Toward GPU-centric Networking on Commodity Hardware

Massimo Girondi
girondi@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Mariano Scazzariello
marianos@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Gerald Q. Maguire Jr
maguire@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Dejan Kostić
dmk@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

ABSTRACT

GPUs are emerging as the most popular accelerator for many applications, powering the core of machine learning applications. In networked GPU-accelerated applications input & output data typically traverse the CPU and the OS network stack multiple times, getting copied across the system's main memory. These transfers increase application latency and require expensive CPU cycles, reducing the system's efficiency, and increasing the overall response times. These inefficiencies become of greater importance in latency-bounded deployments, or with high throughput, where copy times could quickly inflate the response time of modern GPUs. We leverage the efficiency and kernel-bypass benefits of RDMA to transfer data in and out of GPUs without using any CPU cycles or synchronization. We demonstrate the ability of modern GPUs to saturate a 100-Gbps link, and evaluate the network processing time in the context of an inference serving application.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; • **Networks** → **Cloud computing**.

KEYWORDS

GPUs, Commodity Hardware, Inference Serving, RDMA

ACM Reference Format:
Massimo Girondi, Mariano Scazzariello, Gerald Q. Maguire Jr, and Dejan Kostić. 2024. Toward GPU-centric Networking on Commodity Hardware. In *7th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3642968.3654820

1 INTRODUCTION

Many companies already run large numbers of inferences [23, 39], which represents the major cost in many AI-based applications deployments [3, 12]. Most of these applications run on commodity hardware equipped with several Graphics Processing Units (GPUs), as modern trends suggest [15]. With the growth in data volumes, advancements in networking speeds, and the increase processing capabilities of GPUs, the cost of data transfers in these scenarios is becoming increasingly important [5, 18], although it is highly overlooked with respect to other aspects of these technologies. While this performance gap introduces limitations in all GPU-accelerated applications, this is exacerbated in bandwidth-intensive ones, *e.g.,* in video-related use-cases.

The importance of this issue will become more prominent with the deployment of GPU-demanding applications at the edge, close to large groups of users actively generating and consuming vast amounts of data. In fact, positioning computational resources closer to the end-users improves overall performance by reducing latency and increasing availability. This is the case for Generative AI (GenAI) videos (*e.g.,* with OpenAI SORA [30]), as well as Augmented Reality and Virtual Reality technologies. In these scenarios, traditional offline compression techniques cannot be used, making the need for an efficient data transport mechanism paramount. Furthermore, GPU-driven GenAI videos use large models, which in turn require powerful GPUs. Similarly, video processing (*e.g.,* transcoding) workloads might require GPUs from particular vendors. All these reasons underscore the necessity for a service architecture that seamlessly connects a scalable, user-facing frontend to heterogeneous GPU inference servers through a commodity network.

In these contexts, the ability to move data to and from GPUs without needing direct CPU intervention is a key factor to enable efficient device integration. This enables the deployment of GPU-accelerated networked applications with little to no CPU involvement, thereby maximizing throughput and efficiency. Edge deployments favor cost-effective solutions; thus, being able to use low-end CPUs or even bypassing them would be highly beneficial. Given that power and cooling resources are often limited in these setups, reallocating energy from CPUs dedicated to network processing to GPUs can significantly enhance efficiency.

Remote Direct Memory Access (RDMA) technology can play a prominent role because it can potentially eliminate CPU usage beyond the basic initialization tasks. However, existing approaches (*e.g.,* GPUDirect [25] and GPUrdma [9]) cannot be directly applied since they were designed for homogeneous, dedicated networking and GPU equipment. In addition, they are geared for "collective" style communication (*e.g.,* AllReduce operations), rather than user-facing inference serving.

Contributions. Our contributions are as follows:

- **GPU-driven RDMA networking.** We propose the first, minimal implementation to control RDMA data transfers directly from GPUs, moving data directly between the network and the GPU memory avoiding expensive memory copies. We focus on commodity Ethernet and vanilla CUDA GPUs.
- **Zero CPU usage during actual inference.** Exploiting the ability of self-launching CUDA kernels, we propose an architecture to serve AI inferences without any CPU resources

EdgeSys '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0539-7/24/04.
https://doi.org/10.1145/3642968.3654820

during runtime; therefore, the CPU is freed from any GPU-related task, allowing it to be allocated for other processes.

We evaluate our contributions in the contest of a Machine Learning (ML) inference serving scenario, showing the benefits of controlling network transmissions directly from a GPU. Our solution bypasses the costly CPU-GPU synchronizations and reduces response times for common inference serving tasks by *up to 50%*. We demonstrate that (*i*) it is possible to run the RDMA stack on commodity GPUs, (*ii*) our approach introduces negligible runtime overhead in the GPU domain, (*iii*) no modification is needed in device drivers or operating system kernel, and (*iv*) standard protocols and interfaces could be used, fostering integration in already existing solutions.

2 BACKGROUND

The need to transfer ever-increasing data bandwidths, and the spread of bandwidth-intensive applications, fostered the development of alternative network stacks, with the aim of enhancing the efficiency of networked applications [4]. In modern datacenters, most of these approaches aim to bypass the kernel network stack, by either (*i*) using CPU cycles to poll packets from the network (*e.g.,* DPDK) or (*ii*) leveraging hardware accelerators to deliver network packets directly to main memory. The most common implementation of the second category is represented by RDMA, which allows network hosts to exchange data (*e.g.,* read/write memory regions) directly to each other's memory space, without CPU intervention [38].

2.1 RDMA Technology

Originally designed for High Performance Computing (HPC) settings to speed up large-scale parallel computations over InfiniBand, RDMA has emerged as an effective solution for commodity Ethernet datacenter networks as well. This broader application has been facilitated by enhancements in the Ethernet standard that enable RDMA to work even in lossy networks [16, 35]. RDMA transactions occur between two RDMA-enabled NICs, transporting data (*e.g.,* content of memory areas) between them over a specific network protocol (*i.e.,* RoCE). These network packets are generated and consumed directly by NICs, as configured and instructed by the application. RDMA support has been integrated into a wide range of commodity NICs, with software support embedded in most operating systems. In Linux, RDMA functionalities are provided through the `rdma_core` stack [1].

RDMA Verbs. RDMA communication is enabled by RDMA *verbs*, the semantic units representing operations to be executed between NICs involved in the RDMA communication. These verbs act as direct extensions of DMA mechanisms, extending them beyond the boundaries of individual machines. Verbs are implemented in specialized hardware accelerators embedded in RDMA-enabled NICs, called *Processing Units (PUs)*. The operation of requesting a transaction in the RDMA context is commonly referred as *posting a verb*. Beside some complex functions that RDMA supports (*e.g.,* atomic operations), RDMA verbs can be divided into *1-sided* and *2-sided* verbs. We mainly focus on the 1-sided verbs, which enable the receiver (*i.e.,* the server) to process data transfers without any CPU involvement. All the transfers are carried out by the NICs' PUs, making it an effective method for reducing CPU cycles consumption. The two main 1-sided verbs are *READ* and *WRITE*, allowing to fetch/send data from/to a remote destination, respectively.

RDMA in practice. Applications trigger network data transfers by instructing the NIC through a *Work Request (WR)*, a data structure that describes the specific verb to be executed. This WR includes details such as memory addresses, the memory area size, the type of verb, and additional flags that affect how the verb is processed. The memory address of the WR, located in the system's main memory, is then transferred to the NIC by writing its address into a specific PCIe register. To notify the NIC that a verb execution is pending, a specific counter called the *doorbell register* is advanced. In `rdma_core`, these steps are executed through a single API call (*i.e.,* `ibv_post_send`), which typically requires constant time, regardless of the data size to be transferred. The PUs in the NIC fetch the WR, and then the memory areas specified in the WR. These memory areas are in turn transferred to the remote party without any alteration (*i.e.,* the memory region is transmitted at byte-level) and with minimal network encapsulation. On the receiving end, when a verb from the NIC is received and all security checks are passed (*e.g.,* the memory addresses and requested operations are valid), the data is transferred via DMA transactions directly into the remote peer's memory. Optionally, acknowledgements and completion notifications may be issued to inform the sender or the receiver that the transfer has been completed. RDMA includes basic security mechanisms that allow to control *which* memory areas can be accessed, *how*, and by *whom*. These boundaries are generally configured during the initialization phase of the RDMA stack.

2.2 GPUs

GPUs[1] and the NVIDIA CUDA architecture are the *de facto* industry standard for AI workloads. The execution model of GPUs differs from the traditional execution model of CPUs as they usually feature a very large number of threads to execute each instruction [24]. The CUDA primitive execution unit is represented by *CUDA cores*, which are grouped into *Streaming Multiprocessors (SMs)*. Within each SM, CUDA cores are further grouped into fixed subsets called *warps*, with each *thread* assigned to execute on one or more of these warps. This hardware design of CUDA cores implies that a GPU achieves maximum efficiency when all 32 threads in a warp execute the same instruction concurrently. Therefore, GPUs are especially well-suited for large mathematical computation that can be split into smaller, parallel tasks, *e.g.,* matrix multiplications. GPU operations can be associated to a *stream*, which serves as an abstraction for a queue of operations awaiting execution. Streams allow finer control of some mechanisms and higher parallelism among different operations running on the same GPU. Additionally, CUDA operations can be further improved by using *Graphs*. This mechanism allows to record a sequence of actions and replay them in the same order, achieving greater efficiency than launching individual *kernels*.[2]

3 MOTIVATIONS

GPUs are widely used to accelerate AI workloads. The highly parallel nature of most AI workloads calculations makes GPUs an ideal execution target. GPUs, as well as other specific Deep

[1] More specifically, General Purpose Graphics Processing Units (GP-GPUs).
[2] Traditionally, functions running on a GPU are referred as *kernels*.

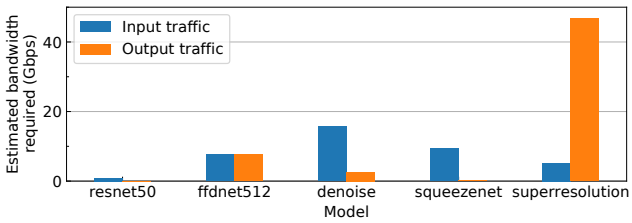

Figure 1: Input and output bandwidth for some common ML models, when running on a NVIDIA A100 with standard I/O sizes. The bandwidth is estimated by multiplying the I/O sizes by the measured inference rate.

Learning (DL) hardware, are often used as *offload accelerators* in an asynchronous fashion, *i.e.,* applications can continue processing other data or tasks while computations are run on the dedicated hardware. This allows scaling of accelerators and CPUs independently for each application, according to the specific requirements and the type of workload (*e.g.,* more intensive pre-processing, hybrid architectures, or memory-bounded models). Most of these accelerators are specifically optimized for running ML operations and have limited utility for general-purpose programming [21, 37]. Consequently, this situation necessitates the use of heterogeneous platforms, where certain operations are carried out on general-purpose CPUs, highlighting the critical role of the communication mechanisms in place for data exchange between the different devices involved. GPUs are typically installed in host machines (*e.g.,* servers), where the CPU has the burden of initializing, orchestrating, and issuing workloads to them. Most of these systems operate on commodity x86 platforms and are interconnected via a PCIe bus, the *de facto* industry standard for servers, desktops, and edge devices. Current GPUs (*e.g.,* NVIDIA H100 [28]) support PCIe 5.0, which provides a maximum theoretical bidirectional bandwidth of 512 Gbps. In addition to the PCIe bus, most datacenter NVIDIA GPUs can leverage the proprietary NVLink protocol [19]. The latest, fourth-generation NVLink achieves up to 7 200 Gbps of bandwidth between two GPUs, either by direct connection or through an NVSwitch fabric [6].

Inference serving requires CPU intervention. The growing interest in embedding AI functionalities into various products is increasing the demand for *inference-serving* systems, where AI models process data received from remote clients (*e.g.,* through an API). Although popular frameworks (*e.g.,* TensorFlow and PyTorch) provide mechanisms to integrate with other systems, almost all of these approaches predominantly rely on CPU processing, involving multiple data copies within the system. Typically, serving an inference requires at least *four* memory transfers: (*i*) NIC to RAM, (*ii*) RAM to GPU, (*iii*) GPU to RAM, and (*iv*) RAM to NIC. In applications where no additional CPU processing is required, these transfers only represent an overhead for the system, increasing total latency, power consumption, and limiting the throughput of the entire processing chain. Clockwork [13] carefully controls low-level mechanisms to meet consistent and reliable SLOs, while scaling among a large number of clients. Similarly to many production frameworks [8], Clockwork uses gRPC to move I/O across the different parts of the system, using CPU cycles to copy data between the protocol buffers and the GPU's memory.

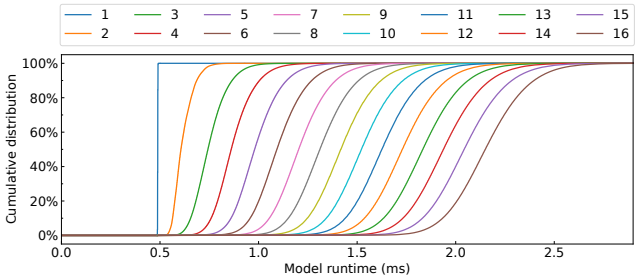

Figure 2: Inference time distribution at various concurrency levels for the *squeezenet* model. Lines are ordered left-to-right according to the concurrency level.

Networking tasks in inference-serving systems demand high CPU loads. The growing performance of GPU-accelerated applications significantly influences the rate at which data is processed by a GPU and, in turn, the volume of data exchanged over the network in client-server setups. Fig. 1 shows the bandwidth measured for a set of state-of-the-art AI models running on a NVIDIA A100 GPU. The sub-millisecond inference times of some models, combined with the high network bandwidth, results in large CPU loads, which are further exacerbated in multi-GPU chassis installations (*e.g.,* with a 4-to-1 or 8-to-1 GPU-to-CPU ratio). References from the industry confirm our intuition, showing up to 50% of inference serving time spent in networking tasks [5].

Parallel execution leads to less predictable inference performance. The deterministic sequence of GPU operations in most AI systems enables applications to reduce inter-device synchronization delays by carefully scheduling operations on CPU and GPUs [13]. However, this assumption is effective only when the GPU performs only *one* operation at a time. As shown in Fig. 2, while the GPU throughput increases logarithmically with higher levels of concurrency, the predictability of completion times decreases. Having the ability to control networking operations directly through the GPU could enable the execution of GPU applications in a fully asynchronous manner, thereby eliminating the unpredictability of inter-device synchronization times. Unfortunately, this approach is largely impractical at present, as almost all existing network transports are CPU-controlled.

Existing RDMA-from-GPU solutions still rely on the CPU to post verbs. Previous works already tackled the problem of offloading RDMA operations to the GPUs. GPUrdma [9] represents one of the pioneering works addressing the interaction between GPUs and NICs without CPU intervention. Nevertheless, this approach requires host stack modifications (*i.e.,* a custom driver) and the CPU is still involved in handling network transmissions. Most NVIDIA GPUs implement PCIe peer-to-peer (P2P) transactions[3] via the GPUDirect technology. More specifically, GPUDirect RDMA [25] provides support for RDMA verbs targeting memory areas in the GPU memory. However, the CPU remains responsible for posting these transactions, retaining control over the network stack. To mitigate this problem, NVIDIA recently introduced GPUNetIO [2] into its DOCA framework, enabling direct posting

[3]PCIe P2P transactions allow system peripherals to directly access each other's exposed memory addresses.

of RDMA operations from GPU kernels to the NIC. However, this library depends on NVIDIA's proprietary SmartNICs, limiting its applicability to commodity, heterogeneous hardware in existing deployments.

4 SYSTEM DESIGN

In this section, we illustrate our system's design that enables direct RDMA data transfers from GPUs without any CPU intervention. The system implements three main components, leveraging functionalities available in off-the-shelf NVIDIA GPUs (*e.g.,* Unified Memory, Host Memory Registration, CUDA Graphs, and GPUDirect RDMA). These features are implemented as separate *kernels*, which can be recorded in a *CUDA Graph* to be executed in an infinite loop. The main logical loop running on the GPU is shown in Listing 1. We package these routines as a shim library around `rdma_core`, using the default library definitions for all unmodified code – maximizing the re-usability and maintainability of our stack.

Network transmissions from the GPU. The action of posting a verb, as described in § 2.1, can be abstracted as transferring a sequence of data structures between the application and the network stack. Yet, standard `rdma_core` APIs are not compiled for CUDA devices and thus cannot be executed directly as device code, requiring manual code porting. We take advantage of the ability of modern GPUs to access memory addresses of other system devices to implement equivalent GPU-side routines for posting verbs. We exploit compiler conditional compilation (*i.e.,* through `#ifdef __CUDA_ARCH__`) to execute the same code on both the CPU and GPU, with minimal device-specific routines for PCIe registry manipulation and synchronization. The interface exposed to the application is the same as the `rdma_core` posting routine, offering a unified API function that can be invoked from both a GPU kernel or from CPU-side programs.

Trigger RX/TX processing from the GPU. The vanilla RDMA stack provides mechanisms that alert applications when a verb is received at the NIC. In the context of GPU-accelerated programs, this would result in costly synchronizations between the CPU and the GPU, with CPU resources being inefficiently consumed in waiting for these operations to complete. To circumvent this issue, we devise a GPU-side *busy-waiting routine*, which polls through the *registered buffer memory* of the RDMA stack, at fixed offsets, to detect when new data is ready to be processed. This process is executed as a separate kernel, pipelined before the main application processing. As for the transmission side, the RDMA stack notifies the application whenever an operation successfully completes. The application must consume these notifications (called *Completion Queue Entries*) to avoid triggering errors. In our approach, we devise a mechanism to consume these events on the GPU side, without the need to involve the CPU in this processing.

CPU-less continuous GPU operation. The ability of CUDA GPUs to record sequences of kernels into graphs enhances execution efficiency by minimizing CPU involvement in GPU-related processes, provided these operations are accurately captured within a graph. We employ *self-launching device Graphs* [27] to create an infinite loop of GPU-side operations, thereby achieving *no load* on the CPU. However, the design of CUDA APIs necessitates that the CPU engage in a synchronization process (*i.e.,* `cudaDeviceSynchronize`), leading to a scenario where a single CPU core is dedicated to monitoring the completion of GPU tasks, consuming 100% of that core's resources indefinitely until the application is closed. In contrast, we design an *interruptible sleep* feature, allowing the CPU to enter a sleep state while CUDA Graphs are executed, thereby freeing up CPU resources for other processes. We report the sample code for this mechanism in Listing 2. Note that the `stop` variable value changes in response to external interruptions or events on the GPU side.

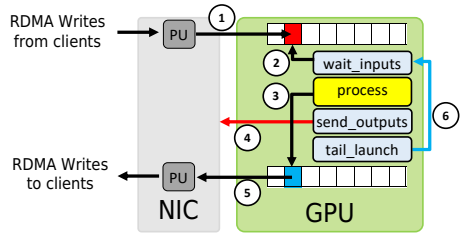

Figure 3: Overview of the system functioning, showing the main data paths.

Example of execution. Fig. 3 shows an overview of the system's functioning. First, data are transferred to the GPU memory via NIC's DMA engines ①, then ② the CUDA kernel reads the data directly, without intermediate copies, and ③ write back the results in another RDMA-enabled GPU buffer area. A verb is posted ④ to the NIC, which will ⑤ read the new data from the buffer. Finally, ⑥ the Graph will self-launch for the next execution.

5 EVALUATION

To demonstrate the benefit of our contribution and measure its performance, we implemented a prototype and ran it on a physical testbed. The server machine that runs our prototype is equipped with two Intel Xeon 6336Y CPUs, 256 GB of RAM, and a 200-Gbps NVIDIA Mellanox ConnectX-6 NIC. We run the GPU application on a NVIDIA A100 and a NVIDIA L40, both connected to the same PCIe 4.0 switch as the NIC, allowing direct P2P PCIe transactions. We use a separate server machine, equipped with a 100-Gbps NVIDIA Mellanox ConnectX-5 NIC, to act as a remote endpoint for RDMA connections. We interconnect the machines with an OpenFlow 100-Gbps switch, from which we collect network throughput metrics. Both systems run upstream software packages (*e.g.,* Ubuntu 20.04 LTS, NVIDIA CUDA 12.3, and Mellanox OFED). All ML models have been compiled with Apache TVM [11]. We fix the clock frequencies to their nominal values and disable energy saving mechanisms to

Listing 1: Pseudo-code for our GPU-side processing, to be recorded as a CUDA Graph.

```
__host__ void execute(){
    wait_inputs<<<1,1>>>();
    process<<<1,1>>>();
    send_outputs<<<1,1>>>();
    tail_launch<<<1,1>>>();
}
```

Listing 2: Pseudo-code for our CPU-side interruptible sleep approach. `cudaDeviceSynchronize` is called only when the program needs to terminate.

```
cudaGraphLaunch(i,stream);
while(!stop){sleep(1);};
cudaDeviceSyncrhonize();
```

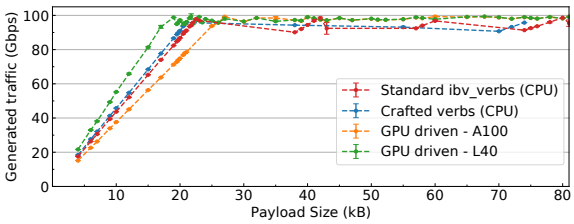

Figure 4: Throughput when generating packets from two different GPUs and CPU.

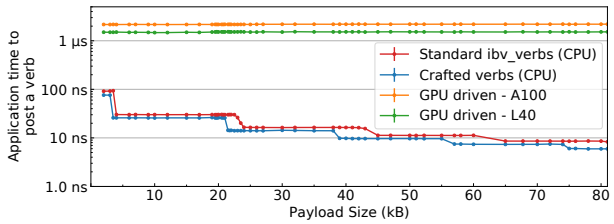

Figure 5: Average time to post a RDMA verb from CPU and GPUs.

reduce the dynamic variables in the system and ensure predictable results, which are collected across multiple runs.

5.1 Microbenchmarks

GPU-controlled RDMA can saturate a 100-Gbps link. Considering GPUs' limited efficiency in handling sequential, single-threaded tasks, such as *verb posting*, it is important to evaluate whether a modern GPU could initiate RDMA verbs efficiently. Fig. 4 shows the throughput (in Gbps) of a single-thread packet generator running on the CPU and on two GPUs models while generating payloads of varying size. To demonstrate that our system performs on par with vanilla RDMA verbs, we performed the experiment on the CPU side using two approaches: (*i*) using vanilla `ibv_verbs` from the `rdma_core` upstream library (red line) and (*ii*) utilizing our custom functions (blue line), which are identical to those compiled as CUDA kernels (orange and green lines). We note that (*i*) all four implementations can saturate a 100-Gbps link when packet sizes are large enough and (*ii*) GPU throughput scales similarly to CPU, but with different slopes. These different slopes are related to the different clock speeds of the devices, where the CPU operates at a higher frequency, and with the L40 running faster than the A100 (*i.e.,* 2.49 GHz vs. 1.41 GHz).

Posting verbs from the GPU requires constant time. We further investigated if our system could introduce additional overheads in verbs posting with respect to the vanilla, CPU-based `ibv_verbs` implementation. Fig. 5 shows the latency (measured from the application) to post a RDMA operation in the NIC. As it is possible to notice, the posting time is constant across all implementations and packet sizes, though GPUs exhibit longer times due to their slower clock rate. Notice that the latency for CPU postings decreases as the payload size increases. We hypothesize that this trend is due to PCIe dynamics and limits in our CPU-side prototype. We leave the analysis of these dynamics as future work.

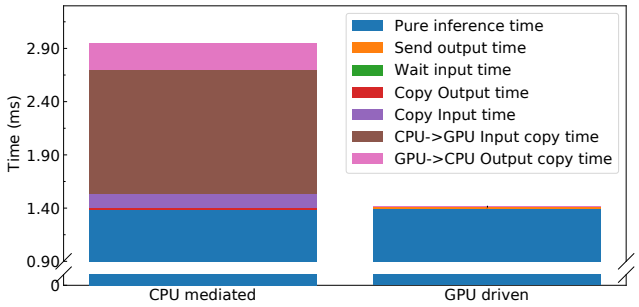

Figure 6: Total inference time contribution when controlling network on the CPU vs. on the GPU running *superresolution* [34] on a NVIDIA A100.

5.2 GPU-controlled RDMA for inference serving

To examine how our system affects performance of a realistic application, we built an inference-serving system prototype that runs some state-of-the-art ML models compiled with Apache TVM [11]. On modern GPUs, such ML models are capable of performing inferences within milliseconds, as demonstrated by MLPerf results [22]. We believe that deploying similar inference-serving systems at the edge could enhance response times to user requests with respect to centralized approaches.

Posting verbs from the GPU has negligible impact on the overall inference time. Fig. 6 shows the break-down of a CPU-mediated (*i.e.,* where data is transported through RDMA but the networking is CPU controlled) and a GPU-driven (*i.e.,* running entirely on the GPU) version of our serving prototype while running the *superresolution* model. The posting and busy-waiting times become negligible while serving common ML models. Using our mechanism leads to a *50% reduction* of total inference time, despite the longer verb posting times.

Eliminating CPU utilization during inference serving. Fig. 7 shows the CPU usage when using our *interruptible sleep* approach introduced in § 4 (green and dark-green lines) compared to standard CUDA synchronization APIs (red and orange lines). The application runs for 60 seconds before a termination signal is sent. For standard API calls, the CPU usage quickly ramps up to 100% on one core due to the polling nature of the API (*i.e.,* the immediate invocation of `cudaDeviceSynchronize`). In contrast, our approach results in

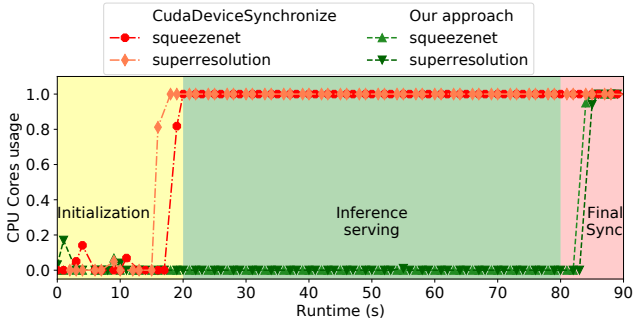

Figure 7: Average CPU utilization when using standard NVIDIA synchronization techniques and our mechanism. All operations run on a NVIDIA A100 GPU within a self-launched CUDA Graph.

minimal CPU usage, using it only during the final stage when the synchronization is effectively invoked, and maintaining 0% CPU usage during inference serving. The throughput and processing times of the application are unaffected by these mechanisms.

6 RELATED WORKS

A number of industrial efforts have already addressed the problem of providing interconnection to GPUs. NVIDIA NCCL [26] allows GPUs to collaborate in large HPC clusters, while UCX [29] and MSCCL [7] provide mechanisms to integrate these routines in bigger MPI-style computations. Although our approach is powered by the same underlying technology, we address the problem of direct connectivity with other heterogeneous nodes in a network, without *any* CPU intervention, focusing to a client-server scenario instead of collective operations.

Aside from the works already discussed in § 3, GPUnet [36], GPU-ether [17] and Lynx [37] have proposed approaches to move more networking processing to GPUs. However, these solutions require custom drivers or dedicated SmartNICs. Our approach is orthogonal to these works, providing general data-transfer mechanisms with minimal costs on the GPU and minimal modifications to the networking stack of the host systems.

Some works explore the inference-serving domain, proposing efficient methods for serving large volumes of traffic [8, 13, 32, 40] without specifically focusing on optimizing the underlying network transport mechanism, but rather relying on CPU-mediated networking. SplitRPC [18] presents an approach to deliver inference data directly on GPUs, characterizing the networking time required in common inference serving scenarios.

Many works address the concerns of low-latency video manipulation and network transport for real-time applications [10, 14, 20, 31, 33]. Our contribution helps these efforts by optimizing the underlying network architecture, fostering the adoption of these applications with minimal infrastructure adaptations.

7 CONCLUSIONS

We have proposed a novel RDMA stack running on the GPU side, showing how performing RDMA operations from a GPU is beneficial for both throughput and latency, reducing the need to deploy powerful CPUs at the edge. Our solution could be easily integrated with existing systems and leverages standard protocols to maximize the compatibility with other applications. Our work helps towards improving the availability of GPU-accelerated applications on edge computing platforms, minimizing the required resources around GPU devices.

ACKNOWLEDGMENTS

This work has been supported by the European Research Council (ERC) under the grant agreement No. 770889.

REFERENCES

[1] [n. d.]. *RDMA core userspace library.* https://github.com/linux-rdma/rdma-core
[2] Elena Agostini, et al. 2023. *Realizing the Power of Real-Time Network Processing with NVIDIA DOCA GPUNetIO.* https://developer.nvidia.com/blog/realizing-the-power-of-real-time-network-processing-with-nvidia-doca-gpunetio/
[3] Guido Appenzeller, et al. 2023. *Navigating the High Cost of AI Compute.* https://a16z.com/navigating-the-high-cost-of-ai-compute/
[4] Luiz Barroso, et al. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (2017).
[5] Alexis Bjorlin. 2022. *Infrastructure for Large Scale AI: "Empowering Open".* https://drive.google.com/file/d/1qqjo-5JtYAcRlK_LWYuQFH-b9MoFOP02/view
[6] Jack Choquette. 2023. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro* 43, 3 (2023).
[7] Microsoft Corporation. 2022. *Microsoft Collective Communication Library.* https://github.com/microsoft/msccl
[8] NVIDIA Corporation. [n. d.]. *NVIDIA Triton Inference Server.* https://www.nvidia.com/en-us/ai-data-science/products/triton-inference-server/
[9] Feras Daoud, et al. 2016. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *ROSS 2016.* New York, NY, USA.
[10] Chris Dickson. [n. d.]. *The Technology Behind A Low Latency Cloud Gaming Service.* https://parsec.app/blog/description-of-parsec-technology-b2738dcc3842
[11] The Apache Software Foundation. [n. d.]. *Apache TVM.* https://tvm.apache.org/
[12] Aishwarya Goel. 2023. *Unraveling GPU Inference Costs for Fine-tuned Open-Source Models V/S Closed Platforms.* https://mlops.community/unraveling-gpu-inference-costs-for-fine-tuned-open-source-models-v-s-closed-platforms/
[13] Arpan Gujarati, et al. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *OSDI 2020.* USA, Article 25.
[14] Yashuang Guo, et al. 2020. Adaptive Bitrate Streaming in Wireless Networks With Transcoding at Network Edge Using Deep Reinforcement Learning. *IEEE TVT* 69, 4 (2020).
[15] Kim M. Hazelwood, et al. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. *HPCA 2018* (2018).
[16] InfiniBand™Trade Association. 2010. *RDMA over Converged Ethernet (RoCE).* https://web.archive.org/web/20160309123709/https://cw.infinibandta.org/document/dl/7148
[17] Changue Jung, et al. 2021. GPU-Ether: GPU-native Packet I/O for GPU Applications on Commodity Ethernet. In *IEEE INFOCOM 2021.*
[18] Adithya Kumar, et al. 2023. SplitRPC: A Control + Data Path Splitting RPC Stack for ML Inference Serving. *SIGMETRICS 2023* 7, 2 (2023).
[19] Ang Li, et al. 2019. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *TPDS* 31, 1 (July 2019).
[20] Pietro Lungaro et al. 2019. Immersivemote: combining foveated AI and streaming for immersive remote operations. In *SIGGRAPH 2019.* Article 17.
[21] Sparsh Mittal et al. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *CSUR 2015* 47 (2015).
[22] MLCommons. 2023. *Inference Datacenter v3.0 Results.* https://mlcommons.org/en/inference-datacenter-30/
[23] Timothy Prickett Morgan. 2023. *Meta Platforms Is Determined To Make Ethernet Work For AI.* https://www.nextplatform.com/2023/09/26/meta-platforms-is-determined-to-make-ethernet-work-for-ai/
[24] John Nickolls, et al. 2008. Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For? *Queue* 6, 2 (March 2008).
[25] NVIDIA. 2023. *GPUDirect Documentation.* https://docs.nvidia.com/cuda/gpudirect-rdma/index.html
[26] NVIDIA. 2023. *NVIDIA NCCL.* https://developer.nvidia.com/nccl
[27] NVIDIA Corporation. 2023. *Device Graph Launch.* https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-graph-launch
[28] NVIDIA Corporation. 2023. *NVIDIA H100 GPU DataSheet.* https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet
[29] NVIDIA Corporation. 2024. *Unified Communication - X Framework.* https://docs.nvidia.com/networking/display/HPCXv215/Unified+Communication+-+X+Framework+Library
[30] OpenAI. 2024. *SORA.* https://openai.com/sora/
[31] Kevin Röbert, et al. 2023. Latency-Aware Scheduling for Real-Time Application Support in Edge Computing. In *EdgeSys 2023.*
[32] Francisco Romero, et al. 2021. INFaaS: Automated Model-less Inference Serving. In *USENIX ATC 2021.*
[33] Eurovision Services. 2021. *Eurovision Services successfully tests full cloud production solution.* https://www.eurovision.net/insights/technical/eurovision-services-successfully-tests-full-cloud-production-solution
[34] Wenzhe Shi, et al. 2016. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *IEEE CVPR 2016.*
[35] Alexander Shpiner, et al. 2017. RoCE Rocks without PFC: Detailed Evaluation. In *KBNets 2017.*
[36] Mark Silberstein, et al. 2016. GPUnet: Networking abstractions for GPU programs. *TOCS* 34, 3 (2016).
[37] Maroun Tork, et al. 2020. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *ASPLOS 2020.*
[38] Manoj Wadekar. 2013. Chapter 11 - InfiniBand, iWARP, and RoCE. In *Handbook of Fiber Optic Data Communication (Fourth Edition)* (fourth ed.), Casimer DeCusatis (Ed.). Academic Press, Oxford.
[39] Matt Walsh. 2023. *ChatGPT Statistics (2023) — The Key Facts and Figures.* https://www.stylefactoryproductions.com/blog/chatgpt-statistics
[40] Hong Zhang, et al. 2023. SHEPHERD: Serving DNNs in the Wild. In *NSDI 2023.*