SplitRPC: A {Control + Data} Path Splitting RPC Stack for ML Inference Serving

ADITHYA KUMAR, The Pennsylvania State University, USA
ANAND SIVASUBRAMANIAM, Penn State University, USA
TIMOTHY ZHU, The Pennsylvania State University, USA

The growing adoption of hardware accelerators driven by their intelligent compiler and runtime system counterparts has democratized ML services and precipitously reduced their execution times. This motivates us to shift our attention to efficiently serve these ML services under distributed settings and characterize the overheads imposed by the RPC mechanism ('RPC tax') when serving them on accelerators. The RPC implementations designed over the years implicitly assume the host CPU services the requests, and we focus on expanding such works towards accelerator-based services. While recent proposals calling for SmartNICs to take on this task are reasonable for simple kernels, serving complex ML models requires a more nuanced view to optimize both the data-path and the control/orchestration of these accelerators. We program today's commodity network interface cards (NICs) to split the control and data paths for effective transfer of control while efficiently transferring the payload to the accelerator. As opposed to unified approaches that bundle these paths together, limiting the flexibility in each of these paths, we design and implement **SplitRPC** - a {control + data} path optimizing RPC mechanism for ML inference serving. **SplitRPC** allows us to optimize the datapath to the accelerator while simultaneously allowing the CPU to maintain full orchestration capabilities. We implement SplitRPC on *both commodity NICs and SmartNICs* and demonstrate how GPU-based ML services running different compiler/runtime systems can benefit. For a variety of ML models served using different inference runtimes, we demonstrate that SplitRPC is effective in minimizing the RPC tax while providing significant gains in throughput and latency over existing kernel by-pass approaches, without requiring expensive SmartNIC devices.

CCS Concepts: • **Computer systems organization → Client-server architectures**; • **Software and its engineering → Client-server architectures**; **Communications management**; • **Networks → Network servers**; **Network adapters**.

Additional Key Words and Phrases: Remote Procedure Call, ML Inference, Data path, Orchestration, SmartNIC

ACM Reference Format:
Adithya Kumar, Anand Sivasubramaniam, and Timothy Zhu. 2023. SplitRPC: A {Control + Data} Path Splitting RPC Stack for ML Inference Serving. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 2, Article 30 (June 2023), 26 pages. https://doi.org/10.1145/3589974

1 INTRODUCTION

Executing Machine Learning (ML) models to make predictions and inferences is an important workload across numerous domains (e.g., image recognition, NLP, advertising, recommendation systems). These ML services are usually hosted on high-end servers, which not only have multiple

Authors' addresses: Adithya Kumar, The Pennsylvania State University, University Park, PA, 16801, USA; Anand Sivasubramaniam, Penn State University, University Park, PA, 16801, USA; Timothy Zhu, The Pennsylvania State University, University Park, PA, 16801, USA.

2476-1249/2023/6-ART30 $15.00
https://doi.org/10.1145/3589974

CPU cores, but also accelerators (e.g., GPUs, TPUs) that are vital for efficient processing of ML inference tasks. Clients typically utilize these services via Remote Procedure Call (RPC) mechanisms [15, 22, 46, 64], invoking specific functions along with their arguments, rather than through basic socket interfaces. Recent works have proposed methods for improving the throughput and latency of general RPC systems [4, 30, 33, 34, 52], but the impact of RPC systems on ML workloads that utilize accelerators has not been explored in depth. To address this void, this paper: *(i)* points to

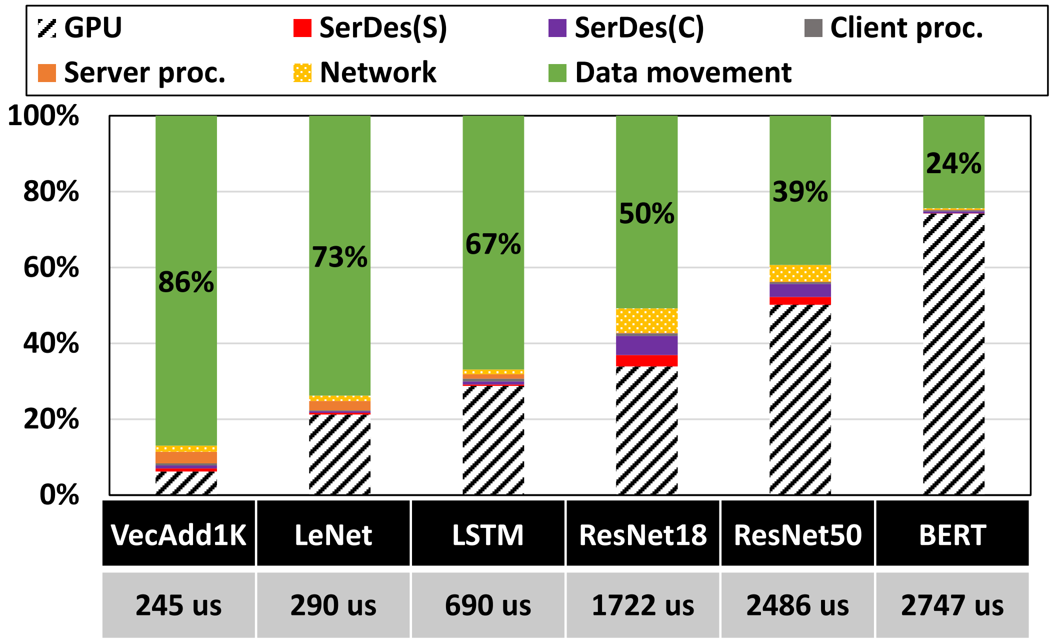


Fig. 1. Breakdown of end-to-end execution time of services executing on an Nvidia-A100 accelerator using gRPC. 'SerDes' is the cost of serializing and deserializing the payload into protobuf at the (C)lient and (S)erver side respectively. A large fraction of time is spent moving data between the GPU/CPU/NIC.

the need for customizing RPC implementations for ML-like services (i.e., those which are offloaded to GPUs/accelerators rather than performed by the CPU); *(ii)* demonstrates that simply moving the entire RPC stack to the GPU is highly inefficient; *(iii)* proposes a novel approach called **SplitRPC** that avoids these inefficiencies by decomposing/de-multiplexing the RPC stack into control and data flows and managing them separately across the CPU/GPU/NIC.

Processing an RPC includes many components such as serialization/deserialization (SerDes) as well as other client and server processing overheads. There are also network transfer overheads and data movement overheads within the server (e.g., NIC to main memory transfer). This "RPC tax" [52] is typically viewed as small relative to the execution time of the RPC function itself. However, our experiments (Figure 1) show that the RPC tax is a large percentage of time across multiple real-world ML models and a microbenchmark (VecAdd1K). This is due to shrinking execution times over the years. Since GPUs/TPUs have been optimized substantially to reduce execution times, the RPC overheads are now a dominant factor in end-to-end latency. For instance, if we consider the well-known ML model ResNet50, the latency has come down from 47ms in 2017 [15] to 2.5ms today. If we consider a tax of ≈980us (as in a gRPC [20] implementation today), the overhead for ResNet50 has risen from 2.3% in 2017 to 39% today as a consequence of the execution time reduction.

One way to reduce this tax is to incorporate additional hardware (either separate hardware as suggested in [52, 62] or more sophisticated intelligence within the NIC [18, 34], referred to as SmartNICs) to take on some of the RPC processing work that is traditionally done by the CPU. This can reduce some of the data transfers (e.g., ensuring only what is needed is transferred rather than complete packets) and reduce/hide some of the computations in the tax (e.g., protocol processing, marshaling/un-marshaling arguments). However, RPC implementations and their proposed optimizations [8, 18, 30, 33, 34, 52, 62, 70] have all inherently assumed that requests are serviced entirely on the server's CPUs.

ML tasks that utilize accelerators (e.g., GPUs, TPUs) incur extra overhead when composed as RPC services since their core execution is not on the CPU. This results in unnecessary data movement and copying, which can be seen as a major component in Figure 1. Lynx [66] is a recent work that shows that using a SmartNIC to bypass the CPU and directly transfer data from the NIC to the GPU can significantly lower data movement costs. However, we observe the need to take a more nuanced approach while dealing with real-world ML models. While their approach is beneficial for simple microbenchmarks comprising a single kernel, we demonstrate that the control and orchestration mechanisms are the predominant factors when running ML models with multiple kernels. As we will see (Section 3.2), a sub-optimal orchestration mechanism that optimizes just the data movement is extremely inefficient and slow. Fundamentally, the control and orchestration of many kernels within ML models is best suited to general purpose CPU processors, while the parallel execution of the kernels is best suited to optimized accelerators.

These observations highlight the intuition behind **SplitRPC** – split the control and orchestration path from the data path. In SplitRPC, we configure the (Smart)NIC to selectively switch incoming components of an RPC request to the host CPU (control) and the GPU (data). This allows the CPU to efficiently handle the orchestration of the many kernels within an ML model while avoiding unnecessary data transfers between main memory and the GPU. Specifically, this paper makes the following contributions towards implementing an efficient RPC stack for ML services:

Contributions:

- We measure and identify two equally important factors of the RPC tax of ML inference: (i) data movement to/from the accelerator, and (ii) orchestration of ML kernels.
- We design and build **SplitRPC**, an RPC stack for ML services on commodity NICs (SplitRPC-pNIC) & SmartNICs (SplitRPC-sNIC) that promotes a split {control + data} path design to minimize data movement while maintaining efficient control and orchestration. SplitRPC is available at `https://github.com/minus-one/splitrpc`.
- SplitRPC implements a novel queueing mechanism to enable simultaneous transmission/reception of data and dynamic batching of the inference tasks on the accelerator.
- We evaluate SplitRPC against (i) traditional gRPC based mechanisms with kernel bypass optimizations, and (ii) newly proposed SmartNIC-based approaches that service the entire RPC call on the GPU [66]. Our design provides 52% (on average) improvements in the end-to-end execution time and up to 2.4× gains in throughput with dynamic batching for a variety of ML inference services.

2 MOTIVATION: CANNOT IGNORE RPC TAX FOR ML INFERENCE SERVING ANY LONGER

Many datacenter/cloud based ML services today utilize RPC mechanisms [15, 46] such as gRPC [20] or bRPC [3]. For example, an image recognition web application would receive an input image and make an RPC call with the input payload to a backend inference framework (e.g., TensorRT) hosted on a remote server to execute the ML model (e.g., ResNet50), which in turn invokes a sequence of kernels on the GPU/TPU. We study the cost/overheads of serving such ML inference applications written for GPUs as gRPC services, representative of many inference serving systems used in production [46, 68] and in research [22, 76].

Figure 1 shows our motivating results across a simple microbenchmark kernel (VecAdd1K), compiler generated optimized backends (LeNet using TVM [10], LSTM using NNFusion [39]), and runtimes that generate optimized kernels on the fly (ResNet18/50, BERT using TensorRT [68]). We measure the end-to-end (E2E) execution time of these services as measured on the client side in a distributed setting with a high speed low latency network (see Section 6.1 for details). Figure 1 breaks down the E2E execution time of each of these services (shown in μs below each bar) into the percentage of time spent under different components. First, we observe that for all services, the E2E execution time is significantly higher than the compute time (i.e., GPU time) with shorter services having larger overheads. . The 'GPU' time is influenced by different factors including the compiler/runtime optimizations [10, 28, 39, 42, 65] used to orchestrate the GPU in addition to the GPU architecture itself. Interestingly some of the inherent gRPC related tasks like serialization and deserialization into protobuf – labeled as 'SerDes(S)' and 'SerDes(C)' for the server side and client side respectively – do not contribute much to the overheads [31, 47]. Coupled together with other gRPC related tasks, this comes to <9% of the E2E execution time (with the maximum for ResNet18). The main source of overheads arises from the *data movement* through the memory hierarchies. Note that this cost does not include the time spent in the network, which is shown separately as 'Network'. The data movement cost includes both data movement between the host memory and GPU memory as well as the data movement through the network sub-stacks (i.e., the Linux kernel). For shorter models like LeNet, the RPC tax due to data movement is as much as 73% of the E2E execution time. This tax is a growing problem for ML inference due to two reasons. The rapid rise in the compute power of accelerators [6, 7, 54], and more importantly, the plethora of compiler optimizations has resulted in the execution times of these ML models ('GPU' time) to shrink to μs-scale or low ms-scale regimes. This magnified RPC tax necessitates a redesign of the RPC software stack used by distributed ML services.

3 RPCS FOR ML INFERENCE

The RPC mechanism used to execute ML inference on a server with an accelerator is provided (i) a specification of the request/response data for the ML inference (e.g., the dimensions of an image), and (ii) a backend inference framework [48, 68] along with a ML model definition (e.g., ONNX [5]/TF FrozenGraph [63]) containing a dataflow graph (DFG) of ML kernels to run as part of the inference query on the input data. Once the input data from the client is received, the inference framework is called to execute the optimized backend implementation of the ML model on the accelerator (e.g., CUDA kernels on Nvidia GPUs). After the computation completes, the response is sent back to the calling client.

Let us examine the sequence of steps in serving this query. We consider the request path and note that the response path is just the reverse of this path.

3.1 Network transport stack (N):

The NIC has to be programmed to receive the request (physical/link layers) and subsequently some kind of transport mechanism has to be employed for reliable receipt of the request. Invariably in the context of RPCs, simple UDP mechanisms suffice [30, 33]. At the end of this, the request (and its parameters) either rest on the DRAM of the NIC or in the host/accelerator memory of the server (depending on if the implementation takes advantage of hardware support). Note that this request has 2 components - (i) the request (64B), which we will henceforth refer to as *Control (C)*, and (ii) the parameters of the request (up to a few KB for some inference tasks), which we will henceforth refer to as *Data (D)*.

- **Control (C):** Conventionally, the only entity that could implement the RPC function has been one of the server CPUs. However, with the growing popularity of accelerators (e.g., GPUs) for ML tasks, the accelerator takes on more (or even all) of the work compared to the host CPU. This gives us 2 options to route the control flow. (*C1*): As before, the NIC passes the control data to the host (memory) using (R)DMA mechanisms; or (*C2*): the NIC directly transfers the data to the GPU (memory) given direct device-to-device (P2P) [41] transfer capabilities that are available in modern devices.
- **Data (D):** Again, these 2 options are available for the flow of data. (*D1*): DMA-ing the data portion to the host memory for main CPU access (necessitating another copy to the GPU memory); or (*D2*): DMA-ing the data into the GPU memory directly, thus saving one additional hop in the data transfer.

3.1.1 Comparing choices for step (N):. We now explore the tradeoffs of executing step (N) on our prototypical heterogeneous server, with the choice of 3 categories of NICs available in the market today (see Figure 4).

- **Simple:** The simplest (and cheapest) NIC option is one which only provides FIFO queues to/from the external wire, with small buffers and DMA engines to transfer to host memory. These do not have on-board processing capabilities, and rely on the host CPU for their programming. Hence, the execution has to be done by the host CPU, and we can only employ design choices (C1) and (D1) for the control and data path - the NIC has no way of figuring out what portions are the data and/or control in the incoming packet.
- **P2P:** The P2P NIC is an improvement over the plain option above, wherein there are peer-aware channels/queues, which can be programmed a priori to route their contents to different peer-to-peer (P2P) devices' memory locations [41, 43]. This is a commodity NIC that allows all the design choices (C1, C2) and (D1, D2) for the control and data paths. For instance, if we have 2 channels/queues, one can be used to DMA from/to the host memory (C1, D1), while the second can be used to DMA from/to the GPU memory (C2, D2). Such programming, however, has to be done by some intelligent entity a priori, and in this case it has to be done by the host CPU since the NIC itself does not have any other intelligence. These NICs (e.g. Mellanox CX-5) are widely used and only cost slightly more than the first category.

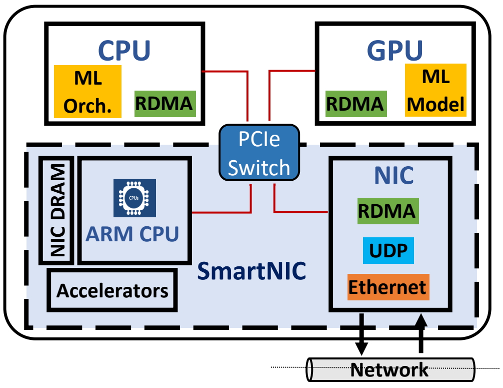


Fig. 2. SmartNICs have programmable compute on them. For e.g., the Mellanox BlueField has ARM cores where the network stack can be offloaded.

- **Smart:** Apart from the hardware supported in the prior two categories, commodity SmartNICs are available which come in a variety of flavors. They have programmable compute available on them apart from specialized accelerators for packet processing making them substantially more expensive (4× the P2P NIC cost). For e.g., Mellanox BlueField has ARM cores and Mellanox Innova Flex has FPGAs as programmable compute.
  As shown in Figure 2, the SmartNIC sits as a PCIe peripheral device on the server and can communicate with the host CPU as well as the GPU. They use conventional DMA engines

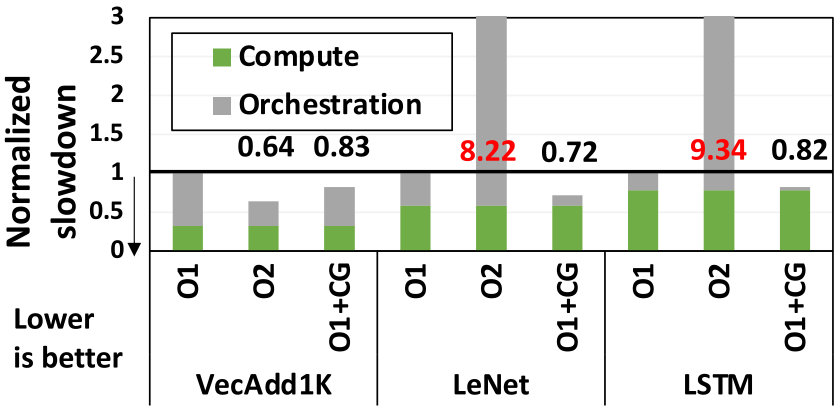


Fig. 3. Normalized breakdown of time spent in computation and orchestration under different ML orchestration mechanisms (Nvidia A100). O1: 'CPU' based orchestration, O2: 'GPU' based orchestration, and O1+CG is O1 performed using the CUDAGraphs API. Lower is better.

(e.g., Netronome Agilio) or RDMA engines (e.g., Mellanox BlueField) implemented over PCIe to establish direct data-paths between the NIC and the host CPU/GPU.
On such SmartNICs, we can offload the entire (N) functionality to the compute engine on the NIC itself rather than the host CPU and leverage the device APIs (e.g., Nvidia's GPUDirect) to program the data-paths to facilitate choices (C2) and (D2).

3.2 ML Orchestration (O):

The next critical step in servicing the request is orchestrating the ML computation on the GPU. This involves launching one or more kernels (e.g., CUDA kernels on Nvidia GPUs) according to the ML model's computation dependency graph. There are two primary approaches to orchestration.

- **CPU-managed (O1):** Typically, a ML inference framework [10, 15, 22, 56, 68] running on the CPU will manage the launching and orchestration of all the kernels in the ML model. While this approach puts the launch overheads of the first kernel in the critical path of the RPC service, it allows for greater control in synthesizing the computation by the ML framework as the host CPU will have full visibility on when to launch each kernel. Additionally, this orchestration style is readily adoptable by all existing ML frameworks requiring minimal/no change to their internal execution mechanisms.
- **GPU-managed (O2):** Alternatively, it is possible to trigger kernels to run from the GPU, but it requires persistently running the kernels on the GPU [23]. The computation itself waits to start executing until it is signaled via a synchronization flag. Kernels of the model would then be launched directly from the GPU. However, triggering subsequent kernels on the GPU suffer from (implicit) inter-kernel synchronization, which we observe to be catastrophically slow for ML models with dozens of kernels. On Nvidia GPUs, this mechanism is referred to as CUDA dynamic parallelism (CDP) [29].

3.2.1 Comparing choices for step (O):. We compare the choices for (O) by taking a few motivating examples and studying the orchestration overheads of the two approaches (O1 & O2) on an Nvidia GPU accelerator. We pick three applications - (i) *VecAdd1K:* Adding a constant to a vector of size 1KiB, written as a simple CUDA kernel, (ii) *LeNet:* the LeNet digit recognition model constructed using the TVM [10] compiler, which has synthesized optimized backend kernels for the different layers, and (iii) *LSTM:* a language comprehension model constructed using NNFusion [39], a recent state-of-the-art runtime system that fuses multiple kernels together (thereby reducing the number of kernels launched) and optimizes the execution on the device. All three applications can be executed directly using the **O1** approach and can be suitably modified to make the kernels run persistently on the GPU using the **O2** approach. In addition to this, we implement an enhanced **O1** approach (calling it as **O1 + CG** by modifying these applications to use the CUDA graphs API [44] to launch the entire set of kernels as a single graph on the GPU and reduce the kernel launch overheads even further.

Figure 3 compares the slowdown (normalized to **O1**) for executing these applications using these different techniques on the Nvidia A100 GPU (see Section 6.1 for full setup). As we can see, there is a clear performance difference among the various schemes. For *VecAdd1K* consisting of just 1 kernel, **O2** outperforms both **O1** and **O1 + CG**. However, as we investigate the breakdown in the time spent in compute vs. orchestration overheads, we can see a different story emerging even for a simple ML service - LeNet - with 12 kernels. The performance of **O2** is orders of magnitude ($\approx 8-9\times$) worse with the major source of overhead being the GPU-managed orchestration. This is because using the GPU to launch multiple large kernels (typical for ML models) one after another forces kernel launches and the associated synchronization to happen on the accelerator itself, which cripples the performance of the whole model. Since ML models typically have dozens of kernels (see Table 1), the *O2 mechanism is catastrophically bad.* Thus, *the RPC mechanism that we implement for serving ML models needs to be considerate of the accelerator orchestration mechanism that is employed by the service.* While CDP on GPUs could be useful in many other scenarios, it is ineffective for orchestrating entire kernels of ML models. Under CDP, as we launch child kernels inside a parent GPU kernel, the cost associated with creating dozens of child kernels inside the GPU and the final synchronization for the completion of the last child kernel is prohibitive. Alternatively, on the host CPU, kernel launches on the CPU and kernel execution on the GPU can overlap, thus hiding this overhead.

Therefore, this observation makes an important case that the *control path* considerations are far more critical in executing ML models on accelerators. While the choice (O2) on paper is seemingly optimal in reducing the communication costs with the accelerator, it moves critical control path activities to the accelerator, which may lack the capability to perform those activities in an optimized fashion.

3.3 Key considerations in ML Inference

There are 3 key considerations/questions/challenges in building RPCs tailored for ML inference.

(i) The key factor, as seen in Figure 3, is in picking the right orchestration technique for the accelerator to fully realize the benefits of the runtime + compiler sub-systems. Sub-optimal orchestration can negate all the benefits of other optimizations performed by the implementation. Towards that, *the RPC mechanism that we implement has to correctly interface with the compiler and runtime techniques developed for the accelerator.*

(ii) While there are prior RPC stacks (including kernel bypass mechanisms) targeting distributed applications, most of these are geared towards applications running on the host CPU. The key challenge is in carefully *expanding the RPC data-path approaches to be "accelerator-aware".*

(iii) SmartNICs have emerged as a panacea to offload parts of the RPC mechanism [34, 71] or even parts of the computation itself [32, 36, 37] to optimize the end-to-end execution time for distributed services. This raises the question, *should we dedicate additional hardware if all we need is to simply transfer data to the accelerator?*

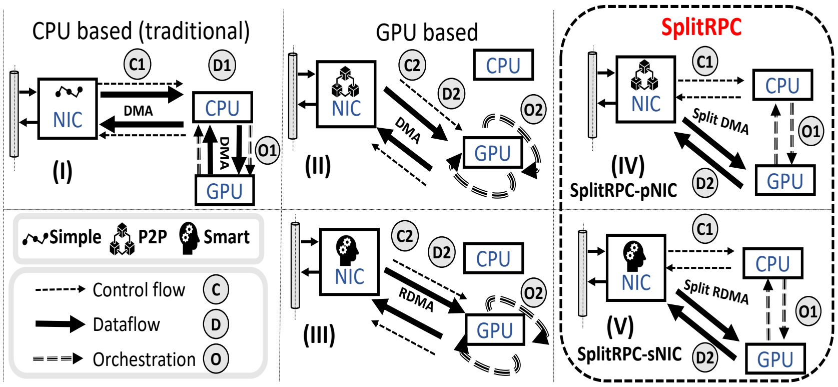


Fig. 4. Implementation choices for building accelerated services. Approach (I) is tailored for Simple NICs lacking any data-path optimizations; approaches (II) & (III) optimize for the data-path while forcing the accelerator to perform the orchestration. The proposed split designs (IV) & (V) combine data-path optimizations while dynamically managing the control flow on the CPU.

4 TAILORING RPCS FOR ML INFERENCE

Based on the preceding discussion on the choices of the control path (C), data path (D), and more importantly the considerations to be made in orchestration (O) for ML inference, we outline 5 different implementation choices in Figure 4 on the 3 kinds of NIC hardware (Simple, P2P, and Smart).

4.1 Traditional CPU based designs:

For simple NICs, the ML inference service deployment is depicted in implementation choice (**I**), which corresponds to design choices {C1, D1, O1}. This choice is commonly used today in many deployments [15, 22, 56, 76]. The design is simple, straightforward, and readily deployable on all systems with little additional cost. As described earlier (Section 2), the key limitation of this approach (I) is the additional data movement overheads as the NIC lacks direct data-paths to the accelerators.

4.2 GPU based designs:

P2P NICs and SmartNICs facilitate direct (R)DMA paths to different devices (accelerators/CPU) on the server, thus allowing all 4 control/data flows designs {C1, D1, C2, D2} to be implemented. GPU based designs [16, 61, 66] choose to route the control *and* data to the accelerator {C2, D2} using these data-paths on P2P and SmartNICs. As a result, the orchestration is also shifted to the GPU. This design overcomes the limitations of CPU based designs by enabling direct data-paths to the accelerator (D2), but unfortunately, the orchestration choice is limited to O2, making them counterproductive for serving ML inference workloads (see Figure 3).

4.3 SplitRPC:

The intuition behind our work is to explore the possibility of getting the best of both worlds while avoiding their individual deficiencies - *data is not needed by the host, but orchestration is more effective if done by the host.* That is, SplitRPC performs D2 in combination with O1 by selectively routing the control information to the host CPU (C1). This design choice of {C1, D2} allows us to be flexible and choose the better orchestration scheme O1 while allowing us to leverage direct data-paths to the accelerators. We implement this design using *both* P2P NICs (**IV**) and SmartNICs

(**V**). To achieve this, *we have to route the control portion to the host (C1) while sending the payload portion to the accelerator (D2).* Fortunately, as we will show, this is achievable on commodity P2P NICs available in the market today, without requiring the extensive processing capabilities of expensive SmartNICs. We program the NIC (in both P2P NICs and SmartNICs) to *split* the incoming request into predefined control and payload portions and use the DMA engines to forward this request to their appropriate destinations.

5 BUILDING SPLITRPC

We implement the SplitRPC design for both P2P NICs (SplitRPC-pNIC) and SmartNICs (SplitRPC-sNIC). Both these NICs allow for peer-aware transfer of {Control + Data} and therefore can implement any of the design choices. We specifically focus on (C1) where the control information is sent to the CPU, (D2) where the data is directly sent to the accelerator, and (O1) where the accelerator is orchestrated dynamically from the CPU, which works well for ML models. In the case of SplitRPC-sNIC, the SmartNIC runs the network stack, while in the case of SplitRPC-pNIC, the host CPU programs the NIC and performs the request/response processing. The NIC's job in this case is only to demux/split the incoming packet components into the appropriate queues. There are 4 high-level implementation details of SplitRPC and we next expand on each for both the implementations.

5.1 How does SplitRPC split the RPC?

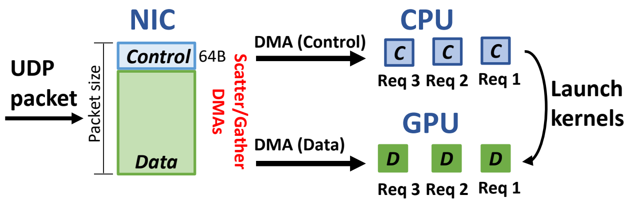


Fig. 5. High level overview of how SplitRPC splits the incoming RPC request into Control (C) and Data (D) portions using Scatter-Gather DMAs.

We first describe the transport mechanism of our network stack and then explain how the splitting is achieved by the NIC. We draw inspiration from recent host-focused RPC proposals [30, 33] and use a UDP-based transport protocol. We add a small service-layer header on top of the UDP header and use that to send/receive the *Control* information about the request/response. The *Control* header contains essential details including - (i) *req_id:* A 64-bit request identifier that is uniquely generated by the client to match the response, (ii) *func_id:* A 16-bit function identifier to match the service to be executed for this request, (iii) *seq_num:* To support large requests, we break down the request into multiple segments and use this 16-bit sequence number identifier to reassemble the segments into the full request. An RPC call is uniquely identified by 4 tuples - the caller's UDP socket info - (a) IP-address and (b) port-number along with the (c) *req_id* and the (d) *func_id*. These 4 tuples are sufficient to uniquely identify a call in the system. The header is padded appropriately so that the total size of the header (including the UDP, IP, and Ethernet headers) is 64B This helps for cache line alignment on the host and cleaner splitting by the NIC's DMA engine. The application related *Data* (which can be the request/response data) comes after this header, and if the payload size is larger than the (pre-established) path MTU (maximum transmission unit), this data will be broken into multiple segments, with each segment having a *Control* header describing the request segment. This allows SplitRPC to be easily adopted across a mix of ML services with small request sizes (few 100Bs) that fit within a single segment (e.g., BERT) and large request sizes (few 100KiBs) requiring segmentation (e.g., ResNet50).

Now that we have defined the *Control* and *Data* portions of the SplitRPC RPC request, we have to program the NIC to send the *Control* portion (containing the *func_id*) to the host CPU so that the corresponding inference framework can be invoked to run the computation. Modern NICs (including P2P and SmartNICs) typically have a shared ring-buffer + doorbell mechanism [50] to store incoming packets in pre-allocated memory regions (typically allocated as huge pages) and notify the host CPU (or the ARM core/FPGA on the SmartNIC) once the packet has been copied into these memory regions. Additionally, these NICs also feature scatter-gather DMAs to scatter incoming packets on predefined size boundaries. These pre-allocated memory regions can also be on the GPU memory, but require mapping the GPU memory regions to be peer-aware both on the GPU and the NIC. We use the corresponding driver APIs - Nvidia's GPUDirect [43] and Mellanox's PeerDirect [41] - to register the memory regions to allow the NICs to directly send information to the accelerator. This is illustrated in Figure 5.

In the case of P2P NICs where the host CPU directly programs the NIC, we create two memory regions - one on the host memory, and one on the GPU memory - and program the NIC's scatter-gather DMA engines to scatter the 64B *Control* portion from the CPU memory (by consuming buffers allocated on the CPU memory region), and send the *Data* portion to the buffers in the GPU memory region. The doorbell containing the address information of both the *Control* and *Data* is notified on the host CPU once *both* these DMA operations are completed by the NIC. Note that scatter-gather DMAs are pretty common and have even been explored in the context of RPCs [53] to scatter the payload itself into different regions on the host CPU. A recent work [51] proposes to do splitting of packets between the host memory and the NIC's on-device memory for processing network functions. In contrast to these approaches, we leverage the DMA features on the NIC to split an RPC request between the host memory and the accelerator memory, which has not been explored before.

On the SmartNIC, the ARM core allocates two separate memory regions on the NIC's DRAM for *Control and Data*. When it receives the incoming packet it performs a similar split into these local memory regions on the NIC DRAM. Then, the *Data* portion alone is RDMA-ed to the application specific memory region on the GPU.

On both these NICs, we implement this split using DPDK [27], a software packet processing framework that facilitates programming the NIC and additionally helps by-pass the kernel stacks altogether to maximize performance.

5.2 How SplitRPC facilitates data-transfer to/from the ML application?

We expose two APIs to send/receive request/response data to/from the ML application. In the case of P2P NICs where the address spaces are shared between devices, it is efficient to just send the pointer to the *Data* portion. We expose a *Zero-Copy (ZC)* API that directly provides this buffer information about the *Data* as ZC-Entries (ZCE) to the inference framework. There are two other cases that require the use of the second slightly more complex API. First, the request itself can be multi-segmented (i.e., have multiple *Data* portions that may be non-contiguous in the device memory) because of packet reordering or other issues. Second, to process a batch of requests together, ML frameworks require these requests/responses to be laid out contiguously. To deal with these challenges, we expose our second API, that implements and provides shared **Tensor-Queues (TQ)** for each input/output tensor between the RPC system and the inference framework. 'Tensors' are typically how the data to an ML model is produced/consumed, which for the purpose of this discussion can be thought of as contiguous virtual memory regions. A TQ is a contiguous region of memory stored on the accelerator/GPU memory, parameterized by an element size and capacity. The element size is specified by the application depending on the size of the tensor inputs consumed per request. Each TQ element (TQE) corresponds to a tensor of a request or a response. Once all *Data* segments of a request are received by the network stack, they are gathered and copied into their corresponding TQ entries and are provided to the inference framework.

When all segments of an RPC request have been received on a P2P NIC, these segments are already resident on the GPU memory, thanks to the splitting mechanism that we have implemented. Therefore, we trigger a simple GPU kernel that simply gathers segments of a request and deposits them into the appropriate TQEs allocated from the request's TQs which are GPU resident using the GPU's DMA engines. Similarly, once a response is ready, the TQE is copied to the appropriate segments of the response. Note that these operations do not move the data across the PCIe bus which is an order of magnitude slower than the high speed GPU memory.

On the SmartNIC, the request is first assembled on the SmartNIC's memory where the buffers reside. Then the entire payload is RDMA-ed using 1-way `RDMA-WRITE` in a single shot to the service's TQ residing on the GPU utilizing GPU Direct RDMA capabilities. In the response path, the response TQE is read using one-way `RDMA-READ` and scattered onto one or more buffers before transmitting it to the corresponding client.

5.3 Orchestration mechanism of SplitRPC

To coordinate between the network stack and the inference framework, SplitRPC implements a side-car data structure called *Inference-Monitor (IM)* for each service. The IM maintains a state-machine of all the RPC calls containing (i) *RPC_call_id:* an ID to uniquely identify an active RPC call, (ii) *client_info*: the client information (4 tuples explained previously), (iii) *TQE_ptr*: pointers to the appropriate TQEs (or ZCEs in case of Zero-Copy), and (iv) a flag representing the `state` stored on the host memory. This `state` is an integer representing the RPC call being in one of these states: `{READY, RX_COMPLETE, WORK_COMPLETE, TX_COMPLETE}`. Once the payload is gathered by the network stack, it marks the request in the IM as *RX_COMPLETE*. At this stage the accelerator can run the computation. Once the execution is complete, the RPC call's state is updated to *WORK_COMPLETE*. The network stack then picks up these completed RPCs and transmits the response back to the clients, marking them as *TX_COMPLETE*. The IM automatically recycles the RPC calls to be used by subsequent requests.

On the SmartNIC, the IM is distributed between the host and the SmartNIC's memory. The client information and the pointer information is stored on the SmartNIC's memory while the *state* is replicated on both the host memory and the SmartNIC's memory. The SmartNIC writes both the request payload into their TQ and the *state* information using `RDMA-WRITE` as a work-request with multiple scatter-gather entries. Once the inference is complete on the accelerator, the SmartNIC is notified about this using an `RDMA-SEND` with the *RPC_call_id* sent inline as an *immediate* data. We experimented with a polling mechanism using one-sided `RDMA-READ`s and found it to be a less performant mechanism.

5.4 Supporting dynamic batching of requests

As discussed earlier, ML computations are often batched together to boost the throughput and increase the utilization of the accelerator. The inputs to the batch of requests need to be contiguously laid out in the memory (or require additional memory copies to make them contiguous). The batch sizes can dynamically vary depending on the system load and the available outstanding requests. SplitRPC automatically lays out incoming requests into its TQs in a contiguous fashion by allocating subsequent TQEs next to each other in memory. However, there is one important case to be handled which is illustrated in Figure 6. Let us say, the current batch that is executing corresponding to TQEs *T6, T7* shown in green, and the next batch requires TQEs *T8, T1, T2* shown in orange. These 3 TQEs (*T8, T1, T2*) need to be contiguous. Consequently, we need to carefully handle the wrap-around scenario where the current TQE can point to the end of the TQ and the next TQE can point to the start of the TQ. To deal with this scenario, we implement a clever trick to the TQ memory region allocation as shown in Figure 6 and explained as follows. We first allocate a contiguous physical

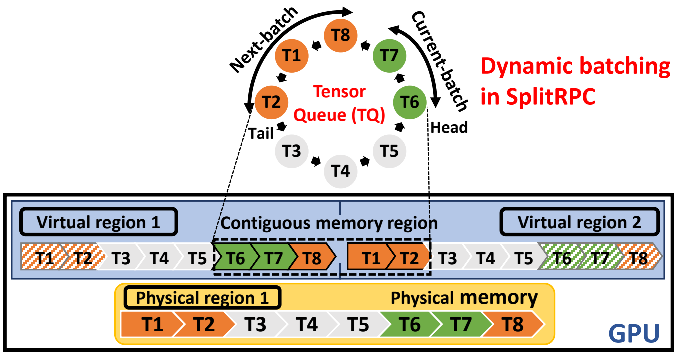


Fig. 6. SplitRPC performs dynamic batching using a novel memory allocation mechanism of the Tensor Queues to easily handle the wrap-around case while accessing contiguous Tensor Queue Entries (TQEs).

region ('Physical region 1') of memory corresponding to each TQ. We then map the same physical region twice in a back-to-back fashion to form a large contiguous virtual region as 'Virtual region 1' and 'Virtual region 2'. The lower half of this virtual region ('Virtual region 1') is then exposed as the TQ_MR from which TQEs are allocated one after the other. This provides the illusion to the ML inference framework that a batch of requests, even if they wrap around the TQ, are always contiguous in virtual memory. We use the recently introduced CUDA Virtual Memory APIs [45] to achieve this on Nvidia GPUs [1].

5.5 Using SplitRPC

SplitRPC is implemented in C++ and exposes C++ APIs to be consumed by ML services. We briefly describe how the `client` and `server` components of an ML service can use SplitRPC to serve requests on an accelerator/GPU in Figure 7. SplitRPC exposes two sets of APIs - (i) SplitRPC-Server library APIs to be consumed by the ML App, and (ii) SplitRPC-Client library APIs to be used to write application clients. A `client` can be any other workload (e.g., a web application) that wants to avail ML inference as a service. This client can be running on same or another machine in the datacenter. Clients use the SplitRPC-Client side APIs to generate a SplitRPC request by passing the application payload (e.g., Image). To make an RPC call, the application client simply needs to make the API call with the *func_id* and payload information to the client library. These APIs can be consumed as a static/shared SplitRPC library similar to how gRPC applications are linked to gRPC libraries. The application configures the client library during application startup to point to

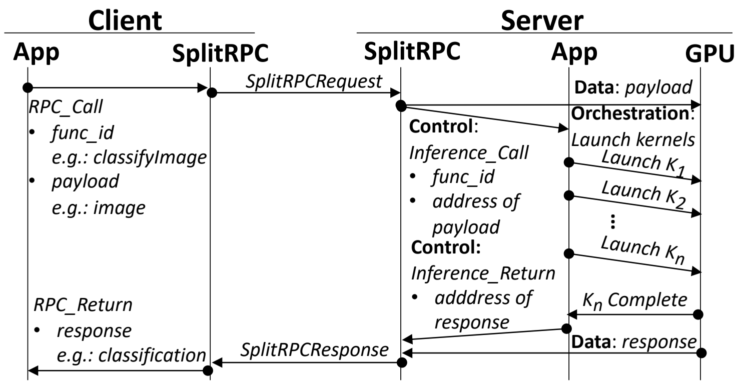


Fig. 7. Shows how the Client/Server components of an ML service can use SplitRPC.

[1]CUDA guarantees the coherence between these two virtual memory addresses as long as they are not simultaneously accessed in a threadblock [45].

Service	Domain	Dataset	Framework	# Kernels	Duration	GPU (us)	IO size	Request	Response
LeNet	Image (CNN)	MNIST	TVM [10]	12	Short	61	Tiny	784 B	40 B
LSTM	Text (RNN)	Synthetic	NNFusion [39]	32	Short	198	Small	8 KiB	1 KiB
ResNet18	Image (CNN)	ImageNet	TensorRT [68]	50	Medium	588	Large	588 KiB	3.9 KiB
ResNet50	Image (CNN)	ImageNet	TensorRT [68]	95	Long	1248	Large	588 KiB	3.9 KiB
BERT	NLP (Transformers)	SQUADv2	TensorRT [68]	172	Long	2042	Small	6152 B	2056 B

Table 1. Evaluation covers ML services spanning different application domains, inference frameworks, duration, and IO sizes.

a specific service endpoint (*IP address, Port number*) and *func_id* (provided as user input) along with other information such as payload size etc. In our experiments, the client application is a load generator application which (i) generates requests (application payload) corresponding to the configured load, (ii) calls the SplitRPC client APIs to send the request, (iii) captures response returned by the client, and (iv) tracks performance measurements corresponding to a model. The `server` process needs to be created using the SplitRPC-Server library APIs to process this request as described in Section 5.1. The ML application needs to provide function callbacks to the server side library which will be invoked once a RPC request is received on the server.

SplitRPC-Server side APIs: SplitRPC exposes server side APIs that can easily be adopted by inference frameworks (like TensorRT) or simple CUDA kernels. To setup/register a RPC function, the application provides: (i) a *func_id* which identifies the service for the client, (ii) a descriptor for the set of inputs and outputs including the device on which the ML service will consume/produce the data, (iii) function callbacks to the inference framework to be executed upon receiving a request and/or finishing a response, and (iv) a flag to enable dynamic batching and specify a maximum batch size. SplitRPC will then call the application's function and provide the location of the payload (within the GPU's memory) whenever a RPC request is received. The application then has control over the orchestration and launching of the kernels on the GPU. Once the ML computation is complete and the application's function returns (marked by *WORK_COMPLETE*), SplitRPC's network stack gathers the response data from the GPU and host memory and send it back to the client. This entire API is simple to use and adds 25 - 65 LOC per application for the ones that we have evaluated.

SplitRPC-Client side APIs: The client calls SplitRPC with a *func_id* indicating the RPC function to call, the input payload (e.g., image to classify), and the server endpoint. We implement two clients to be consumed by any service. (i) a DPDK-based client which is mainly used by our load generator test harness, and (ii) a traditional socket-based client to be used by any general application. The client API takes the user provided payload, splits it into appropriate segments, inserts the SplitRPC header with the client-specified *req_id*, *func_id* and *seq_num*, and routes it to the appropriate RPC server endpoint. Once the response has arrived for the *req_id*, it returns the response payload back to the client function.

6 EVALUATION

Our goal in this work is to illustrate that RPC mechanisms for accelerated services must be cognizant of accelerator orchestration in addition to just optimizing the data-path. While prior proposals have focused on individually performing each, we have built split designs that achieve both simultaneously, leveraging the full capabilities of commodity NICs available in the market today rather than incurring SmartNICs. We first measure the improvements in end-to-end (E2E) execution time of SplitRPC implemented on (a) Mellanox

	Client	Server
CPU	Intel Xeon Gold 6230 LLC: 27.5MB	Intel Xeon Gold 6226 LLC: 44MB
Memory	188GiB DDR4	251GiB DDR4
NIC	Mellanox CX-5 EN 100GbE	pNIC: Mellanox CX-6 EN 100GbE sNIC: Bluefield DP-2 EN 100GbE
GPU	-	Nvidia A100 40GB
PCIe Switch	-	PLX PEX 8796 Gen 3

Table 2. Hardware setup.

CX-6 P2P NIC (SplitRPC-pNIC) and (b) Mellanox BlueField DP-2 SmartNIC (SplitRPC-sNIC) over traditional gRPC-based approaches and Lynx [66], a recent SmartNIC-based data-path optimizing implementation (which is representative of a type (III) implementation depicted in Figure 4) (Section 6.2). We then study the latency performance (both mean & tail) of different SplitRPC based user-facing ML services under stochastic load conditions (Section 6.3). Finally, we analyze the dynamic batching capabilities of SplitRPC (Section 6.4).

6.1 Setup and methodology

Hardware setup: Table 2 lists the configuration details of the client and server machines used for evaluation. The machines are connected in a back-to-back configuration using a high speed 100GbE QSFP DAC cable, in order to minimize latency and maximize the throughput. We enable jumbo frames on the link and set the MTU to be 9000B. We disable power saving features on the CPU and GPU on all the machines and isolate cores to minimize interference while running our experiments. The NICs (SmartNIC/P2P NIC) and the GPU on the host are connected to each other using a PCIe switch to maximize their peer bandwidth, and all services are run on the closest host socket CPU to both these devices.

Schemes compared:

- **gRPC-VMA:** This scheme represents state-of-the-art ML inference services [15, 22, 56, 76]. Most (if not all) inference services use gRPC as the RPC framework. We prototype the services shown in Table 1 using the C++ APIs of gRPC [20] and additionally enhance it using VMA [40], a network accelerator by Mellanox that allows by-passing the network stack directly to user space.
- **Lynx-sNIC:** This is a recent state-of-the-art [66] that uses SmartNICs to directly transfer data to the accelerator. This represents a GPU based approach (see Figure 4) which implements orchestration on the GPU without any explicit support for batching/expose an RPC interface. We compare the LeNet and LSTM models using this approach as only certain models are amenable to this kind of orchestration. We specifically chose LeNet as it was explored in depth by Lynx [66].
- **SplitRPC-pNIC:** This is our split design scheme implemented with a Mellanox CX-6 P2P NIC. It combines the benefits of having a good control path while also optimizing for the data-path to the accelerator.
- **SplitRPC-sNIC:** This represents our SplitRPC scheme implemented on the Bluefield DP-2 SmartNIC.
- **SplitRPC-local:** This is a version of SplitRPC running locally on the server without traversing the network/NIC. This serves as a theoretical upper bound version with lowest data movement.

ML inference services evaluated: The models that we had access to and could run on our platform largely determined their choice in our evaluations. However, we demonstrate the flexibility and versatility of SplitRPC by choosing different kinds of ML models with short/long execution times and different request/response sizes. Most importantly our models are built using different kinds of ML inference engines (TVM [10], NNFusion [39], TensorRT [68]). We have compiled all models/inference engines against CUDA version 11.4. Table 1 lists the full details of the ML inference applications that are evaluated. In addition to these, we also use microbenchmarks performing vector addition (VecAdd) and matrix multiplication (MatMul) to articulate the importance of kernel size on orchestration. The results indicate that even simple models require careful orchestration. SplitRPC, by design is not tied to any specific model that we have used here in the evaluation.

Load generator: The client machine runs a common test harness which calls the corresponding client (gRPC/SplitRPC) to benchmark all systems and captures all the measurements (except certain breakdown measurements) on the client machine only. We use a closed loop load generator

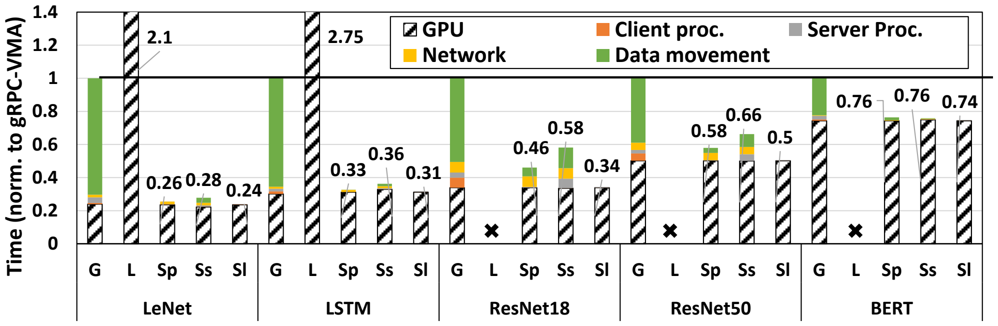


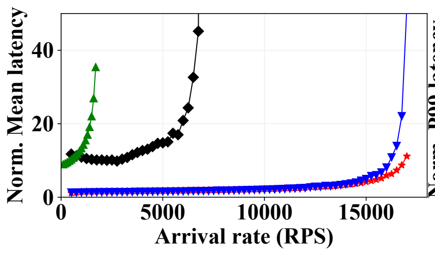
Fig. 8. End-to-end (E2E) execution time (normalized to gRPC-VMA). Lower is better. **G**: gRPC-VMA, **L**: Lynx-sNIC, **Sp**: SplitRPC-pNIC, **Ss**: SplitRPC-sNIC, **Sl**: SplitRPC local to the host (ideal). SplitRPC's split {control + data} path performs better than just optimizing for the data (Lynx) or control (gRPC) paths.

to measure (i) end-to-end execution time (single client) and the (ii) peak throughput (multiple concurrent clients). We use an open loop load generator to evaluate the (iii) latency under a range of load settings. These metrics are roughly equivalent to the distributed versions of the MLPerf [55] evaluation scenarios - Single stream, Offline, and Server, respectively. We model request arrivals in the open loop load generator as a Poisson Process to mock user-facing workloads, which are parameterized by an arrival rate measured as requests-per-second (RPS) representing the load imposed on the system. We run a validation phase to warm up the application and validate the response before benchmarking each service.

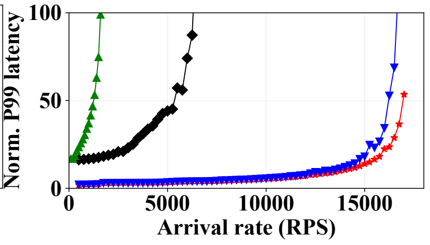
6.2 Improvements in end-to-end (E2E) execution time

We first compare and study the break down of the normalized end-to-end (E2E) execution time for each model in Figure 8. All times are normalized to the baseline gRPC-VMA. The E2E execution time is broken down along 5 dimensions. (i) `GPU time:` execution time spent on the accelerator. (ii) `Client proc.:` processing time spent on the client including (de/)serialization for gRPC. (iii) `Server proc.:` processing time spent on the server including (de/)serialization for gRPC. (iv) `Network:` wire latency spent on the network. (v) `Data movement:` the remaining time, including the data movement between the GPU/CPU/NIC and the time spent managing/moving data in the network stack on the host.

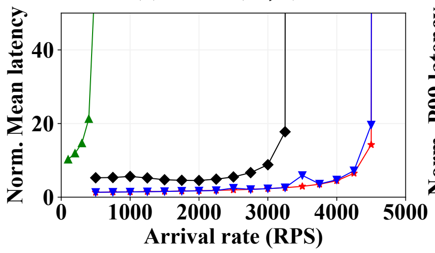
Across all the real-world ML models evaluated in this experiment, SplitRPC implementations perform much better than the conventional gRPC approach and the more recent Lynx approach. Under SplitRPC-pNIC, the E2E execution time has reduced over existing CPU based schemes (i.e., gRPC-VMA) by 52% on average with a maximum savings of 74% for LeNet. Most of these savings are from reducing the data movement costs (shown in solid green). The contribution of data movement alone has reduced from 49% (on average) under gRPC-VMA to 2% (on average) under SplitRPC-pNIC. As expected, Lynx-sNIC does reduce the data movement costs, but the overall performance is poor (2.1×, 2.7× worse than gRPC-VMA) for both the ML models due to the sub-par orchestration mechanism employed in serving them. On the other hand, on the SmartNIC, our SplitRPC-sNIC scheme reduces not just the data movement but has reduced the execution time by 47% on average (max of 72% for LeNet) over the baseline gRPC-VMA scheme. While this is good, it performs worse than our SplitRPC-pNIC scheme (by 5 % points) as it performs an extra copy operation to the SmartNIC's DRAM using the less capable ARM cores on the SmartNIC. Lastly, SplitRPC-pNIC scheme is within 14% (on average) of an ideal SplitRPC-local scheme where the RPC is made local to the host. For models with large data transfers such as ResNet18 SplitRPC-pNIC has



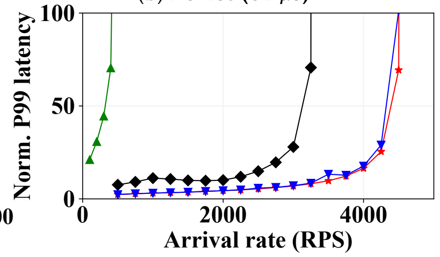
(a) **LeNet (61μs)**



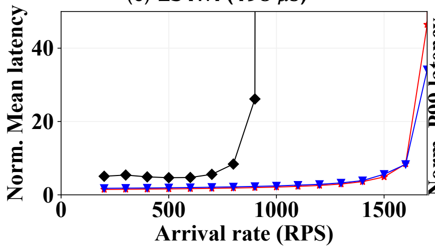
(b) **LeNet (61 μs)**



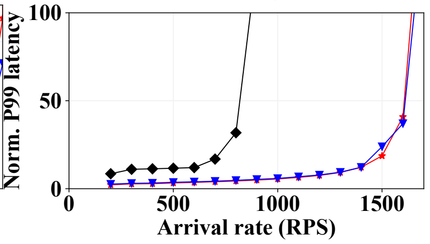
(c) **LSTM (198 μs)**



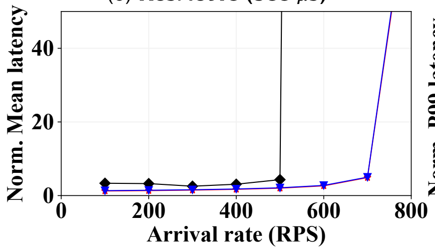
(d) **LSTM (198 μs)**



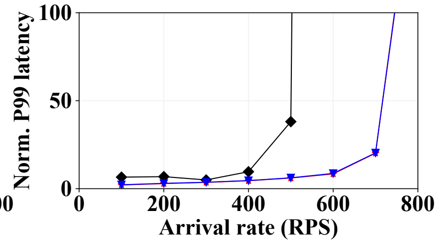
(e) **ResNet18 (588 μs)**



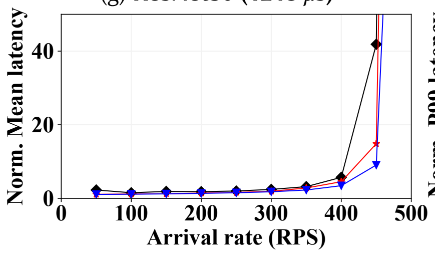
(f) **ResNet18 (588 μs)**



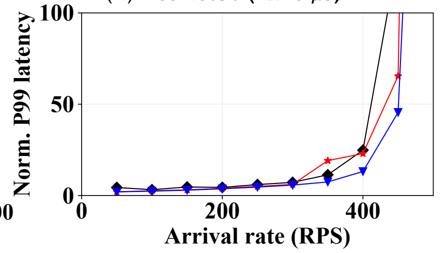
(g) **ResNet50 (1248 μs)**



(h) **ResNet50 (1248 μs)**

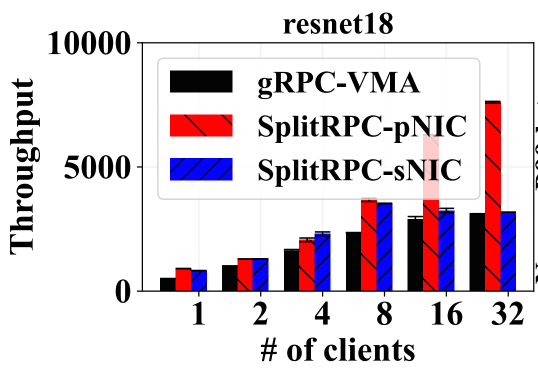


(i) **BERT (2040 μs)**

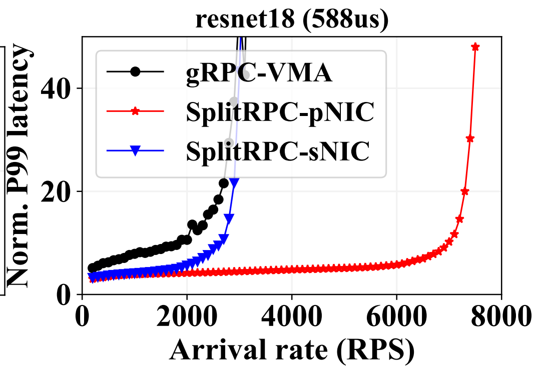


(j) **BERT (2040 μs)**

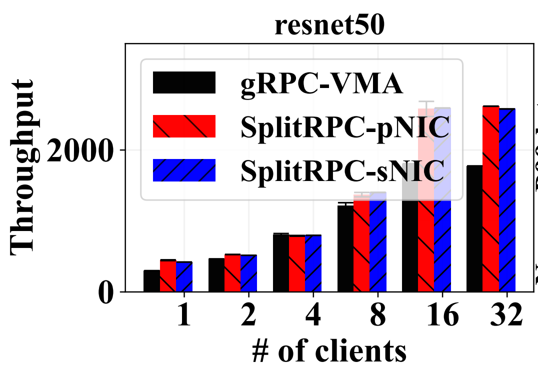
Fig. 9. Mean latency (left column) and tail latency (right column) under different load regions. ◆- gRPC-VMA, ▲- Lynx-sNIC, ★- SplitRPC-pNIC, ▼- SplitRPC-sNIC



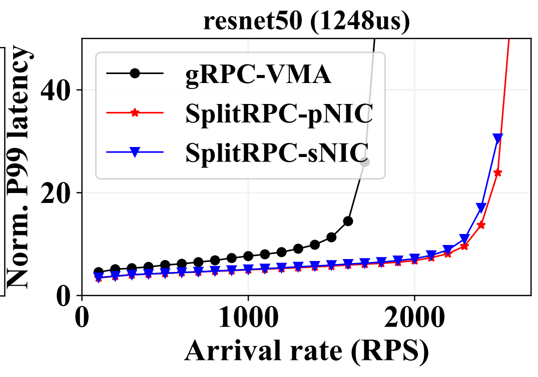
(a) **ResNet18 - Throughput**



(b) **ResNet18 - P99 tail latency**



(c) **ResNet50 - Throughput**



(d) **ResNet50 - P99 tail latency**

Fig. 10. Throughput and tail latency performance with dynamic batching. Maximum batch-size is set to 8.

a higher latency over the ideal (35%) and the overhead is almost negligible (2%) with models like BERT.

These results show that: *For ML inference serving applications, achieving savings in data movement is important, but it is even more important to achieve these savings while being aware of the orchestration overheads in coordinating the launch of kernels on the GPU.* Additionally, as long as we are able to split the control and data flow to the two different entities (host and GPU for control and data respectively), deploying additional hardware (in the form of SmartNICs) is not really required.

6.3 Mean and tail latency under different load regions

Many of these ML services are typically deployed as user-facing applications, making it imperative to study their mean and, importantly, tail latency under various load conditions. We vary the arrival rate of requests to the server and study their normalized (to 'GPU' time) latency. Figure 9 shows the mean (left column) and P99 tail latency (right column) for all the schemes and services under different load regions (shown as requests-per-second (RPS) on the x-axis). There are two key observations. (i) SplitRPC schemes significantly increase the load regions under which the service can safely operate (i.e., the knee points in the graph move to the right). These can be as high as 2.6× higher than the baseline gRPC-VMA scheme as in the case of LeNet. On the other hand, gRPC-VMA and Lynx are saturated easily because of the overheads imposed by the data movement and the orchestration respectively. This is fundamentally due to the differences in the maximum throughput provided by each approach. Even without batching, these experiments show the ability of SplitRPC to handle higher loads, and in the next section, we'll see how batching provides even greater benefits to throughput. (ii) Even under low load regions, the SplitRPC schemes provide significant savings in mean and tail latency. For services with a low 'GPU service' time like LeNet, these savings are huge – 90% (mean latency) and 88% (P99 tail latency) savings over gRPC-VMA under low load. For other services like BERT, these savings are a respectable – 22% (mean latency) and 18% (P99 tail latency) savings over gRPC-VMA under low load.

6.4 Dynamic batching

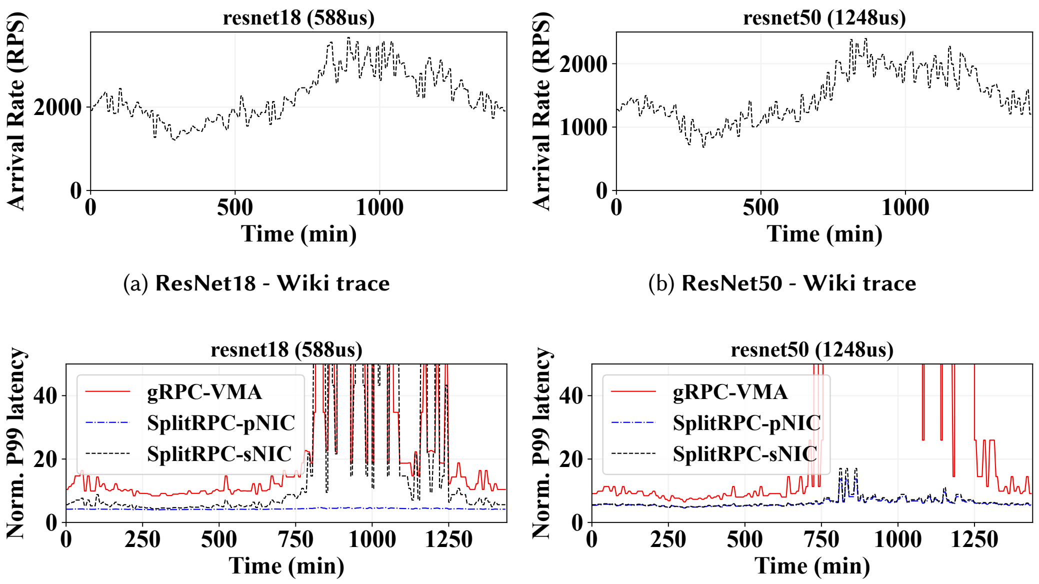
We next turn our attention to the benefits of batching. Recall that our Tensor-Queues (TQ) dynamically batch multiple inputs together when the ML model supports batching. Our goal in this work is not to come up with the best policy for batching, but to illustrate that SplitRPC can complement and be easily extensible for batching. We have implemented a work-conserving batching policy that dynamically caps and varies the batch size depending on the amount of outstanding work and runs these batched inferences. That is, once an execution with a particular batch-size is complete, we run another execution with (all) the outstanding requests capped by a maximum batch size. In these experiments, we fix the maximum batch-size as 8 on the inference framework as these are the typical batch-sizes deployed for user-facing scenarios. SplitRPC by itself does not have any batch-size limitations.

We focus on two models, ResNet18 and ResNet50, that support batching. Despite similar payload sizes, ResNet50 is larger and has a significantly longer GPU execution time than ResNet18. We first study the peak throughput under this dynamic batching scheme by varying the number of clients concurrently making requests to the service. As number of clients increase, the dynamic batch size the policy picks is expected to increase, resulting in a higher throughput. Figure 10a and Figure 10c show the peak throughput for ResNet18 and ResNet50 models under different number of concurrent clients (x-axis). Overall, under a high concurrency value of 32, the SplitRPC schemes provide 2.4× and 47% higher throughput over the gRPC-VMA baseline, with SplitRPC-pNIC providing the maximum throughput. The SplitRPC-sNIC is limited in performance for the ResNet18 model as it is running on a slow ARM NIC core, which becomes a bottleneck. Note that for the ResNet50 model, this is not observed, as the GPU is the bottleneck due to higher execution times than ResNet18. On comparing the tail latency in Figure 10b and Figure 10d, we see that the SplitRPC schemes provide consistently lower tail latency across the entire load region. In a low/mid load region of 1500RPS, the savings in tail latency over the gRPC-VMA baseline with (SplitRPC-pNIC, SplitRPC-sNIC) is (54%, 48%) and (49%, 47%) for ResNet18 and ResNet50 respectively.

These results highlight two observations: (i) the benefits of split designs amplify for larger batch sizes, and (ii) the limited compute on the SmartNIC may become a bottleneck due to the limited processing power of the SmartNIC.

6.5 Performance with workload traces

We analyze the performance of SplitRPC on a real workload trace [67] with dynamic batching. We use the Wikipedia traces [67] as they are representative of user facing ML inference services. We scale the wiki trace to ensure maximum load is servable on our experimental setup and show the performance for 1-day's trace. In addition to this, to simulate bursty conditions, we add Gaussian noise to the load (10%). The generated trace is shown in Figure 11a and Figure 11b for ResNet18 and ResNet50 models. Figure 11c and Figure 11d show the P99 tail latency of gRPC-VMA baseline, SplitRPC-pNIC, and SplitRPC-sNIC schemes for ResNet18 and ResNet50 models. As we can clearly see, SplitRPC-pNIC scheme consistently outperforms the other two across the entire trace. For



(a) **ResNet18 - Wiki trace** (b) **ResNet50 - Wiki trace**

(c) **ResNet18 - P99 tail latency** (d) **ResNet50 - P99 tail latency**

Fig. 11. Tail latency performance (norm.) under Wikipedia workload traces [67] (scaled and 10% Gaussian noise added for burstiness.

ResNet18, SplitRPC-sNIC starts degrading in performance even under moderate loads (>2500 RPS) because of the slower ARM cores, similar to the behavior we saw earlier in Section 6.4. For ResNet50, both SplitRPC-pNIC and SplitRPC-sNIC schemes are competitive and significantly outperform the baseline gRPC-VMA scheme.

7 DISCUSSION

7.1 Impact of orchestration:

So far, we saw that approaches that fail to optimize the control path (i.e., choices II/III in Figure 4) perform poorly when serving ML models. To further reiterate the importance of considering orchestration, we study microbenchmarks with single kernels which are not representative of realistic ML models. The two microbenchmark kernels are implemented as simple CUDA kernels: VecAdd1K (adding a constant to a vector of size 1KiB) and MatMul8K (multiplying two 32x32 matrices with a total size of 8KiB).

Figure 12 shows the E2E execution time (left) and peak throughput (right) achieved under the 4 schemes that we have studied so far. The E2E execution time is comparable for Lynx-sNIC and SplitRPC-pNIC. SplitRPC-sNIC has a slightly higher execution time (10μs overhead) due to the extra notification step involved with the host CPU. The peak throughput for Lynx-sNIC is significantly higher as it completely avoids the kernel launch overheads resulting in these gains. For simple microbenchmarks, as kernel launch overheads start to dominate, the best way to run them is directly on the accelerator using O2 design that completely avoids the overhead. Note that our SplitRPC-pNIC is not inherently limited by the orchestration choice itself, as it can also perform the O2 orchestration. We build and run the same microbenchmarks with O2 orchestration, labeled **Sp-O2** in Figure 12. Our experiments show it performs better than Lynx-sNIC. *Fundamentally, the*

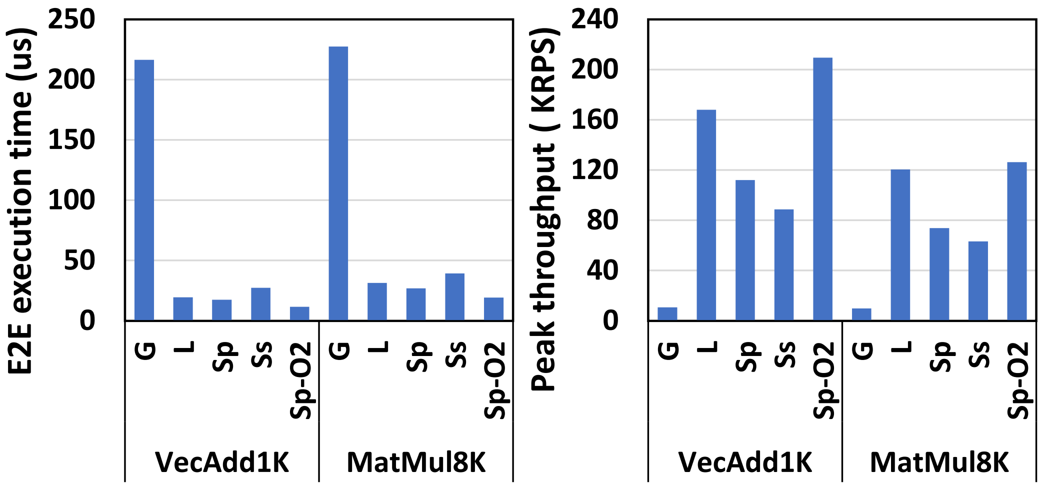


Fig. 12. **G**: gRPC-VMA, **L**: Lynx-sNIC, **Sp**: SplitRPC-pNIC, **Ss**: SplitRPC-sNIC, **Sp-O2**: SplitRPC-pNIC built to use O2 design.

O2 orchestration is superior when launch overheads is dominant, but for complex ML models with multiple kernels, the O1 orchestration is far superior.

Since SplitRPC can support both O1 (CPU-managed) and O2 (GPU-managed) orchestration choices, the versatility of SplitRPC allows for maximum flexibility in control-path orchestration while simultaneously achieving data-path optimizations. In addition, services can be engineered to employ the dynamic batching capabilities offered by SplitRPC to amortize the orchestration overheads even further.

7.2 Extending SplitRPC to other accelerators and SmartNICs:

The SplitRPC design is currently implemented on Nvidia GPUs and Mellanox SmartNICs. Its implementation can be extended to other GPUs and SmartNICs. SplitRPC primarily relies on the vendor for the availability of: (i) a mechanism to allocate (e.g., using *cudaMalloc* on Nvidia devices) and access device memory across the PCIe in a peer-to-peer fashion (e.g., Nvidia GPUDirect, AMD DirectGMA) and (ii) APIs for creating direct data-paths between the (R)DMA engines on the NIC to accelerator / host CPU on the server. Using the data-path APIs and mechanism to allocate device memory, the SmartNIC can selectively copy the control and data portions to the respective memory locations. Besides these requirements, the core components of SplitRPC itself - the Inference Monitor and the Tensor Queues - can easily be ported to other GPUs/accelerators and SmartNICs.

8 RELATED WORK

Optimizing ML model serving: There have been two broad research directions to improve ML model serving: (i) There has been a whole suite of hardware [11, 13, 38] and software solutions [10, 39, 42, 60, 75, 77] to reduce the execution time (i.e., 'GPU' time) of the ML application that focus on the execution within the accelerator and do not consider the 'RPC Tax' (see Figure 1) of running them under distributed settings. (ii) Solutions that optimize for deployment challenges [13, 15, 26, 48] and democratize ML services provide higher level support for deployment under public/private clouds by (a) optimizing for cost [24, 74] and SLOs [21, 22, 56, 73], (b) dealing with multi-tenancy [17, 25, 72], and (c) constructing optimal service pipelines on GPU [14, 59, 76] and CPU [35] clusters. These solutions typically employ a CPU based RPC design (choice (I) in Figure 4). As the rising RPC tax start limiting the full capabilities of these solutions, these serving systems stand to gain immensely in realizing their cost/SLO goals by using SplitRPC to serve ML models.

Optimizing RPCs: RPCs have been an intense topic of discussion for many years [2, 8, 70], with recent software [33] and hardware [52, 62] mechanisms aiming to generalize their usage and/or reduce the 'RPC tax'. These proposals are not designed for accelerated ML inference queries that require us to handle a number of specific issues as described in Section 5. We build upon existing approaches [30, 33, 53] and identify the distinct factors that impact an RPC mechanism for serving ML inference tasks. Our SplitRPC design addresses these issues through a split {control + data} path approach.

Network stacks and accelerators: There are network stacks that have been accelerated using custom hardware [4, 9, 18, 58], for GPUs [1], and to run ML computations on custom FPGA deployments [13]. In contrast to such approaches, SplitRPC is designed for commodity NICs facilitating immediate and wide deployment in the datacenter. Similarly, there have been a few proposals that build software based network stacks for GPUs such as GPUNet [61] and GPUrdma [16] following GPU based implementation designs (implementation choice (III) in Figure 4). These require infiniband networks and primarily expose socket-like APIs directly on the GPU. These are designed for simple GPU applications, and not for ML services which are inherently more complex with dozens of kernel launches as shown in Table 1. They do not implement an RPC mechanism, and thus do not support features like dynamic batching. They require the GPU to perform the orchestration, limiting their suitability for ML inference. Similarly, SmartNICs have become a hot proposition to offload many parts of distributed applications [12, 19, 32, 36, 37, 49, 57]. Lynx [66] offloads the network stack to the SmartNIC and uses the compute on them to directly establish data-paths to the accelerator. It assumes the accelerator will orchestrate the computation, which works well for a single simple kernel. But as we have shown in Section 6.2, such approaches are counter-productive for complex ML models. On the other hand, we show SplitRPC-pNIC achieves even better benefits on cheaper commodity P2P NICs without requiring expensive SmartNICs with extensive processing capabilities. FractOS [69] is a recent work that takes the right step in identifying the 'tax' for running applications on remote accelerators and proposes RDMA based data-paths between different devices. It works well for a face verification kernel, but as we have extensively analyzed in this work, complex ML models (as opposed to simple kernels) are impacted by the orchestration mechanism even more so than the data-paths to accelerators.

9 CONCLUSION

We explore RPC mechanisms used to serve ML applications on accelerators (e.g., GPUs). We consider ML inference, an important workload in the datacenter, and quantify the rising RPC tax in serving them. We show the need to co-optimize data-paths and the accelerator orchestration mechanism by systematically deconstructing the steps involved in serving an ML inference. While prior works expressly search for data-path optimizations, we show that these must not come at the cost of control-path inefficiencies. Consequently we design SplitRPC, which implements *split* designs leveraging the P2P capabilities of commodity NICs and show their efficacy over control-path-only and data-path-only optimized designs. As workloads are increasingly accelerated, we highlight the control-path awareness in serving them. SplitRPC takes a step in this direction, and we hope to extend this to different types of accelerators (e.g., FPGAs, near memory compute devices) targeting different application domains (e.g., data-analytics).

10 ACKNOWLEDGEMENTS

We thank our shepherd Wei Bai, and the anonymous reviewers at SIGMETRICS for their helpful feedback. We also thank Prasanna Rengasawmy, Iyswarya Narayanan, Saambhavi Baskaran, Balachandran Swaminathan, and Siddhartha Rai Balakrishna for reading through drafts of this paper and providing useful suggestions and feedback. This research was supported in part by National Science Foundation grants 2211018, 1912495, 1909004, 1714389, 1912495, 1629915, 1629129, 1763681, a grant from Intel Corporation and by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] Roberto Ammendola, Andrea Biagioni, Ottorino Frezza, Gianluca Lamanna, Alessandro Lonardo, F Lo Cicero, Pier Stanislao Paolucci, F Pantaleo, Davide Rossetti, Francesco Simula, et al. 2014. NaNet: a flexible and configurable low-latency NIC for real-time trigger systems based on GPUs. *Journal of Instrumentation* 9, 02 (2014), C02023.

[2] A. Ananda, B. Tay, and E. Koh. 1991. ASTRA-an asynchronous remote procedure call facility. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 172,173,174,175,176,177,178,179. https://doi.org/10.1109/ICDCS.1991.148661

[3] Apache. 2020. bRPC Framework. https://brpc.apache.org/. [Online; accessed 09-Aug-2022].

[4] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. 2021. NanoTransport: A Low-Latency, Programmable Transport Layer for NICs. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. Association for Computing Machinery, New York, NY, USA, 13–26.

[5] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[6] Saambhavi Baskaran, Mahmut Taylan Kandemir, and Jack Sampson. 2022. An architecture interface and offload model for low-overhead, near-data, distributed accelerators. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, IEEE, New York, NY, 1160–1177.

[7] Saambhavi Baskaran and Jack Sampson. 2020. Decentralized offload-based execution on memory-centric compute cores. In *The International Symposium on Memory Systems*. Association for Computing Machinery, New York, NY, USA, 61–76.

[8] Andrew D. Birrell and Bruce Jay Nelson. 1983. Implementing Remote Procedure Calls. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)* (Bretton Woods, New Hampshire, USA) *(SOSP '83)*. Association for Computing Machinery, New York, NY, USA, 3. https://doi.org/10.1145/800217.806609

[9] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Boston, MA, USA, 973–990.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. USENIX Association, Boston, MA, 578–594.

[11] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[12] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. 2020. λ-nic: Interactive serverless compute on programmable smartnics. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE, IEEE, New York, NY, USA, 67–77.

[13] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving dnns in real time at datacenter scale with project brainwave. *iEEE Micro* 38, 2 (2018), 8–20.

[14] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In *Proceedings of the International Symposium on Cloud Computing (SoCC)*. Association for Computing Machinery, New York, NY, USA, 477–491. https://doi.org/10.1145/3419111.3421285

[15] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Boston, MA, USA, 613–627.

[16] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers.*

Association for Computing Machinery, New York, NY, USA, 1–8.

[17] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, New York, NY, USA, 492–506.

[18] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. {NICA}: An infrastructure for inline acceleration of network applications. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*. USENIX Association, Boston, MA, USA, 345–362.

[19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking:{SmartNICs} in the Public Cloud. In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Boston, MA, USA, 51–66.

[20] Google. 2018. GRPC Framework. https://grpc.io/. [Online; accessed 17-Apr-2022].

[21] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. 2017. Swayam: distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. USENIX Association, Boston, MA, USA, 109–120.

[22] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Boston, MA, USA. https://www.usenix.org/conference/osdi20/presentation/gujarati

[23] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar)*. IEEE, New York, NY, USA, 1–14. https://doi.org/10.1109/InPar.2012.6339596

[24] Matthew Halpern, Behzad Boroujerdian, Todd Mummert, Evelyn Duesterwald, and Vijay Janapa Reddi. 2019. One Size Does Not Fit All: Quantifying and Exposing the Accuracy-Latency Trade-Off in Machine Learning Cloud Service APIs via Tolerance Tiers. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, IEEE, New York, NY, USA, 34–47.

[25] Yitao Hu, Swati Rallapalli, Bongjun Ko, and Ramesh Govindan. 2018. Olympian: Scheduling Gpu Usage in a Deep Neural Network Model Serving System. In *Proceedings of the 19th International Middleware Conference*. ACM, New York, NY, USA, 53–65.

[26] Microsoft Inc. 2016. ONNX Runtime inference engine. https://github.com/Microsoft/onnxruntime/. [Online; accessed 17-Apr-2022].

[27] DPDK Intel. 2014. Data plane development kit.

[28] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, New York, NY, USA, 47–62.

[29] Stephen Jones. 2012. Introduction to dynamic parallelism. In *GPU Technology Conference Presentation*, Vol. 338. NVIDIA, Santa Clara, CA, 2012.

[30] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter {RPCs} can be General and Fast. In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Boston, MA, USA, 1–16.

[31] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE, New York, NY, USA, 462–478.

[32] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 756–771.

[33] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. {R2P2}: Making {RPCs} first-class datacenter citizens. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*. USENIX Association, Boston, MA, USA, 863–880.

[34] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 36–51.

[35] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. {PRETZEL}: Opening the Black Box of Machine Learning Prediction Serving Systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Boston, MA, USA, 611–626.

[36] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. ACM, New York, NY, USA, 318–333.

[37] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*. USENIX Association, Renton, WA, 363–378. https://www.usenix.org/conference/atc19/presentation/liu-ming

[38] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing {CNN} Model Inference on CPUs. In *2019 {USENIX} Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Boston, MA, 1025–1040.

[39] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Boston, MA, USA, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[40] Mellanox. 2022. LIBVMA. https://github.com/Mellanox/libvma. [Online; accessed 17-Apr-2022].

[41] Mellanox. 2022. Mellanox NV Peer memory. https://github.com/Mellanox/nv_peer_memory. [Online; accessed 17-Apr-2022].

[42] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion. In *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI)* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 883–898. https://doi.org/10.1145/3453483.3454083

[43] NVIDIA. 2022. CUDA GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html. [Online; accessed 17-Apr-2022].

[44] NVIDIA. 2022. CUDA Graphs API. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html. [Online; accessed 17-Apr-2022].

[45] NVIDIA. 2022. CUDA VMM Aliasing. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#virtual-aliasing-support. [Online; accessed 17-Apr-2022].

[46] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. arXiv:1712.06139 [cs.DC]

[47] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 293–307. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. Curran Associates, Inc., NY, USA, 8026–8037.

[49] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Carlsbad, CA, 663–679. https://www.usenix.org/conference/osdi18/presentation/phothilimthana

[50] Solal Pirelli and George Candea. 2020. A Simpler and Faster {NIC} Driver Model for Network Functions. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Boston, MA, USA, 225–241.

[51] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The benefits of general-purpose on-NIC memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 1130–1147.

[52] Arash Pourhabibi Zarandi, Mark Johnathon Sutherland, Alexandros Daglis, and Babak Falsafi. 2021. Cerebros: Evading the RPC Tax in Datacenters. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE, New York, NY, USA, –.

[53] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. 2021. Breakfast of champions: towards zero-copy serialization with NIC scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. USENIX Association, Boston, MA, USA, 199–205.

[54] Siddhartha Balakrishna Rai, Anand Sivasubramaniam, **Kumar, A**, Prasanna Venkatesh Rengasamy, Vijaykrishnan Narayanan, Ameen Akel, and Sean Eilert. 2021. Design space for scaling-in general purpose computing within the DDR DRAM hierarchy for map-reduce workloads. In *Proceedings of the 18th ACM International Conference on Computing Frontiers (CF)*. Association for Computing Machinery, New York, NY, USA, 113–123.

[55] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, New York, NY, USA, 446–459.

[56] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2020. INFaaS: A Model-less and Managed Inference Serving System. arXiv:1905.13348 [cs.DC]

[57] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 740–755.

[58] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. {FlexTOE}: Flexible {TCP} Offload with {Fine-Grained} Parallelism. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX, Bostom, MA, USA, 87–102.

[59] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 322–337.

[60] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently Compiling Dynamic Neural Networks for Model Inference. arXiv:2006.03031 [cs.PL]

[61] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. 2016. GPUnet: Networking abstractions for GPU programs. *ACM Transactions on Computer Systems (TOCS)* 34, 3 (2016), 1–31.

[62] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. 2020. The NEBULA RPC-optimized architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, IEEE, New York, NY, USA, 199–212.

[63] Tensorflow. 2022. Tensorflow Frozen Graph. https://docs.snap.com/lens-studio/references/guides/lens-features/machine-learning/ml-frameworks/export-from-tensorflow/. [Online; accessed 17-Apr-2022].

[64] **Kumar, A**, Iyswarya Narayanan, Timothy Zhu, and Anand Sivasubramaniam. 2020. The Fast and The Frugal: Tail latency aware provisioning for coping with load variations. In *Proceedings of The Web Conference (WWW)*. Association for Computing Machinery, New York, NY, USA, 314–326.

[65] **Kumar, A**, Anand Sivasubramaniam, and Timothy Zhu. 2022. Overflowing Emerging Neural Network Inference Tasks from the GPU to the CPU on Heterogeneous Servers. In *Proceedings of the 15th ACM International Systems and Storage Conference (SYSTOR)*. Association for Computing Machinery, New York, NY, USA.

[66] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 117–131.

[67] Erik-Jan van Baaren. 2009. Wikibench: A distributed, wikipedia based web application benchmark. *Master's thesis, VU University Amsterdam* 1 (2009).

[68] Han Vanholder. 2016. Efficient Inference with TensorRT.

[69] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. 2022. Slashing the disaggregation tax in heterogeneous data centers with FractOS. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, New York, NY, USA, 352–367.

[70] Brent B Welch. 1986. *The Sprite Remote Procedure Call System*. Technical Report. University of California at Berkeley, USA.

[71] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. 2021. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. USENIX Association, Boston, MA, USA, 206–212.

[72] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. arXiv:1902.04610 [cs.DC]

[73] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*. USENIX Association, Boston, MA, USA.

[74] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, Boston, MA, USA.

[75] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. Deepcpu: Serving rnn-based deep learning models 10x faster. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*. USENIX Association, Boston, MA, 951–965. https://www.usenix.org/conference/atc18/presentation/zhang-minjia

[76] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. 2022. Astraea: towards QoS-aware and resource-efficient multi-stage GPU services. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 570–582.

[77] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Boston, MA, USA, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

Received August 2022; revised February 2023; accepted April 2023