PREDICTING PERFORMANCE OF HETEROGENEOUS AI SYSTEMS WITH DISCRETE-EVENT SIMULATIONS

Vyacheslav Zhdanovskiy
Institute for Information
Transmission Problems, RAS
Bolshoy Karetny per. 19, Moscow, 127051, Russia;
Moscow Institute of Physics and Technology
(National Research University)
Institutskiy per. 9, Dolgoprudny, 141701, Russia
E-mail: zhdanovskiy.vd@phystech.edu

Lev Teplyakov
Institute for Information
Transmission Problems, RAS
Bolshoy Karetny per. 19, Moscow, 127051, Russia
E-mail: teplyakov@visillect.com

Anton Grigoryev
Institute for Information Transmission Problems, RAS
Bolshoy Karetny per. 19, Moscow, 127051, Russia
E-mail: me@ansgri.com

KEYWORDS

Heterogeneous computing; Discrete-event simulations; Artificial intelligence; Video analytics; Software architecture

ABSTRACT

In recent years, artificial intelligence (AI) technologies have found industrial applications in various fields. AI systems typically possess complex software and heterogeneous CPU/GPU hardware architecture, making it difficult to answer basic questions considering performance evaluation and software optimization. Where is the bottleneck impeding the system? How does the performance scale with the workload? How the speed-up of a specific module would contribute to the whole system? Finding the answers to these questions through experiments on the real system could require a lot of computational, human, financial, and time resources. A solution to cut these costs is to use a fast and accurate simulation model preparatory to implementing anything in the real system. In this paper, we propose a discrete-event simulation model of a high-load heterogeneous AI system in the context of video analytics. Using the proposed model, we estimate: 1) the performance scalability with the increasing number of cameras; 2) the performance impact of integrating a new module; 3) the performance gain from optimizing a single module. We show that the performance estimation accuracy of the proposed model is higher than 90%. We also demonstrate, that the considered system possesses a counter-intuitive relationship between workload and performance, which nevertheless is correctly inferred by the proposed simulation model.

INTRODUCTION

In recent years, there has been a significant growth of interest in machine learning and artificial intelligence (AI) technologies. Analysts expect the AI market to grow to more than USD 660 billion by 2028 (GrandViewResearch.com 2021). Nowadays, AI technologies are used in various fields, such as urban services (Wang and Sng 2015), retail (Weber and Schütte 2019), medicine (Chen et al. 2021), etc.

One of the key technologies that allowed for the progress is deep learning. In particular, deep neural networks made it possible to achieve almost human-like object recognition quality in problems like image classification (Rawat and Wang 2017), object detection (Arnold et al. 2019; Liu et al. 2020) and image segmentation (Guo et al. 2018). In order to achieve this accuracy and still provide reasonable recognition speed, deep neural networks have to exploit special hardware providing fast matrix multiplication, most commonly — GPUs. This poses a problem for software developers, because they have to design the architecture of their AI-based applications with heterogeneous CPU/GPU computations in mind.

In detail, software developers have to utilize the parallelism provided by modern multi-core CPUs along with the GPU acceleration. Meanwhile the GPU is used for the neural network inference, the CPU is used to prepare the neural networks's input and postprocess its output, run various computer vision algorithms like tracking, localization, keypoint detection. The CPU also handles all the additional modules delivering AI results to the end user — API calls, business logic, database management, etc. In order to deliver on all these tasks on advanced heterogeneous hardware with the given time, memory and energy constraints, software developers have to design rather complex architectures (Sutter et al. 2005).

The complex software architecture, in turn, complicates performance evaluation and software optimization of such systems (Voss et al. 2019c). In particular,

Communications of the ECMS, Volume 36, Issue 1,
Proceedings, ©ECMS Ibrahim A. Hameed, Agus Hasan,
Saleh Abdel-Afou Alaliyat (Editors) 2022
ISBN: 978-3-937436-77-7/978-3-937436-76-0(CD) ISSN 2522-2414

it is hard to locate the bottleneck module and to predict, how its optimization will affect the performance of the entire system. It may lead to lots of human and financial resources as well as time resources being spent on optimization without any significant increase in performance of the entire system. For a new module to be designed and implemented, it might be hard to map the system's constraints to the constraints of a specific module. It may turn out - after the resources are spent on implementing a new module and its integration into the system! - that the module results in unaffordable slowdown of the system.

An elegant way to avoid the aforementioned problems is to design a simulation model of the system and infer the feasibility of optimizations on the model prior to implementing them in the code. Discrete-event simulations of software have been used before in other areas, for example, planning, evaluation and optimization of Hadoop clusters (Bian et al. 2014; Wang et al. 2014; Liu et al. 2016; Chen et al. 2016). However, to the best of our knowledge, no one yet has tried to apply this approach to applications with heterogeneous CPU/GPU computing, such as AI systems. Moreover, such a simulation model can be used for other tasks like capacity planning. This can be achieved by evaluating the model on a collected set of suitable hardware configurations (Korobov et al. 2020).

In terms of performance modelling, different AI applications have their own unique specialties. The key difference is which hardware components to consider. For example, video analytics (Wang and Sng 2015) and self-driving cars (Badue et al. 2021) are mostly CPU/GPU intensive, meanwhile AI on pure-cloud solutions also rely heavily on the network performance, which requires to consider the I/O subsystem in the model.

In this work, we consider a video analytics system as a typical example of AI application utilizing CPU/GPU computing. The possibility of applying our research to other AI applications is discussed in Section III. While the GPU is used to infer the neural network responsible for complex object detection tasks such as human detection, the CPU is used for preprocessing the video frames, postprocessing the neural network output and running classic computer vision algorithms such as ORB keypoint detector and descriptor (Rublee et al. 2011). The considered system has the following functionality:

- processing video feed from multiple video cameras;
- person detection using YOLOv4 (Bochkovskiy et al. 2020);
- person 3D localization;
- single- and multi-camera person tracking.

We design a discrete-event simulation model which is low-cost both in terms of its development and simulation speed and can easily be adopted by the software developers. We use it to estimate:

1. the performance scalability with the increasing number of video cameras;
2. the performance gain from optimizing a single system module;
3. the performance impact caused by integrating of a new module in the system.

The rest of the paper is structured as follows: Section I describes the software architecture of the video analytics system, Section II provides a description of the proposed simulation model, Section III contains numerical results. Section IV concludes the paper.

SYSTEM DESCRIPTION

Flow graph paradigm

A common design pattern to efficiently implement parallelism in heterogeneous and parallel systems is to use the flow graph paradigm (Grigoryev et al. 2015; Badue et al. 2021; Huang et al. 2021). With it, the algorithm is represented as a data flow graph (Voss et al. 2019b), see Fig. 1. Each graph node (vertex) receives a message from its predecessors, processes it and broadcasts the output to the successors. Input and output nodes are the exception:

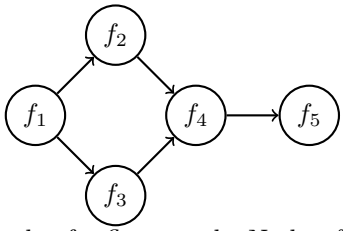

Fig. 1: Example of a flow graph. Nodes f_2 and f_3 can process messages broadcasted from f_1 in parallel. $f4$ can process messages from f_2 and f_3 independently or wait until messages from both nodes are present — it depends on the desired behaviour.

- input nodes either produce an input message by itself or receive it somewhere from outside the graph;
- output nodes do not broadcast the result, but instead store it or deliver it outside the graph.

The considered video analytics system has multiple places where parallelism can be utilized. In particular:

- frames from different video streams can be processed in parallel;
- a single frame can be processed in parallel by multiple data-independent detectors, for example, by the neural network and the keypoint detector.

Flow graphs allow to efficiently utilize these types of parallelism by executing different nodes of the graph in parallel at the same time.

There are multiple frameworks implementing this paradigm. Notable examples include oneTBB[1] and Taskflow[2] (Huang et al. 2021). In this work, we use oneTBB.

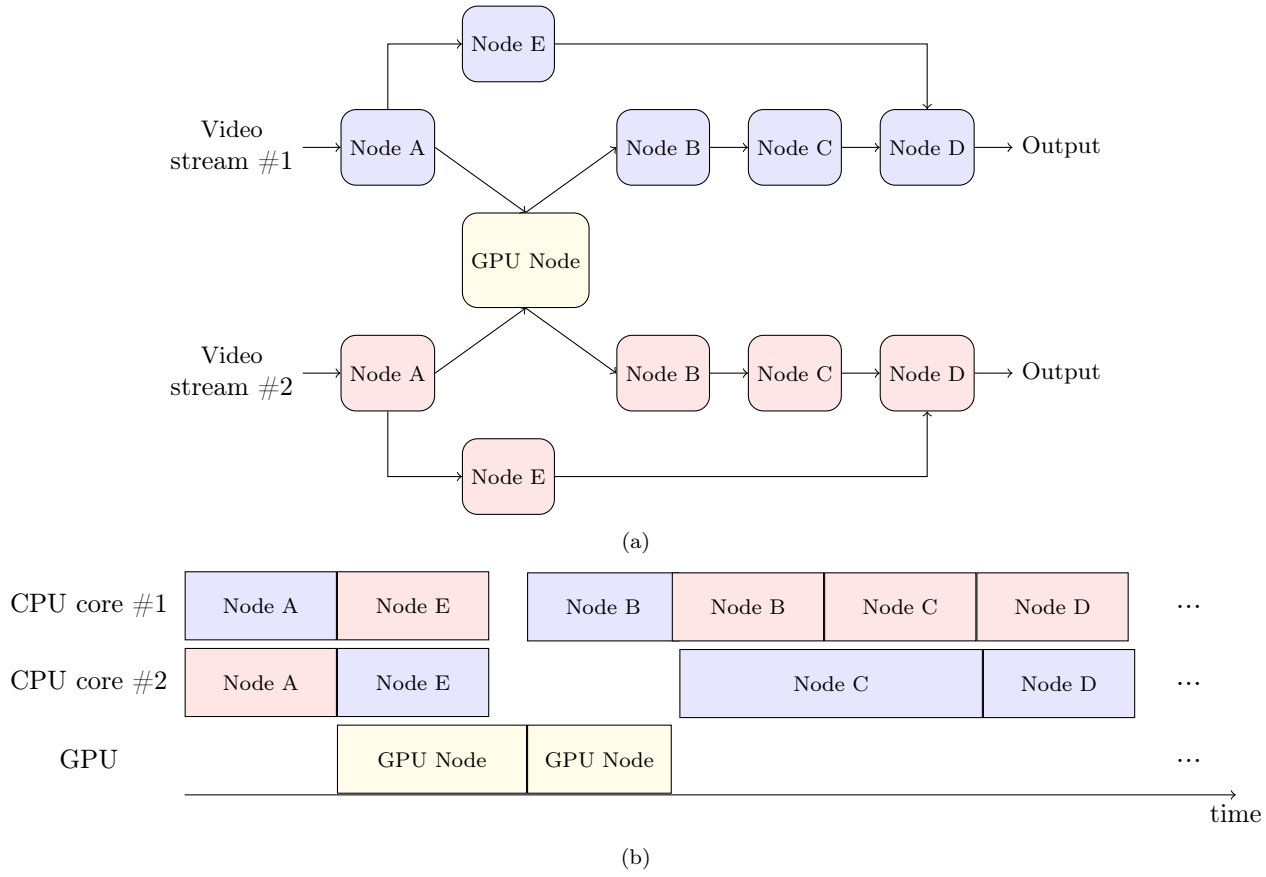

Fig. 2: Software architecture of the studied video analytics system represented as a flow graph (a). Sample timing diagram of the system execution (b). Note than `Node E` is processed in parallel with the `GPU Node`, meanwhile `Node B` has to wait until `GPU Node` finishes processing its task. `Node E` is the CPU-intensive module we consider in Section III. It is turned off for all experiments except "Integration of a new module in the system".

Software architecture of the video analytics system

In this section, we provide a brief description of the software architecture used in the considered video analytics system.

Overall, the whole system is represented by a flow graph, see Fig. 2a. Each video stream from a single camera is represented by a separate graph component. We shall notice that we deliberately show a less detailed version of the real graph for the sake of simplification. All graph components are executed in a single thread pool.

Meanwhile many flow graph nodes can be executed in parallel, some of them can not because their functions are not thread safe. In particular, this is true with the neural network nodes. Because neural networks are usually implemented in a way (NVIDIA 2022) that each neural network instance can be executed on GPU by only one context at a particular moment, multiple video streams have to put their tasks for the neural network inference in a shared queue and then wait for their completion, see Fig. 2b. We implement this functionality with oneTBB's *async nodes* (Voss et al. 2019a).

SIMULATION MODEL

In this section, we describe the details of the proposed simulation model. Overall, the system flow graph has a near-direct representation in the model. We implemented the model in Python using the SimPy[3] library.

Algorithm 1 Flow graph node simulation for a basic node

Q — `input queue (from predecessors)`
S — `output queue (to successors)`
C — `pool of free CPU cores`
P — `distribution of the node's running time`
while `not all frames are processed` **do**
  $m \leftarrow$ `Q.pop() (blocks if` Q `is empty)`
  $c \leftarrow$ `C.pop() (blocks if` C `is empty)`
  $t \sim P$
  `wait(`t`)`
  S`.push(`m`)`
  `C.push(`c`)`
end while

[1]Formerly TBB; more details at: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html`

[2]More details at: `https://taskflow.github.io/`

[3]More details at: `https://simpy.readthedocs.io/en/latest/index.html`

Algorithm 2 Flow graph node simulation for a GPU async node

```
Q — input queue (from predecessors)
S — output queue (to successors)
C — pool of free CPU cores
G — GPU
P — distribution of the node's running time
while not all frames are processed do
    m ← Q.pop() (blocks if Q is empty)
    c ← C.pop() (blocks if C is empty)
    G.lock() (blocks if G is busy)
    C.push(c)
    t ∼ P
    wait(t)
    G.release()
    c ← C.pop() (blocks if C is empty)
    S.push(m)
    C.push(c)
end while
```

The source code of the implementation is available at GitHub[4]. It contains only ≈ 400 LoC of Python (including the profiling trace parsers), meanwhile the real system contains ≈ 15000 LoC of C++ (not including the dependencies).

We consider a CPU with N logical cores in the model. Each flow graph node waits until the input message is present, see Algorithm 1. Then it waits until there is a free CPU core, then the node occupies the CPU core for the execution time. The execution time is sampled from independent empirical distributions measured by profiling the real system. Input messages that have not yet been processed are stored in a queue. Effectively, this represents the work of oneTBB's thread pool.

The GPU neural network node is modelled in a slightly different way, see Algorithm 2. Like a basic flow graph node, it waits for an input message and a CPU core. Then it waits until the GPU is free, then locks it for the execution time, while yielding the CPU core for some another flow graph node. When the GPU is done with processing the message, the node waits again for a free CPU core in order to broadcast its output to the successors. Effectively, this represents the mechanism shown in Fig. 2b.

The flow graph nodes' execution times are sampled from empirical distributions measured by profiling the real system. Fig. 3 contains an example of such distribution for the neural network inferencer. The oneTBB flow graph nodes are profiled using Intel Flow Graph Tracer and Flow Graph Analyzer (Voss et al. 2019c). The non-oneTBB activities like the video decoding and the neural network inference are sampled by our in-house tracing library.

We deliberately use flow graphs in the model instead of some well-known formalisms like Petri Nets (Peterson 1977) and Queuing Networks (Chandy et al. 1975). It makes designing the simulation model easier because we can explicitly use the same graph structure and synchronization primitives used as the real system. This approach also has potential for the model to be automatically generated by parsing the profiling traces. Moreover, our approach is easier to be adopted by the software developers who are unlikely to be experts in modelling and simulation.

[4]More details at: `https://github.com/iitpvisionlab/heterogeneous-ai-system-simulator`

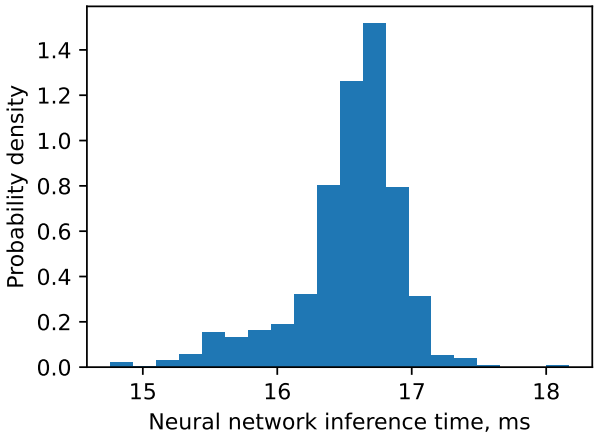

Fig. 3: Empirical distribution of neural network inference time.

We shall notice that we do not consider the overhead caused by communications between the nodes, as the execution time of each node significantly exceeds the average communication time in our case. However, our simulation model can easily be upgraded by introducing additional timings between consecutive graph nodes. This will allow to model other AI applications like AI on pure-cloud solutions, where there is significant communication overhead cause by the network. The communication overhead can also be important in self-driving cars, where the TCP/IP stack is often used for communication even within a single machine, e.g. the ROS framework(Quigley et al. 2009).

EVALUATION

In this section we evaluate the proposed simulation model. In order to do it, we compared the performance metrics predicted by the simulator with those measured on the real video analytics system. In our case, the performance metrics is the average frames per second (FPS) per each video stream.

The hardware specifications are listed in Table 1.

TABLE 1: Testbed hardware specifications

Parameter	Value
CPU	Intel Core i7-7800X, 3.5 GHz, 6 Cores, 12 Threads
GPU	NVIDIA GeForce GTX 1080 Ti, 11 GB VRAM
RAM	64 GB

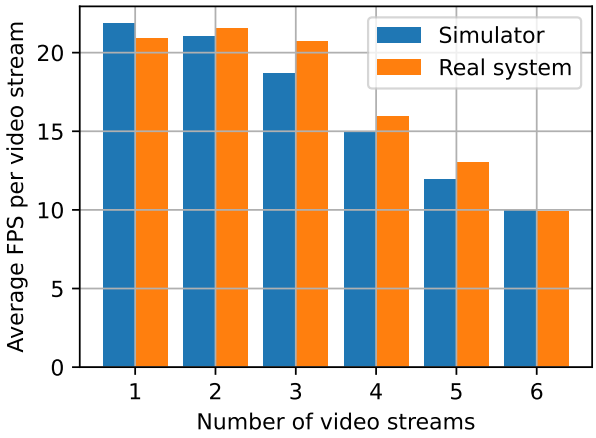

Fig. 4: FPS per video stream depending on the number of video streams.

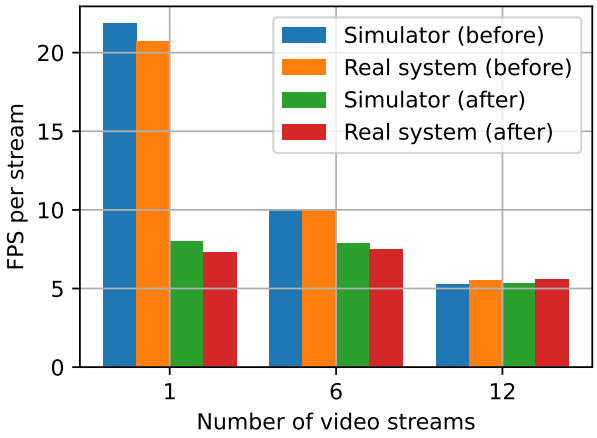

Fig. 5: Performance impact caused by integrating a new CPU-intensive module

Estimating the model accuracy and the system scalability

In this experiment, we varied the number of video streams in order to estimate the accuracy of the simulation model. The results presented in Fig. 4 demonstrate, that the performance prediction error rate of the simulation model is less than 10%, which is accurate enough to rely on the model's prediction in planning the scalability of the system. It is also lower than the error rate threshold of 20% used in a related work (Bian et al. 2014).

On small number of video streams (up to three) the system experiences almost no performance drop due to efficient parallelism. However, when the number of video streams increases, the performance drops significantly, because the neural network becomes the bottleneck that hinders the parallelism: threads stand idle waiting for a task to execute. When the number of video streams exceeds the number of CPU logical cores, the performance drops even more because there is not enough free threads to execute appearing tasks.

Integration of a new module in the system

To study an impact of a new module on the system, we conducted the following experiment. We added a CPU-intensive module (`Node E` in Fig 2a) in the system and measured the overall slowdown on both the real model and the simulator. The results, see Fig 5, show that the module produces approximately 2.5X slowdown on 1 video stream, meanwhile producing no noticeable slowdown on 12 video streams. The result is counter-intuitive: the increase in workload makes the overhead of the module unnoticeable instead of it growing linearly with the workload. This occurs because the neural network, while being comparably fast on one stream, becomes the bottleneck on high number of video streams.

This experiments demonstrates, that for systems with high degree of parallelism the impact of adding a new module or changing the existing one on system's performance could be nontrivial.

Estimating the feasibility of software optimizations

In this section we study the feasibility of optimizations in the considered system with the help of the proposed simulation model by the example of the following optimization.

The quality of service (QoS) of the considered video analytics system highly depends on tracking algorithms. Tracking accuracy, in turn, depends on many parameters of the algorithm. In order to improve QoS, it is needed to find the optimal values of the parameters.

The simplest way to find the optimal values is the grid search. However, for the high-dimensional search space of parameters it is computationally unfeasible. More sophisticated optimization methods like Bayesian optimization allow to mitigate but not fully alleviate this problem, still requiring many runs of the system.

Fortunately, all the required runs use the same input data with different values of parameters. Therefore, in theory, it is possible to save a lot of time by caching neural network predictions and reusing them on each run instead of inferring the real network.

However, the cache may not provide the desired performance gain, because some other module can become the bottleneck. Therefore, prior to implementing and testing the optimization (which takes about one week for a software engineer), we study the feasibility of the optimization on the simulation model (which takes a couple of hours).

First, we used the developed simulation model to estimate the performance gain of implementing the neural network cache without considering the overhead of the cache itself. Effectively, we set the execution time of the GPU node to zero. The speed-up appeared to be 13.8x (Table 2, "ideal" cache).

Then we implemented a LevelDB-based[5] cache module detached from the system, benchmarked its performance and used the data in simulation model to ac-

[5]More details at: `https://github.com/google/leveldb`

TABLE 2: Overall system speedup on 6 video streams from implementing the cache

Experiment	Overall system speedup
Real system	11.3x
Simulator, "ideal" cache	13.8x
Simulator, "real" cache	12.0x

count for the overhead. The speed-up appeared to be 12.0x (Table 2, "real" cache).

Finally, we integrated the developed cache module into the system. The real achieved speed-up was equal to 11.3x (Table 2, real system), close to the predicted value of 12.0x.

The experiment demonstrates, that with a negligible overhead in time spent on simulating each step of the implementation, it allows to correctly estimate the limit of optimization's impact. Moreover, it could save a lot of time, if it had emerged that a specific optimization step is not worth implementing.

CONCLUSIONS

In this work, we proposed a discrete-simulation model to predict the performance of a heterogeneous CPU/GPU video analytics system. The proposed model can easily be adopted by the software developers who are not experts in simulation and modelling. We showed that the accuracy of performance estimation using the proposed system is higher than 90% in each experiment.

We used the simulation model to predict performance scale with workload; to estimate the impact of a new module on the whole system, which demonstrated counter-intuitive results yet correctly predicted by the simulator; to predict the feasibility of an optimization.

We believe such a simulation model should become a workplace tool for software designers and it could save lots of resources by easily inferring the feasibility of optimizations and modifications prior to doing some costly work on a real system.

Possible future research includes taking into consideration the hardware configuration to predict, for example, an optimal hardware price — quality of service trade-off of a system.

REFERENCES

Arnold, Eduardo; Omar Y Al-Jarrah; Mehrdad Dianati; Saber Fallah; David Oxtoby; and Alex Mouzakitis. 2019. "A survey on 3D object detection methods for autonomous driving applications." *IEEE Transactions on Intelligent Transportation Systems*, 20(10):3782–3795.

Badue, Claudine; Rânik Guidolini; Raphael Vivacqua Carneiro; Pedro Azevedo; Vinicius B Cardoso; Avelino Forechi; Luan Jesus; Rodrigo Berriel; Thiago M Paixao; Filipe Mutz; et al. 2021. "Self-driving cars: A survey." *Expert Systems with Applications*, 165:113816.

Bian, Zhaojuan; Kebing Wang; Zhihong Wang; Gene Munce; Illia Cremer; Wei Zhou; Qian Chen; and Gen Xu. 2014. "Simulating Big Data clusters for system planning, evaluation, and optimization." In *2014 43rd International Conference on Parallel Processing*, 391–400.

Bochkovskiy, Alexey; Chien-Yao Wang; and Hong-Yuan Mark Liao. 2020. "YOLOv4: Optimal speed and accuracy of object detection." *arXiv preprint arXiv:2004.10934*.

Chandy, K. Mani; Ulrich Herzog; and Lin Woo. 1975. "Parametric analysis of queuing networks." *IBM Journal of Research and Development*, 19(1):36–42.

Chen, Jianguo; Kenli Li; Zhaolei Zhang; Keqin Li; and Philip S Yu. 2021. "A survey on applications of artificial intelligence in fighting against COVID-19." *ACM Computing Surveys (CSUR)*, 54(8):1–32.

Chen, Qian; Kebing Wang; Zhaojuan Bian; Illia Cremer; Gen Xu; and Yejun Guo. 2016. "Cluster performance simulation for Spark deployment planning, evaluation and optimization." In *International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, 34–51.

GrandViewResearch.com. 2021. "Artificial intelligence market size, share trends analysis report by solution, by technology (deep learning, machine learning, natural language processing, machine vision), by end use, by region, and segment forecasts, 2021 - 2028." `https://www.grandviewresearch.com/industry-analysis/artificial-intelligence-ai-market`. Accessed: 18.01.2022.

Grigoryev, Anton; Timur Khanipov; Ivan Koptelov; Dmitry Bocharov; Vassily Postnikov; and Dmitry Nikolaev. 2015. "Building a robust vehicle detection and classification module." In *ICMV 2015*.

Guo, Yanming; Yu Liu; Theodoros Georgiou; and Michael S Lew. 2018. "A review of semantic segmentation using deep neural networks." *International journal of multimedia information retrieval*, 7(2):87–93.

Huang, Tsung-Wei; Dian-Lun Lin; Chun-Xun Lin; and Yibo Lin. 2021. "Taskflow: A lightweight parallel and heterogeneous task graph computing system." *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1303–1320.

Korobov, Nikita; Oleg Shipitko; Ivan Konovalenko; Anton Grigoryev; and Marina Chukalina. 2020. "SWaP-C based comparison of onboard computers for unmanned vehicles." In *ER(ZR)-2019*, volume 154, 573–583 (Springer, Singapore, 2020).

Liu, Jun; Bianny Bian; and Samantika Subramaniam Sury. 2016. "Planning your SQL-on-Hadoop deployment using a low-cost simulation-based approach." In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 182–189.

Liu, Li; Wanli Ouyang; Xiaogang Wang; Paul Fieguth; Jie Chen; Xinwang Liu; and Matti Pietikäinen. 2020. "Deep learning for generic object detection: A survey." *International journal of computer vision*, 128(2):261–318.

NVIDIA. 2022. "NVIDIA TensorRT Documentation." `https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#threading`. Accessed: 28.01.2022.

Peterson, James L. 1977. "Petri nets." *ACM Computing Surveys (CSUR)*, 9(3):223–252.

Quigley, Morgan; Ken Conley; Brian Gerkey; Josh Faust; Tully Foote; Jeremy Leibs; Rob Wheeler; Andrew Y Ng; et al. 2009. "Ros: an open-source robot operating system." In *ICRA workshop on open source software*, volume 3, page 5.

Rawat, Waseem and Zenghui Wang. 2017. "Deep convolutional neural networks for image classification: A comprehensive review." *Neural computation*, 29(9):2352–2449.

Rublee, Ethan; Vincent Rabaud; Kurt Konolige; and Gary Bradski. 2011. "ORB: An efficient alternative to SIFT

or SURF." In *2011 International conference on computer vision*, 2564–2571.

Sutter, Herb et al. 2005. "The free lunch is over: A fundamental turn toward concurrency in software." *Dr. Dobb's journal*, 30(3):202–210.

Voss, Michael; Rafael Asenjo; and James Reinders, *Beef Up Flow Graphs with Async Nodes*, 513–534 (Apress, Berkeley, CA, 2019a). ISBN 978-1-4842-4398-5. doi:10.1007/978-1-4842-4398-5_18.

Voss, Michael; Rafael Asenjo; and James Reinders, *Flow Graphs*, 79–107 (Apress, Berkeley, CA, 2019b). ISBN 978-1-4842-4398-5. doi:10.1007/978-1-4842-4398-5_3.

Voss, Michael; Rafael Asenjo; and James Reinders, *Flow Graphs: Beyond the Basics*, 451–511 (Apress, Berkeley, CA, 2019c). ISBN 978-1-4842-4398-5. doi:10.1007/978-1-4842-4398-5_17.

Wang, Kebing; Zhaojuan Bian; Qian Chen; Ren Wang; and Gen Xu. 2014. "Simulating Hive cluster for deployment planning, evaluation and optimization." In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, 475–482.

Wang, Li and Dennis Sng. 2015. "Deep learning algorithms with applications to video analytics for a smart city: A survey." *arXiv preprint arXiv:1512.03131*.

Weber, Felix Dominik and Reinhard Schütte. 2019. "State-of-the-art and adoption of artificial intelligence in retailing." *Digital Policy, Regulation and Governance*.

AUTHOR BIOGRAPHIES

VYACHESLAV ZHDANOVSKIY was born in Verkhnyaya Pyshma, Russia. He obtained his B.Sc. in Applied Physics and Mathematics in 2020 from Moscow Institute of Physics and Technology (MIPT). Currently he is a M.Sc. student in Computer Science and Engineering at MIPT. Since 2020, he works at the Vision Systems Lab at the Institute for Information Transmission Problems. His research interests include heterogeneous and parallel computing, computer vision and distributed systems. His e-mail address is zhdanovskiy.vd@phystech.edu.

LEV TEPLYAKOV was born in Arkhangelsk, Russia. He obtained his B.Sc. and M.Sc. in Applied Physics and Mathematics from Moscow Institute of Physics and Technology (MIPT) in 2017 and 2019 correspondingly. Since 2016, he has been developing industrial computer vision systems with the Vision Systems Lab at the Institute for Information Transmission Problems. His research interests include heterogeneous and parallel computing, object detection and tracking. His e-mail address is teplyakov@visillect.com.

ANTON GRIGORYEV was born in Petropavlovsk-Kamchatskiy, Russia. Having graduated from Moscow Institute of Physics and Technology, he has been developing industrial computer vision systems with the Vision Systems Lab at the Institute for Information Transmission Problems since 2010. His research interests are image processing and enhancement methods, autonomous robotics and software architecture. His e-mail address is me@ansgri.com.