the experimental graph of the dependence of the total processing time of packets on the size of the dump file with processed packets presented in Figure 6 was obtained.

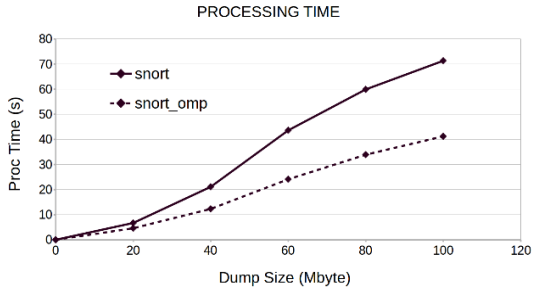

Fig. 6. Graph of the total processing time of packets on the size of the dump file with processed packets

VII. THE VIDEO STREAM DETECTOR VNF

Modern technologies require the solution of a wide range of computer vision problems, including, in particular, a set of tasks related to orientation in external space (for example, the problem of simultaneous localization and mapping). Together with NFV, network services and video related functions, especially such as video processing functions, are of paramount importance, given that currently the video content takes up a lot of global network traffic and is growing every year. VNFs related to video, in particular those related to video stream processing (pattern search / motion tracking), are usually computationally-oriented and often require rather strict real-time restrictions.

In this paper, we analyzed and discussed the use of the GPGPU module to accelerate VNF to monitor motion in a video stream. To recognize the movement of objects in the video stream, the OpenCV computer vision library is used. The choice of this library is due to a wide range of algorithms for working with computer graphics and pattern recognition.

Video stream processing takes place on a graphics processor, which, by its architecture, is perfectly suitable for parallel processing. The main technology for processing frames on the GPU is NVIDIA CUDA, which allows sending blocks of information to be processed by the GPU of the video card. To recognize the images, a pre-trained object discovery network was used from the OpenCV dnn module. The developed architecture of a particular VNF video processing unit was used to obtain results in a number of experiments to determine the performance of its operation. The results clearly show that the use of the GPU can significantly increase the performance of video processing in VNF.

VIII. CONCLUSIONS

In this paper, we conducted an experimental study of such virtual network functions as AES encryption, information compression (LZSS), video stream processing and pattern search in intrusion detection systems, as well as the VNF deployment process management system. The developed VNF supports the launch in both a hybrid and a CPU-only environment. Functional testing of VNF was carried out both as a separate module and as part of a chain. Testing allowed to make sure that the developed system works correctly. Load testing of the virtual network function in a hybrid environment was carried out. The testing showed that a significant increase in the performance of the developed VNF was achieved: when using the NVIDIA Tesla K20Xm GPU, encryption is performed about 60 times faster than using the Intel Core i5-4210U CPU.

Thus, the results proved the possibility of efficient use of graphics processors to accelerate virtual network functions. This approach turned out to be well applicable in implemented network functions, since they have a high degree of parallelism.

REFERENCES

[1] Gerő B., Jocha D., Szabó R., Czentye J., Haja D., Németh B., Sonkoly B., Szalay M., Toka L., Cano C. J. B., Murillo L. M. C. The Orchestration in 5G Exchange – a Multi-Provider NFV Framework for 5G Services. 2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN). Berlin, Germany, pp. 1-2, 6-8 Nov. 2017. DOI: 10.1109/NFV-SDN.2017.8169865.

[2] C. J. Bernardos, B. P. Ger, M. Di Girolamo, A. Kern, B. Martini, and I. Vaishnavi, “5gex: realising a Europe-wide multi-domain framework for software-defined infrastructures,” Trans. Emerging Tel. Tech., vol. 27, no. 9, pp. 1271–1280, Sep. 2016.

[3] Network Functions Virtualisation (NFV). Management and Orchestration. Network Functions Virtualisation (NFV) ETSI Industry Specification Group (ISG). Dec. 2014

[4] Skitev A. A., Rovnyagin M. M., Martynova E. N., Zvyagina M. I., Shelopugin K., D., Chernova A. A. Methods for implementation of Pseudo-Random Number Generator based on GOST R 34.12-2015 in hybrid CPU/GPU/FPGA high-performance systems. 2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), St. Petersburg, Russia, pp. 555 - 559, 1-3 Feb. 2017.

[5] Ortega, J. Trefftz, H. Trefftz, "Parallelizing AES on multicores and GPUs", IEEE International Conference on Electro/Information Technology (EIT), pp. 1-5, May 2011.

[6] Keisuke Iwai , Naoki Nishikawa, Takakazu Kurokawa, “Acceleration of AES encryption on CUDA GPU“, International Journal of Networking and Computing, Volume 2, Number 1, pages 131–145, January 2012.

[7] “RFC 1951 DEFLATE Compressed Data Format Specification version 1.3” May 1996.

[8] Gailly, J.-L., and Adler, M., GZIP documentation and sources, available as gzip-*.tar in ftp://prep.ai.mit.edu/pub/gnu/.

[9] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search”, Commun., ACM, vol. 18, no. 6, (1975), pp. 333-340.

[10] L. Schaelicke; T. Slabach; B. Moore; and C. Freeland. (2003). Characterizing the performance of network intrusion detection sensors. In Proceedings of Recent Advances in Intrusion Detection (RAID 2003), September 2003.

[11] H. Song; T. Sproull; M. Attig; and J. Lockwood. (2005). Snort offloader: A reconfigurable hardware NIDS Filter. International conference on Field programmable logic and applications, 493-498.

[12] “Snort.TheOpenSourceNetworkIntrusionPreventionandDetectionSystem ”, http://www.snort.org.

[13] Comi P., Secondo Crosta P., Beccari M., Paglierani P., Grossi G., Pedersini F., Petrini A. Hardware-accelerated High-resolution Video Coding in Virtual Network Functions. 2016 European Conference on Networks and Communications (EuCNC), Athens, Greece, pp 32 – 36, 27-30 June 2016.

[14] Rosebrock A. Practical Python and OpenCV + Case Studies. Available at: https://www.pyimagesearch.com/practical-python-opencv/ (accessed 10 November 2017).

[15] Solem J. E. Programming Computer Vision with Python: Tools and algorithms for analyzing images. O'Reilly Media, 2012, 264 p.