Application-Level Function Pipelines on In-Network FPGA Accelerators

Ziyi Yang*, Zsolt István†, Marco Canini*, and Suhaib A. Fahmy*
*King Abdullah University of Science and Technology, Thuwal, Saudi Arabia.
†Systems Group, Technical University of Darmstadt, Germany.
ziyi.yang@kaust.edu.sa

***Abstract*—In-network computing can dramatically improve the performance of application pipelines through accelerated processing and predictable chaining behavior. But to achieve practical adoption, we need to evolve in-network computing from network packet-level offloads towards application-level service. This requires networked computing nodes that preserve application semantics while executing functions on the network fast path. We design and implement such capability on a hostless FPGA framework that enables chaining of application-level functions executing on FPGAs across the network without host intervention. Our proposed hardware accelerator datapath has predictable latency and throughput, simplifying provisioning to meet application throughput. We use pre-processing for deep learning recommendation model training as a case study. Our evaluation shows that our framework adds minimal abstraction overhead while providing stable latency, reducing cross-device composition latency by 40% compared to strong CPU baselines.**

***Index Terms*—In-Network Computing, Field Programmable Gate Arrays, Hardware Acceleration.**

I. Introduction

Modern applications increasingly decompose into multi-stage function pipelines that must serve many concurrent clients with predictable latency. This pattern spans ML inference preprocessing [1], [2], microservice call graphs [3], [4], and media transcoding, where each request traverses multiple operator stages (e.g., decompress $\rightarrow$ transform $\rightarrow$ recompress). CPU-based servers struggle to meet processing demands in these use-cases, especially under high client concurrency. Computationally, FPGAs can overcome these constraints, though traditional deployments treat them as virtualized accelerators similar to GPUs [5], [6], hence not benefiting from tight coupling of computing and communication [7].

In-network computing (INC) is a promising approach for the efficient execution of function pipelines: computation is moved into the data path, with in-network and network-attached devices that have high compute capacity but deterministic service times [8], [9]. This allows bypassing host ingestion and scheduling overheads and makes inter-stage communication more efficient, thanks to the closeness to the network. However, existing INC platforms deployed in datacenters and clouds, such as Programmable Switches [10] and CPU-based DPUs [11] face practical constraints that limit application-level (L7) acceleration (see Section II for details). In this work we focus on enabling FPGAs to natively support compute-rich L7 operators for INC (e.g., floating-point operations and complex payload transforms used in machine learning) with tightly coupled network ingestion on-chip, which we believe elevates them to an ideal platform for INC [12].

Although related work has demonstrated the potential of FPGAs for INC [13]–[15], there remains a gap to achieve application-level (L7) processing for four reasons:

First, they often lack request level semantics. Many related works prototype computation on a per-network packet basis, however, in practice, the system must preserve request boundaries to enable per-request operator execution. This is especially important for ML pipelines, where requests commonly carry tensors or mini-batches that exceed a single network MTU and therefore span multiple packets [2], [16].

Second, FPGA prototypes often focus on a single device, without the ability to pipeline operators across several devices. In practice, computation will not always fit on a single in-network compute element and intermediate results must be forwarded between computational stages with minimal data movement and bounded buffering.

Third, related work often optimizes for maximum performance and not predictable scaling. In practice, provisioning resources should provide predictable capacity increases and stable latency under concurrency. This is one of the main selling points of INC over traditional, CPU-based, computation.

Fourth, FPGA-based prototypes are not commonly designed with multiple clients and performance isolation in mind. In practice, e.g. in machine learning use-cases, no single function pipeline will saturate bandwidth (e.g., due to source rate limits) but, instead, they share device resources. When sharing, they should not experience large latency disparities nor lead to resource dominance by a single flow.

In this work, we show how to address the four limitations of FPGA-based INC. We achieve this by a *hostless* FPGA-based framework that exposes request-level services invoked directly from the network, to support both in-FPGA and cross-FPGA function chaining, with performance isolation. We ground the discussion with a representative case study from Deep Learning Recommendation Model (DLRM) pre-processing [1], whose bounded-granularity operators are throughput-critical and often CPU-bottlenecked in production training/serving stacks.

ISBN 978-3-903176-82-9 © 2026 IFIP

We make the following contributions:

- Extending an existing single-node hostless FPGA accelerator platform [17] with a **multi-hop chaining mechanism** via a compact in-header routing field to enable in-FPGA streaming pipelines and cross-FPGA composition, neither supported by the prior platform.
- Proposing and validating a lightweight provisioning model that translates target throughput into required hardware parallelism, enabling deterministic system scaling.
- Through a DLRM pre-processing case study, demonstrating that our framework achieves better latency stability and cross-client fairness compared to high-performance DPDK-based software baselines.

II. Background and Motivation

A. Beyond Packet Offload: Hostless L7 In-Network Computing

CPU limitations on predictable latency: To achieve predictable latency under high CPU concurrency, prior work has evolved through a sequence of increasingly "structural" fixes. Early approaches mask variance in software. Dean and Barroso [18] reduce tail latency via request replication, but do so by wasting resources, using extra network traffic and duplicated computation. Heracles [19] targeted this inefficiency by improving utilization through co-locating high- and low-priority jobs with isolation for shared resources. However, this approach does not fundamentally remove contention in key microarchitectural structures (e.g., under hyperthreading) leaving residual interference. Shenango [20] and Caladan [21] address predictability more directly at the OS/runtime level by pairing kernel-bypass datapaths with core-centric scheduling (dedicated cores and careful placement), but this predictability comes at the cost of underutilization, often burning cores for polling and control. Finally, DPDK-based efforts such as Metronome [22] aim to reclaim that wasted core time using adaptive sleep-wake polling, but prediction errors can still trigger latency spikes and the mechanism typically requires workload-dependent tuning.

The CPU scheduling and execution model makes it hard to deliver predictable latency under high concurrency, and this variability is further amplified when processing is chained across multiple servers. These inherent sources of jitter motivate INC, moving request execution and chaining control to network-attached devices with more deterministic service times reduces reliance on host scheduling and makes end-to-end latency more predictable at high concurrency.

Packet-level offload to network-attached devices: Datacenter architectures have steadily pushed latency and throughput critical work out of the central host CPU into network-attached devices. Packet-oriented offloading systems such as ClickNP [14], PANIC [13], and SuperNIC [15] demonstrate that programmable hardware can accelerate network stream with complex network functions (e.g., encryption/decryption), but they remain largely host-centered and lack the protocol support needed to expose application-level semantics. With the rise of P4-programmable switches, some functionality moved further into the fabric to reduce host round trips (e.g., NetCache [23]) or accelerate specific communication primitives (e.g., SwitchML [24]). However, match-action pipelines and limited on-chip state constrain these designs to relatively simple logic, making compute-rich processing (e.g., floating-point operations) difficult. While programmable DPUs (e.g., BlueField [11]) can run transport and processing offloads on the NIC, they typically rely on general-purpose ARM cores for computation. Consequently, they inherit the exact same scheduling and contention issues as host CPUs described above, leaving the DPU's operating system in the critical path and compromising performance predictability.

L7 request semantics and service graphs: Packet-level offloads are effective for network functions. They typically operate on individual packets but offer limited support for preserving request boundaries and manipulating application payloads. These limitations are driving a shift toward L7 acceleration, where in-network devices process data at the granularity of application requests. The rise of serverless and microservice architectures further strengthens this trend: applications are decomposed into stateless, request-driven functions, making the request the natural unit of work and demanding in-network logic that preserves request boundaries and understands application-level semantics. While recent INC transports and frameworks [8], [9], [25] have begun providing the necessary architectural and protocol support for message-level processing, they still largely rely on host-controlled co-designs where the CPU manages scheduling and data movement. This host-side control reintroduces communication overhead and scheduling jitter that INC aims to avoid, motivating *hostless* L7 acceleration where both control and data movement remain on the network-attached device.

Supporting L7 request semantics and service graphs in the network requires (i) efficient request-boundary reconstruction, (ii) compute-rich operator execution, and (iii) low-overhead multi-stage chaining, all while maintaining predictable latency under high concurrency. This places pressure on P4-based switches whose compute capability is limited, as well as CPU-based network device implementations, whose per-request service time and queueing behavior can vary significantly as load and core count increase.

In contrast, FPGA pipelines provide the necessary compute density for L7 logic while maintaining predictable, deterministic service times. This substrate enables the *hostless* execution and deterministic chaining required to bypass both the scheduling jitter of CPUs and the processing constraints of P4 switches.

Fig. 1 summarizes this evolution, contrasting our hostless L7 approach, where request processing occurs entirely on the network-attached FPGA.

B. Case Study: DLRM Data Pre-processing

Motivation for INC: DLRMs underpin ranking and personalization in many large-scale services [26]. At production scale, the input pipeline, typically implemented as a Data Pre-Processing Service (DPP), must transform massive datasets

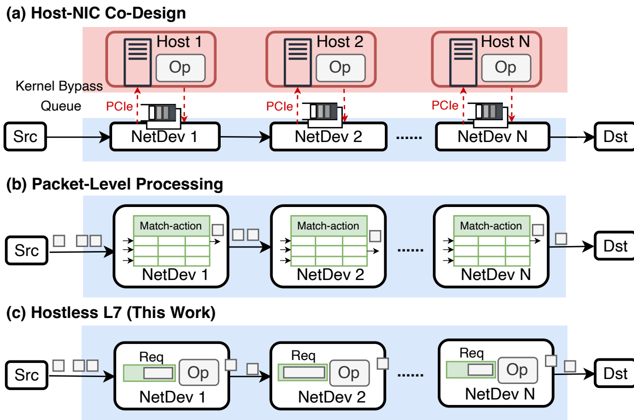

Fig. 1: Comparison of networked function pipeline processing. Blue represents the network; 'Op' denotes operators; 'NetDev' denotes network device. (a) Host-NIC co-design uses kernel-bypass to offload processing to host CPUs. (b) Packet-level processing handles packets individually through match-action tables. (c) Our hostless L7 approach assembles packets into requests within the device, processing them on-chip with header-determined cross-device chaining.

at high throughput, and is commonly distributed across tens to hundreds of CPU nodes per training job [2]. This bottleneck can dominate provisioning at scale. Meta reports that DPP requires dozens of CPU servers per GPU server, and Google reports thousands of preprocessing workers for a single model [16]. The resulting cost, power, and data-movement overhead motivate offloading preprocessing into the network to relieve host CPUs and improve performance.

Granularity and feasibility of L7 in-network offload: To offload preprocessing into the network, we first identify the *minimum request-level granularity* at which each operator can run (i.e., the smallest unit of data that can be processed without accessing beyond the request boundary). Based on the operators described in [2], we classify preprocessing functions into two granularities: *Per-row* operators are applied independently to each row, and *Per-tensor* operators operate on (or depend on statistics across) a tensor/batch of rows. Table I summarizes this classification.

Given this granularity, we next ask *whether the corresponding request sizes are practical to offload*. For the commonly used Criteo dataset [27], each sample contains 13 dense features and 26 sparse features, resulting in $\sim$150 B per row. At the per-tensor granularity, a mini-batch of thousands of rows is therefore around hundreds of KB, which remains well within the capacity of FPGA on-chip memory (The Alveo U280 [28] we use has over 40MB of on-chip memory). Even for the larger synthetic datasets used in [2] (with $\sim$5 KB per row), offloading per-row operators remains feasible.

In summary, the predictable granularity and manageable request sizes of DLRM preprocessing make it an ideal workload for request-level in-network acceleration on FPGAs.

Limitations of current approaches: Recent work has explored offloading DLRM pre-processing to network-attached devices, but existing solutions remain task-specific rather than providing a general framework for L7 in-network acceleration. PreSto [29] offloads pre-processing at storage nodes. However, it hard-couples storage nodes to specific FPGA resources, limiting flexibility under dynamic traffic and reducing overall accelerator utilization. Piper [30] offloads preprocessing to a SmartNIC/FPGA, but relies on explicit data movement (e.g., PCIe DMA and RDMA) to pull training data from host or remote memory; its reported throughput drops to less than 8 Gbps for small (KB-scale) transfers, which is ill-suited to fine-grained, high-throughput streaming. Our work targets a general L7 in-network acceleration framework that delivers predictable scaling and fairness and supports both in-FPGA and cross-FPGA chaining, in contrast to PreSto and Piper, which are single-device point solutions.

TABLE I: Granularity of representative DLRM preprocessing operators (subset from [2]).

Operations	Description	Granularity
Cartesian	Cartesian product of sparse features	Per-row
Bucketize	Bucketize dense values by boundaries	Per-row
IdListTransform	Intersection of two sparse feature lists	Per-row
GetLocalHour	Extract local hour from timestamps	Per-row
NGram	Build n-grams from sparse feature lists	Per-row
MapId	Map feature IDs to fixed values	Per-row
Onehot	One-hot encode using vocab/distribution	Per-row
Logit	Logit transform for normalization	Per-row
FirstX	Truncate sparse list to first X (based on global stats) elements	Per-tensor
BoxCox	Box-Cox transform for normalization	Per-tensor
SigridHash	Hash value to normalize sparse lists	Per-tensor

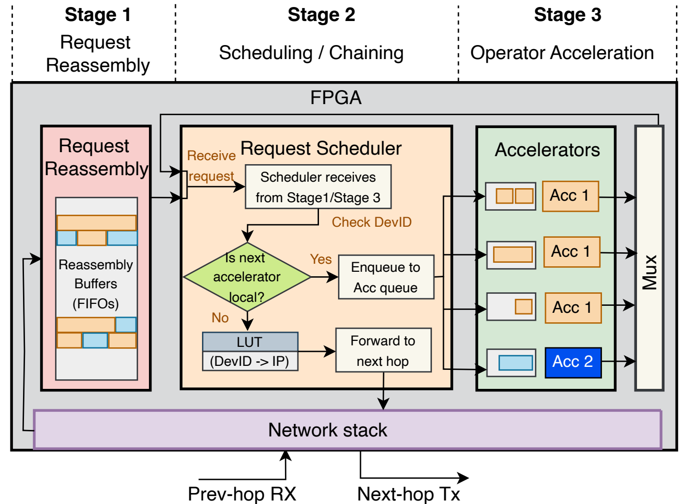

Fig. 2: Architecture of the hostless L7 acceleration framework. Adopting the request reassembly (Stage 1) from the underlying platform, we implement the request scheduler (Stage 2) for hostless chaining and accelerator execution (Stage 3) enabling our provisioning model.

III. Design and Approach

We structure our design to address the four limitations highlighted in Section I. First, **preserving request boundaries** is provided by the underlying hostless platform we build

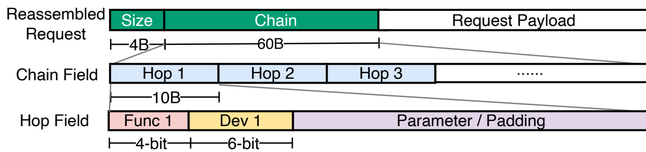

Fig. 3: Request format for function execution pipeline showing a concatenation of function identifiers to be read by each node in the pipeline.

upon, which reconstructs complete requests on the FPGA and exposes operators to a request-level stream. Second, **low-overhead chaining** is achieved by making routing decisions in hardware using a compact request header field, supporting both in-FPGA operator chaining and multi-hop inter-FPGA chaining. Third, **predictable scaling** is addressed through a deterministic provisioning model that converts a target throughput into the minimum required accelerator instance parallelism. Finally, **performance isolation** is evaluated by mapping representative DLRM preprocessing operators onto these mechanisms to exercise both small, single-packet requests and large, multi-packet mini-batches under multi-client concurrency.

A. Decoupled Request-level Acceleration

Our design is shown in Fig. 2, where the FPGA directly ingests network traffic and executes application operators without host involvement. The network stack is implemented entirely in hardware, guaranteeing in-order packet delivery with no host involvement. The abstraction decouples *request reassembly* from *operator execution*, so the request scheduler and accelerators deal with request-level streams while the framework can multiplex many concurrent clients and functions efficiently.

Stage 1: Request Reassembly: Each request carries a small application header at the beginning of its first packet (Fig. 3 *Reassembled Request*), which encodes the total request size, the target function chain, and associated metadata. Using this header, the FPGA aggregates packets into a complete request before releasing it downstream. Single-packet requests take a fast path, while multi-packet requests are buffered until all fragments arrive.

Stage 2: Scheduling and Chaining: The request scheduler receives reassembled requests (Stage 1) and accelerator outputs (Stage 3), and selects the next action. If the next hop targets the local device, it enqueues the request into the corresponding accelerator queue; otherwise, it forwards the request to the network stack toward the specified destination. We describe the header-based chaining decisions in Section III-B.

Stage 3: Operator Acceleration: Operations are executed on accelerators, which can be provisioned (scaling number of instances) based on traffic demand. Performance scaling is discussed in Section III-C. Each accelerator processes a request as an atomic unit at the request granularity, avoiding intra-request context switches and simplifying state management. Because Stage 1 delivers complete requests, accelerator queues feed requests continuously, without stalling, improving utilization and avoiding pipeline bubbles.

In summary, the FPGA reconstructs requests from the application header, buffers multi-packet requests in on-chip FIFOs, and dispatches complete requests through queued scheduling to keep the accelerator pipelines busy.

B. Chaining Hostless Accelerators

Fig. 3 illustrates how the request header implements function chaining. Each request begins with a fixed 64B header containing a 4B *Size* field and a 60B *Chain* field. The *Chain* field encodes up to 6 hops using 10B *Hop* entries. Each hop contains a 4-bit function ID, a 6-bit device ID, and a parameter region consumed by the target accelerator. We reserve *Func*=`0xF` to indicate the end of the chain in this prototype. To support up to 64 devices, each FPGA maintains a Lookup Table encoding which 6-bit device ID maps to which physical IP address. Assuming source routing, each client constructs the *Chain* vector using a pre-computed chain descriptor (e.g., provided by a control-plane service) that encodes a logical pipeline into the corresponding sequence of (*Func*, *Dev*, *Parameter*) hop words. Automating this encoding under dynamic topology changes is orthogonal to our datapath mechanism and is left to future work. Runtime chaining decisions are made by the request scheduler by decoding the head *Hop* in *Chain*. When a request arrives from Stage 1, the scheduler reads the first 10B hop word and checks whether the encoded device ID matches the current device's ID. When a request returns from Stage 3 after completing one operator, the scheduler shifts *Chain* left by 10B to expose the next hop, and repeats the same check until it encounters the end-of-chain marker. If the device ID matches the local device, the scheduler enqueues the request into the corresponding accelerator queue specified by *Func*; otherwise, it translates *Dev* to a destination IP address using the Lookup Table and forwards the request to the network stack for the specified target. *Parameter* is used to pass any required function parameters to the accelerator for each invocation. For example, in Fig. 3, the first word might trigger a local dispatch to accelerator 1. After execution, the scheduler shifts *Chain* left by 10B to dispatch accelerator 2, and continues until it reaches the end-of-chain marker. Mid-chain node failures are handled by relying on the underlying TCP timeouts or RST packets, which propagate back to the originating client as a request failure. Application-level retries are kept out of the fast path, delegated instead to the client or orchestration layer.

C. Scaling on Demand

FPGA-based accelerators exhibit a distinct *break point* in their latency-throughput profile. Because operator execution is deterministic and deeply pipelined, the service time per request for a given request size s is known a priori. As a result, end-to-end latency remains nearly constant until the offered load approaches the accelerator's aggregate service capacity, at which point queueing delay begins to dominate and latency increases sharply. In contrast, CPU-based execution suffers from

non-deterministic service times due to cache/TLB contention, OS preemption, and scheduling jitter [22], [31], leading to a more gradual latency degradation with significantly higher variance. This deterministic behavior allows us to derive a precise provisioning rule to maintain stable latency and satisfy performance isolation under a target throughput Θ.

Breakpoint throughput. Let s be the request size (bits), $T_{\text{acc}}(s)$ the compute time per request, and T_{oh} the per-request framework overhead (e.g., parsing, scheduling). With N identical accelerator instances, each contributes throughput $s/(T_{\text{oh}}+T_{\text{acc}}(s))$, so the breakpoint throughput scales linearly:

$$\Theta_{\text{break}}(s,N) = N \cdot \frac{s}{T_{\text{oh}} + T_{\text{acc}}(s)}.$$

Provisioning. For a target throughput Θ, the minimum number of accelerator instances N required is:

$$N = \left\lceil \frac{\Theta \cdot (T_{\text{oh}} + T_{\text{acc}}(s))}{s} \right\rceil. \quad (1)$$

D. Mapping DLRM Preprocessing to Framework

DLRM preprocessing is a representative L7 INC workload because its operators have well-defined request granularity (Table I) and naturally exercise both small and large requests.

The framework uses the request size to accommodate these application granularities. *Per-row* operators run on individual samples and, for common DLRM datasets (e.g., Criteo [27]), the requests typically fit within a single large packet, so they take the single-fragment fast path. *Per-tensor* operators process mini-batches, so requests span multiple packets and are reassembled before execution. For larger synthetic datasets (up to $\sim$5 KB per row), even per-row requests may exceed a single packet, but can still be serviced by our framework due to the request reassembly.

IV. Implementation

A. Network Stack and Abstraction Layer

We implement our FPGA prototype on an AMD/Xilinx Alveo U280 card [28]. Existing FPGA SmartNIC frameworks like PANIC [13] and SuperNIC [15] only process at the packet-level, and rely on a host for coordination. Meanwhile there are TCP/IP stack implementations on FPGA, but designed for host NIC functionality [32]. Hence, we choose to build on top of a hostless FPGA-based platform [17], that builds hostless accelerator functionality on top of a TCP/IP stack, with request reassembly. However, it natively only supports a single RPC-style client-server invocation. Hence, we extend the framework's abstraction layer with a request scheduler and mechanism for opening connections to enable multi-hop INC and operator chaining across the network (Fig. 2).

B. DLRM Pre-processing Operators

We implement four DLRM pre-processing operators, in Verilog, designed to cover the diverse execution patterns and data granularities (Per-row vs. Per-tensor) found in production pipelines. Crucially, these operators are selected to be composable, allowing us to evaluate both standalone performance and multi-stage operator chaining. Specifically: (i) *MapId* and *Sparse Tensor Transform* represent sparse, memory-bound processing that can be chained to stress hostless communication and routing; (ii) *Logit Transform* provides a representative per-row dense arithmetic workload; and (iii) *Normalization* represents per-tensor transforms that exercise the framework's buffering and scheduling logic for large, multi-packet mini-batches.

MapId maps 32-bit hashed values to contiguous 32-bit IDs, utilizing a pre-defined dictionary that stores unique integer indices for each hash value. Using the Criteo dataset [27], we filter hash keys appearing more than 1000 times to bound dictionary size, resulting in a 128 KB table, exploiting the dual-port nature of on-chip memory to perform two dictionary lookups in parallel per cycle. In contrast, a CPU implementation is typically restricted to serializing such irregular memory accesses.

Sparse Transform converts a raw feature vector into a compact sparse stream by filtering out zero-valued entries. It transforms the data into Compressed Sparse Row (CSR) format, a standard representation for sparse matrices. The operator scans the incoming features in a fixed order and emits non-zero values along with their original indices. For example, a dense row $[0, 4.2, 0, 0, 7.1]$ is compressed into the value stream $\{4.2, 7.1\}$ and index stream $\{1, 4\}$, while the offset array is updated to reflect the new cumulative count. This enables downstream stages to process only non-zero data, significantly reducing memory bandwidth and computation.

Logit Transform represents per-row dense transforms, applying an element-wise logit function to 32-bit floating-point inputs, producing 32-bit floating-point outputs. For each element x, it computes $\log(\frac{x}{1-x})$ using a sequence of pipelined floating-point operations (subtraction, division, and logarithm), yielding a non-linear feature transform commonly used for probability-like inputs.

Normalization represents per-tensor dense processing, performing min-max normalization on a tensor of 32-bit floating-point elements, producing a 32-bit floating-point output tensor of the same shape. It first processes the tensor to find $\min$ and $\max$, computes the range $(\max - \min)$, and then transforms each element to $\frac{x-\min}{\max-\min}$ using floating-point subtraction and division.

C. Resource Utilization

Table II reports resource usage for the underlying TCP/IP stack, our modified abstraction layer (Fig. 2), and one instance of each of the four DLRM preprocessing operators integrated as accelerators. It also includes the execution time (T_{acc}) of each operator, which will be used in our subsequent scaling and provisioning analysis (Section III-C). Overall, the abstraction layer and application operators consume only a modest fraction of the FPGA resources, leaving room to scale accelerator instances and support higher concurrency.

D. CPU Baseline using DPDK

We build our CPU baseline on top of Libtpa [33], a DPDK-based TCP stack. We extend it from a transport microbenchmark into a compute-serving endpoint by implementing the

TABLE II: FPGA resources consumed on the Alveo U280. T_{acc} is the per-operator execution time for a 1 KB request at 250 MHz, obtained from cycle-accurate simulation.

Component	LUT	FF	BRAM	URAM	DSP	T_{acc} (μs)
Infrastructure						
TCP/IP Stack	150.5k (11.5%)	130.2k (11.9%)	200.5 (10.0%)	16 (1.7%)	0 (0%)	–
Extended Abstraction	10.3k (0.8%)	20.8k (0.8%)	309 (15.3%)	54 (5.6%)	0 (0%)	–
Application Operators						
MapId	0.97k	0.98k	58	0	0	0.692
Sparse Tensor Trans.	0.91k	2.04k	42	0	0	1.240
Logit Transform	2.13k	4.11k	21	0	6	1.092
Normalization	3.02k	4.83k	25	0	4	4.092

same four operators and integrating them into the datapath, so the software baseline matches our FPGA "network + L7 abstraction + compute" service model. At the stack level, we make two key changes. First, we add support for fragmented requests, ensuring that requests spanning multiple packets are reassembled before computation, mirroring the FPGA L7 abstraction design. Second, we implement a connection opening mechanism to emulate an in-network device and enable multi-machine chaining. The server can initiate separate TCP connections to return processed results instead of always responding on the original request connection. While this would not be a typical approach for implementing INC, or the type of microservices/serverless pipeline that is sometimes used, we believe this is a more challenging high-performance baseline to compare against.

V. Evaluation

A. Testbed and Setup

Fig. 4 illustrates our testbed setup. The testbed comprises a traffic generator (Section V-B) and two inline processing nodes connected to an EdgeCore DCS810 switch via 100 Gbps links with an MTU of 9000 B. The traffic generator runs on a server equipped with an AMD EPYC 7763 64-core processor, 512 GB DDR4 RAM, and an NVIDIA ConnectX-6 NIC, running Ubuntu 20.04. Each inline processing node is instantiated in one of two configurations: (i) **FPGA-based INC node**: Two AMD/Xilinx Alveo U280 FPGAs running our design. (ii) **CPU baseline node**: Two servers identical to the traffic generator, running the DPDK-based operator processing pipeline in the same inline position to mimic the INC deployment role in software, providing a more challenging baseline than a lightweight network appliance. We evaluate two topological setups. (i) **One-hop:** A single inline processing node sits between the client and the endpoint (①→②→③→⑥ in Fig. 4). (ii) **Two-hop:** Two inline processing nodes are chained in the network (①→②→③→④→⑤→⑥). All experiments use closed-loop clients where each client sends the next request only after receiving the response to the previous request.

B. Traffic Generator

The traffic generator functions as both the client and the endpoint server. To efficiently manage bidirectional traffic on

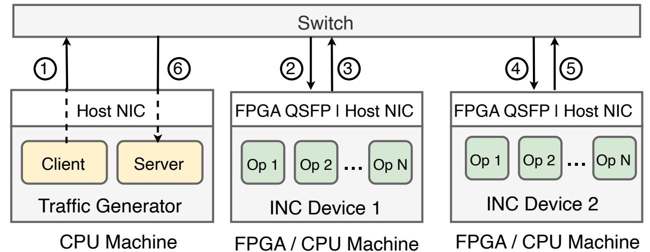

Fig. 4: Testbed setup; 'Op' denotes DLRM preprocessing operators (Section IV-B) deployed on each machine.

a single interface, physical NIC queues are partitioned, allocating half for DPDK-based sending (client) and the other half for receiving (server). Each traffic generator client is pinned to a hardware thread within the NIC-attached NUMA node. This co-location allows end-to-end latency to be measured on a single machine with synchronized timestamps. Utilizing Libtpa [33], we make three key modifications. First, we elevate Libtpa from a packet-level transport stack into an L7 RPC generator that repeatedly issues single-fragment or multi-fragment requests and measures end-to-end response completion latency at high rates across many concurrent threads. Second, to model disaggregated INC execution, each client thread maintains separate sockets for requests and responses, allowing replies to arrive on different TCP connections than their corresponding requests. Third, we pre-allocate huge pages and add per-thread logging hooks to reliably capture raw latency data under multi-threaded load for latency distribution analysis.

C. Experimental Results

We evaluate our approach by answering the following questions: *1. What baseline communication overhead does the framework introduce? 2. How do in-FPGA and cross-FPGA operator chaining behave in terms of latency and throughput, compared to CPU? 3. Can the proposed scaling model in Section III-C accurately predict performance and determine the minimum accelerator parallelism needed to meet a throughput target? 4. Does the system maintain performance isolation under high concurrency and mixed-size request traffic?*

Communication overhead baseline: We quantify the intrinsic communication overhead of FPGA pipeline chaining compared to a DPDK-based software baseline for one-hop and two-hop configurations using a pure forwarding pipeline with no operator execution. For the CPU baseline, we vary the number of cores, while for the FPGA we measure the underlying TCP stack [32] without our abstraction to isolate framework overhead. The *Traffic Generator* sends 1KB requests, matching the settings used in subsequent experiments so that overhead is measured under identical conditions. The FPGA framework can sustain more than 85 Gbps, but we intentionally avoid peak throughput since in realistic workloads the bottleneck is operator execution, consistent with later experiments. Fig. 5 shows that for the CPU, increasing the number of allocated cores raises saturation throughput, but low-load baseline latency remains around 10 μs for one hop and 15 μs for two

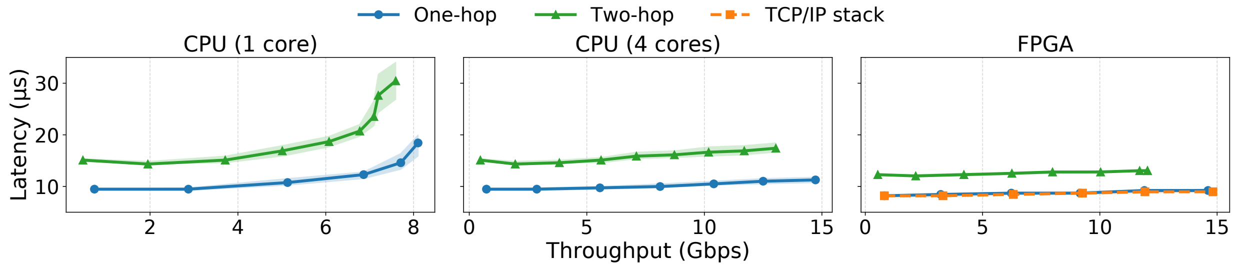

Fig. 5: Baseline communication overhead for single function and two function pipelines; the shaded regions indicate 25th–75th percentile latency range.

hops. In contrast, the FPGA design lowers latencies to 9 μs and 12 μs, resulting in a smaller per-hop increment of 3 μs versus 5 μs on the CPU, while our abstraction layer introduces at most 0.26 μs additional latency over the raw TCP stack. The FPGA framework thus adds sub-microsecond abstraction cost and has small and predictable per-hop overhead.

Function chaining behavior: We evaluate the chaining of *MapId* and *Sparse Transform* in both one-hop (in-node chaining) and two-hop (cross-node chaining) configurations. In the one-hop setup, we test three scenarios on a single device: executing *MapId* alone, executing *Sparse Transform* alone, and executing both as a pipeline on the same FPGA (or CPU) node. In the two-hop configuration, we split the functions across two devices in a cross-network pipeline. For the CPU baseline, we compare the one-hop, in-machine pipeline with both single-core and 2-core execution. The client generates 1KB closed loop requests, utilizing the header format shown in Fig. 3. Fig. 6 illustrates several key performance differences. First, the narrow P95 latency bands for the FPGA demonstrate highly predictable execution behavior, in stark contrast to the wide variance and long tail latencies observed on the CPU. Second, while the CPU implementation with a single core exhibits an end-to-end latency nearly equal to the sum of standalone execution times, the in-FPGA pipeline achieves close to the same latency as the slower of the two operators by overlapping their execution in a streaming pipeline. Furthermore, comparing the 2-core CPU in-machine pipeline against the cross-machine deployment shows that, even with the same total CPU core budget, the cross-machine setting reaches saturation at a substantially lower throughput. In contrast, FPGA cross-network chaining maintains the same saturation point as the in-FPGA case, with a stable, throughput-independent latency increase of approximately 3 μs before reaching saturation. This confirms that in-FPGA streaming masks latency via overlapped execution, while inter-FPGA chaining enables multi-device scaling with minimal latency overhead and no throughput loss.

Performance scaling: To validate the scaling model in Section III-C, we deploy 1–4 accelerators/cores for each of the four operators on both the FPGA platform and the CPU baseline, using 1KB requests to make the throughput saturation point clearly visible, while sweeping the number of concurrent clients from 1 to 32 to generate different request throughputs. Fig. 7 shows that the FPGA exhibits a clear breakpoint

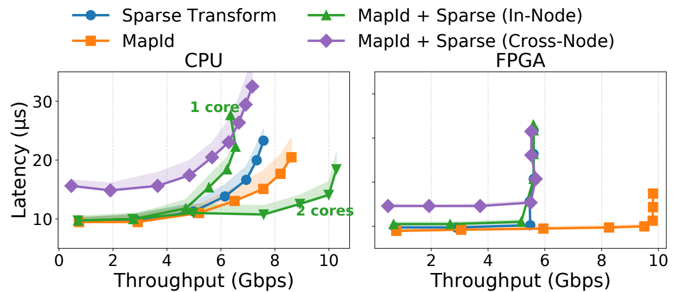

Fig. 6: Latency comparison between FPGA and CPU for single function and two function pipelines. The shaded regions indicate the P95 latency.

TABLE III: Breakpoint Throughput (Gbps) and Provisioning Accuracy. P: Predicted, E: Experimental. Missing entries are due to traffic generator limitations; using larger requests or additional generator machines would allow us to reach higher load and observe more breakpoints.

Operator	**N=1**		**N=2**		**N=3**		**N=4**	
	P	E	P	E	P	E	P	E
MapId	7.5	9.5	15.0	16.5	22.5	–	30.0	–
Sparse Trans.	5.0	5.5	10.0	10.3	15.0	14.9	20.0	–
Logit	5.5	5.2	11.0	11.1	16.5	14.1	22.0	–
Normalization	1.8	1.6	3.6	3.1	5.5	4.7	7.3	6.2

where latency rises sharply, while the CPU shows a more gradual latency increase. Adding FPGA instances shifts the breakpoints to higher throughputs, while allocating more CPU cores affects how quickly latency deteriorates under load. *Sparse Transform* and *Normalization* as individual operators perform well on CPU as they involve repeated accesses to their input data to produce their output and this fits in CPU caches, while they do not benefit as much from the streaming pipeline advantages of FPGAs.

We compare these observed breakpoints with theoretical predictions from $\Theta_{\text{break}}(N) = N \cdot s/(T_{\text{oh}} + T_{\text{acc}})$ proposed in Section III-C. We set a conservative $T_{\text{oh}} = 0.4$ μs to cover header parsing, scheduler enqueue/dequeue, and measurement overhead, using T_{acc} from Table II. Across all four operators, the model matches the observed breakpoints within $1 - 17\%$ relative error (Table III). *MapId* shows slightly larger deviation because its short execution time makes overhead a larger and

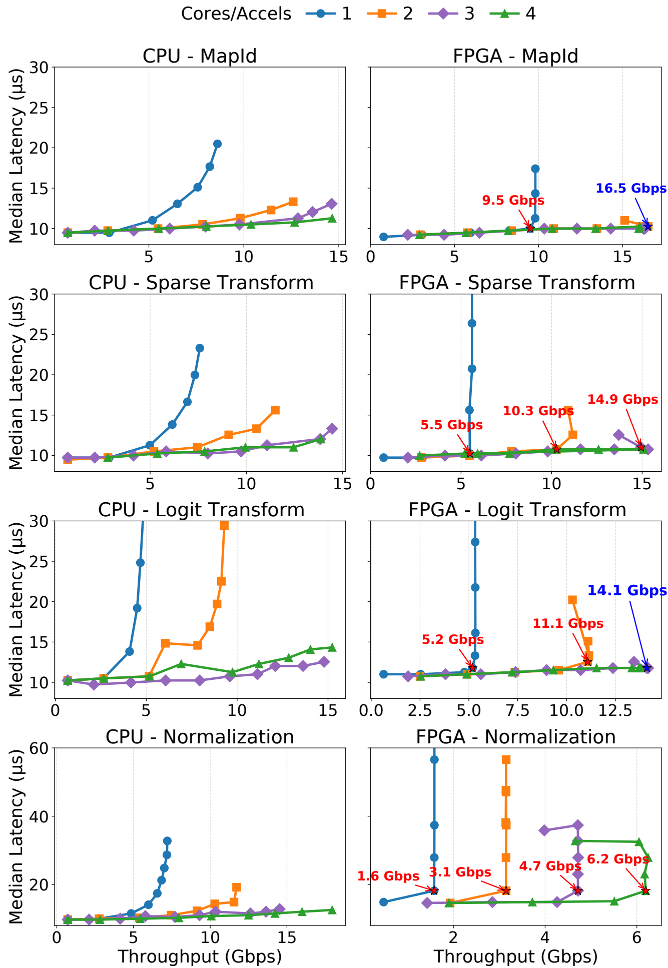

Fig. 7: Median latency vs. throughput when scaling the number of cores (CPU) or accelerators (FPGA). FPGA latency breakpoints are determined as the onset of a $>$10% latency increase (red) or the point of maximum throughput before saturation (blue).

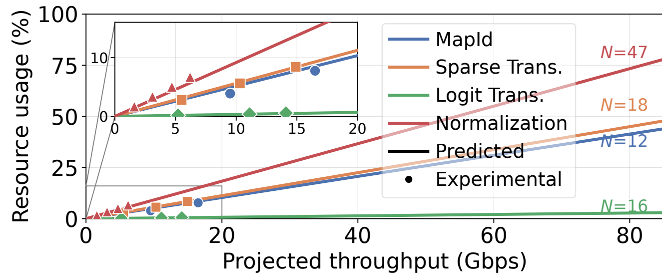

Fig. 8: Projected throughput vs. resource utilization when scaling accelerator instances, with experimental validation. Solid lines show predictions; markers denote measured configurations. N is the number of accelerator instances required to achieve the maximum throughput supported by the underlying platform (85Gbps).

more variable fraction of end-to-end latency. The consistent rightward shift with increasing N supports our claim that provisioning can reliably translate a throughput target into the required accelerator parallelism.

Figure 8 projects throughput scaling, plotting achievable throughput against FPGA resource utilisation as we add accelerator instances. Resource usage is computed from each operator's critical resource on the Alveo U280 FPGA: *MapId*, *Sparse Transform*, and *Normalization* are BRAM-bound; *Logit Transform* is LUT-bound. The figure indicates that all four operators could be replicated to reach the theoretical maximum 100 Gbps well within FPGA resource constraints. Though we clarify that some minor modifications of the framework would be needed to host larger numbers of accelerators.

Performance isolation (high concurrency): We evaluate performance isolation by running the same four operators on our FPGA framework and a CPU baseline with 24 closed-loop clients issuing 1KB requests, recording per-client latency traces. We use 24 clients as a representative high-concurrency point; with 1 accelerator/core the system is near saturation, while with 4 accelerators/cores it remains in the stable-latency region. Fig. 9 shows cross-client latency variability. Across all operators and resource budgets, the FPGA shows consistently tighter latency distributions than the CPU, indicating more uniform per-client latency under contention. The CPU is able to achieve a lower median for *Normalization*, which offers limited streaming pipeline benefits on the FPGA, even though the FPGA still exhibits smaller cross-client variance.

Fig. 10 extends this analysis to a two-operator pipeline with four accelerators/cores (to stay within the stable region) by plotting each client's P5, P50, and P95 latencies. For in-device chaining, both platforms inherit end-to-end jitter from the noisier standalone stage. For cross-network chaining, the FPGA continues to track the higher-jitter stage, whereas the CPU baseline shows a noticeably larger jitter increase, suggesting weaker fairness when the pipeline spans machines. This demonstrates that FPGA-based operators are individually predictable in terms of latency and this extends to chained accelerators, even across the network.

Performance isolation (mixed-sized requests): Such a framework accepts mixed size requests to support the different functions discussed in Section IV-B. We deploy a single instance of each operator on the FPGA and construct a mixed workload with one large request flow: per-tensor *Normalization* with 4KB requests, and three short request flows: *MapId*, *Sparse Transform*, and *Logit Transform* with 1KB requests. We implement one closed-loop client per short flow and increase the number of closed-loop clients for the long flow from 1 to 4 to raise its offered load. Fig. 11 shows that the short-flow latencies remain essentially unchanged as the long-flow load increases, indicating that the framework preserves performance isolation under mixed-size request traffic.

VI. Discussion

Our results suggest the following broader implications for FPGA-accelerated in-network computing.

Enabling Fine-Grained Application Disaggregation: Hostless L7 chaining effectively eliminates the "communication

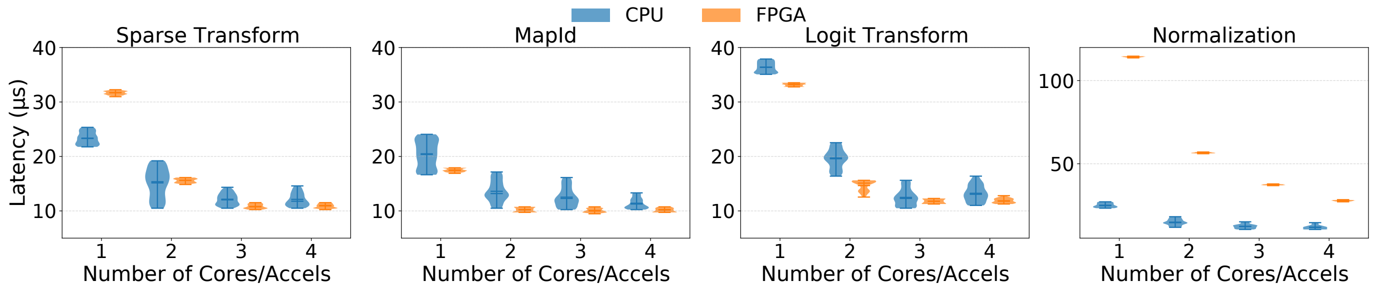

Fig. 9: Latency distribution across clients under high concurrency (24 concurrent clients).

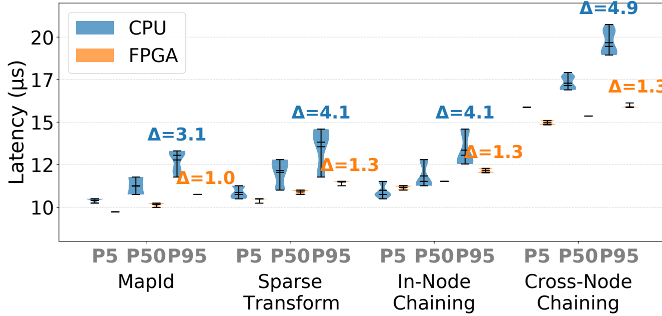

Fig. 10: Per-client latency distribution for chained and standalone operators under high concurrency with 4 accelerators/cores; Δ denotes P95−P5 cross-client jitter.

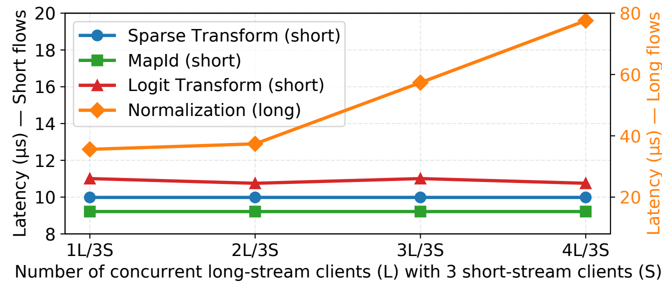

Fig. 11: Performance isolation under mixed-size request traffic.

tax” of distributed applications by maintaining throughput saturation across devices with minimal latency overhead. This is a key enabler for microservice/serverless pipelines (e.g., data sanitization → inference → encryption) which currently suffer from significant RPC stack overheads in software.

Hardware Accelerator Suitability: Overall operator speedup depends on how much it can exploit pipeline overlap within a request. Streaming-friendly accelerators benefit most, while those based on repeated accesses to chunks of larger memory perform better on powerful CPUs. That being said, the determinism and composition benefits of FPGAs still hold.

Provisioning efficiency: Deterministic scaling behavior allows “right-sized” provisioning without the large safety margins required to absorb software jitter. In real world deployment, an INC device can allocate the precise number of accelerator instances required for a specific target throughput, minimizing wasted capacity and power consumption in the network fabric.

Strong multi-tenant isolation: Hardware-enforced scheduling ensures strong performance isolation even under high concurrency and mixed-sized requests. This is critical for co-located applications, where distinct function pipelines might share the same physical accelerators while guaranteeing distinct SLAs, ensuring robust service quality of service in shared infrastructure.

Minimal Software Modifications: Since our FPGA INC approach builds atop standard TCP/IP transport, no significant software changes are needed on the client. It simply formats the application payload with the custom header, allowing standard socket APIs to trigger accelerated functions seamlessly.

Transport layer optimization: While our prototype leverages a standard TCP/IP stack since it is open-source and widely available, the proposed abstraction is transport-agnostic. Such an abstraction over a specialized, lightweight transport layer could yield further significant latency improvements.

Cost and Energy Efficiency: Both the Alveo U280 ($10k to $15k) and our DPDK baseline server (AMD EPYC 7763, $8k to $12k) have comparable acquisition costs, but their energy profiles differ dramatically. The FPGA prototype consumes only 28–31 W under load [17]. In contrast, the CPU server baseline draws 188–250 W to provide the same inline service, representing an approximately 6–9× difference in system-level power. Furthermore, applications with lower throughput requirements can be served by a smaller, less expensive FPGA platform, reducing both acquisition cost and power consumption even further.

VII. Conclusion

We evaluated application-level function pipelines on host-less, network-attached FPGAs and showed that our on-chip header-driven chaining mechanism enables low-overhead, multi-hop composition of accelerators, reducing cross-device chaining latency by 40% compared to CPU baselines with predictable per-hop increments of only $3\,\mu s$ and deterministic throughput scaling. Using deep learning recommendation model training preprocessing operators, we demonstrated a simple provisioning model that translates a throughput target into the required accelerator parallelism, closely matching observed behavior across all four operators. Under high concurrency and mixed-size request traffic, the FPGA maintains tight cross-client latency distributions, demonstrating stronger

performance isolation and fairness than software baselines. This hostless FPGA-based INC framework eliminates the communication tax of distributed applications, providing a highly efficient deployment path for disaggregated applications. Furthermore, it integrates seamlessly without requiring client-side software modifications, and achieves these performance and isolation benefits while delivering a 6–9× reduction in system-level power compared to equivalent CPU baselines.

REFERENCES

[1] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," 2019. [Online]. Available: https://arxiv.org/pdf/1906.00091

[2] M. Zhao, N. Agarwal, A. Basant, B. Gedik, S. Pan, M. Ozdal, R. Komuravelli, J. Pan, T. Bao, H. Lu, S. Narayanan, J. Langman, K. Wilfong, H. Rastogi, C.-J. Wu, C. Kozyrakis, and P. Pol, "Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product," in *International Symposium on Computer Architecture (ISCA)*, 2022, pp. 1042–1057.

[3] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 3–18.

[4] A. Sriraman and T. F. Wenisch, "μSuite: A benchmark suite for microservices," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 1–12.

[5] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in *International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[6] C. Bobda, J. M. Mbongue, P. Chow, M. Ewais, N. Tarafdar, J. C. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser, M. Herbordt, H. Shahzad, P. Hofste, B. Ringlein, J. Szefer, A. Sanaullah, and R. Tessier, "The future of FPGA acceleration in datacenters and the cloud," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 3, pp. 1–42, 2022.

[7] R. A. Cooke and S. A. Fahmy, "Characterizing latency overheads in the deployment of FPGA accelerators," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 347–352.

[8] X. Wan, L. Li, H. Tian, X. Liao, X. Huang, C. Zeng, Z. Wang, X. Yang, K. Cheng, Q. Ning, G. Liu, L. Luo, and K. Chen, "A generic and efficient communication framework for message-level in-network computing," in *IEEE Conference on Computer Communications (INFOCOM)*, 2025, pp. 1–10.

[9] T. Ji, R. Vardekar, B. Vamanan, B. E. Stephens, and A. Akella, "MTP: Transport for in-network computing," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2025, pp. 959–977.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, 2014.

[11] NVIDIA Corporation, "NVIDIA BlueField-3 DPU," 2025. [Online]. Available: https://www.nvidia.com/en-us/networking/products/data-processing-unit/

[12] S. A. Fahmy, Z. Yang, Y. Chen, G. Alonso, Z. István, and M. Canini, "FPGAs are the hero in-network computing needs," in *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2025, pp. 131–139.

[13] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, "PANIC: A high-performance programmable NIC for multi-tenant networks," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 243–259.

[14] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "ClickNP: Highly flexible and high performance network processing with reconfigurable hardware," in *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2016, pp. 1–14.

[15] W. Lin, Y. Shan, R. Kosta, A. Krishnamurthy, and Y. Zhang, "SuperNIC: An FPGA-based, cloud-oriented SmartNIC," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2024.

[16] M. Zhao, E. Adamiak, and C. Kozyrakis, "cedar: Optimized and unified machine learning input data pipelines," *Proceedings of the VLDB Endowment*, 2024.

[17] Z. Yang, K. B. Iyer, Y. Chen, R. Shu, Z. István, M. Canini, and S. A. Fahmy, "OffRAC: Offloading through remote accelerator calls," 2025. [Online]. Available: https://arxiv.org/pdf/2504.04404

[18] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, 2013.

[19] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[20] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[21] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[22] M. Faltelli, G. Belocchi, F. Quaglia, S. Pontarelli, and G. Bianchi, "Metronome: Adaptive and precise intermittent packet retrieval in DPDK," in *International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2020.

[23] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing key-value stores with fast in-network caching," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[24] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling distributed machine learning with in-network aggregation," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[25] T. Wang, J. Lin, G. Antichi, A. Panda, and A. Sivaraman, "Application-defined receive side dispatching on the NIC," 2024. [Online]. Available: https://arxiv.org/pdf/2312.04857

[26] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of Facebook's DNN-based personalized recommendation," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[27] Criteo AI Lab, "Criteo research datasets," 2025. [Online]. Available: https://ailab.criteo.com/ressources/

[28] "Alveo U280 data center accelerator card data sheet (DS963)," 2026. [Online]. Available: https://docs.amd.com/r/en-US/ds963-u280

[29] Y. Lee, H. Kim, and M. Rhu, "PreSto: An in-storage data preprocessing system for training recommendation models," in *International Symposium on Computer Architecture (ISCA)*, 2024, pp. 340–353.

[30] Y. Zhu, W. Jiang, and G. Alonso, "Multi-tenant SmartNICs for in-network preprocessing of recommender systems," 2025. [Online]. Available: https://arxiv.org/pdf/2501.12032

[31] M. Faltelli, G. Belocchi, F. Quaglia, and G. Bianchi, "COREC: Concurrent non-blocking single-queue receive driver for low latency networking," *The International Journal of Computer and Telecommunications Networking*, 2025.

[32] Z. He, D. Korolija, and G. Alonso, "EasyNet: 100 Gbps network for HLS," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2021.

[33] ByteDance, "Libtpa: A DPDK-based userspace TCP stack," 2026. [Online]. Available: https://github.com/bytedance/libtpa