# Understanding the Benefits of Hardware-Accelerated Communication in Model-Serving Applications

Walid A. Hanafy[a], Limin Wang[b], Hyunseok Chang[b], Sarit Mukherjee[b], T. V. Lakshman[b], and Prashant Shenoy[a]

[a]University of Massachusetts Amherst
[b]Nokia Bell Labs

***Abstract*—It is commonly assumed that the end-to-end networking performance of edge offloading is purely dictated by that of the network connectivity between end devices and edge computing facilities, where ongoing innovation in 5G/6G networking can help. However, with the growing complexity of edge-offloaded computation and dynamic load balancing requirements, an offloaded task often goes through a multi-stage pipeline that spans across multiple compute nodes and proxies interconnected via a dedicated network fabric within a given edge computing facility. As the latest hardware-accelerated transport technologies such as RDMA and GPUDirect RDMA are adopted to build such network fabric, there is a need for good understanding of the full potential of these technologies in the context of computation offload and the effect of different factors such as GPU scheduling and characteristics of computation on the net performance gain achievable by these technologies. This paper unveils detailed insights into the latency overhead in typical machine learning (ML)-based computation pipelines and analyzes the potential benefits of adopting hardware-accelerated communication. To this end, we build a model-serving framework that supports various communication mechanisms. Using the framework, we identify performance bottlenecks in state-of-the-art model-serving pipelines and show how hardware-accelerated communication can alleviate them. For example, we show that GPUDirect RDMA can save 15–50% of model-serving latency, which amounts to 70–160 ms.**

***Index Terms*—GPUDirect RDMA, model-serving, low-latency communication, edge computing**

## I. Introduction

The concept of edge computing was pioneered more than a decade ago [1], [2], and yet the role of edge computing remains critical even today because functional requirements and user expectations for applications have constantly been surpassing even the most sophisticated on-device capabilities [3]. For example, multi-user cloud gaming, machine learning (ML)-based wearable cognitive assistance, immersive 360-degree point cloud video streaming, industrial robot control, etc. heavily rely on geographically close-by compute resources, accessible to end devices via low-latency, high-throughput interconnects. In making edge computing a reality, there have been two important industry trends. On the networking side, the advances in 5G/6G technologies have been instrumental in enabling offloaded computation and associated data delivery to meet stringent latency/throughput requirements. On the computation side, the arrival of new chip technologies (e.g., GPU, TPU) has been a catalyst for accelerating and scaling required computation within server hardware.

As more and more practical use cases of edge offloading are emerging, driven by these trends, the research community has been dedicating significant research efforts to optimize the latency performance of edge offloading within a given edge computing infrastructure. There have been works on utilizing adaptive computation for low latency [4]–[6], and proposing intelligent workload scheduling in compute clusters or a single node [7]–[9]. Although the existing works differ in their approaches and scopes, one commonality they share is that the primary focus is on compute resources or computation itself, but not on the underlying networking, in particular, *the network fabric within an edge computing infrastructure*. A common assumption is that the end-to-end networking performance of edge offloading is purely dictated by that of the network connectivity between end devices and edge computing facilities, where ongoing innovation in 5G/6G networking can help. However, with the growing complexity of offloaded computation and dynamic load balancing requirements within a single edge domain, an offloaded task often goes through a multi-stage pipeline which spans across multiple compute nodes and proxies interconnected via a dedicated network fabric within a given edge computing infrastructure. Therefore, the performance of such internal network fabric and its interaction with task execution can also play a nontrivial role in the end-to-end latency of edge offloading.

The latest hardware-accelerated transport technologies such as Remote Direct Memory Access (RDMA) and GPUDirect are suitable for building the network fabric for interconnecting compute nodes in current edge computing environments [10]. Unlike TCP/IP-based communication, where a server CPU is involved in packetizing and transferring data via the operating system's protocol stack, RDMA and GPUDirect bypass the server CPU and the operating system, and directly write data into a destination processor's memory (it could be CPU memory or GPU memory). This remote zero-copy mechanism allows these technologies to decrease data transfer latency and increase service throughput, presenting a "lower-bound" latency to other remote computation offloading techniques. However, there is still a lack of understanding on the full potential of the existing hardware-based transport technologies in the context of computation offload and the effect of different factors (e.g., GPU scheduling, computation type and size) on

979-8-3503-9973-8/23/$31.00 ©2023 IEEE

the net performance gain achievable by them.

This paper aims to unveil detailed insights into the performance overhead of typical model-serving pipelines, which are often hard to get from existing feature-rich model-serving systems, and in turn, to highlight potential performance gains that could be achieved by adopting hardware-accelerated transport in different deployment scenarios. To achieve this goal, we build a model-serving application framework with support for different communication mechanisms (e.g., TCP, RDMA) and with the capability to provide fine-grained visibility into model-serving pipeline stages.[1] Such exploratory features are not available in existing off-the-shelf model-serving systems. We use the framework to explore a wide-range of scenarios that resemble real-world edge deployments.

Our systematic evaluation demonstrates that hardware-accelerated transport, in particular GPUDirect RDMA (GDR), presents a promising approach to build low-latency edge offloading infrastructures, saving 15–50% of model-serving latency, which translates to 70–160 ms, compared to TCP-based transport in a wide range of setups. This study helps us understand the benefits of hardware-accelerated communication in model-serving applications, as summarized below.

**(1) Communication fraction matters.** Hardware-accelerated transport provides the most benefit when communication takes a significant fraction of time in a given model-serving pipeline. With increasing GPU processing capabilities and application network I/O requirements, the proportion of communication overhead is expected to become more significant.

**(2) Protocol translation is worthwhile.** Adopting hardware-accelerated transport within a given compute cluster can provide substantial latency benefit compared to end-to-end TCP pipelines, even at the cost of protocol translation.

**(3) Data copies are bottlenecks.** Host-to-device (H2D) and device-to-host (D2H) copies can quickly become a bottleneck as concurrency increases within a GPU. Issuing copy commands interferes with execution in a GPU. GDR can alleviate these problems by skipping the GPU copy queues all along.

**(4) Effectiveness of prioritization is limited due to copy-engine's coarse granular interleaving.** GPU copy-engine's coarse granular interleaving limits the ability of high-priority clients to prioritize their execution over other clients.

## II. Background

In this section, we provide an overview of the technologies we evaluate in this paper.

### A. Edge Offloading

In a typical edge offloading architecture (Figure 1), end devices offload computational tasks (e.g., object recognition in a camera view, collision-free robot navigation) to a nearby edge computing facility via request/response transactions. When an end device requests for an offloading service, it submits corresponding data to a frontend gateway of an edge computing facility over existing access networks. The gateway

[1]The source code for the model-serving system is available at https://github.com/nokia/accelerated-offloading.

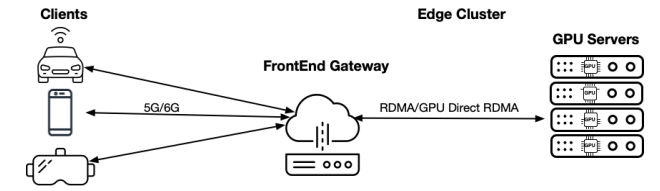
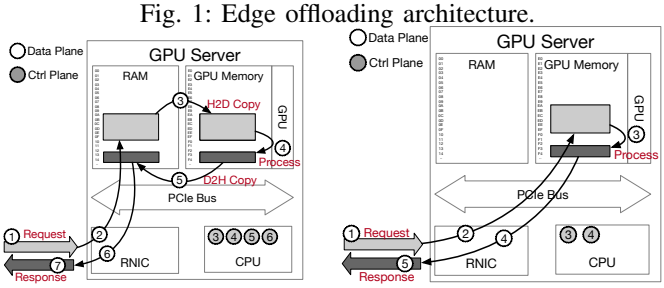



Fig. 1: Edge offloading architecture.




(a) RDMA (b) GDR

Fig. 2: Request-response transaction over RDMA/GDR.

then dispatches the request to compute servers available at the facility over an internal network fabric. Once the required computation gets executed on the data, a response is generated and sent back to the requesting device via the gateway. Unlike the external access networks interconnecting end devices and edge computing facilities, the edge-internal network fabric is under the control of a given edge computing facility, and can be enhanced by leveraging the latest hardware-accelerated network fabric technologies such as RDMA and GPUDirect. In the following, we describe how an edge offloading task can be accelerated within a given edge computing facility via RDMA and GPUDirect RDMA.

### B. Edge Offloading over RDMA

RDMA is a hardware mechanism through which a local peer can directly access a remote peer's memory without the intervention of the remote peer's CPUs and the network stack traversal overhead. RDMA was originally designed to interconnect high-performance computing (HPC) clusters on specialized high-throughput, low-latency InfiniBand (IB) networks. Many of today's RDMA deployments are based on RoCEv2 (RDMA over Converged Ethernet), where packets are encapsulated in UDP/IP packets and carried over the commodity Ethernet fabric. Figure 2(a) depicts a typical workflow of a compute-intensive request utilizing a GPU on a remote server over RDMA. When the request along with necessary data arrives at the server's RDMA-capable NIC (RNIC) as RDMA traffic, it gets DMAed to the server's RAM (steps 1–2). Then it gets copied from RAM to GPU memory and serviced in the GPU, after which the result gets copied back to RAM (steps 3–5). As a response, the result is DMAed from RAM to the RNIC and sent out as outgoing RDMA traffic by the RNIC (steps 6–7). Throughout the workflow, the CPUs of the server may issue control plane instructions at certain steps (e.g., steps 3–6), but is not directly involved in data movement.

### C. Edge Offloading over GPUDirect RDMA

GPUDirect is a suite of technologies introduced by NVIDIA to enhance data movement and access for their GPUs. In par-

ticular, GPUDirect RDMA (GDR) enables PCIe devices like RNICs to directly access GPU device memory. This eliminates the involvement of CPUs and the staging buffer copies of data via main memory for inter-node GPU communication, thereby reducing CPU overhead and improving latency. To support GDR, NVIDIA provides an operating system extension that enables DMA bus mapping of GPU device memory to allow GPU memory to be directly used as RDMA target memory regions just like normal main memory. With GDR, leveraging a remote GPU for an edge offloading task can be substantially simplified, as shown in Figure 2(b). Since GPU device memory can be directly employed as RDMA target memory regions, an incoming service request and its data can be DMAed by the RNIC to GPU memory without going through system RAM (steps 1–2), and GPU processing can take place right away (step 3). Similarly, the service result can also be directly DMAed from GPU memory to RNIC for output (steps 4–5). The copy in/out of data between RAM and GPU memory in Figure 2(a) is completely avoided.

### D. GPU Scheduling

Typical GPU hardware contains an array of multi-threaded execution engines called Streaming Multiprocessors (SMs), as well as a multi-level memory system and dedicated copy engines that copy data to and from host memory over the PCIe bus, in parallel to execution engines. Each SM comprises a large number of processing cores known as CUDA cores in the case of NVIDIA GPUs. NVIDIA GPUs are programmed on the CUDA platform, which presents a programming model and easy-to-use APIs for utilizing available CUDA cores. A typical CUDA program is composed of multiple kernels (functions) designed to exploit the parallel processing of CUDA cores. Kernels are grouped into blocks, where each block contains multiple threads.

Although GPU scheduling algorithms on NVIDIA hardware platforms are proprietary, researchers have been trying to gain insights on them [11], [12]. In summary, to issue kernels or to allocate memory, programs create a CUDA context which communicates with the GPU driver and holds the execution state. CUDA contexts issue kernels to a default stream called *NULL* stream. For higher GPU utilization and faster execution, it is possible to use multiple streams. With multiple streams, the execution engine schedules kernels' blocks across streams in a priority-accommodating round-robin fashion [11], [12]. Once kernels are in the execution engine queue, their blocks are scheduled in an FCFS fashion, where each block is launched based on the availability of CUDA cores and memory. CUDA streams also support priority-based scheduling, where different CUDA streams have different priorities. The priorities affect the execution on the block level in a non-preemptive way. Another way of sharing GPUs is by using multiple contexts over multiple threads or processes. In this case, GPU execution engines are shared among contexts in a time-sliced fashion. Finally, NVIDIA GPUs support Multi-Process Service (MPS), which allows packing threads from multiple contexts to be running at the same time [13]. This

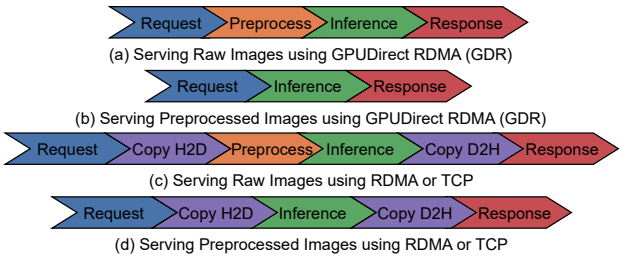


Fig. 3: Model-serving stages.

resembles the execution of multiple streams without potential head-of-queue blocking in streams. Although all these resource sharing methods increase system efficiency at scale, they come with the cost of lower performance predictability. Researchers have developed methods to cope with the unpredictability [7], or improve predictability by disabling sharing methods [8]. However, the trade-off between predictability and utilization has not been fully explored.

## III. Methodology

In this paper, we seek to investigate the role of hardware-accelerated network fabric in enabling low-latency computation offload at the edge. Given that ML model-serving is a popular type of edge-offloaded computation, we focus on model-serving offload. In order to study the complex interplay between network fabric technologies and the rest of model-serving pipelines and derive generalizable findings, what is needed is somewhat open-ended deployment environments, where we can easily enable different features and analyze their impact. Off-the-shelf model-serving systems [14], [15] are not suitable in this respect due to the following reasons. First of all, the existing systems only support TCP-based application protocols such as HTTP and GRPC, but do not support hardware-accelerated transport primitives such as RDMA and GDR. Second, as deployment-ready systems, they do not come with fine-grained profiling and tracing capability that will help us understand performance bottlenecks in model-serving pipelines. Finally, production-grade systems often come with various add-on features such as security, clients priority, etc., which may not help with or even complicate our investigations. These factors motivate us to build our own model-serving system from scratch that is flexible and generic enough to serve as a reference model-serving testbed. In the following, we describe the functional details of our model-serving system.

### A. Model-Serving Pipeline Stages

In our framework, a model-serving pipeline consists of request handling, preprocessing, inference, and response handling stages. The preprocessing stage ensures that client-submitted data is compatible with the model requirements (e.g., input size) in case that the client submits raw data. The inference stage executes a given model with client data. The request/response handling stages proceed differently based on the underlying transport mechanism, as explained below.

To support model-serving over RDMA, a server and a client first go through an RDMA-specific connection setup procedure. During this time, they create a set of queues for send-

TABLE I: Performance metrics.

| Category | Metric | Description |
|---|---|---|
| | `total-time` | End-to-end model-serving latency |
| Transport | `request-time` | Time taken to send a request |
| | `response-time` | Time taken to send a response |
| GPU | `copy-time` | H2D copy time + D2H copy time |
| | `preprocessing-time` | Time taken in preprocessing |
| | `inference-time` | Time taken in model inference |
| CPU | `cpu-usage` | CPU usage in user and kernel |
| Memory | `memory-usage` | RAM and GPU memory usage |

ing/receiving data over RDMA and for receiving work completion events, allocate memory buffers to hold request and response data, and exchange connection-related metadata [16]. Once a connection is created, the client sends a model-serving request to the server by posting a work request (WR) to its send queue, and blocks until it receives work completion (WC) events for the request as well as for a corresponding response from the server. We use `RDMA_WRITE` operation for both the request and the response. When the server receives a WC event for the request, as shown in Figure 2(a), it first copies the data from the client's request buffer to the GPU memory using `cudaMemcpy` with `cudaMemcpyHostToDevice` flag. Then it processes the request according to the application requirements. Lastly, it copies the data back to the client's response buffer, this time with `cudaMemcpyDeviceToHost` flag, and pushes a WR to its send queue and waits for a WC event. GDR follows the same steps as RDMA, except that, on the server, we allocate GPU memory rather than host memory, and that H2D and D2H copies are omitted (Figure 2(b)).

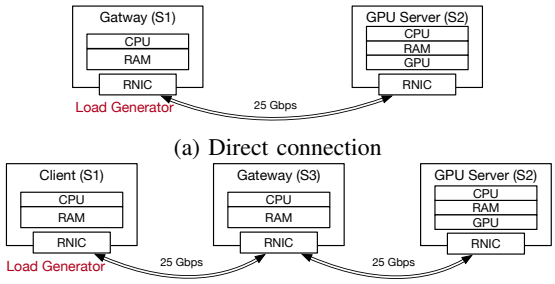
For TCP-based transport, we choose ZeroMQ [17] over HTTP and GRPC for the following reason. The RDMA-based transport in our framework allows data to be transmitted with memory read/write semantic, and hence does not incur data (de)serialization overhead. Whereas common TCP-based protocols like HTTP and GRPC require data (de)serialization, and therefore comparing end-to-end latency between HTTP/GRPC and RDMA is not fair. Unlike HTTP/GRPC, ZeroMQ does not require data serialization, and hence it can be a fair comparison with RDMA protocol. We use a Router-Dealer proxy where the server allocates the same number of threads as the number of clients. Each thread reuses its memory buffers to avoid memory allocation overheads. Figure 3 summarizes the model-serving pipelines for different communication mechanisms. The difference lies in the processing stages for raw and preprocessed data and the steps on the server side where data copies are selectively needed.

### B. Performance Metrics

To understand the performance bottlenecks of a model-serving pipeline, fine-grained visibility into the pipeline is required. Thus, we enable detailed time profiling for individual pipeline stages in the model-serving system. The client-perceived end-to-end model-serving latency is broken into two components: (i) transport latency and (ii) GPU latency. Within GPU latency, `copy-time` is only applicable to TCP/RDMA-based communication as GDR moves data directly to GPU memory. To measure GPU-related latency

TABLE II: DNN models used.

| Model | Task | GFLOPS | Input Shape | Output Shape |
|---|---|---|---|---|
| `MobileNetV3` | Classification | 0.06 | 3×224×224 | 1×1000 |
| `ResNet50` | Classification | 4.1 | 3×224×224 | 1×1000 |
| `EfficientNetB0` | Classification | 0.39 | 3×224×224 | 1×1000 |
| `WideResNet101` | Classification | 22.81 | 3×224×224 | 1×1000 |
| `YoloV4` | Detection | 128.46 | 3×416×416 | S×S×3×85, S = {13, 26, 52} |
| `DeepLabV3_ResNet50` | Segmentation | 178.72 | 3×520×520 | 2×21×520×520 |




(a) Direct connection

(b) Proxied connection

Fig. 4: Connection modes.

components, we inject CUDA events between steps and measure the time between the events. Transport delay components `request-time` and `response-time` capture client-to-server and server-to-client communication overhead, respectively, which are measured differently for different transport methods. For RDMA/GDR-based transport, which is offloaded to an RNIC, we measure the delay as the time between posting an RDMA WR and receiving a corresponding WC event. TCP-based ZeroMQ communication overhead is measured with processing time for `zmq_send()` API on server response, while the request time is the time difference between the `total-time` and total server time. Besides measuring model-serving latency, we also capture CPU/memory resource usages using Linux `/proc` file system and `nvidia-smi`. All reported performance metrics are summarized in Table I. The metrics are collected with a varying number of clients, where each client sends 1000 requests in a closed-loop fashion.

### C. Experimental Scenarios

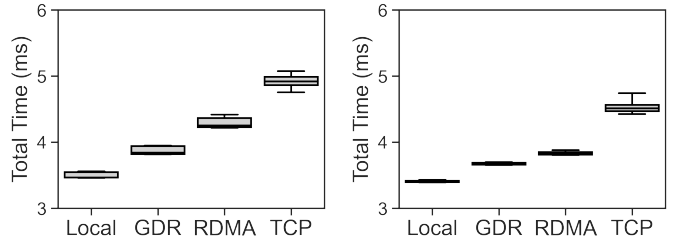
The flexibility of our model-serving system allows us to evaluate model-serving pipelines across a wide range of deployment environments as explained below.

**Transport mechanism.** It supports four types of transport mechanisms: (i) local, (ii) RDMA, (iii) GDR, and (iv) TCP (ZeroMQ). In "local" processing, a client processes data on a local GPU without offloading. Hence it only incurs processing and inference latency, but no delay from data movement. This presents a lower bound on achievable end-to-end latency.

**Connection mode.** It supports two common connection scenarios between a client and a GPU server: (i) direct connection, and (ii) proxied connection. They are compared in Figure 4. The direct connection mode illustrates the connectivity between a gateway and a GPU server within an edge computing facility. In this case, we deploy our load generator on the gateway server. Note that both the gateway and the GPU server must be equipped with RNICs to support hardware-accelerated transport. The proxied connection mode represents the case where client-to-server communication is proxied by

TABLE III: Testbed configuration.

| Name | Server type | CPU | GPU | NIC |
|---|---|---|---|---|
| S1 | Dell PowerEdge R740 | Intel Xeon-G 6240 | - | ConnectX-5 25GbE |
| S2 | Dell PowerEdge R740 | Intel Xeon-G 6240 | NVIDIA A2 | ConnectX-5 25GbE |
| S3 | Dell PowerEdge R750 | Intel Xeon-G 6330 | - | ConnectX-5 25GbE |




(a) Raw images (b) Preprocessed images

Fig. 5: Total latency across mechanisms for `ResNet50`.

an intermediate gateway. To focus on the effect of networking rather than the gateway's scheduling decision, the gateway is configured to forward client requests to a fixed server.

**Workload.** To evaluate the effect of different jobs and data sizes, we deploy several different DNN models as shown in Table II. The models differ in terms of functionality, model complexity, and communication overhead (i.e., input/output sizes). The classification models are trained with ImageNet [18], while detection and segmentation models are trained with Microsoft COCO [19].

**GPU configuration.** Finally, we evaluate common methods to control processing latency within a GPU by varying client concurrency, client priority, and GPU sharing modes, namely `multi-stream`, `multi-context`, and `MPS`.

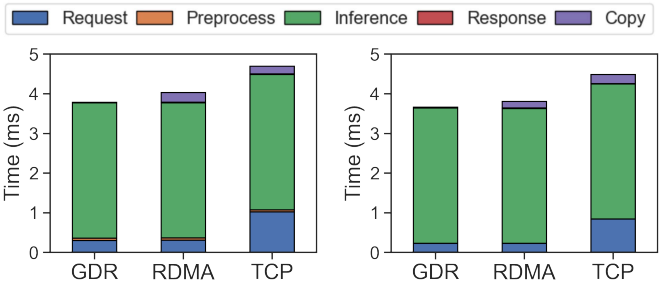
### D. Implementation and Deployment

In implementing the aforementioned system, we use NVIDIA OFED v5.6 for RDMA communication, and ZeroMQ v2.1 for TCP-based communication. For model-serving pipelines, we use CUDA toolkit v11.6.2, OpenCV v4.5.5, and TensorRT v8.4. The client and server are written in C++ and comprise ∼4.5k SLOC. The model-serving system is deployed on three servers described in Table III. S2 is equipped with NVIDIA A2 GPU, which has 10 execution engines, 16 GB memory, and two copy-engines. All servers are running on Ubuntu v20.04 LTS and kernel v5.15.

In the rest of the paper, we utilize our platform to systematically examine the latency performance of model-serving under different scenarios. Starting from employing a single client session to identify the bottlenecks of the model-serving pipeline without resource sharing and contention, we further study the impact of concurrency when offload resources are subject to competition from multiple clients. Lastly, we explore the trade-offs of the mechanisms used to tame the overhead of concurrency. In all cases, different transport, workload, and connection mode combinations are explored.

## IV. Single Client Performance

In general, when a client offloads a model-serving computation to a server, its end-to-end latency is determined by how the client's model-serving request is delivered to the target compute resource (transport delay), as well as how the request is executed within the compute resource (execution delay). Within an edge computing facility, the transport delay can vary with different transport mechanisms, while the execution delay can be influenced by how the compute resource is shared to handle concurrent client requests. In the first set of evaluations, we focus on the transport delay, while discounting the effect of a specific GPU scheduling algorithm. For this purpose, we evaluate the latency performance of model-serving across different transport mechanisms when running a *single client*, which shows the performance of model-serving without any interference from sharing edge network or compute resources.



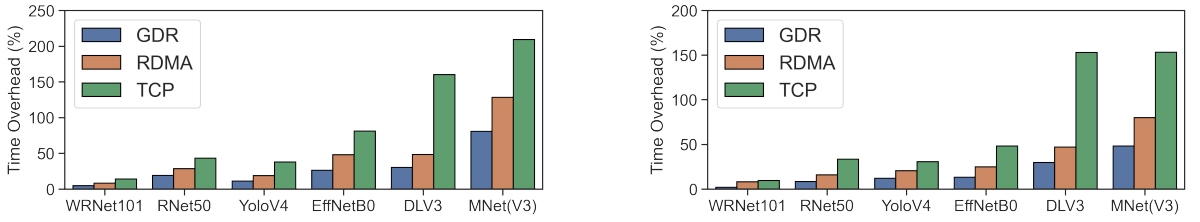

(a) Raw images (b) Preprocessed images

Fig. 6: Latency breakdown across mechanisms for `ResNet50`.

### A. Direct Connection

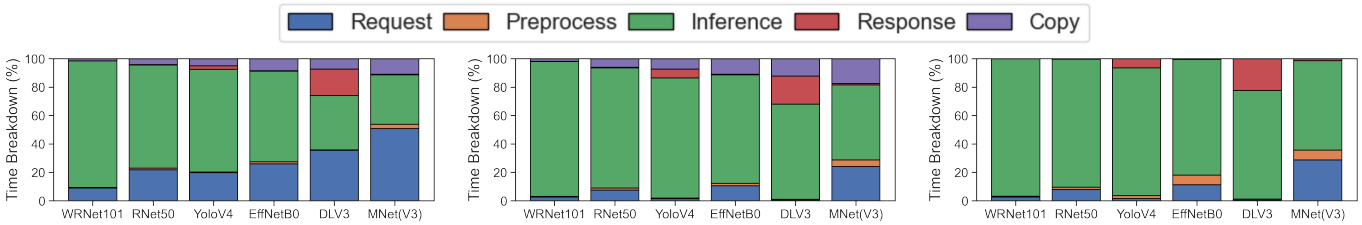
We start by evaluating model-serving performance in the most simplistic scenario, where model-serving requests are handled in the direct connection mode. In this mode, we exclude the the client-to-edge latency and focus on the latency with the edge network fabric. In this case, we run the load generator on the gateway server itself. In Figure 5, we compare model-serving latency across different transport mechanisms when `ResNet50` is used (Table II). We add "local processing" as a reference. We repeat the experiments with and without preprocessing. The figure shows that GDR and RDMA perform better than TCP-based transport. When the server performs preprocessing (Figure 5(a)), GDR and RDMA incur 20.3% and 11.4% less latency than TCP, respectively. Without preprocessing (Figure 5(b)), GDR and RDMA lead to 23.2% and 15.2% shorter delay than TCP, respectively. The relative performance of hardware-accelerated transports, compared to TCP, is more pronounced when preprocessing is not needed because the overall model-serving pipeline takes less time to execute. Compared to local processing, GDR-based model-serving adds as low as 0.27–0.53 ms, while TCP adds 1.2–1.5 ms, depending on whether or not preprocessing is needed.

To understand the source of difference, we plot the latency breakdown in Figure 6. The figure shows that the difference between GDR/RDMA and TCP comes from data transfer time. For example, TCP-based transport takes 0.73 ms and 0.61 ms more to send raw and preprocessed data than GDR and RDMA-based counterparts, respectively. GDR outperforms RDMA as it skips H2D and D2H copies, which saves extra 0.3 ms and 0.2 ms when handling raw and preprocessed data, respectively. This highlights the advantage of GDR and quan-




(a) Raw Images (b) Preprocessed Images

Fig. 7: Model-serving latency overhead with respect to local processing for different DNN models.




(a) TCP (b) RDMA (c) GPUDirect RDMA (GDR)

Fig. 8: Latency breakdown across different transport mechanisms for different DNN models.

tifies potential bottlenecks across protocols when the number of clients increases, which will be demonstrated in Section V.

To generalize these findings, we repeat the experiments with other ML models of varied complexity and I/O sizes, as listed in Table II. Figure 7 shows the latency overhead with respect to local processing for different models. That is, with each model, it shows how much longer latency is incurred by offloading the model compared to executing it locally. Figures 7(a) and (b) show the latency overhead when sending raw and preprocessed images, respectively. In both cases, GDR outperforms the alternatives as expected. However, the overhead greatly varies across models. It shows that smaller models tend to have higher overhead than bigger models, and that models with larger I/O have higher overhead as well. This is because smaller models and models with larger I/O sizes have a higher fraction of time spent in the communication stage, making the role of transport mechanisms more vital than bigger models with smaller I/O sizes. For example, offloading `MobileNetV3` adds at least 80.8% and 48.1% overhead compared to local processing, while offloading `WideResNet101` adds just about 4.5% and 2% overhead when serving raw and preprocessed images, respectively. Models with large I/O sizes (e.g., `DeepLabV3`) also show very high overhead, especially with TCP.

To quantify these overheads, Figure 8 depicts the fraction of time spent in each stage. The result confirms our hypothesis. For instance, when serving `MobileNetV3`, 62%, 42%, and 30% of total time is spent in data movement (`copy-time` + `request-time` + `response-time`) when serving requests over TCP, RDMA, and GDR, respectively, while in `WideResNet101`, the fraction of communication overhead does not surpass 10% in all cases. The result also shows the merits of hardware-accelerated transport for large I/O. For example, when serving raw data using `DeepLabV3`, TCP spends 60% of the overall latency in data movement, while

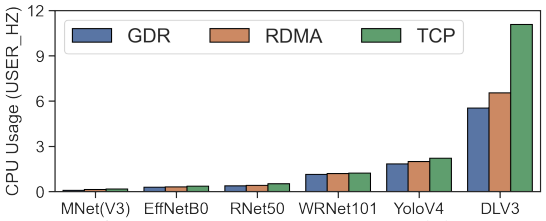



Fig. 9: CPU Usage across models.

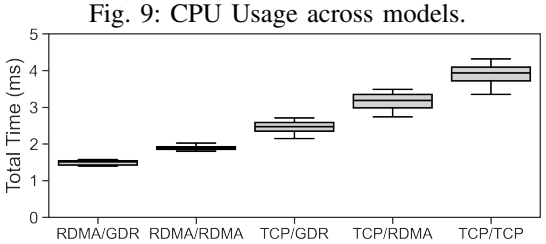


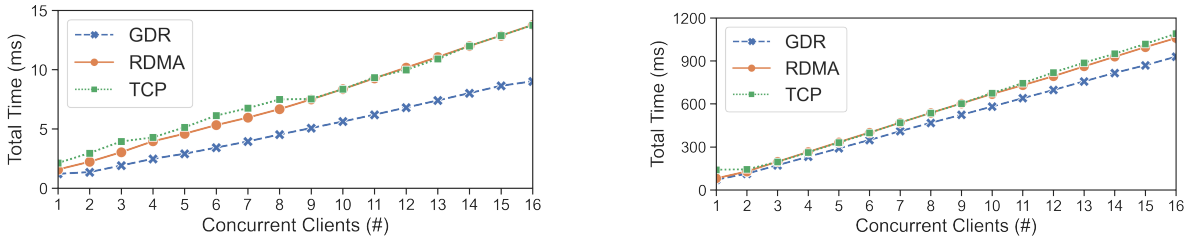

Fig. 10: End-to-end latency with proxied connection.

RDMA and GDR spend only 32% and 23%. In this case, the overhead of large I/O size is translated to higher latency difference, where TCP-based transport adds 71 ms and 68 ms, compared to GDR and RDMA-based transports, respectively. Note that, with more powerful accelerators and more I/O-intensive immersive application offloading, the fraction of time spent in actual processing will become smaller, which will further increase the importance of transport methods.

Finally, Figure 9 shows the CPU usage per request across different models. It shows that TCP-based transport incurs the highest CPU usage as the CPU is involved in communication. The overhead is most visible when serving `DeepLabV3` as its I/O size is high, where TCP adds 100% more CPU usage than GDR-based transport. It also shows that issuing copy operations for RDMA adds only a minor effect.

### B. *Proxied Connection*

Next, we switch to a more realistic scenario, where a client sends its requests to an intermediate gateway which then steers the requests to an appropriate GPU server. To focus on the




(a) `MobileNetV3` (b) `DeepLabV3`

Fig. 11: Total time across clients when processing raw images.

effect of transport mechanisms in such "proxied-connection" scenario, we exclude the overhead of server selection within the gateway by letting the gateway forward requests to a fixed server. In this scenario, client-to-gateway and gateway-to-server communication can be realized with available transport mechanisms independently. This results in the following configuration pairs for client-to-gateway and gateway-to-server transports: (i) RDMA/GDR, (ii) RDMA/RDMA, (iii) TCP/GDR, and (iv) TCP/RDMA. Lastly, we add TCP/TCP as a representative of existing model-serving frameworks.

Figure 10 plots model-serving latency results when a client submits raw data to `MobileNetV3`. It shows that hardware-accelerated transports can improve model-serving latency even if applied only to the last hop of the communicating path. Compared to end-to-end TCP connections (i.e., TCP/TCP), adopting hardware-accelerated transport between the gateway and the GPU server saves 23% and 57%, when replaced with RDMA (i.e., TCP/RDMA) and GDR (i.e., TCP/GDR). The results also show that TCP introduces higher performance variation, but the usage of hardware-accelerated communication, even partially, can reduce its effects.

***Key takeaways:*** *GDR can provide significant gains when communication comprises a high fraction of end-to-end latency. Hardware-accelerated transport can alleviate the overhead of proxied connections which are common in large-scale and dynamic model-serving environments.*

## V. Performance Scalability

Next, we study performance scalability and the effect of sharing the compute infrastructure across multiple clients. In this set of experiments, our load generator starts multiple instances of a client application, each issuing model-serving requests concurrently. On the GPU-server side, requests from each client are handled by a dedicated stream.

### A. Direct Connection

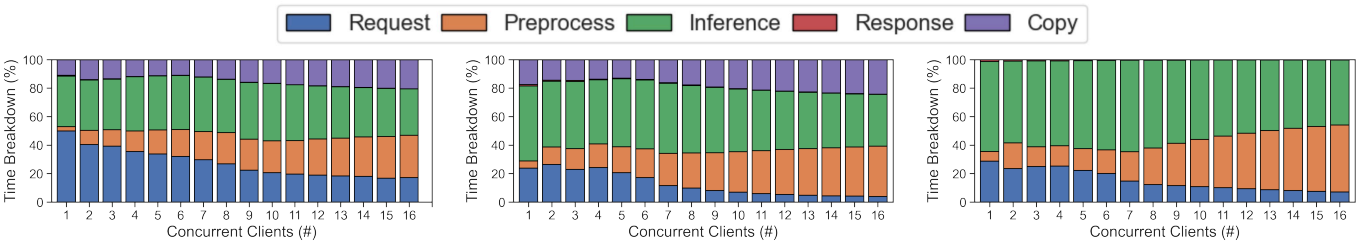
First, we evaluate performance scalability using the direct connection mode. Figure 11 shows the model-serving latency across a varying number of clients when `MobileNetV3` and `DeepLabV3` are used with raw images. In both cases, GDR outperforms both RDMA and TCP, and the gap between the two increases with the number of clients. For instance, with 16 clients, GDR saves 4.7 ms and 160 ms for `MobileNetV3` and `DeepLabV3`, respectively, compared to TCP. Surprisingly, however, the gain from using RDMA is lost with more clients, making its performance equivalent to that of TCP.

To understand the reason for this behavior, we examine how latency breakdown changes as we increase the number of clients. Figures 12 and 13 show the fraction of time spent in each stage when serving raw images using `MobileNetV3` and `DeepLabV3`. As the number of clients changes, different models and transport mechanisms develop different bottlenecks. For instance, for `MobileNetV3`, the fraction of processing time (`preprocessing-time + inference-time`) increases from 38% to 62%, from 58% to 72%, and from 70% to 92% when TCP, RDMA, and GDR are used, respectively. Having the processing delay as the largest component makes network overheads negligible, which is a desired goal in edge offloading. However, for RDMA and TCP, increasing the copy time presents a steady source of overhead, where the GPU copy engine becomes the bottleneck. This explains the performance similarity between RDMA and TCP. The figure also shows that network I/O (request and response) never becomes a bottleneck in these two cases.

On the other hand, when the more complex `DeepLabV3` is served, processing time has less impact on the performance, especially for RDMA and TCP. For instance, in case of GDR shown in Figure 13(c), processing time dominates the pipeline, which is indicated by the low transport overhead. However, in case of TCP and RDMA (Figures 13(a) and 13(b)), although processing time still dominates the overhead, the copy-time overhead becomes significant. The copy-time changes from 7% to 36% (10–366 ms) for TCP, and from 12% to 28% (9–264 ms) for RDMA.

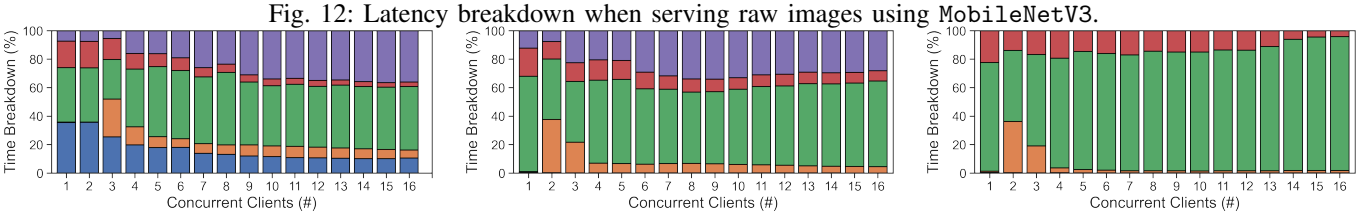
### B. Proxied Connection

Figure 14 shows performance scalability in the proxied connection mode. Similar to Section IV-B, we compare five different configurations for the proxied connection when serving raw images with `MobileNetV4`: (i) RDMA/GDR, (ii) RDMA/RDMA, (iii) TCP/GDR, (iv) TCP/RDMA, and (v) TCP/TCP. The figure shows that, as the number of clients increases, the behavior of different configurations changes in a counter-intuitive way, compared to the single client case discussed in Section IV-B. For instance, TCP/GDR outperforms the hardware-accelerated connection (RDMA/RDMA), and it even becomes comparable to the RDMA/GDR case. Moreover, the performance of end-to-end TCP-based transport (TCP/TCP) becomes similar to that of




(a) TCP (b) RDMA (c) GDR

Fig. 12: Latency breakdown when serving raw images using `MobileNetV3`.




(a) TCP (b) RDMA (c) GDR

Fig. 13: Latency breakdown when serving raw images using `DeepLabV3`.

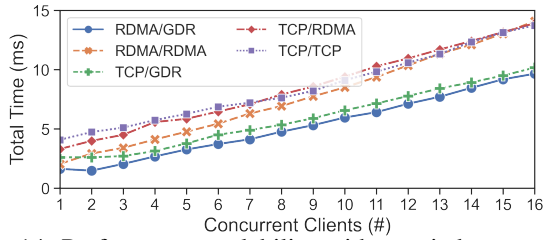



Fig. 14: Performance scalability with proxied connection.

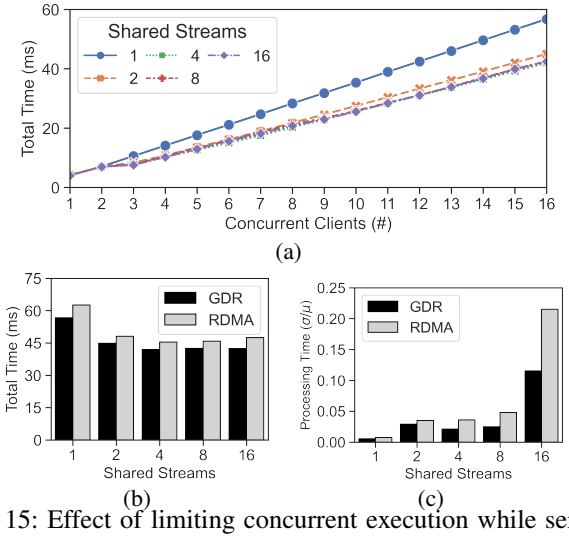



Fig. 15: Effect of limiting concurrent execution while serving `ResNet50`. (a) Scalability using GDR, (b) Total latency when serving 16 clients, (c) CoV in processing time.

hardware-accelerated transport in the last hop (TCP/RDMA) or end-to-end hardware-accelerated network (RDMA/RDMA). The resulting performance similarity between these methods (RDMA/RDMA, TCP/RDMA, and TCP/TCP) is caused by the bottleneck created by the copy engine, as explained earlier. We note that the usage of GDR in the last hop saves 27% compared to end-to-end TCP-based transport, while adding only 4% compared to the best case (RDMA/GDR).

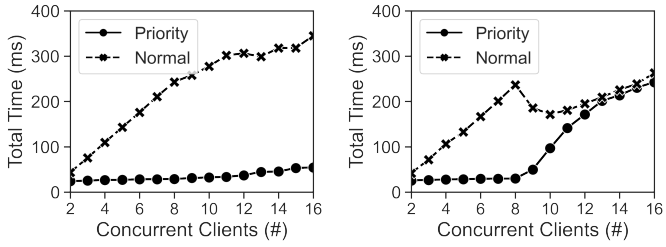
***Key takeaways:*** *The H2D/D2H copy quickly becomes a bottleneck in model-serving, which can easily remove any gain from hardware-accelerated RDMA communication. By skipping this step, GDR can yield higher scalability. This also applies to proxied communication, where skipping GPU's copy-engine with GDR greatly benefits the end-to-end latency. Adopting GDR at the last hop communication is comparable to utilizing hardware-accelerated communication end-to-end in terms of the overall model-serving latency.*

## VI. GPU Processing Management

As already shown in Section V, sharing a GPU among concurrent clients can add a significant overhead to the end-to-end model-serving latency. When we consider GPU resource sharing in the context of computation offload, the way GPU management can affect GPU processing is slightly different across different transport mechanisms. For instance, when using GDR, only execution engines are shared among clients. In the case of RDMA and TCP, *both* execution engines and copy engines are shared. To shed light on its implications, we evaluate different GPU management approaches and find out how effective they are in limiting GPU sharing overhead. Since TCP-based and RDMA-based transports use GPU resources similarly, we consider GDR and RDMA only.

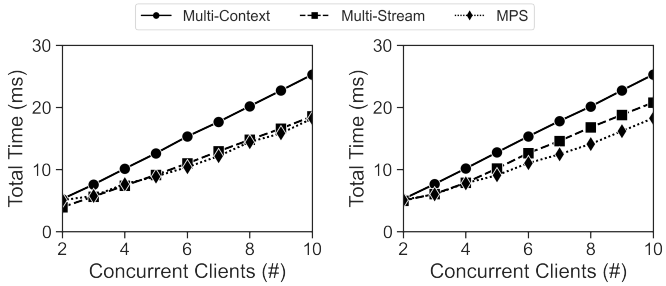
### A. Concurrent Execution

Sharing GPU resources among multiple clients increases device efficiency, but at the cost of increased variability in processing time [8]. A common approach to reducing this variability is to contain the level of execution concurrency. For GPUs, this can be achieved through limiting the number of execution streams. With a fixed number of streams, client requests are placed in a job queue till a stream becomes available. This presents a trade-off between efficiency and variability regarding concurrent execution. Here we try to quantify the trade-off with different spans of GPU concurrency, i.e., concurrency in execution engines only (with GDR) and concurrency in execution and copy engines (with RDMA).




(a) GDR (b) RDMA

Fig. 16: Single priority client with a varying number of regular clients, serving preprocessed images with `YoloV4`.




(a) GDR (b) RDMA

Fig. 17: Serving raw images using `EfficientNetB0` with different GPU sharing methods.

Figure 15 shows the effect of limiting the level of concurrency by scheduling clients with a varying number of streams on `ResNet50`. Figure 15(a) shows how model-serving scales when using GDR with different levels of concurrency. When there is only one stream to be shared by all clients (no-concurrent execution), end-to-end latency is 33% higher compared to the one-stream-per-client case (i.e., 16 streams for max concurrency). This is because limiting the number of execution streams increases requests' queuing delay. Figure 15(b) zooms in on the total time for 16 clients under different levels and spans of concurrency. As the number of streams increases, both GDR and RDMA yield latency reduction, but at a monotonically decreasing rate. This is because the level of multiplexing is limited by the model and device sizes. Since GDR skips the copy engine, it performs better than RDMA.

In Figure 15(c), we examine the effect of concurrency by quantifying variability in processing time with the coefficient of variation (CoV) computed as $\sigma/\mu$. In both RDMA and GDR, as expected, the processing time (which excludes copy process) is less variable when concurrency is limited. However, surprisingly, the variability is different between RDMA and GDR. For instance, for 16 clients, the CoV is 0.11 and 0.21 for GDR and RDMA, respectively. This contradicts with the assumed independence between execution and copy engines in GPUs. This is mostly an artifact of the fact that GPUs are managed using a single central unit (i.e., GigaThread Engine).

### B. Priority Clients

Next, we evaluate the effect of varying the priority of different clients under two different concurrency domains (i.e., GDR and RDMA). In this experiment, we run one high-priority client along with other normal-priority clients, and vary the total number of clients from 2 to 16. Figure 16 compares model-serving latency experienced by the priority and normal clients across GDR and RDMA. We use `YoloV4` with preprocessed images. For both RDMA and GDR, the latency for the priority client remains roughly the same with the number of clients until it reaches eight clients. In this case, the performance of the priority client greatly differs between RDMA and GDR. For instance, using GDR, the latency experienced by the priority client is $54\,\mathrm{ms}$ which is much lower than normal clients. However, in the case of RDMA, with more clients, the priority client's latency grows comparable to that of normal clients. This noticeable difference stems from the fact that, when using GDR, the priority is applicable only to execution that can be prioritized at a fine granularity of kernel block level [11]. In contrast, when using RDMA, the copy engine is interleaved at a coarse granularity of a request, limiting the ability of the priority client to occupy the copy engine. Note that, for models with smaller I/O, data copies will be interleaved at a relatively finer granularity, decreasing this effect on high-priority clients.

### C. Comparison of GPU Sharing Methods

Lastly, we compare several common GPU sharing methods under different concurrency domains. We adjust the levels of concurrency by (i) changing the number of streams (`multi-stream`), (ii) increasing the number of deployed application instances with multiple contexts (`multi-context`), or (iii) time-sharing multiple application instances via `MPS`, where no streams or applications are shared between users. Figures 17(a) and (b) compare model-serving latency among these three schemes when serving `EfficientNetB0` over GDR and RDMA. As expected, `MPS` always performs better than `multi-context` [13]. GDR and RDMA yield similar latency with `multi-context` and `MPS` across a varying number of clients, showing no clear benefit for GDR compared to RDMA. On the other hand, `multi-stream` exhibits different behavior between the two. As in Figure 17(a), when using GDR, the performance is almost identical between `multi-stream` and `MPS`, but using RDMA, `MPS` shows better performance. We hypothesize that, across processes, GPU copy engines are interleaved in a different way, which hides the copy overhead. We note that `multi-stream` uses multi-threading which shares the CUDA libraries on the GPU, while `multi-context` and `MPS` use multi-processing which restricts memory sharing, and hence limits the number of clients.

***Key takeaways:*** *Data exchange between the host and GPU memory imposes an interfering effect on processing. Stream priorities are more effective in sharing execution engines than the copy engine as the scheduling decision for execution engines is made on a fine granularity. The copy engine is shared differently between multiple streams and contexts.*

## VII. Limitations of RDMA and GDR

We show that RDMA and GDR are promising alternatives to TCP-based transport for latency-sensitive compute offloads. However, we acknowledge their drawbacks as follows.

**Memory overhead:** With RDMA and GDR, it is common that memory buffers are reserved and pinned per-client. This implies that the total number of sessions that can be supported will be limited, especially for GDR, as GPU memory is often more limited that host memory.

**Homogeneity:** RDMA transfers raw bytes between a client and a server. This requires data are stored homogeneously on both sides' memory, which might limit the interoperability. Proxied communication could alleviate this problem.

**GPU pinning:** GDR operates by allocating GPU memory buffers for each client. This ties a client to a specific GPU, or forces it to pay the data copying overhead between GPUs.

**GPU inadequacy:** With dedicated ASIC-based accelerators (e.g., image decoder), GPUs may not be an optimal choice for certain preprocessing tasks, where transferring data to the host memory via RDMA may be a better option than GDR. GPUDirect may still be used to move preprocessed data from an accelerator to a GPU directly, avoiding multiple data copies.

## VIII. Related Work

The performance of hardware-accelerated transports has been studied previously [20]–[25]. In contrast, we performed detailed performance evaluation of model-serving systems across a wide range of realistic scenarios. The most related works are [26]–[28]. Lynx [26] offloads the network stack from CPUs to SmartNIC cores. FlexDriver [28] leverages FPGA to build an on-accelerator hardware data-plane driver. GPU-Ether [27] implements native network I/O on GPUs themselves. While these works demonstrate the performance benefits of their point solutions, these efforts do not provide in-depth insights into the role of hardware-accelerated transport such as GDR actually plays. Our study bridges the gap and sheds light on this topic with important findings beyond simply showing which communication mechanism is the best.

## IX. Conclusion

In this paper, we presented a reference model-serving application framework with support for multiple communication mechanisms (TCP, RDMA, GDR) and the capability to profile a model-serving pipeline on a fine time granularity. Our evaluation results indicate that hardware-accelerated communication provides the most improvement when communication takes up a significant portion of the pipeline. Adopting hardware-accelerated communication within the compute cluster can significantly reduce latency compared to TCP-based pipelines. Our study also highlights that data copies and concurrent compute sharing can affect latency, and these insights can be used for better utilization of hardware-accelerated communication in various applications.

## Acknowledgement

The work was started while Walid was a summer intern at Nokia Bell Labs. Walid and Prashant were partly supported by NSF grants 2105494, 1908536, 2211302, and 2211888.

## References

[1] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, vol. 8, no. 4, 2009.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," in *Proc. MCC Workshop on Mobile Cloud Computing '12*, 2012.

[3] M. Satyanarayanan *et al.*, "The Role of Edge Offload for Hardware-Accelerated Mobile Devices," in *Proc. ACM HotMobile '21*, 2021.

[4] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency Online Prediction Serving System," in *Proc. USENIX NSDI '17*, 2017.

[5] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, "DeepSlicing: Collaborative and Adaptive CNN Inference With Low Latency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, 2021.

[6] S. Naveen, M. R. Kounte, and M. R. Ahmed, "Low Latency Deep Learning Inference Model for Distributed Intelligent IoT Edge Clusters," *IEEE Access*, vol. 9, 2021.

[7] Q. Liang, W. A. Hanafy, A. Ali-Eldin, and P. Shenoy, "Model-Driven Cluster Resource Management for AI Workloads in Edge Clouds," *ACM Trans. Auton. Adapt. Syst.*, 2023.

[8] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up," in *Proc. USENIX OSDI '20*, 2020.

[9] J. Soifer, J. Li, M. Li, J. Zhu, Y. Li, Y. He, E. Zheng, A. Oltean, M. Mosyak, C. Barnes, T. Liu, and J. Wang, "Deep Learning Inference Service at Microsoft," in *Proc. USENIX OpML '19*, May 2019.

[10] Ramki Krishnan and Chris Wright, "Microservices on the Edge: The Infrastructure Impact," in *Proc. IETF 98*, 2017.

[11] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, pp. 104–115, 2017.

[12] M. Yang, "Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems," in *Proc. 30th Euromicro Conference on Real-Time Systems*, 2018.

[13] Nvidia, "Multi-Process Service (MPS) ." https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2022.

[14] NVIDIA, "NVIDIA Triton Inference Server." https://developer.nvidia.com/nvidia-triton-inference-server, 2022.

[15] C. Olston *et al.*, "TensorFlow-Serving: Flexible, High-Performance ML Serving," in *Proc. ML Systems Workshop at NIPS 2017*, 2017.

[16] "RDMA Aware Networks Programming User Manual." Mellanox Technologies. Rev 1.7.

[17] ZeroMQ, "ZeroMQ." https://zeromq.org/.

[18] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *Int. Journal of Computer Vision*, vol. 115, no. 3, 2015.

[19] T.-Y. Lin *et al.*, "Microsoft COCO: Common Objects in Context," in *Computer Vision – ECCV 2014*, (Cham), pp. 740–755, 2014.

[20] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi, "Design and Performance Evaluation of NUMA-Aware RDMA-Based End-to-End Data Transfer Systems," in *Proc. SC '13*, 2013.

[21] J. Jose *et al.*, "Memcached Design on High Performance RDMA Capable Interconnects," in *Proc. ICPP 2011*, 2011.

[22] H. Subramoni, P. Lai, M. Luo, and D. K. Panda, "RDMA over Ethernet – A preliminary study," in *Proc. IEEE Cluster 2009*, 2009.

[23] A. Li *et al.*, "Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," *IEEE Trans. on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2020.

[24] A. Venkatesh *et al.*, "A high performance broadcast design with hardware multicast and GPUDirect RDMA for streaming applications on Infiniband clusters," in *Proc. IEEE HiPC 2014*, 2014.

[25] C.-H. Chu *et al.*, "Exploiting Hardware Multicast and GPUDirect RDMA for Efficient Broadcast," *IEEE Trans on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 575–588, 2019.

[26] M. Tork, L. Maudlej, and M. Silberstein, "Lynx: A smartnic-driven accelerator-centric architecture for network servers," in *Proc. ASPLOS '20*, p. 117–131, 2020.

[27] C. Jung, S. Kim, I. Yeom, H. Woo, and Y. Kim, "Gpu-ether: Gpu-native packet i/o for gpu applications on commodity ethernet," in *Proc. IEEE INFOCOM 2021*, 2021.

[28] H. Eran, M. Fudim, G. Malka, G. Shalom, N. Cohen, A. Hermony, D. Levi, L. Liss, and M. Silberstein, "Flexdriver: A network driver for your accelerator," in *Proc. ASPLOS '22*, 2022.