Introduction of Benchmarking Client for OpenVINO™ Model Server – Release 2022.1

Intel Corporation

February 16, 2022

Abstract

In this paper we introduce an example of an inference benchmark client designed for stressing the OpenVINO™ Model Server. It is written in Python 3 and is based on the TensorFlow API. The benchmark client is part of the HAProxy docker container which includes the Prometheus monitoring service. This is the first public version that supports transmission over the gRPC protocol, and it includes many advanced features like stateful models, binary input, workload data generation based on service metadata and reporting of comprehensive metrics. We present 2 different measurement methods in which the client plays an important role: (first – throughput vs. latency and second – quality of service), as well as a number of interesting measurement cases that demonstrate OpenVINO™ Model Server performance tuning, backend service configuration, batching, demultiplexing, and binary input over a relatively slow network. The benchmark client is distributed under the Apache License 2.0.

1 Introduction

1.1 OpenVINO™ Model Server

OpenVINO™ Model Server (OVMS) is a scalable, high-performance inference service designed to serve machine learning models based on convolutional neural networks (CNN). OVMS is optimized for Intel® architectures. The server provides inference-as-a-service over wired and wireless networks supporting the 2 popular communication protocols gRPC and REST. The Application Programming Interface (API) of the proposed solution is modeled after TensorFlow Serving (TFS) but contrary to TFS it uses OpenVINO™ (OV) as the inference execution provider (backend). Both OV and OVMS are implemented in C++ and this contributes to better scalability and higher performance when compared to the previous OVMS version which was implemented mainly in Python [blog/OVMS]. The OVMS is a part of the cross-platform OpenVINO™ toolkit which consists of libraries, inference engine plugins, model downloader and optimizer as well as C, C++, and Python application examples.

Let's briefly review the most important features available in the current release. Repositories of models served may reside on the following media:

- local and network mounted file systems,
- Google Cloud Storage (GCS),
- Azure Blob Storage,
- Amazon S3 / Minio.

OVMS Supports multiple frameworks including the folowing popular framework formats:

- TensorFlow,
- Caffe,
- MXNet,
- ONNX.

In addition to CPU devices, both Intel® Core™ and Intel® Xeon® lines, there are many compatible artificial intelligence (AI) accelerators including Intel® interated GPU and vision processing units (VPU), which include the Intel® Movidius Myriad and the High Density Deep Learning (HDDL) plugins. The OVMS has implemented many interesting features such as:

- Serving models that are commonly referred to as *stateful*, these operate on sequences of input data and maintain an internal state between inference requests. This allows recognition of dependencies between consecutive portions of input data which can be especially useful in processing multimedia streams.

- Interconnection of multiple models and operators in order to deploy complex processing solutions and reduce overhead of sending data back and forth – briefly referred to as *pipelines*.
- Inference model or data transformations can be implemented by a custom node in C/C++ loaded as an external dynamic linked library.
- Input data can be sent in binary format to reduce traffic and offload the client applications. At the moment, we support 2 common graphical data types: JPEG and PNG, but other formats can be covered by user custom nodes.

OVMS also supports configuration changes at runtime, including:

- deployment of new models, versions, pipelines, and custom nodes,
- removing these elements,
- performance tuning through parameter updates,
- model updating such as reshaping.

For additional information on OpenVINO™ Model Server please refer to the official documentation published on the GitHub home page [GitHub/OVMS] where you will also find many practical real-life examples including how to create and use custom nodes and other advanced features.

1.2 Benchmark Client

The benchmark client introduced in this paper is written in Python 3 and it is a part of OpenVINO™ Model Server repository [GitHub/OVMS]. The source code is located in *demos/benchmark/python* directory[1]. It is recommended to use the benchmark client as a docker container – please see Section 1.2.1 to learn more about the architecture details and how to build the docker image from the repository. Prior to transmission, the client downloads metadata from the server, which contains a list of available models, their versions as well as accepted input and output shapes. Then it generates tensors containing random data with shapes matched to the models served by the service. Both the length of the dataset and the workload duration can be specified independently. The synthetic data created is then served in a loop iterating over the dataset until the workload length is satisfied. As the main role of the client is performance measurement all aspects unrelated to throughput and/or latency are ignored. This means the client does not validate the received responses nor does it estimate accuracy as these activities would negatively affect the measured performance metrics on the client side.

In addition to the standard data format, the client also supports stateful models (recognizing dependencies between consecutive inference requests) as well as binary input for select file formats (PNG and JPEG)[2]. Both channel types, insecure and certificate secured, are supported. Secrets/certificates have to be mounted on a separated volume as well as their path has to be specified by command line [doc/GRPC]. The secure connection can be used, for example, to benchmark the Nginx OVMS plugin whith the built-in Nginx reverse proxy load balancer. It can be built as a docker image using the public source code in the OVMS repository [GitHub/OVMS/nginx].

A single docker container can run many parallel clients in separate processes. Measured metrics (especially throughput, latency, and counters) are collected from all client processes and then combined upon which they can be printed in JSON format/syntax for the entire parallel workload. If the docker container is run in the deamon mode the final logs can be shown using the *docker logs* command. Results can also be exported to a Mongo database. In order to do this the appropriate identification metadata has to be specified in the command line – see examples in Section 1.2.2.

1.2.1 Building & Architecture

We present the usage of the client as a docker container. It can be built using the *Dockerfile* located in the root client directory – *demos/benchmark/python*. You can rename the container image as desired but in this paper it will be referred to as *benchmark_client*. Let's build the image with the following commands:

```
cd demos/benchmark/python
docker build . -t benchmark_client
```

[1] Please note, that *demos/benchmark/python* path can be changed in future OVMS releases.

[2] The client supports both the OVMS and the Nvidia Triton Server. However, full functionality is only available for OVMS. Only the basic operations were implemented for Nvidia Triton, this means certificate secured channel, stateful model, and binary input are not supported.

The client docker image is built based on the HAProxy official public image [dockerhub/HAProxy]. It allows the use of the Prometheus monitoring system, which is build into this image. To use this functionality, the workload has to be redirected to the internal load balancer. You can easily do this by adding the $--proxy$ switch in the command line and mapping to our default Prometheus metric port which is 11888. Also, the HAProxy configuration can be changed by modification of the *haproxy.cfg* file which is also located in the repository. The Prometheus service is an open-source system monitoring and alerting toolkit, which allows capturing various network communication metrics during workload execution [www/Prometheus]. The client also generates its own metrics as a plain text and/or JSON report after finishing all inferences and stopping all processes.

The client architecture is designed as multiple independent processes launched at the beginning of the workload. The implementation is based on the multiprocessing Python module. After a required duration or an executed iteration number, each process generates a dictionary with recorded metrics. The final step in a benchmark run is to combine metrics from all the processes and then export them by pushing them to a Mongo database and/or printing to the *stdout*. The metrics include counters, latency, timestamps, and basic statistics. The metrics are reported for 3 different time windows:

- warmup (U) – an initial time interval (preceded by $\_warmup$ prefix),
- internal window (W) – which should be the most stationary (preceded by $\_window$ prefix),
- total workload duration (T) – (without $\_warmup$ and $\_window$ prefixes),

where the following requirement has to be fulfilled: $U + W \leq T$, however, we recommend to assume this relation: $T = W + 2U$. Then, after the window, the workload will be realized U more seconds. Instead of expressing the total workload duration in seconds, the number of iterations can be specified. As these options are mutually exclusive the benchmark−client will terminate with an error message if both these are specified at the command line. The warmup interval and the internal window length only support time specified in seconds. All mentioned window ranges can be defined by using command line switches (see $-t$, $--duration$, $-u$, $--warmup$, $-w$, $--window$, and $-n$, $--steps\_number$). Some examplary JSON report for the 3 mentioned windows is presented in Appendix A.

The client source code and associated configuration files are distributed under the Apache License, Version 2.0 and the base HAProxy docker image is published under the GPL with the additional exemption that compiling, linking, and/or using OpenSSL is allowed [dockerhub/HAProxy]. As stated earlier the client generates requests compatible with TensorFlow API in gRPC protocol but there are also plans to add support for REST and KServ API.

1.2.2 Selected Command Examples

For more usage details and to check other available options, please see further examples in this paper, where the most important selected options are discussed. You can also display the help menu by using the $--help$ or $-h$ switch. The command should print all available options with a brief clarification – here only a synopsis is listed:

```
docker run benchmark_client --help

[-h] [-i ID] [-c CONCURRENCY] [-a SERVER_ADDRESS]
[-p GRPC_PORT] [-r REST_PORT] [-l] [-b [BS [BS ...]]]
[-s [SHAPE [SHAPE ...]]] [-d [DATA [DATA ...]]] [-j]
[-m MODEL_NAME] [-k DATASET_LENGTH] [-v MODEL_VERSION]
[-n STEPS_NUMBER] [-t DURATION] [-u WARMUP] [-w WINDOW]
[-e ERROR_LIMIT] [-x ERROR_EXPOSITION] [--max_value MAX_VALUE]
[--min_value MIN_VALUE] [--step_timeout STEP_TIMEOUT]
[--metadata_timeout METADATA_TIMEOUT] [-y DB_CONFIG]
[--print_all] [--certs_dir CERTS_DIR] [-q STATEFUL_LENGTH]
[--stateful_id STATEFUL_ID] [--stateful_hop STATEFUL_HOP]
[--nv_triton] [--sync_interval SYNC_INTERVAL]
[--quantile_list [QUANTILE_LIST [QUANTILE_LIST ...]]]
[--hist_factor HIST_FACTOR] [--hist_base HIST_BASE]
[--internal_version]
```

The versioning of the client uses 2 digits (major and minor): the first digit relates to OV and OVMS releases (1 corresponds to 2022.1) and the second digit is used for internal purposes. It indicates the number of GIT commits pushed to the OVMS repository that include changes to the client. As of writing this paper it was at 17, however upon publication of this paper it should be expected to be higher. The version can be checked by using $--internal\_version$ switch as follows:

```
docker run benchmark_client --internal_version

1.17
```

The client is able to download the metadata of the served models. If you are unsure which models and versions are served and what status they have, you can list this information by specifying the *−−list_models* switch (also a short form *−l* is available):

```
docker run benchmark_client -a 10.91.242.153 -r 30002 --list_models

XI worker: try to send request to endpoint: http://10.91.242.153:30002/v1/config
XI worker: received status code is 200.
XI worker: found models and their status:
XI worker:  model: resnet50-tf-fp32, version: 1 - AVAILABLE
XI worker:  model: resnet50-tf-int8, version: 1 - AVAILABLE
```

As can be seen from the example above an IP address (*−a*) and a REST port number (*−r*) have to be specified. In the presented example only 2 *Resnet 50* models are availabe. The IP address in the above example is private and non-routing. Users must set their own IP address on the host running the OVMS. In general, the model status can be as well checked by commonly known applications – for instance, *WGET*.

Names, model shape, as well as information about data types of both inputs and outputs can also be downloaded for all available models using the same listing switches and adding *-m <model-name>* and *-v <model-version>* to the command line. The option *−i* is only to add a prefix to the standard output with a name of an application instance. For example:

```
docker run benchmark_client -a 10.91.242.153 -r 30002 -l -m resnet50-tf-fp32 -p 30001 -i id

XI id: try to send request to endpoint: http://10.91.242.153:30002/v1/config
XI id: received status code is 200.
XI id: found models and their status:
XI id:  model: resnet50-tf-fp32, version: 1 - AVAILABLE
XI id: request for metadata of model resnet50-tf-fp32...
XI id: Metadata for model resnet50-tf-fp32 is downloaded...
XI id: set version of model resnet50-tf-fp32: 1
XI id: inputs:
XI id:  input_name_1:
XI id:   name: input_name_1
XI id:   dtype: DT_FLOAT
XI id:   tensorShape: {'dim': [{'size': '1'}, {'size': '3'}, {'size': '224'}, {'size': '224'}]}
XI id: outputs:
XI id:  output_name_1:
XI id:   name: output_name_1
XI id:   dtype: DT_FLOAT
XI id:   tensorShape: {'dim': [{'size': '1'}, {'size': '1001'}]}
```

Be sure the model name specified is identical to the model name shown when using the *–list_models* parameter. A model version is not required but it can be added when multiple versions are available for a specific model name. Port numbers are required both for REST and gRPC APIs. In the above example, the model has a single input named *input_name_1* with the size (1, 3, 224, 224) and a single output referred to as *output_name_1* with the size (1, 1001) – see Section 2.2.1 for clarification of *Resnet 50* input and output meaning.

2 Material & Methods

2.1 Measurement Methodology

The measurements presented in this paper are focused on end-to-end and black-box approach which, from a customer's perspetive, would be the most important. A generic benchmark setup would be a single OpenVINO™ Model Server instance on a multicore processor as the hardware backend which is stressed by multiple clients over the network. The client number is referred to as concurrency. We assume that requests are generated asynchronously and that clients are independent. Each client waits for the response from the OVMS returning the inference results. Both the server and clients are run in docker containers on separate hardware platforms. The collected metrics are measured and reported by the individual clients and then combined into more comprehensive statistics. Using this architecture (see Figure 1), we analyze 2 types of characteristics:

- throughput vs. mean latency for increasing concurrency,
- quality of service by latency nonlinear statistics.

Let's dive deeper into their details in the next 2 sections. Both mentioned characteristics or profiles are estimated for a windowed part of the workload. In each measurement some warmup and closing margins are ignored in order to eliminate intervals which are potentially non-stationary since some clients may not yet be fully operational. The ranges of these non-stationary intervals should be determined prior to actual measurements for each workload individually.

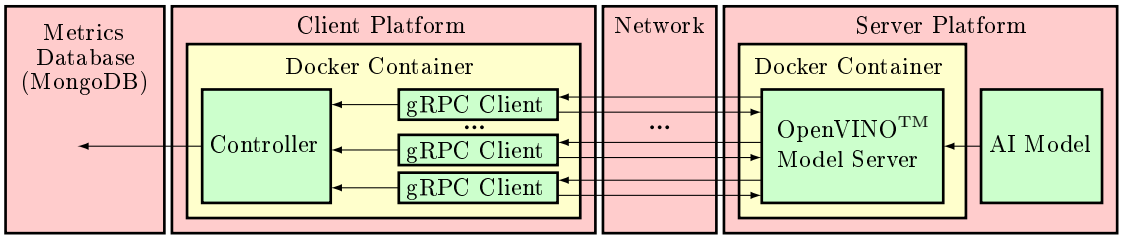

Figure 1: Setup schema.

2.1.1 Throughput vs. Mean Latency

The throughput vs. latency profile is the system characteristic best suited for estimating its capacity and investigating throughput as a function of latency in a multicore and/or multiprocessor system serving concurrent clients. This can be particularly useful when determining an optimal trade-off between both these metrics for any system by deploying an appropriate concurrency. We exploit such an approach in order to compare overall performance of different OVMS configurations for certain AI models and hardware setups.

The final curve denoted by f is estimated by ordering independent measurement points $(L_{c,i}, T_{c,i})$ according to increasing client number c. The approximation of the profile: $\hat{T}_c = f(\hat{L}_c)$ is based on many measurement realizations $i = 1, 2, 3, ...$, where $\hat{T}_c$ and $\hat{L}_c$ are estimates of, respectively, throughput and mean latency. In special cases, latency can locally clearly decrease when concurrency increases, therefore distribution f is not the function in the strict mathematical sense. It is a kind of trajectory over latency-throughput plane.

The concurrency can be explicitly specified by calling the benchmark client image with $-c$ or $--concurrency$ switches together with a workload duration, for example, as follows:

```
1  docker run benchmark_client -c C -n N (...)
2  docker run benchmark_client -c C -t T (...)
```

where C parallel clients will, respectively, send N requests each or work over T interval.

The measurement procedure is started from a single client. For the folowing iterations, the concurrency is increased as can be seen in Figure 2. The maximum number of considered clients depends on the system capacity. In each investigated point (for each concurrency), mean latency and throughput are measured for an internal windowed part of the workload, the internal windowed being the most stationary period of a workload. Finally, the collected metrics are graphed in a common coordinate system. Both throughput and mean latency are linear estimates, therefore, an additional investigation should be conducted for workloads with outliers (outliers can be lost by averaging). In the next Section 2.1.2, we propose another method to analyze performance outliers in the workload.

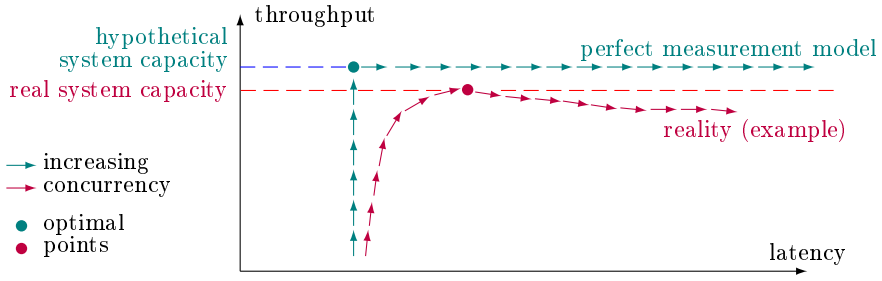

Figure 2: Interpretation of throughput vs. latency profile.

2.1.2 Latency-based Quality of Service

Quality of service (QoS) is a commonly used characteristic in telecommunication to asses the reliability of the network and services. In general, it can be applied to analyze errors, packet loss, delay, etc. However, in this paper, we consider latency-based QoS to present a statistical probability distribution of inference latency. The resulting profile is a kind of cumulative distribution function. We graph it in a coordinate system with swapped axes - the abscissa axis represents probability (on a logarithmic scale) and the ordinate refers to the response latency. In other words, this metric is a set of latency quantiles drawn in the logarithmic perspective (see interpretation in Figure 3).

The QoS profile is a nonlinear and highly sensitive indicator especially in the high quantiles range. Therefore, its measurement should be repeated many times over before drawing any conclusions. It is a very useful tool for analyzing value and probability of latency outliers in the workload. For example, measuring time-to-first-inference which can include the model reloading time, which is usually the longest among all infer-requests. As QoS in the current release of benchmark_client is estimated only for the internal window of the workload, the warmup duration command line parameter has to be set to 0 in order to measure the first request.

In order to estimate a specific quantile $q \in < 0, 1 >$, we have to collect a sufficient number of samples greater than $1/(1-q)$. A common rule of thumb used for achieving stable results, is that the number of analyzed samples should be at least $10/(1-q)$ or even $100/(1-q)$ – depending on workload stability and the latency probability distribution. Any specific quantiles can be requested by using $--quantile\_list$ switch. The quantile values are estimated based on a histogram. To improve the histogram's resolution the following 2 switches can be adjusted, however their default values should be optimal for most workloads:

- linear coefficient $--hist\_factor$,
- logarithmic mapper $--hist\_base$.

Some examplary usage of the benchmark clients for the quality of service could look like that:

```
1  docker run benchmark_client --quantile_list 0.9 0.99 0.999 -t 120 -u 30 -w 60 (...)
```

which means that the whole workload should last 2 minutes. The QoS, which includs 3 quantiles (mainly: 90%, 99%, and 99.9%), will be estimated only for window which length is equal to 1 minute. The window will be preceded by the 30-second warmup.

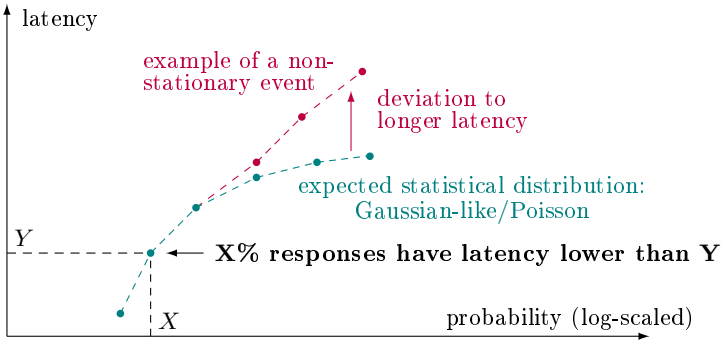

Figure 3: Interpretation of latency-based quality of service.

2.2 Artificial Neural Networks

Artificial neural networks, commonly referred to as *models*, are one of the most important aspects of machine learning. During the inference, decisions are made in a manner defined by the model. The main idea of machine learning is that models contain information derived from the input data applied during their training. Model complexity, input/output size, the numerical precision, and composition of layers have substantial impact on the final performance of inference as well as training. Therefore, each performance comparison of different hardware and software stack/setup conducted on different models (even if their names are the same) are not reasonable.

2.2.1 Model *Resnet 50*

Resnet 50 is a convolutional neural network consisting of 50 layers [arXiv/Resnet50]. Its purpose is classification of images into 1000 categories. The categories include among others: individual animals, vehicles, commonly used items, etc. Its output has a length equal to 1001 which corresponds to the categories (plus one extra category). Input images have a height × width resolution of 224 × 224. The model used for our benchmarking is generated using TensorFlow and can be obtained from the Open Model Zoo github repo [GitHub/OMZoo] directory *public/resnet-50-tf*. The intermediate representation (IR) of the model has a size of approx. 98 MB for floating point 32 and 25 MB for integer 8 precision. *Resnet 50* is one of a few models widely used in machine learning benchmarking.

2.3 Hardware Configurations

In this section we introduce the hardware configurations of both client and server platforms used for benchmarking. In Table 1 we also list configuration details like operating system and other important software elements installed. We have focused on Intel® scalable architecture dedicated to server solutions – Intel® Xeon®. Technical details of Intel products are also published on the official website [spec/Intel/HW]. OVMS configuration, especially threading configuration and streams numbers should correspond with the CPU parameters to optimally utilize it in order to achieve the greatest possible performance or the lowest latency.

Table 1: Platforms for server solution benchmarking

	OVMS Platform	**Remote Client Platform**
CPU Name	Intel® Xeon® Platinum 8260M	Intel® Xeon® Gold 6252
CPU Microarchitecture	Cascade Lake	Cascade Lake
CPU Power (TDP)	150 W	162 W
CPU Speed	2.40 GHz	2.10 GHz
CPU Cores	2 × 24	2 × 24
Motherboard Vendor	Intel® Corporation	Inspur Group
Motherboard Model	Server Board S2600WF H48104-872	YZMB-00882-104 NF5280M5
BIOS Vendor	Intel® Corporation	American Megatrends Inc.
BIOS Version	SE5C620.86B.02.01	4.1.16
BIOS Release	03/26/2020	06/23/2020
Memory Name	Hynix® DDR4	Samsung® DDR4
Memory Speed	2666 MT/s	2666 MT/s
Memory Size	16 × 16GB	16 × 16GB
Operating System	Ubuntu 20.04.3 LTS	Ubuntu 20.04.3 LTS
Linux Kernel	5.4.0-81-generic	5.4.0-81-generic
Docker Version	20.10.8	20.10.8
Network Speed	40 Gb/s (1 Gb /s)	

3 Performance Tuning & Measurements

OVMS performance can be managed by parameters passed by configuration file or/and by command line especially if *single-model* mode is used. Summarizing, we can list the following parameters for CPU backend which can have an impact on end-to-end performance without any accuracy degradation:

- `grpc_workers` – gRPC workers (specified only in command line),
- `rest_workers` – REST workers (specified only in command line),
- `CPU_THROUGHPUT_STREAMS` - the number of CPU streams,
- `CPU_BIND_THREAD` - binding option allows to map threads to cores,
- `CPU_THREADS_NUM` - the number of threads used for inference,
- `PERFORMANCE_HINT` - auto-config (a new 2022.1 feature),
- `nireq` - inference request queue size,
- batch size – a workload parameter.

The last parameter – batch size is dependent on a request content, but it can be considered in the context of the demultiplication, which is one of the OVMS features and it can be applied through an appropriate pipeline definition (see Section 3.3). Let's review some of the parameters in following sections, where we will try to investigate their impact. However, determining the best values for parameters for optimal end-to-end performance should be done for each specific workload and hardware configuration. Additional information regarding performance tuning can be found in the official documentation – [GitHub/OVMS/perf] and [doc/OV/CPU].

The newest 2022.1 OV release offers a new intersting configuration option to easy tune the performance, which is PERFORMANCE_HINT. It can be set as "THROUGHPUT" or "LATENCY" in order to optimize, respectively, throughput and latency. You shall consider this approach, if you did not reserch the OV backend configuration before or your Best Known Configuration (BKC) seems to not meet expectations. This is especially recommended for the "THROUGHPUT" value when the target device is set as CPU or AUTO. Please note that this option is novel and still validated on a limited number of configurations. Therefore, we wait for your feedback in case of any issues.

3.1 CPU Throughput Streams & Nireq

Let's review a small OpenVINO™Model Server configuration file for *Resnet 50* model, where X denotes a number of CPU throughput streams. *Nireq* is set as double the value of the CPU streams number ($2X$).Threads number has the constant value equal to 48 (which is equal to the number of Intel® Xeon® physical cores in 2 sockets). The thread binding flag is set as "yes". The number of gRPC workers are equal to 16. Many conducted experiments confirmed that such values are appropriate for a parallel system. The whole configuration includs a directory path with the served model (*/models/resnet50-tf-fp32* – it has to exist and it must be avalable at least to read for the service) and it looks as follows:

```
"model_config_list": [
  {
    "config": {
      "name": "resnet50-tf-fp32",
      "base_path": "/models/resnet50-tf-fp32",
      "target_device": "CPU",
      "batch_size": "auto",
      "nireq": 2X,
      "plugin_config": {
        "CPU_THROUGHPUT_STREAMS": "X",
        "CPU_BIND_THREAD": "YES",
        "CPU_THREADS_NUM": "48"
      }
    }
  }
]
```

In this experiment, the throughput vs. mean latency profile is measured for different $X = 1, 2, 4, 8, 12, 16, 24, 25, 28, 32, 36, 48, 56, 60, 64, ...$ However, not all of the CPU stream values are presented in the graph. We display only the most relevant results. The batch size on server site is set to *auto*, which means that OpenVINO™ Model Server will accept all positive batch sizes and process them without any additional reshaping. In general, any model reloding connected with reshaping can result in decreasing performance when batches have many different sizes in a single workload. However, in this experiment all batch sizes are constant and they are set to 1. Throughput is simply defined as the ratio of the number of samples processed divided by the sample processing time.

The total workload duration is equal to 3 minutes for each measurement point. The first and last minutes are rejected. The final metrics are estimated based on a time window of approximately 1 minute at the mid point of the workload duration. The concurrency is adjusted from 1 to as much as 120. However, not all the results are presented in the corresponding Figures 4 and 5. We see that if the CPU throughput streams parameter is lower than 25, it allows the management throughput vs. latency. For the Resnet 50 models, it is worthwhile lowering the streams number when concurrency is low, this allows a reduction of the mean latency without sacrificing throughput. The maximal throughput for integer 8 precision is observed for 48 streams, which corresponds to the number of physical CPU cores. For floating-point 32 precision, the throughput optimal streams number is lower – around 16.

Finally in this experiment, we considered how concurrency affects the quality of service (QoS) expressed as the latency cumulative probability distribution for *Resnet 50* model in integer 8 precision. We analyzed system behavior for concurrencies from 1 to 8. In Figure 6, we clearly see, that incresing the clients number causes increased latency for quantiles lower than 99.9% which is as expected. Moreover, it seems that we have collected too little data to evaluate these characterisics with a high confidence

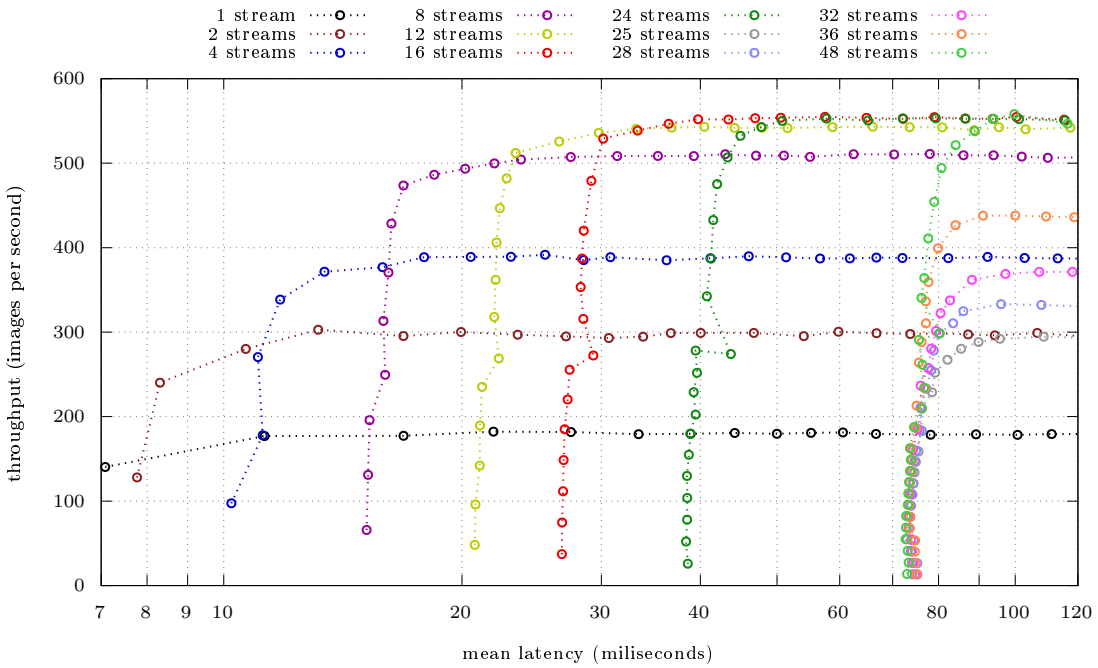

Figure 4: Througput vs. mean latency with changed concurrency for *Resnet 50* model in floating-point 32 precision collected on server platform with Intel® Xeon® Platinum 8260M processor (see Table 1). Common parameters are number of threads and thread binding flag equal to, respectively, 48 and "yes".

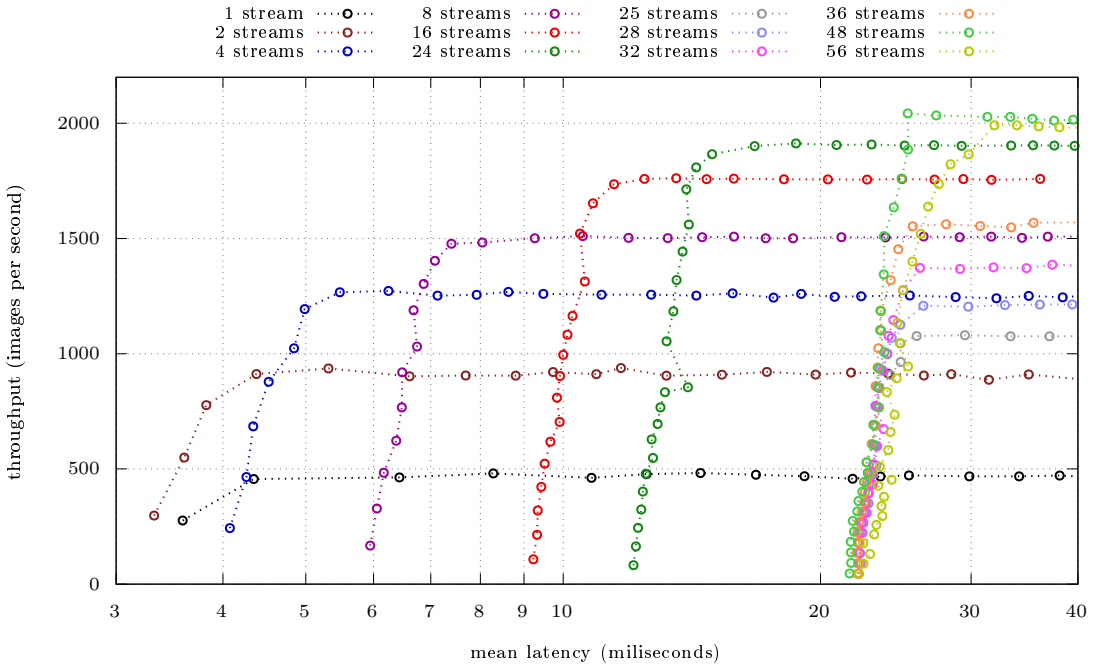

Figure 5: Througput vs. mean latency with changed concurrency for *Resnet 50* model in integer 8 precision collected on server platform with Intel® Xeon® Platinum 8260M processor (see Table 1). Common parameters are number of threads and thread binding flag equal to, respectively, 48 and "yes".

above the 99.9% quantile. Each measurement point was estimated over a 5 minute period and it was repeated at least 3 times for each concurrency. In the figure, measurement points are represented by dotted lines. At the end the median values are estimated and plotted using dashed lines individually for each considered concurrency.

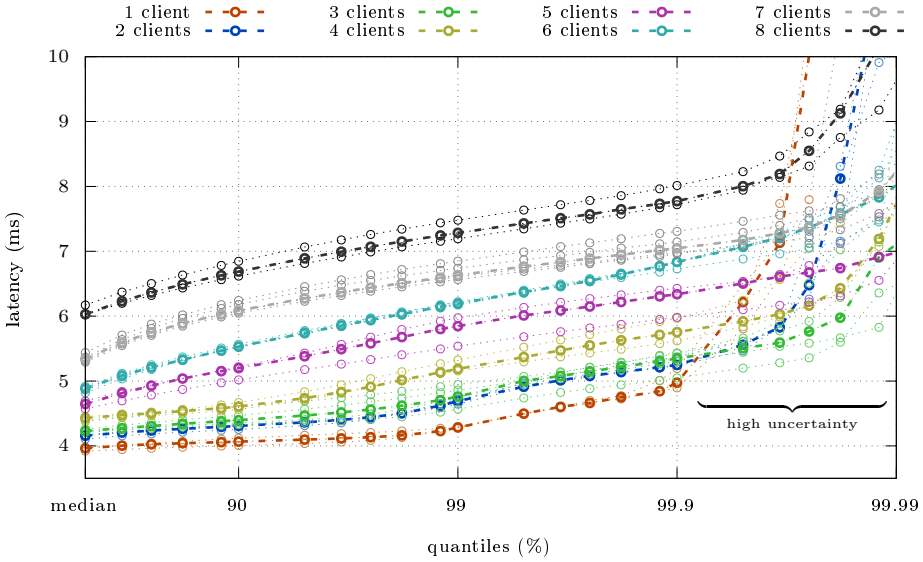

Figure 6: Quality of service estimated for *Resnet 50* model in integer 8 precision collected for selected concurrency (form 1 to 8) on server platform with Intel® Xeon® Platinum 8260M processor (see Table 1). Common parameters are number of threads, thread binding flag, CPU streams, and nireq equal to, respectively, 48, "yes", 48, and 96.

3.2 Binary Input & Slow Network

OpenVINO™ Model Server supports compressed binary input data (including JPEG and PNG formats) for models, which process images. This feature can improve overall performance especially in slow networks where the lower bandwidth constitutes a system bottleneck. A good example of such use could be wireless communication. On the other hand, for systems with high speed links additional decompression can result in a reduction in performance. For some specific models, you have to set an appropriate layout mapping in OVMS configuration file to support this variant. For instant, the *resnet 50* model considering in this paper requires adding the following layout value:

```
"model_config_list": [
  {
    "config": {
      "name": "resnet50-tf-int8",
      "layout": "NHWC:NCHW",
  (...)
```

In order to test this feature, we have added the option to generate Portable Network Graphics (PNG) images in the benchmark client. There is also an example implemented using an embedded JPEG file. However, let's focus on PNG testing data. In order to send it to the input named Y, you have to specify an appropriate value of the switch $-d$, $--data$ as follows:

```
docker run benchmark_client --data Y:png (...)
docker run benchmark_client --data Y:png4 (...)
docker run benchmark_client --data Y:png8 (...)
docker run benchmark_client --data Y:png16 (...)
```

If a single input is available, there is no needed to specify the input name:

```
docker run benchmark_client --data png (...)
docker run benchmark_client --data png4 (...)
docker run benchmark_client --data png8 (...)
docker run benchmark_client --data png16 (...)
```

At the moment, 4 different options are available: *png*, *png4*, *png8*, and *png16* (there are plans to extend this set). They are distinguished by the radius of the lowpass filter used to smooth uniform noise, which is the foundation of the data. Each of them has a different ability to be compressed, and what follows this, is that each of them has a different duration of compression. All these aspects can affect the final values of the estimated metrics. In Figure 7, examples of generated images are presented for all 4 options and in Table 2 their mean sizes are listed.

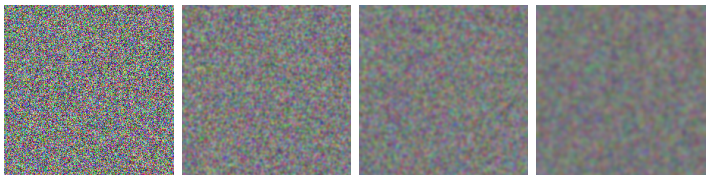

Figure 7: Examples of random PNG files generated with uniform probability distribution, smoothed by a lowpass filter, and used as binary input for options, respectively, *png*, *png4*, *png8*, and *png16*.

Table 2: Mean size of images (224×224) corresponding to Figure 7

Option Name	not compressed	*png*	*png4*	*png8*	*png16*
Mean Request Size	602 KB	148 KB	117 KB	108 KB	99 KB

In this section, we analyze the AI inferencing performance of 2 communication networks: a 1 Gb/s (slow) and a 40 Gb/s (fast). We focus on the *Resnet 50* model in integer 8 precision. We use a sequence of PNG files as the testing workload since this model expects images as its input. OpenVINO™ Model Server decompresses the images on-the-fly and then forwards them to the OpenVINO™execution engine in a decompressed format as plain tensors. The PNG format has been choosen (although JPEG could be compressed more) because of its lossless information compression. The results for binary input and

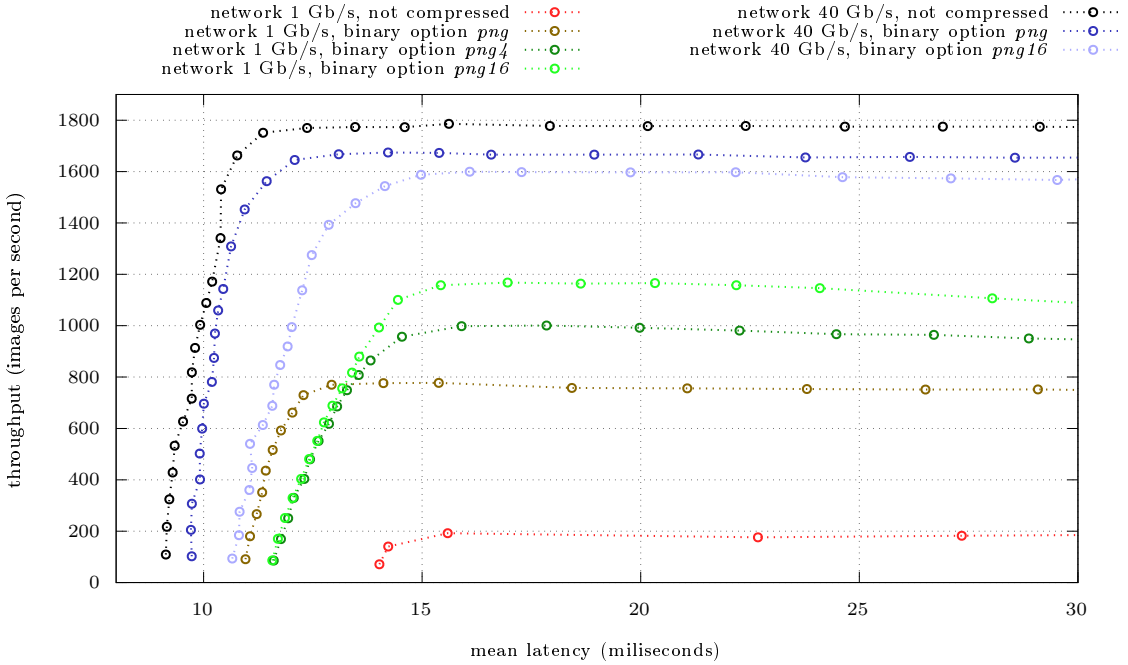

Figure 8: Througput vs. mean latency with changed concurrency for *Resnet 50* model in integer 8 precision collected on server platform with Intel® Xeon® Platinum 8260M processor (see Table 1). Common parameters are number of threads, thread binding flag, CPU streams, and nireq equal to, respectively, 48, "yes", 16, and 32. 2 different network links – 1 and 40 Gb/s, and various input types including binary PNG format were investigated.

those without compressed tensors are compared in Figure 8. As expected the binary input mode has considerably better performance in comparison to the uncompressed input for slow networks. However, in general, the benefit depends on how much the data can be compressed. On the other hand, for fast networks the opposite situation is observed, where binary input causes a performance degradation. This is because, the decompression requires additional computing power. Therefore, it is recommended to analyze the benefits of using binary input application in each particular situation.

3.3 Batching & (De)multiplexing

Batching is the mechanism of joining single requests in some greater groups. Further, these data structures can be processed more effectively with relatively higher throughput by parallel approach and to minimize communication overload. On the other hand, the cost that has to be paid is usually a longer response (greater latency) and increases complexity of the whole system. The throughput improvement may vary depending on different models as well as backend configurations. Basically, OpenVINO™ Model Server requires to declare accepted batch sizes or shapes. However, a few suggestions have been proposed to address the problem of *a priori* unknown request shape (or even changable/dynamic shape accross a single workload), mainly:

- the automatic mode (*auto*),
- a pipeline with demultiplexing,
- dynamic shape (experimental feature, avaliable starting from version 2022.1).

The first option, referred to as the automatic mode (or shorter *auto*) is used, for example, in the experiment presented in Section 3.1. This option causes the model to reload in the memory each time the shape of request is changed. Therefore, the final performance is expected to be very poor for workloads

```
{
  "model_config_list": [
    {
      "config": {
        "batch_size": 1,
        "name": "resnet50_model_name",
        "base_path": "/models/resnet50_model_name",
        "target_device": "CPU",
        "nireq": 16,
        "plugin_config": {
          "CPU_THREADS_NUM": "48",
          "CPU_THROUGHPUT_STREAMS": "8",
          "CPU_BIND_THREAD": "YES"
        }
      }
    }
  ],
  "pipeline_config_list": [
    {
      "name": "demux_pipeline_name",
      "demultiply_count": -1,
      "inputs": ["input_name_1"],
      "nodes": [
        {
          "type": "DL model",
          "name": "internal_label",
          "model_name": "resnet50_model_name",
          "inputs": [
            {
              "original_model_input_1": {
                "node_name": "request",
                "data_item": "input_name_1"
              }
            }
          ],
          "outputs": [
            {
              "data_item": "original_model_output_1",
              "alias": "output_alias"
            }
          ]
        }
      ],
      "outputs": [
        {
          "output_name_1": {
            "node_name": "internal_label",
            "data_item": "output_alias"
          }
        }
      ]
    }
  ]
}
```

Figure 9: Examplary of OpenVINO™ Model Server configuration file which consists of the *Resnet 50* model and a pipeline with demultiplexing of input requests. The model configuration includs performance tuning parameters and the batchsize which has to be equal to 1.

in which batch size is frequently changed. Such situation can occur when in a system, many parallel clients generate requests with different shapes at the same time.

The second option, which is a pipeline with demultiplexing, is to create your own algorithm for the Directed Acyclic Graph scheduler. For this purpose, you have to use the *demultiply_count* parameter, which adds ability to any node to slice outputs into separate sub outputs. The following nodes will be executed many times by an event loop, independently, and results will be gathered and packed into one output just before sending a response. A *demultiply_count* parameter value has to match the first dimension of all node outputs or it should be -1, which means that it will be automatically set to be matched [GitHub/OVMS/demux]. At the moment, the pipeline with demultiplexing is recommended[3] to address the issue with dynamic bash size becaue:

- the performance of such solution is the best amongst all investigated variants,
- the method was introduced in previous releses therefore it is validated most accurately,
- all existing models should support such approach,
- it is well documented (see [GitHub/OVMS/demux]).

Finally, the dynamic batch size issue is also addressed in the newest OpenVINO™ release (2022.1) by the feature referred to as *dynamic shape*. This allows to set an expected range of each shape dimension including the batch size. For example, "2:4" means that OVMS will accept three size values: 2, 3, and 4. Moreover, if we do not know what range will be appropriate, it is possible to set universal -1, which means that each value will be accepted. The dynamic shape is implemented in OpenVINO™, however not all models support it yet. Unfortunately, the performance of this solution estimated on selected models is relatively poor for low concurrency and not stable for high concurrency. Which is also presented on Figure 10. We proactively and continuously work on these issues and we expect improvement of them in the future.

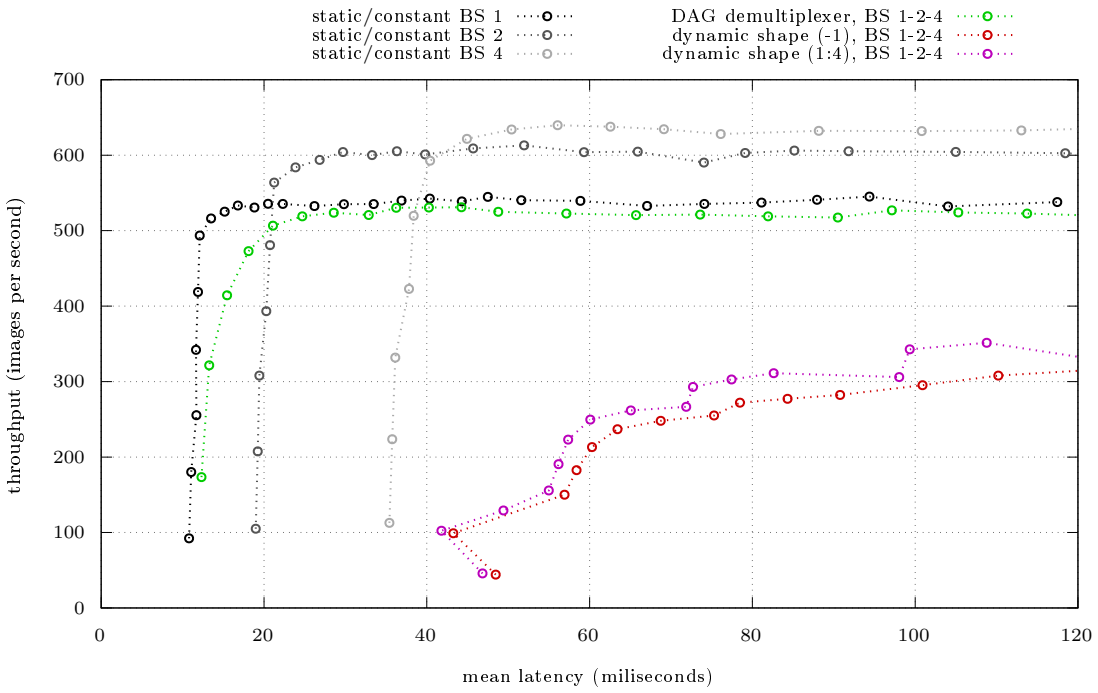

Figure 10: Througput vs. mean latency with changed concurrency for *Resnet 50* model in floatingpoint 32 precision collected on server platform with Intel® Xeon® Platinum 8260M processor (see Table 1). Common parameters are number of threads, thread binding flag, CPU streams, and nireq equal to, respectively, 48, "yes", 6, and 12. Characteristics of systems with 3 static batch sizes, 2 dynamic shapes and the pipeline with demultiplexer are presented.

[3]Please note that this recommendation can be changed in future releases

The introduced benchmark client supports generation of requests with multiple and different batch sizes in a single workload. The switches $-b$, $--bs$ can be used to specify this parameter. For example, in order to set multiple batch sizes – let's say 1, 2, 4, 2 – the following command can be called:

```
docker run benchmark_client -a 10.91.242.153 -p 30001 -m resnet50-tf-fp32 -b 1-2-4-2 -n 8
```

This phrase means that 4 batches with random data will be generated. They will include respectively 1, 2, 4, and 2 images – 9 in total. The client will send each batch 2 times (in the following order: 1, 2, 4, 2, 1, 2, 4, 2) because 8 iterations are required.

4 Conclusions

In this paper we introduced a client dedicated to benchmarking the OpenVINO™Model Server. The client's objective is generating a workload in order to stress the service, and then measure final metrics such as counters, latency, throughput, etc. We described the main features of the benchmark application and we presented several interesting use cases covering OVMS configuration, parallel processing, batching, demultiplexing, and binary data processing. Additionally, we presented 2 examples of measurement methods demonstrating comprehensive system characterisation by the benchmark client.

There are plans to continue the development of the benchmarking client. We are going to support REST and KServ APIs. We will implement dataset loading form external resources, increasing the number of generated patterns, as well as support future and improve current OpenVINO™ and OpenVINO™ Model Server features. Furthermore, our intention is to extend this paper by presentation of more performance measurement scenarios, especially, related to practical use cases with new OpenVINO™Model Server features.

References

[GitHub/OVMS] Openvinotoolkit GitHub – Model Server (OVMS), *release 2021.4*, https://github.com/openvinotoolkit/model_server/ (GH/OVMS≫), Sep. 2021.

[GitHub/OVMS/perf] Openvinotoolkit GitHub – Model Server (OVMS), *release 2021.4*, GH/OVMS≫blob/main/docs/performance_tuning.md

[doc/OV/CPU] OpenVINO™ documentation, Deploying Inference, *release 2021.4*, https://docs.openvino.ai/latest/openvino_docs_IE_DG_Device_Plugins.html

[GitHub/OVMS/demux] Openvinotoolkit GitHub – Model Server (OVMS), *release 2021.4*, GH/OVMS≫blob/develop/docs/demultiplexing.md

[GitHub/OVMS/nginx] Openvinotoolkit GitHub – Model Server (OVMS), *release 2021.4*, GH/OVMS≫blob/main/extras/nginx-mtls-auth/README.md

[GitHub/OVMS/demux] Openvinotoolkit GitHub – Model Server (OVMS), *release 2021.4*, GH/OVMS≫blob/main/docs/demultiplexing.md

[www/Prometheus] Prometheus Home Page, https://prometheus.io/, January, 2022.

[GitHub/OMZoo] Openvinotoolkit GitHub – Open Model Zoo, *release 2021.3*, https://github.com/openvinotoolkit/open_model_zoo, Apr. 2021.

[arXiv/Resnet50] Kaiming He, Xiangyu Zhang, Shaoqing Ren Jian Sun, *Deep Residual Learning for Image Recognition*, arXiv, CoRR, http://arxiv.org/abs/1512.03385, 2015.

[doc/OV/CPU] OpenVINO™ documentation, Deploying Inference, CPU plugin, *release 2021.4*, https://docs.openvinotoolkit.org/latest/openvino_docs_IE_DG_supported_plugins_CPU.html

[blog/OVMS] D. Trawinski, K.Czarnecki, *What's New in the OpenVINO™Model Server*, 2021, https://medium.com/openvino-toolkit/whats-new-in-the-openvino-model-server-3a060a029435.

[spec/Intel/HW] Intel Product Specification – official website, October, 2021, https://ark.intel.com/content/www/us/en/ark.html

[doc/GRPC] GRPC online documentation with examples, 2021,
https://grpc.io/docs/guides/auth/#python and https://grpc.github.io/grpc/python/grpc.html

[dockerhub/HAProxy] Official HAProxy docker image, version 2.3.10, 2021,
https://hub.docker.com/_/haproxy

Disclaimer

The performance results presented here are meant to illustrate how to use the benchmark client. They are not intended to serve as official hardware or software benchmark results for OpenVINO™ Model Server. For benchmarking purposes we strongly encourage the readers to measure their own metrics on their specific system configurations. Performance results may vary for many reasons. Therefore, official benchmark results should be based on multiple measurements where statitical uncertainty is estimated and reported together with the performance results.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure. Your costs and results may vary.

Authorship

- **Krzysztof Czarnecki**, Ph.D.

  K. Czarnecki is an employee of Intel Technology Poland, IOTG, in Gdansk. He received MS and PhD degrees from Gdansk University of Technology in digital signal processing. Currently, he works on development of software for supporting machine learning especially inference, benchmarking, and automation. Email: krzysztof.czarnecki@intel.com

- **Kamil Matejuk**

  K. Matejuk is an intern of Intel Technology Poland, IOTG in Gdańsk. He is currently studying computer science in Wrocław University of Science and Technology and writing his BSc thesis on machine learning in games. Email: kamil.matejuk@intel.com

A Appendix: Examplary metrics printed in JSON format


```
{
  "submetrics": 14,
  "warmup_total_batches": 2694,
  "warmup_total_frames": 2694,
  "warmup_pass_batches": 2694,
  "warmup_fail_batches": 0,
  "warmup_pass_frames": 2694,
  "warmup_fail_frames": 0,
  "warmup_netto_batch_rate": 1356.0306425275078,
  "warmup_netto_frame_rate": 1356.0306425275078,
  "warmup_mean_latency": 0.010318353062477128,
  "warmup_pass_max_latency": 0.01539468765258789,
  "warmup_fail_max_latency": 0,
  "warmup_first_latency": 0.014332771301269531,
  "warmup_start_timestamp": 1643120764.8586872,
  "warmup_stop_timestamp": 1643120766.906712,
  "warmup_total_duration": 2.048024892807007,
  "warmup_brutto_batch_rate": 1315.4136990530544,
  "warmup_brutto_frame_rate": 1315.4136990530544,
  "warmup_batch_passrate": 1,
  "warmup_frame_passrate": 1,
  "warmup_pass_mean_latency": 0.010318353062477128,
  "warmup_fail_mean_latency": 0,
  "warmup_mean_latency2": 0.00010690618853063 4,
  "warmup_stdev_latency": 0.0006616484026300306,
  "warmup_cv_latency": 0.06412345057624812,
  "warmup_pass_mean_latency2": 0.00010690618853063 4,
  "warmup_pass_stdev_latency": 0.0006616484026300306,
  "warmup_pass_cv_latency": 0.06412345057624812,
  "warmup_fail_mean_latency2": 0,
  "warmup_fail_stdev_latency": 0,
  "warmup_fail_cv_latency": 0,
  "window_total_batches": 7907,
  "window_total_frames": 7907,
  "window_pass_batches": 7907,
  "window_fail_batches": 0,
  "window_pass_frames": 7907,
  "window_fail_frames": 0,
  "window_netto_batch_rate": 1323.5655405366879,
  "window_netto_frame_rate": 1323.5655405366879,
  "window_mean_latency": 0.010577998565358708,
  "window_pass_max_latency": 0.013636589050292969,
  "window_fail_max_latency": 0,
  "window_first_latency": 0.013448953628540039,
  "window_start_timestamp": 1643120766.8596165,
  "window_stop_timestamp": 1643120772.9084067,
  "window_total_duration": 6.048790216445923,
  "window_brutto_batch_rate": 1307.2035427021144,
  "window_brutto_frame_rate": 1307.2035427021144,
  "window_batch_passrate": 1,
  "window_frame_passrate": 1,
  "window_pass_mean_latency": 0.010577998565358708,
  "window_fail_mean_latency": 0,
  "window_mean_latency2": 0.00011206056858306057,
  "window_stdev_latency": 0.0004080624147476522,
  "window_cv_latency": 0.03857652392617946,
  "window_pass_mean_latency2": 0.00011206056858306057,
  "window_pass_stdev_latency": 0.0004080624147476522,
  "window_pass_cv_latency": 0.03857652392617946,
  "window_fail_mean_latency2": 0,
  "window_fail_stdev_latency": 0,
  "window_fail_cv_latency": 0,
  "total_batches": 13259,
  "total_frames": 13259,
  "pass_batches": 13259,
  "fail_batches": 0,
  "pass_frames": 13259,
  "fail_frames": 0,
  "netto_batch_rate": 1330.8045962184033,
  "netto_frame_rate": 1330.8045962184033,
  "mean_latency": 0.010518890567313092,
  "pass_max_latency": 0.01539468765258789,
  "fail_max_latency": 0,
  "first_latency": 0.014332771301269531,
  "start_timestamp": 1643120764.8586895,
  "stop_timestamp": 1643120774.9077034,
  "total_duration": 10.049013853073 12,
  "brutto_batch_rate": 1319.432950721351,
  "brutto_frame_rate": 1319.432950721351,
  "batch_passrate": 1,
  "frame_passrate": 1,
  "pass_mean_latency": 0.010518890567313092,
  "fail_mean_latency": 0,
  "mean_latency2": 0.00011087626771836501,
  "stdev_latency": 0.00047875771665496417,
  "cv_latency": 0.04551408854301413,
  "pass_mean_latency2": 0.00011087626771836501,
  "pass_stdev_latency": 0.00047875771665496417,
  "pass_cv_latency": 0.04551408854301413,
  "fail_mean_latency2": 0,
  "fail_stdev_latency": 0,
  "fail_cv_latency": 0,
  "qos_quantile_0": "0.5",
  "qos_latency_0": 0.010580348253788 06,
  "qos_error_0": 1.3339214689263995e-06,
  "qos_quantile_1": "0.66",
  "qos_latency_1": 0.010761586482980283,
  "qos_error_1": 1.315893046997 46e-06,
  "qos_quantile_2": "0.75",
  "qos_latency_2": 0.010874164102808802,
  "qos_error_2": 1.304972121796838e-06,
  "qos_quantile_3": "0.82",
  "qos_latency_3": 0.010963328537935064,
  "qos_error_3": 1.2965423302011697e-06,
  "window_hist_factor": 10000,
  "window_hist_base": 1.8
}
```