Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers

Maroun Tork
Technion – Israel Institute of Technology
Haifa, Israel

Lina Maudlej
Technion – Israel Institute of Technology
Haifa, Israel

Mark Silberstein
Technion – Israel Institute of Technology
Haifa, Israel

Abstract

This paper explores new opportunities afforded by the growing deployment of compute and I/O accelerators to improve the performance and efficiency of *hardware-accelerated computing services* in data centers.

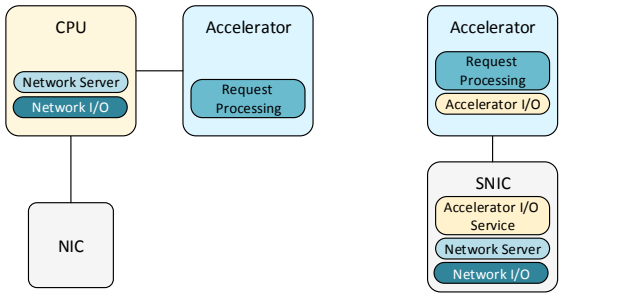
We propose *Lynx*, an accelerator-centric network server architecture that offloads the server data and control planes to the SmartNIC, and enables direct networking from accelerators via a lightweight hardware-friendly I/O mechanism. Lynx enables the design of hardware-accelerated network servers that run without CPU involvement, freeing CPU cores and improving performance isolation for accelerated services. It is portable across accelerator architectures and allows the management of both local and remote accelerators, seamlessly scaling beyond a single physical machine.

We implement and evaluate Lynx on GPUs and the Intel Visual Compute Accelerator, as well as two SmartNIC architectures – one with an FPGA, and another with an 8-core ARM processor. Compared to a traditional host-centric approach, Lynx achieves over 4× higher throughput for a GPU-centric face verification server, where it is used for GPU communications with an external database, and 25% higher throughput for a GPU-accelerated neural network inference service. For this workload, we show that a single SmartNIC may drive 4 local and 8 remote GPUs while achieving linear performance scaling without using the host CPU.

ACM Reference Format:
Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378528

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378528



(a) Traditional host-centric **(b)** Lynx: Accelerator-centric

Figure 1. Accelerated network server architectures.

1 Introduction

Modern data centers are increasingly heterogeneous, with a variety of compute accelerators deployed to accommodate growing performance demands. Many cloud vendors leverage them to build hardware-accelerated network-attached computing services. For example, Amazon offers Elastic Inference [4] on top of GPUs, Microsoft Azure exposes FPGA-accelerated ML services [33], and Google runs AutoML service on TPUs [15]). In such systems, accelerators run the dominant fraction of the application logic.

At the same time, growing network rates drive the adoption of programmable Smart Network Adapters to offload data center networking workloads. Microsoft has been among the first to deploy *SmartNICs (SNICs)* at scale, with the Catapult FPGA-based SNICs installed in each of over a million of Azure servers [34]. Today, SNIC adoption by cloud vendors is on the rise: they are deployed in, e.g., China Mobile [52], Tencent [28], Huawei [19], and Selectel [45].

So far, SNICs have been used primarily for accelerating low-level packet processing applications, such as network functions and software defined networking [14, 28]. However, we posit that SNICs also create new opportunities for improving the efficiency and performance of hardware-accelerated network-attached computing services.

To demonstrate this idea, we propose an accelerator-centric network server architecture we call *Lynx*[1]. Lynx *executes much of the generic server data and control planes on the SNIC,*

[1]Lynx is a wild cat. We allude to its pronunciation as *links*, i.e., *Link*ing a*X*elerators to network

thereby enabling network I/O from and to accelerators without using the host CPU for network processing, and without running the network stack on accelerators.

In a conventional hardware-accelerated server design (Figure 1a), the CPU performs two main tasks: (1) it runs the boilerplate logic, such as packet processing in the network stack, and interacts with network clients; (2) it dispatches requests to compute accelerators, taking care of the associated data transfers, accelerator invocation and synchronization. In contrast, Lynx (Figure 1b) offloads these two tasks to the SNIC, allowing the application code on accelerators to directly interact with the network, bypassing the host CPU.

The Lynx architecture provides several benefits over the traditional CPU-centric design:

Lightweight networking from accelerators. Lynx enables accelerators to communicate with other machines over the network at any point of the accelerator execution. While several earlier works have demonstrated the advantages of GPU-side networking API [8, 26, 39], they run resource-heavy GPU-side network stack, support only Remote Direct Memory Access (RDMA) as the primary protocol, and are only suitable for GPUs. In contrast, Lynx runs a lightweight API layer on the accelerator, natively supports TCP/UDP, and is deployable across accelerators of different types (§5.4).

High CPU efficiency. The host CPU is freed from the network processing and accelerator management tasks. As a result, the host may run other tasks that can better exploit the latency-optimized CPU architecture. At the same time, specialized SNIC cores, which are less efficient for general-purpose computations, are sufficient to drive hardware-accelerated network services with negligible performance cost. For example, a GPU-accelerated neural network inference service managed by an SNIC is only 0.02% slower than its CPU-driven version (§6.3), whereas the extra host CPU core is better utilized by the `memcached` key-value store, which scales linearly with additional CPU cores.

Performance isolation. Lynx achieves strong performance isolation between the SNIC-driven hardware-accelerated services and other applications running concurrently on the same machine (e.g., from other cloud tenants). For example, as we observe experimentally (§3.2), a memory intensive *noisy neighbor* application, co-executed with the hardware-accelerated network server, leads to high variations in server response latency, unacceptable in latency-sensitive soft real-time workloads, e.g., in image processing for autonomous driving (§6.3).

To achieve these benefits, the Lynx's design builds on two key ideas:

Offloading the network server logic to an SNIC. The SNIC runs a full network stack and a generic network server that listens on the application-specified ports, and delivers application messages to and from the accelerators. No application development is necessary for the SNIC.

The server dispatches the received messages to the appropriate accelerator via *message queues (mqueues)*, retrieves the responses and sends them back to clients. Accelerators run a lightweight I/O layer on top of mqueues, providing zero-copy networking. An mqueue is a user-level abstraction similar to an RDMA Queue Pair, but optimized to reduce the complexity of accessing it from the accelerator (§4).

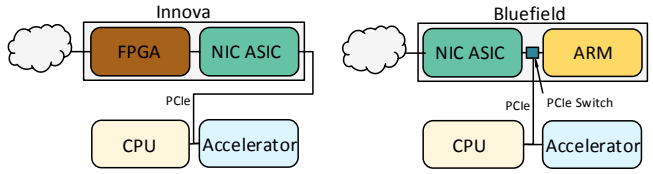
Using SNIC RDMA for portability and scalability. An accelerator stores the mqueue data and control buffers in its own memory, whereas the SNIC accesses the mqueues remotely, and executes the I/O operations on behalf of the accelerator. For remote access, the SNIC uses its internal hardware-accelerated RDMA engine to efficiently read from/write to the mqueues via one-sided RDMA. Note that RDMA is used only between the SNIC and the accelerators, transparently for the external clients which connect to the accelerated service via UDP/TCP.

This design choice ensures both Lynx portability and scalability: it allows the SNIC to support a variety of accelerators without requiring the SNIC to run an accelerator driver (§4.5). Moreover, the use of RDMA makes it possible to scale beyond a single machine, enabling Lynx to seamlessly provide I/O services to *remote* accelerators across the network (§5.5).

A key goal of Lynx is to facilitate the implementation of network servers that follow an accelerator-centric design which minimizes the host CPU involvement. This is advantageous in low-latency accelerated servers which may run entirely on the accelerator [8, 26, 39, 47, 50], but is less suitable for hybrid applications which must use the CPU. However, as we discuss in §3, host CPU involvement is often elided in accelerated applications by emerging optimization frameworks [7], making them an ideal candidate for Lynx.

We prototype Lynx on a system with multiple local and remote NVIDIA GPUs, as well as one Intel Visual Compute Accelerator (VCA) [21] which hosts three Intel E3 processors with Software Guarded Extensions (SGX) trusted execution support. Furthermore, we implement Lynx on the ARM-based Mellanox Bluefield SNIC, and also prototype on the Mellanox Innova SNIC with Xilinx FPGA.

We evaluate the system performance and scalability using microbenchmarks and realistic applications. For example, we develop a LeNet [27] model-serving server for digit recognition, implemented entirely on the GPU. Lynx on Bluefield achieves 3.5 K requests/sec at 300μsec latency, which is 25% higher throughput and 14% lower latency than an optimized host-centric server, but without using the host CPU. To demonstrate the scalability, we run a LeNet server with 12 GPUs distributed across three physical machines. Lynx achieves linear scaling, and is projected to scale up to 100 and 15 GPUs for UDP and TCP network services respectively, saving CPU cores otherwise required for network processing and GPU management.



(a) Innova - FPGA bump in the wire **(b) Bluefield - ARM multi host**

Figure 2. SNIC architectures.

Lynx generalizes the accelerator-centric server design concepts of prior works beyond GPUs, making them applicable for other accelerators. To achieve portability and efficiency, it leverages recent advances in SNIC hardware architecture. While the current design focuses on servers running on a single accelerator, Lynx will serve as a stepping stone for a general infrastructure targeting multi-accelerator systems which will enable efficient composition of accelerators and CPUs in a single application.

This paper makes the following contributions:

- We propose Lynx, a new SNIC-driven server architecture that offloads to SNICs the data and control planes in accelerated network servers.
- We design a system that enables fast network I/O from accelerators, with minimal requirements imposed on accelerator hardware and software.
- We prototype Lynx on two different SNICs, local and remote GPUs, Intel VCA, and show that SNIC-driven accelerated servers provide high performance and good scalability.

2 Background

SNICs are NICs that feature a programmable device to perform custom processing of the network traffic. In this paper we use two SNIC architectures:

Bump-in-the-wire FPGA-based NIC. Each packet passing through the NIC is processed by the FPGA logic customized by the programmer. In this paper, we use Mellanox Innova Flex SNIC (Figure 2a). The FPGA is located in front of the Mellanox ConnectX-4 NIC ASIC (relative to the network) which is used in non-programmable NICs. This architecture reuses the optimized NIC-to-host data path, and is compatible with the standard I/O software stack. For Lynx, the application logic on the FPGA includes a network server that handles incoming connections and interacts with external compute accelerators via the PCIe.

Processor-based SNIC. These SNICs vary in their choice of the processor architecture (i.e., MIPS [6], ARM [29]), internal connectivity (i.e., PCIe switch between the NIC and the CPU), and software stack (proprietary OS or embedded Linux). Figure 2b shows the architecture of the Mellanox Bluefield SNIC we use in this paper. It features eight 64-bit ARM A72 cores running at 800 MHz, connected to the NIC ASIC and to the host via an internal PCIe switch. The CPU runs BlueOS Linux, and can execute regular applications. The SNIC may work in several connectivity configurations, but in this paper we focus on a *multi-homed mode*. Here the SNIC CPU runs as a separate machine with its own network stack and IP address. In addition, the CPU can communicate with the main host via reliable RDMA connections. We leverage RDMA to access message queues in accelerator memory.

3 Motivation

We explain that the use of CPUs in accelerated network services is often reduced to generic network processing and accelerator management. However, we claim that executing these tasks on a CPU hinders system's efficiency and performance isolation, whereas the recent proposals to mitigate these issues by exposing accelerator-side I/O capabilities incur high resource overheads and have functional limitations. These factors motivate the SNIC-driven accelerator-centric server design we propose in Lynx.

3.1 Diminishing role of CPU in accelerated services

Popular cloud services, such as Google AutoML [15] or Amazon elastic inference [4], run on GPUs or other specialized hardware accelerators. Though the details of their implementations are not published, a plausible design would include a frontend running on the host CPU that dispatches client requests to accelerators, and sends the results back. Yet, it is worth considering to what extent the host CPU is involved in the request processing, beyond the boilerplate network processing logic and accelerator management?

To answer this question, we analyze the optimization techniques employed in GPU-accelerated neural network inference systems as a representative case of accelerated services in a cloud. We observe that reducing CPU involvement by building larger GPU kernels with fewer kernel invocations is among the most common optimization targets.

There are several reasons for that. First, kernel invocation involves extra overheads that start dominating the performance if the GPU kernel execution is relatively short (see §3.2). Therefore, GPU kernel fusion [7] is a common technique used to merge several kernels into a single one to reduce these overheads. Second, kernel termination causes the GPU to reset its transient state stored in registers and per-core scratchpad memory. Recent work [9] has shown that keeping the GPU kernel state in registers substantially improves system performance for computations in Recurrent Neural Networks. Last, having to interleave computations on the CPU and on the GPU requires also data transfers between them, which is costly. Therefore, certain tasks are moved to the GPU, even if they run faster on the CPU, only to reduce the data transfer overheads.

As a concrete example showing the practice of eliding CPU involvement, we develop an application to perform neural network inference for the classical LeNet [27] model using TensorFlow [1]. We then optimize it for execution on GPUs using the state-of-the-art TVM optimization compiler for neural network computations [7]. We observe that the resulting implementation does not run *any application logic* on the CPU, besides a series of GPU kernel invocation commands.

In fact, the accelerated system design which minimizes the CPU involvement is not specific to systems with GPUs. For example, the official TensorFlow-light guide that explains the generic `Delegate` API for using custom deep learning inference accelerators recommends offloading all the computations to the accelerator to avoid costly control and data transfers between them and the host CPU [48]. Similarly, the recent Goya inference accelerator from Habana Labs [17] fully offloads the computations into the accelerator engine.

We conclude that *emerging accelerated services are likely to perform the majority of request processing on accelerators, whereas the host CPU is occupied primarily by network message processing and accelerator management.* However, using the host CPU for these tasks has disadvantages as we discuss next.

3.2 Disadvantages of the host CPU-driven design

Accelerator invocation overhead. In a CPU-driven network server the CPU performs the accelerator invocation, synchronization and data movements. These tasks constitute a significant portion of the end-to-end execution for short latency sensitive kernels because they all involve interaction with the accelerator driver.

To verify, we run a simple kernel which implements an echo server on a GPU. This kernel comprises a single GPU thread which copies 4 bytes of the input into the output. Additionally we add a 100-μsecond delay inside the kernel. We run the pipeline composed of the CPU-GPU transfer, kernel invocation, and GPU-CPU transfer. We measure the end-to-end latency of 130μseconds, implying 30μseconds of the pure GPU management overhead. For short kernels, such as LeNet neural network inference (§6) of about 300μseconds, this is about 10% latency overhead per request. Moreover, recent results show that these overheads are the root cause of poor multi-GPU scaling in such workloads [50]. *Lynx strives to mimimize kernel invocation and synchronization overheads.*

Wasteful use of the CPU. Accelerator management and network I/O tasks are inherently I/O- and control-bound. Thus they do not need the full power of super-scalar, out-of-order X86 CPU architecture to achieve high performance. For example, most interactions with the accelerator require access to memory-mapped control registers over PCIe, which in turn are long-latency, blocking operations. Furthermore, polling is commonly used to achieve low latency when interacting with the accelerator, but polling can run on a simple micro-controller instead of wasting the X86 core. Indeed, as we show in our evaluation using a slower multi-core ARM processor for these tasks results in negligible end-to-end performance degradation of the accelerated service. At the same time, latency-sensitive or compute-intensive applications, i.e., from other cloud tenants, may significantly benefit from the extra host CPU cores (§6.3).

Thus, *the CPU could have been put to better use, if it were only possible to offload the network server logic and the accelerator management to a processor better suited for these tasks.*

Interference with co-located applications. Many accelerated services, such as machine learning model serving, require sub-millisecond, predictable response latency. In a multi-tenant system, this requirement translates into the need for performance isolation among co-executing workloads on the same physical machine to prevent the noisy-neighbor effect.

While the noisy-neighbor is a well-known problem in CPU-only applications, does it affect accelerated network services? To answer this question, we measure the response latency of a simple GPU-accelerated network server which computes the product of the input vector by a constant. Each request comprises 256 integers. We model the noisy neighbor as a Matrix product of two integer matrices of size 1140×1140, that fully occupies the Last Level Cache on our machine. Concurrent execution of these two workloads on different CPU cores results in 13× higher 99^{th} percentile latency for the GPU-accelerated server (from 0.13 mseconds to 1.7 mseconds) and 21% slowdown for the matrix product compared to their execution in isolation.

In some cases, performance isolation can be improved via Cache Allocation Technology (CAT) in modern processors [35]. However, it is not always available (as is the case in the Xeon E5-2620 CPU available in our server), and provides only a limited number of isolated cache partitions. In general, cache partitioning is still an open problem in large multi-core systems and is an area of active research [13, 46, 51].

In summary *the CPU-centric design suffers from inefficiencies and poor performance isolation.*

3.3 Limitations of the GPU-centric server design

Recent research demonstrated the advantages of the *GPU-centric application design* whereby the GPU runs a complete network-intensive application without any CPU code development [5, 8, 16, 26, 39]. All these works introduce a GPU-side networking layer for network I/O from GPU kernels. In particular, this layer enables the development of GPU-accelerated servers in which the GPU may run the entire server pipeline, from receiving and parsing network messages through request processing to sending the response. For example, GPUnet proposes a full network socket abstraction and API in order to enable full server implementation

on a GPU, whereas GPUrdma implements full support for RDMA verbs. These works show that the GPU-centric design enables higher performance in certain workloads, is more efficient and easier to program than the traditional CPU-driven approach discussed in §3.2.

While promising, there are several challenges in applying this approach to general hardware-accelerated servers in data centers. First, accelerators are usually inefficient at executing control-intensive logic. Even for a GPU, running a network server requires significant computational resources to achieve high I/O performance, at the expense of reduced throughput of the compute-intensive application logic [26]. Second, accelerator-side complex I/O layers put additional pressure on hardware resources, such as registers, and might result in significant performance degradation in some cases [26]. Third, most of these works require Infiniband transport to connect to the service running on the GPU, and do not support UDP/TCP, which significantly restricts their use in data center systems. Last, the majority of these works require a few host CPU cores to operate the GPU-side network I/O; thus they still suffer from the inefficient use of the CPU and performance interference with co-located tenants.

Therefore, *GPU-side network libraries cannot be easily retrofitted beyond GPU systems and Infiniband transport.*

Opportunity. Growing availability of fully-programmable SNICs and emergence of hardware-accelerated network services that rely heavily on accelerators, motivates us to explore new opportunities for improving their efficiency. In Lynx, we build upon the concepts of GPU-centric server design, while eliminating its disadvantages by offloading most of the network communication logic to the SNIC, and freeing *both the CPU and the Accelerator* for more suitable tasks.

4 Design

We first explain the high level design and then describe the main components in details.

4.1 System overview

Lynx targets the system organization shown in Figure 3. An accelerated network service runs on accelerators located in one or multiple physical machines. The SNIC runs a generic network server which serves as a frontend for the network service; all the clients connect to the service via the standard TCP/UDP protocol. Under the hood, the SNIC communicates with remote accelerators via RDMA using regular (not-programmable) RDMA-capable NICs located in their respective machines. All the devices in the same machine communicate via PCIe peer-to-peer DMA without the CPU involvement in data/control path.

Our design strives to meet the following requirements:

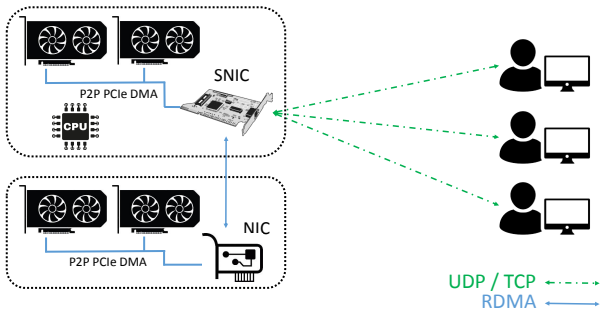


Figure 3. System hardware organization. Lynx runs on the SNIC and serves requests from clients via TCP/UDP. It manages accelerators connected via PCIe in the same server and via RDMA-capable NICs in other servers. The host CPUs in the servers are not involved in request serving.

Provide accelerator-side network I/O. We focus on low-latency server applications that execute their performance-critical logic on accelerators. To this end, Lynx provides network I/O API directly from the accelerator-resident code. Thus, the accelerator can receive and send data, eliminating the need to interrupt its execution when performing I/O via the CPU, thereby reducing the associated overheads.

Avoid running generic server and dispatching logic on accelerators. Unlike prior works on GPU-side I/O support (§3.3), Lynx implements accelerator-side network I/O without running a resource-heavy network server and work dispatch code on the accelerator. Instead, accelerators use a lightweight shim layer with minimal resource demands. This layer can be easily implemented in hardware or software and ported across different accelerators.

Offload accelerator I/O layer to SNIC. Lynx moves the network processing and accelerator management tasks to the SNIC, thereby freeing both the CPU and the accelerators, and enabling TCP/UDP support for clients. The SNIC runs generic, application-agnostic code, therefore the developers of accelerated services do not need to program SNICs.

Maintain portability across accelerators. The SNIC does not run accelerator-specific code. Therefore, Lynx can easily add support for new accelerators. At the same time, we leverage the existing software stack running on the host CPU to set up and configure the accelerators, without having to develop accelerator drivers to run on the SNIC.

4.2 Main components

Figure 4 shows the main Lynx components. The SNIC serves as a mediator between the network and the accelerators. It implements the mechanism to relay the messages to/from the accelerators via message queues, while performing all the network I/O operations on their behalf. The SNIC dispatches requests to the accelerators that execute the server logic.

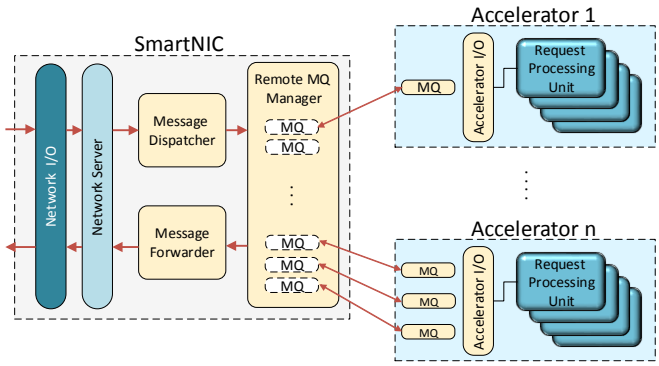


Figure 4. Lynx high level design. MQ: Message Queue. Request Processing Unit represents internal accelerator logic.

Network Server is responsible for handling the network I/O tasks. It performs TCP/UDP transport layer processing, listens on the network ports requested by the server developer, and forwards messages from the network to accelerators and back.

On the ingress side, the *Message Dispatcher* dispatches received messages to appropriate message queues according to the dispatching policy, e.g. load balancing for stateless services, or steering messages to specific queues for stateful ones. On the egress side, *Message Forwarder* fetches the outgoing messages from the message queues, and sends them to respective destinations.

Message Queues (mqueues) are used for passing messages between the accelerator and the SNIC. An mqueue consists of two producer-consumer ring buffers called receive (*RX*) and transmit (*TX*) queues, and their respective notification and completion registers for producer-consumer synchronization.

Mqueues and their status registers are located *in accelerators' local memory*. Therefore, the latency of enqueuing an I/O operation on the accelerator is *exactly the latency of accelerator local memory access*, which is important for reducing the overhead for the accelerator-side I/O.

Remote Message Queue Manager is a key to maintaining the mqueues in accelerator memory. It runs on the SNIC, and uses *one-sided RDMA* to access the mqueues in the accelerator.

The use of RDMA makes it possible for the Lynx to maintain accelerator-agnostic interfaces. Lynx only relies on the NIC's RDMA engine, and the ability to access the accelerator's memory from another PCIe peer, aka peer-to-peer DMA. For example, peer-to-peer GPU access is readily available via GPUdirectRDMA [38] in NVIDIA GPUs, and is also supported by Intel VCA drivers.

In addition, the use of RDMA makes Lynx agnostic to the actual location of mqueues, as long as it is possible to access them via RDMA. As a result, Lynx can manage accelerators spanning multiple physical hosts.

4.3 Accelerator-side networking

The goal of the mqueue abstraction is to support common communication patterns in servers without providing the full flexibility of POSIX sockets while gaining simplicity and implementation efficiency.

We define two types of mqueues: *server* and *client*.

Server mqueue is best suited for simple RPC-like request-response interactions. It is associated with a network port on which the server is listening. From the receiving accelerator perspective, the server mqueue is a connection-less messaging abstraction similar to a UDP socket. Namely, two messages received from the mqueue may be sent from different clients. However, when the accelerator writes the response back to the mqueue, the response will be sent to the client from which the request was originally received. Here we choose a simpler networking interface over flexibility.

This approach achieves good scalability in terms of the number of connections by reusing the same mqueue for multiple client connections instead of creating one per connection.

Each accelerator may have more than one server mqueue associated with the same port, e.g., to allow higher parallelism. For example, in our implementation of the LeNet inference server, the GPU has only one server mqueue, whereas in the Face Verification server there are 28 server mqueues managed in a round-robin manner (§6).

Client mqueue serves for sending messages to other servers and for receiving responses from them. Unlike server mqueues, it cannot be reused for different destinations. The destination address is assigned when the *server* is initialized. This design choice favors simplicity over flexibility of a dynamic connection establishment. Static connections are sufficient to support a common communication pattern for servers to access other back-end services. For example, in the Face Verification server, we use client mqueues to communicate with a remote database stored in `memcached` (§6.4).

Using mqueues. A CPU is responsible for initializing mqueues in accelerator memory. It passes the pointers to the mqueues to their respective accelerators. On the SNIC, it configures the Network Server to dispatch the messages to these mqueues, providing the mqueue pointers to the SNIC. Then, it invokes the accelerator to handle incoming requests, and remains idle from that point. Both the accelerator and the SNIC use polling to communicate via mqueues.

4.4 Accelerator hardware requirements

There are two requirements that an accelerator must fulfill in order to work with Lynx.

First, for the peer-to-peer PCIe DMA to work between the RDMA-capable NIC and the accelerator, the accelerator must be able to expose its memory on the PCIe (via its Base Address Register, BAR). A less efficient and more complex alternative (from the accelerator's hardware perspective) is

to use host memory for NIC DMAs, and for the accelerator to map that memory into its virtual address space. Note that the host CPU is not involved in the transfers in either case, but the former is clearly more efficient, and does not assume virtual memory support in the accelerator. The host is expected to allocate the memory buffers and configure all the mappings ahead of the execution.

Second, to allow producer-consumer interaction between the SNIC and the accelerator over RDMA, the accelerator must have the means to enforce the memory ordering when accessing its local memory, as well as to comply with the PCIe ordering rules for the accesses to its memory from the NIC via PCIe BAR. These requirements are needed to force the strict ordering of updates among the data and the data-ready flag (*doorbell*) in mqueues, in the transmit and receive paths respectively. See §5.1 for the discussion about the GPU.

4.5 Discussion

Accelerators with integrated NICs. In certain systems a NIC is integrated with an accelerator [17]. While such accelerators might run their own networking library, we believe that Lynx might still be beneficial for them.

This is because to support TCP access from clients, the accelerator must run the TCP stack, or implement it in hardware. The former is resource-demanding and inefficient, whereas the latter has well-known maintainability issues. Furthermore, the network processing stack should run on every such accelerator.

Instead, Lynx offloads network processing to the SNIC, which in turn can be shared across multiple accelerators, thereby freeing their resources to run the application logic. Therefore, Lynx can support accelerators with integrated NICs in a way similar to how it manages remote accelerators connected via their RDMA-capable NICs.

Difference from RDMA verbs. Mqueue resembles RDMA Queue Pairs [10] but it is optimized to be used by accelerators. First, an mqueue requires fewer operations to send a message. For example, sending over RDMA requires preparation of a Work Queue Element, followed by access to the NIC doorbell register over PCIe, and lastly polling for completion. In Lynx all these operations are offloaded to the SNIC; the accelerator is only required to write a message to the mqueue and update a control register in its local memory. This is a key to enabling low-overhead I/O from the accelerator. For comparison, enqueuing a single RDMA send request (note, the request itself is asynchronous) requires at least 4.8μsec [8]. This is a long blocking operation which affects multiple GPU threads. Thus, developers are forced to send large (over 64KB) messages to hide the overheads. In Lynx this problem is solved.

Second, the memory layout of the mqueue is flexible, and determined by Lynx runtime rather than RDMA standard. As a result, it can be tailored for the specific accelerator, i.e., eliminating unnecessary fields and aligning the entries as necessary.

Scaling to multiple connections. Lynx architecture allows the scaling of a large number of concurrent incoming connections, which is critical for supporting real-life server workloads. In contrast to prior works [8, 42], where every connection is associated with an RDMA QP or a socket, Lynx allows multiplexing multiple connections over the same server mqueue. In practice, the scalability depends on the compute capacity of the SNIC to multiple connections in its network stack.

Scaling to multiple accelerators. Adding new accelerators to a system requires more mqueues, and increases the load on the Remote Message Queue Manager. As we show in the evaluation, a Lynx on an SNIC may scale to dozens of accelerators, as long as the system performance is bounded by the aggregated throughput of all its accelerators.

Multi-tenancy. Lynx runtime can be shared among multiple servers. For example, users may use different accelerators for their applications, e.g., subscribing for Lynx' services. Lynx is designed to support multiple independent applications while ensuring full state protection among them.

5 Implementation

We prototype Lynx using two SNICs: Mellanox Bluefield with ARM cores and Mellanox Innova Flex with an FPGA (see §2 for details). We implement a complete version for the Bluefield SNIC and a partial prototype for Innova. In addition, the Bluefield version of Lynx is source-compatible to run on X86 CPU in Linux.

We use NVIDIA K40m and K80 GPUs, and integrate Lynx with the Intel Visual Compute Accelerator, which requires only minor modifications.

5.1 Lynx on Bluefield

Using RDMA to access accelerator memory. The main technical idea that underlies our implementation is the use of the one-sided RDMA support in Bluefield. Bluefield runs BlueOS Linux distribution. It includes the Mellanox OFED stack [30] which we use to implement RDMA access to mqueues.

How efficient is it to use RDMA to access mqueue in accelerator memory compared to using the accelerator's internal DMA to do so? The former is used by Lynx and is a device-agnostic mechanism, whereas the latter is the standard way to copy data to/from accelerators, but its DMA engine must be programmed via the driver.

We perform the evaluation on a GPU, and access GPU memory from the CPU (because we do not have an NVIDIA driver for SNIC). We build a GPU-accelerated echo server with a single threadblock that receives data via a single mqueue, and measure the end-to-end throughput. We evaluate three mechanisms for accessing the mqueue from the

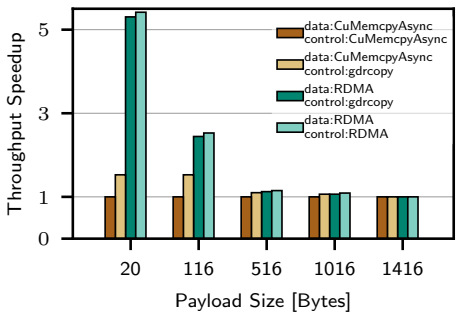


Figure 5. Performance of data transfer mechanisms for managing mqueue, relative to `cudaMemcpyAsync`.

CPU: `cudaMemcpy`, `gdrcpy` [37] and the Ininiband RDMA from the NIC. `gdrcpy` is a tool to allow direct mapping of GPU memory into the CPU virtual address space and to access it directly from CPU programs. We use different access mechanisms for the data path (payload transfers) and control path (access to status registers).

Figure 5 shows the results. Each bar represents the speedup relative to the implementation that uses `cudaMemcpyAsync`. RDMA performs better than any other mechanism, in particular for smaller granularity of accesses. This is because `cudaMemcpyAsync` incurs a constant overhead of 7-8μsec dominating small transfers, whereas `gdrcopy` blocks until the transfer is completed because it is invoked as a write access to memory. These write accesses are on the critical path of the Message Dispatcher, and therefore slow it down. On the other hand, IB RDMA requires less than 1μsec to invoke by the CPU [11]; thus it is more efficient. In summary, the use of RDMA improves system performance compared to cudaMemcpy, confirming the performance advantages of our design.

Metadata and data coalescing in mqueues. To reduce the number of RDMA operations for updating the mqueue, we append control metadata to each message. The metadata occupies 4 bytes, and includes (1) total message size, (2) error status from the Bluefield (if a connection error is detected), and (3) notification register (doorbell) for the queue. The accelerator polls this notification register while waiting for a new message.

Unifying the notification register with the payload works correctly only if the NIC DMA is performed from lower to higher virtual addresses; otherwise the register will be set before the message has arrived. We validated that this is indeed the case for Mellanox NICs.

Data consistency in GPU memory. We prototype Lynx using GPUs that run persistent kernels. It is well known that in such a setup, the peer-to-peer DMA writes from the NIC to the GPU might not preserve the PCIe ordering [38]. This implies that the updates to the mqueue doorbell and the data could be reordered, leading to the data corruption of a received message. We note that we have never observed such a corruption, and similar results were also reported earlier [8, 26].

Recently, NVIDIA published an unofficial technique to overcome the consistency problem [44]. The idea is to use an RDMA read from the GPU memory as a write barrier after data update. Thus, in Lynx, each message to the GPU is performed via three RDMA transactions: RDMA write to the data, blocking RDMA, and RDMA write to the mqueue doorbell. We measured that these operations incur extra latency of 5μseconds to each message. This is a significant per-message overhead that affects the overall system efficiency, and also disables our metadata/data coalescing optimization. However, we hope that NVIDIA will provide a more efficient write barrier in future devices.

We note that for the purpose of this paper, GPU persistent kernels are used to emulate the behavior of hardware accelerators, rather than applied specifically to GPU workloads. Therefore, in our evaluation we disable the consistency enforcement workaround described above.

One RC QP per accelerator. We implement a standard producer-consumer ring buffer, but use RDMA for updating the data and the status registers. To create an mqueue, Lynx initializes an InfiniBand Reliable Connection (RC) QP with buffers in accelerator memory. To reduce the number of RC QPs, Lynx coalesces all the mqueues of the same accelerator to use the same RC QP and the same ring buffer, equally partitioned among the mqueues.

5.1.1 Network Server

Network server performs many `recv`/`send` system calls on sockets. We observed that ARM cores on Bluefield incur high system call cost, making the use of Linux kernel I/O stack on the SNIC too expensive.

Instead, we employ VMA [31], a user-level networking library that allows direct access from user mode to the network adapter bypassing the kernel. The use of the library on Bluefield significantly improves the performance. For example, for minimum-size UDP packets VMA reduces the processing latency by a factor of 4. The library is also effecient on the host CPU resulting in 2× UDP latency reduction.

5.2 Lynx on Innova Flex

We partially implement Lynx on Mellanox Innova Flex SNIC, equipped with Xilinx FPGA. We leverage NICA [12], a hardwar-software co-designed framework for inline acceleration of server applications on SNICs.

Background: NICA architecture. NICA provides hardware infrastructure to build Accelerated Function Units (AFUs) on FPGA-based SNICs, and software support to integrate them with server applications on the host. All the network

traffic passes through the AFU. To simplify AFU development, NICA provides an on-FPGA UDP stack, and implements a network switch to support multiple AFUs.

Our prototype only implements the receive path, so we explain the receive-side mechanisms. The data passing from the network through the AFU can be received on the host in two ways: standard POSIX socket API and a *custom ring API* which directly stores application-level messages in receive buffers bypassing the CPU network stack.

Lynx on NICA. The network server is implemented as an AFU that listens on a given UDP port, appends the metadata to each message, and places the payload onto the available custom ring used as an mqueue. The NICA driver is modified to allocate custom rings in accelerator memory.

This prototype has two primary limitations. First, it does not yet support the send path. Second, it requires a CPU helper thread for the custom ring. This is because NICA uses InfiniBand Unreliable Connection (UC) QP to implement the custom ring in the FPGA, which in turn requires a separate CPU thread to explicitly refill the QP receive queue, and to take care of the flow control. We believe, however, that the requirement to use the CPU thread is not fundamental, and will be removed in the future with the NICA implementation of custom rings using one-sided RDMA.

5.3 Lynx I/O library for accelerators

The implementation of the I/O library consists of a few wrappers over producer-consumer queues of an mqueue that provide familiar `send` and `recv` calls only with zero-copy.

GPU I/O library. The GPU uses a single thread in a threadblock to implement the I/O logic. This is in contrast with other GPU-side network libraries that use multiple threads [8, 26]. The rest of the implementation is a fairly standard producer-consumer queue logic.

5.4 Intel Visual Compute Accelerator

Background. Intel VCA packs three independent Intel E3 processors each with its own memory. These CPUs are interconnected via a PCIe switch, which in turn connects to the PCIe slot on the host. It supports secure computations via x86 Software Guarded Extensions, SGX [20].

From the software perspective VCA appears as three independent machines running Linux, each with its own IP. The host connects to each processor via standard tools such as SSH, thanks to the IP over PCIe tunneling.

Lynx on VCA. We implement the I/O layer for the Intel VCA card as an example of integration of a new accelerator into the system. All we had to do was to modify 4 lines of code in Lynx in order to expose the VCA-accessible memory to the NIC.

Unfortunately, our attempts to allow RDMA into VCA memory were unsuccessful, most likely due to a bug. Therefore, we used CPU memory to store the mqueues but mapped this memory into VCA. This workaround allows us to estimate the system performance, albeit in a sub-optimal configuration.

5.5 Support for remote accelerators

Lynx can manage remote accelerators just as it manages the local ones. From the hardware perspective, the only requirement is that each remote accelerator is connected to InfiniBand via its own network adapter which supports peer-to-peer PCIe access to the accelerator. A remote host sets up the RC QP in the accelerator's memory, and from that point such a remote accelerator is indistinguishable for RDMA access from a local one. Indeed, all what is required from Lynx is to change the accelerator's host IP to the one of the remote host.

6 Evaluation

In our evaluation we seek to answer the following questions:

1. How Lynx compares to a host-centric architecture;
2. How well it scales;
3. How SNIC contributes to the server efficiency when running multiple workloads;
4. How portable Lynx is to run on Intel VCA.

Lynx is available at https://github.com/acsl-technion/lynx.

Hardware setup. We use 2 client and 4 server machines with Intel Xeon E5-2620 v2 CPU, connected via Mellanox SN2100 40Gbps switch. One server uses a 25Gbps Mellanox BlueField SNIC, one with a 40Gbps Mellanox Innova Flex 4 Lx EN SNIC. The two others with ConnectX-4 Lx EN NICs used for hosting remote GPUs. We use 2× NVIDIA K40m and 6× K80 dual GPUs, and an Intel VCA. Hyper-threading and power saving settings are disabled to reduce noise.

Performance measurements. We use `sockperf` [32] with VMA [31] to evaluate the server performance. `sockperf` is a network load generator optimized for Mellanox hardware. We run each experiment 5 times, 20 seconds (millions of requests), with 2 seconds warmup. We report the average. Standard deviation is below 1.5%, not reported.

6.1 Evaluated server designs

- **Host-centric (baseline)**: network messages are received by the CPU, which then invokes a GPU kernel for each request.
- **Lynx on Bluefield**: We use 7 ARM cores (out of 8);
- **Lynx on the host CPU**: runs the same code as on Bluefield;
- **Lynx on Innova FPGA-Based SNIC.**

6.2 Microbenchmarks

For all the experiments here we use the following GPU server implementation: In each threadblock there is 1 thread which copies the input to the output, and waits for a predefined period emulating request processing.

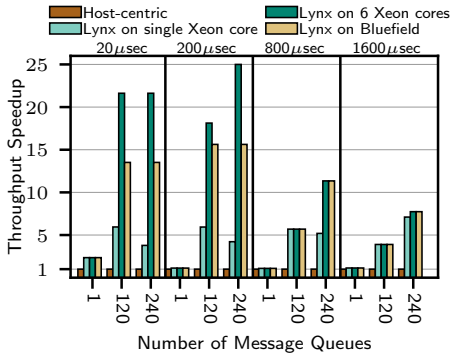


Figure 6. Relative throughput of GPU server implementations for different request execution times (higher is better).

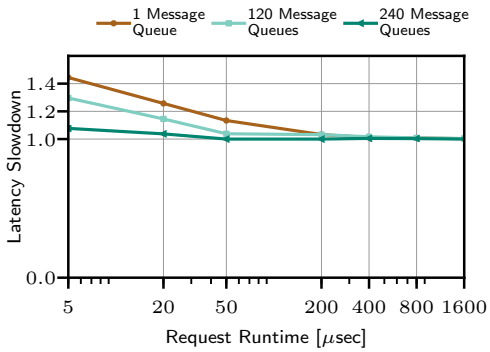


Figure 7. Relative latency of a GPU server with Lynx on Bluefield vs. Lynx on 6-core CPU (lower is better). Absolute numbers are in the text.

For a host-centric server, we use a pool of concurrent CUDA streams, each handling one network request. For each request, a stream performs data transfer to the GPU, kernel invocation (1 threadblock), data transfer back to the CPU. We run on one CPU core because more threads result in a slowdown due to an NVIDIA driver bottleneck.

For Lynx, we run a persistent GPU kernel with up to 240 threadblocks (maximum number of concurrently executing threadblocks on NVIDIA K40m). Each threadblock polls its own mqueue, processes the message and sends the response.

Throughput comparison. We evaluate the performance across two dimensions: different number of mqueues and different request execution times. Varying these two parameters effectively changes the rate at which the system handles execution requests, but their combination stresses different components. Higher execution time reduces the load on the networking logic. Increasing the number of mqueues is equivalent to increasing the number of parallel requests, which stresses the dispatching logic and the mqueue manager. Combining both will show the saturation point of the server. We use 64B UDP messages to stress the system.

Figure 6 shows that the host-centric design performs significantly worse than other implementations. Lynx on Bluefield is faster in particular for short requests with one mqueue (2×), and even more for a larger number of mqueues (15.3×). The slowdown is due to GPU management overheads.

When comparing Lynx on the host and on the Bluefield, we see that the latter is always faster than a single core on the host, but up to 45% slower than 6 host cores for shorter requests and 240 mqueues (200μsec and lower). This is expected because the UDP stack is slower on Bluefield. For these configurations, we find that one needs *4 host CPU cores* to match the Bluefield performance (not shown in the graph).

For larger requests both Bluefield and 6-core CPU achieve the same throughput, but a single host core is not enough to handle 240 mqueues even for 1.6 msec requests.

Latency of Lynx on Bluefield vs. host CPU. As Figure 7 shows, shorter requests are slower on Bluefield, but the difference diminishes for requests of 150μsec and higher. For a larger number of mqueues, both platforms spend more time on handling multiple mqueues (via round-robin), so the relative performance for any request size is within 10%.

The latency breakdown shows that when running Lynx on Bluefield, the request spends 14μsec from the point it completes the UDP processing till the GPU response is ready to be sent, with a zero-time GPU kernel (copy 20 bytes from input to output). The same time on the host CPU is 11μsec. With the end-to-end latency of 25μsec and 19μsec for Bluefield and CPU respectively, the interaction with the GPU is the main source of overheads for short requests.

Lynx on Bluefield results in negligible latency overheads for requests larger than 200μsec.

Bluefield vs. Innova FPGA. This experiment aims to estimate the maximum throughput Lynx can achieve when running on a specialized FPGA hardware.

Since the FPGA prototype is limited to the receive path alone, we measure the system throughput when receiving messages on the GPU rather than end-to-end. One more limitation is that it requires the use of CPU helper threads (see §5.2).

In the experiment, we use 240 mqueues on a single GPU and measure the receive throughput for 64B UDP messages. Innova achieves 7.4M packets/sec compared to 0.5M packets/sec on Bluefield. The CPU-centric design running on six cores is 80× slower.

Using specialized SNICs for running Lynx holds significant performance potential compared to a fully programmable Bluefield and to the host CPU.

Performance isolation. We run Lynx on Bluefield in parallel with the noisy-neighbor application as in Section 3.2. As expected, we observe no interference between them, in contrast to its execution on the CPU.

Using Bluefield provides better isolation between different applications running on the same physical server.

Integration with the Intel VCA. We run a simple secure computing server *inside the SGX enclave* on one of the Intel VCA processors. The server receives an AES-encrypted message (4 bytes) via Lynx, decrypts it, multiplies it by a constant, encrypts it and sends the result back. SGX guarantees that the encryption key is not accessible to any component on the server, besides the client code in the enclave. The Lynx I/O library is small and simple (20 Lines of Code); therefore it is statically linked with the enclave code and included in the Trusted Computing Base.

Our baseline is a server that runs inside the VCA but invokes the enclave on each request. It uses the native Linux network stack on VCA which receives data via a host-based network bridge. This is the Intel preferred way to connect the VCA to the network.

Lynx achieves 56μsec 90^{th} percentile latency, which is 4.3× lower than the baseline under the load of 1K req/sec.

Lynx facilitates the integration of high performance networking with accelerators, and shows significant performance gains compared to their original network I/O design.

6.3 LeNet neural network inference server.

We build an accelerated server to perform written digits recognition using the standard LeNet Convolutional Neural Network architecture [27]. A client sends 28×28 grayscale images from the standard MNIST dataset [53], and the server returns the recognized digit by running the LeNet inference on the GPU.

We evaluate three versions of the server: host-centric (baseline), Lynx on Bluefield, Lynx on CPU.

The LeNet computations for Lynx are performed in a persistent GPU kernel: We use a single GPU thread to poll the server mqueue. Then, it invokes the GPU kernels that implement the actual neural network inference using the *dynamic parallelism* [36] feature of NVIDIA GPUs that allows spawning kernels from another kernel. When these kernels terminate, the response is sent back to the client via an mqueue.

We use TVM [7] to generate the GPU-only LeNet implementation from TensorFlow [1] which does not run any application logic on the CPU (see §3.1). The host-centric design runs the TVM code for each request.

LeNet end-to-end performance. We measure the throughput of all three implementations. We observe that running Lynx on both Bluefield and a Xeon core achieves 3.5 Kreq/sec, 25% faster compared to 2.8 Kreq/sec using the host-centric baseline. We note that the theoretic maximum throughput on a single GPU is 3.6Kreq/sec, only within 3% of the Lynx on Bluefield.

Figure 8a shows the latency distribution at maximum throughput for UDP requests. For 90^{th} latency percentile, Lynx on Xeon and on Bluefield achieve 295μsec and 300μsec respectively, whereas the host-centric server is 14% slower.

Running the server over TCP (not shown) achieves about 10% lower throughput for Bluefield (3.1Kreq/s, 5% for Xeon (3.3Kreq/sec), and introduces additional latency. The end-to-end latency is 322μsec and 346μsec on Xeon and on Bluefield respectively. This result is expected, because TCP processing demands more compute resources, and ARM cores suffer from higher impact.

Comparing CPU efficiency of Lynx and server workloads. One of the goals of Lynx has been to achieve higher system efficiency. In particular, our goal has been to free the CPU for other tasks by offloading Lynx to the SNIC.

At a high level, we want to demonstrate the following. Given two concurrently executing applications $A1$ and $A2$ (perhaps from different tenants), where $A1$ is a hardware-accelerated server and $A2$ is some other multi-threaded application. Lynx makes the system more efficient if offloading $A1$ to the SNIC as enabled by Lynx, and giving the freed CPU cores to $A2$ results in a higher *overall* system performance than mapping them in reverse or keeping them both on CPU.

To illustrate this point, we use a Lynx-based GPU-accelerated LeNet server as $A1$ and a typical server workload, `memcached` key-value store as $A2$.

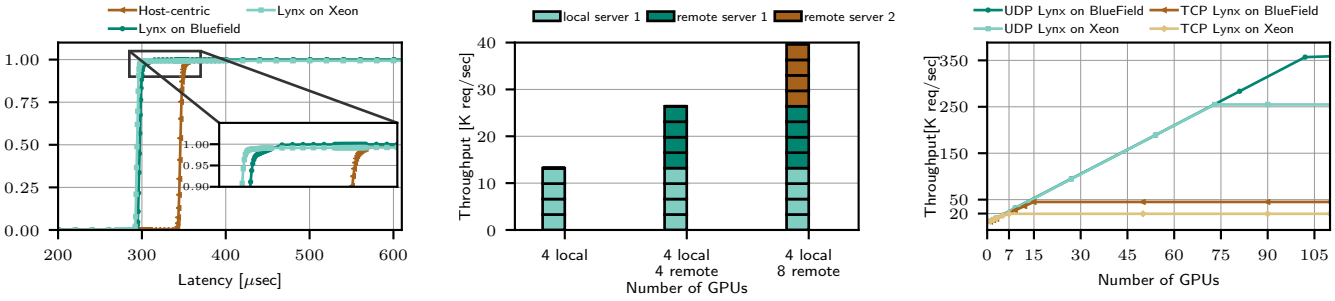
We compare the performance of both applications in two configurations: (1) `memcached` running on all six host CPU cores (six instances) while the LeNet server is managed by Bluefield, and (2) `memcached` running on five host cores and *on Bluefield*, and LeNet with Lynx on the sixth host core.

We run `memcached` on Bluefield in two modes. In the throughput-optimized mode, we evaluate the system at its maximum throughput, whereas in the latency-optimized mode we measure the throughput under a given latency target.

Figure 9 depicts the latency (99^{th} percentile) and throughput of `memcached servers in both configurations`. The performance of the LeNet server *does not depend* on the configuration (3.5Kreq/sec, not shown); therefore the *overall* system performance is dictated by `memcached` alone.

As we see in Figure 9, `memcached` on Bluefield achieves a higher throughput than a Xeon core (400 Ktps vs. 250 Ktps/core), but at the expense of a dramatic latency increase (160μsec vs. 15μsec). For the latency-optimized configuration, we set the latency target to not exceed that of the server latency of 15μsec on Xeon. However, this requirement *cannot be satisfied*, since the use of Bluefield necessarily increases the latency beyond that threshold.

We note that here we allocate only one CPU core for running Lynx, which might be insufficient for matching the performance of the Bluefield as observed in the first Experiment. With more cores for Lynx, offloading it to SNIC would result in even larger performance gains for `memcached`.



(a) Latency distribution at maximum throughput. **(b)** Lynx scaleout to remote GPUs **(c)** Scalability projection with Lynx.

Figure 8. LeNet neural network inference service.

This experiment confirms that *Lynx is able to improve system efficiency via offloading hardware-accelerated server management to a SNIC.*

Scaleout to remote GPUs. We show that Lynx can scale beyond one physical machine. We configure Lynx to manage 12× Tesla K80 GPUs in three machines, one of which runs Lynx on Bluefield. Figure 8b shows that the system throughput scales *linearly* with the number of GPUs, regardless whether remote or local[2]. Using remote GPUs adds about 8μsec latency.

Multi-GPU scalability projection. Once not limited to a single machine, we ask how many GPUs running LeNet one Bluefield fully utilize.

To estimate the scalability, we emulate the request processing by invoking a kernel with a single thread which blocks the amount of time equivalent to the LeNet execution on GPU. We instantiate multiple such kernels on the same GPU and connect one mqueue to each. All kernels are executed concurrently to emulate parallel request processing on different GPUs.

Figure 8c shows that Lynx with Bluefield scales linearly for both UDP and TCP connections. For UDP, Lynx on Bluefield scales up to 102 GPUs, compared to 74 GPUs on a Xeon core. For a TCP connection, Lynx with Bluefield scales up to 15 GPUs, compared to 7 GPUs for Lynx on a Xeon core, because of the TCP processing overheads.

We note that the emulation results precisely match the performance of Lynx on 12 real GPUs. We believe that this methodology provides a reliable estimate of the actual system performance with multiple GPUs as long as the RDMA infrastructure is not the bottleneck.

6.4 Support for multi-tier applications: Face Verification Server.

Face Verification server benchmark has been used in prior works [26] for measuring GPU-side network I/O performance. A client sends a picture (a face) along with a label (person ID). The server verifies the person's identity by comparing the picture received with the picture corresponding to the person's ID in its own database. The comparison is performed using a well-known local binary patterns (LBP) algorithm for Face Verification [3]. The server returns the comparison result to the client.

This is an example of a multi-tier server: the frontend tier that receives all the requests communicates with the database backend tier to fetch data relevant for the request.

In our implementation we use a `memcached` server to store the image database. The verification server runs on a GPU. The server communicates with clients and the database server (running on a different host) via mqueues. We use both UDP for communicating with clients, and TCP for interacting with `memcached`. The goal of this setup is to highlight the Lynx's ability to support complex server applications.

We use images from a color FERET Database [23] resized to 32×32. The labels are random 12-byte strings. Clients issue uniformly-distributed random queries.

We implement two versions of the server: (1) Host-centric (baseline); It fetches the database image for a given label from `memcached`, and launches a kernel to compare the images. The access to `memcached` is asynchronous; the server may start handling the next request while waiting for the response from the database. (2) GPU-centric with Lynx. Here, a persistent GPU kernel issues the request to `memcached` via mqueues. We allocate 28 message queues each connected to a kernel executed by a single threadblock with 1024 threads.

Face Verification performance. Lynx achieves over 4.4× and 4.6× higher throughput for Bluefield and Xeon core respectively compared to the host-centric design, because the overhead of kernel invocation and GPU data transfers are relatively high vs. the kernel execution time (about 50μsec).

The host-centric implementation uses two CPU cores to achieve its highest throughput. Using more cores results in lower performance. Running Lynx on Bluefield is about 5% slower than on a Xeon core, due to the slower TCP stack processing on Bluefield when accessing `memcached`.

[2]Tesla K80 GPU is slower than K40m and achieves 3300 req/sec at most.

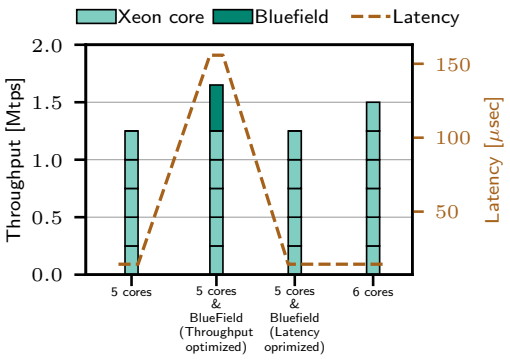


Figure 9. Illustration of the (inefficient) use of Bluefield to run server workloads (`memcached`) vs. a single Xeon core.

7 Related work

Hardware for GPU networking. Network communications from GPUs were first considered in NaNet [5] using proprietary network protocols and a specialized NIC. An early work on Infiniband verbs on GPU [40] concluded that using them is inefficient, and suggested adding specialized hardware to offload the bookkeeping of communication requests. Later, the GPUrdma project [8] overcame some of the overheads of on-GPU Verb processing, but also demonstrated that the latency of posting communication requests is dominated by the access to NIC control registers and work request preparation. Lynx has been inspired by these ideas, but it generalized and implemented them on real hardware.

GPU-side network I/O libraries. Several works focused on supporting high-level network I/O abstractions from GPUs. GPUnet [26] implemented the socket abstraction over reliable Infiniband connections; NVSHMEM [39] implemented Partitioned Global Address Space abstraction on top of Infiniband RDMA. dCUDA [16] offered a partial implementation of Message Passing Interface (MPI) on GPUs. All these works have a common goal, which is to provide intuitive communication support for GPUs, but they differ from Lynx in four aspects: (1) they add a GPU-side network library with complex logic that imposes resource overheads on GPU kernels; (2) they provide GPU-specific implementations, and do not generalize easily to other accelerators; (3) they all use the host CPU to function; (4) they run on Infiniband and provide no TCP/UDP support.

Peer-to-peer PCIe optimizations. GPUDirectRDMA [38] is a low-level mechanism in NVIDIA GPUs to enable access to their memories from other peripherals via PCIe peer-to-peer communication. It has been used together with the GPU-side network I/O to optimize MPI libraries in GPU-accelerated clusters [43]. Lynx also uses GPUDirectRDMA.

Network Servers on GPUs. Building servers on GPUs has been suggested as an efficient alternative to standard server architecture. Rhythm [2] introduced a GPU-based server architecture to run PHP web services. MemcachedGPU [18] demonstrated a GPU-based key-value store server. GPUnet [26] showed a few GPU-native servers and analyzed their performance tradeoffs. Similarly to these works, Lynx also advocates for GPU-centric server architecture, but improves efficiency by offloading much of the server logic from GPU to SNIC.

Network processing acceleration on GPUs. Several works, among them SSLShader [22], NBA [25], and GASPP [49] demonstrated how GPUs can be used for accelerating packet processing and network functions. In contrast, Lynx, in its current form, focuses on application-level acceleration on GPUs, and not on packet processing.

SNIC-based acceleration of network applications. The use of SNICs to accelerate network processing has recently drawn significant interest. For example, AccelNet [14] enables fast in-network packet processing on top of Microsoft Catapult SNICs [34]. FlexNIC [24] proposed a new programmable networking device architecture for offloading network processing tasks from applications. NICA [12] proposes using FPGA-based SNICs to accelerate network servers. Floem [41] proposes a compiler to enable simpler application development to offload computations on SNICs. Lynx also uses SNICs, with a different goal: instead of accelerating the application logic, Lynx aims to enable more efficient servers by running some of their data and control plane.

8 Conclusions

The emerging diversity of compute and I/O accelerators in data centers calls for new ways of building efficient accelerated network services. Lynx introduces an accelerator-centric server architecture in which the generic server networking logic is offloaded to an SNIC, whereas compute accelerators perform network I/O efficiently, without additional architectural burden. Using GPUs and VCA as an example of accelerator architectures, we prototype Lynx on two different SNICs and demonstrate its performance and efficiency benefits over the host CPU.

Our main takeaway is that SNICs offer a viable alternative to the host CPU for driving compute-bound accelerated network servers. The more specialized the SNIC architecture, the higher its performance potential. Yet, as CPU-based SNICs are evolving and becoming more capable, using them for managing accelerators is an appealing approach to building a fast and efficient hardware-accelerated network service.

9 Acknowledgements

We thank our shepherd Chris Rossbach for insightful comments, and Mellanox for hardware donation and assistance. We greatfully acknowledge support from Israel Science Foundation (Grant 1027/18) and Israeli Innovation Authority.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] Sandeep R. Agrawal, Valentin Pistol, Jun Pang, John Tran, David Tarjan, and Alvin R. Lebeck. 2014. Rhythm: harnessing data parallel hardware for server workloads. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. 19–34. https://doi.org/10.1145/2541940.2541956

[3] Timo Ahonen, Abdenour Hadid, and Matti Pietikäinen. 2006. Face Description with Local Binary Patterns: Application to Face Recognition. *IEEE Trans. Pattern Anal. Mach. Intell.* (2006), 2037–2041. https://doi.org/10.1109/TPAMI.2006.244

[4] Amazon Elastic Inference. [n.d.]. Amazon Elastic Inference: Add GPU acceleration to any Amazon EC2 instance for faster inference at much lower cost. https://aws.amazon.com/machine-learning/elastic-inference/.

[5] Roberto Ammendola, Andrea Biagioni, Ottorino Frezza, G. Lamanna, Alessandro Lonardo, Francesca Lo Cicero, Pier Stanislao Paolucci, F. Pantaleo, Davide Rossetti, Francesco Simula, M. Sozzi, Laura Tosoratto, and Piero Vicini. 2014. NaNet: a flexible and configurable low-latency NIC for real-time trigger systems based on GPUs. *JINST* (2014). https://arxiv.org/pdf/1311.4007.pdf

[6] Cavium. [n.d.]. LiquidIO SmartNIC family of intelligent adapters provides high performance industry-leading programmable server adapter solutions for various data center deployments. https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/index.jsp.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 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, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[8] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrdma: GPU-side Library for High Performance Networking from GPU Kernels. ACM, New York, NY, USA, 6:1–6:8.

[9] Gregory Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent RNNs: Stashing Recurrent Weights On-chip. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, 2024–2033. http://proceedings.mlr.press/v48/diamos16.pdf

[10] Dotan Barak. [n.d.]. RDMAmojo âĂŞ blog on RDMA technology and programming. https://www.rdmamojo.com/2013/06/01/which-queue-pair-type-to-use/.

[11] Dotan Barak. [n.d.]. RDMAmojo âĂŞ blog on RDMA technology and programming. https://www.rdmamojo.com/2013/01/26/ibv_post_send/.

[12] Hagai Eran, Lior Zeno, Gabi Malka, and Mark Silberstein. 2017. NICA: OS Support for Near-data Network Application Accelerators. In *International Workshop on Multi-core and Rack-scale Systems (MARS17)*. http://acsl.eelabs.technion.ac.il/publications/nica-os-support-for-near-data-network-application-accelerators/

[13] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr, and Dejan Kostic. 2019. Make the Most out of Last Level Cache in Intel Processors *(EuroSys '19)*. https://people.kth.se/~farshin/documents/slice-aware-eurosys19.pdf

[14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[15] Google AutoML. [n.d.]. AutoML: Train high-quality custom machine learning models with minimal effort and machine learning expertise. https://cloud.google.com/automl/.

[16] Tobias Gysi, Jeremia Bär, and Torsten Hoefler. 2016. dCUDA: hardware supported overlap of computation and communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. 609–620. https://doi.org/10.1109/SC.2016.51

[17] Habana. [n.d.]. Goya deep learning inference accelerator: White paper. https://habana.ai/wp-content/uploads/2019/06/Goya-Whitepaper-Inference-Performance.pdf.

[18] Tayler H. Hetherington, Mike O'Connor, and Tor M. Aamodt. 2015. MemcachedGPU: scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*. 43–57. https://doi.org/10.1145/2806777.2806836

[19] Huawei. [n.d.]. FPGA-Accelerated Cloud Server. https://www.huaweicloud.com/en-us/product/fcs.html.

[20] Intel. [n.d.]. IntelÂő Software Guard Extensions (IntelÂő SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.

[21] Intel. [n.d.]. IntelÂő Visual Compute Accelerator (IntelÂő VCA) Product Brief. https://www.intel.com/content/www/us/en/servers/media-and-graphics/visual-compute-accelerator-brief.html.

[22] Keon Jang, Sangjin Han, Seungyeop Han, Sue B. Moon, and KyoungSoo Park. 2011. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. https://www.usenix.org/conference/nsdi11/sslshader-cheap-ssl-acceleration-commodity-processors

[23] Jonathon Phillips. [n.d.]. color FERET Database. https://www.nist.gov/itl/iad/image-group/color-feret-database.

[24] Antoine Kaufmann, Simon Peter, Thomas E. Anderson, and Arvind Krishnamurthy. 2015. FlexNIC: Rethinking Network DMA. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. https://www.usenix.org/conference/hotos15/workshop-program/presentation/kaufmann

[25] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. 2015. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. ACM, 22:1–22:14.

[26] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 201–216. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kim

[27] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. [n.d.]. Gradient-Based Learning Applied to Document Recognition. In *Proceedings of the IEEE, november 1998*. http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

[28] Layong Larry Luo. 2018. Towards Converged SmartNIC Architecture for Bare Metal & Public Clouds. https://conferences.sigcomm.org/events/apnet2018/slides/larry.pdf. 2nd Asia-Pacific Workshop on Networking (APNet 2018).

[29] Mellanox Technologies. [n.d.]. BlueField SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic.

[30] Mellanox Technologies. [n.d.]. Mellanox OpenFabrics Enterprise Distribution for Linux (MLNX_OFED). http://www.mellanox.com/page/products_dyn?product_family=26.

[31] Mellanox Technologies. 2018. libvma: Linux user-space library for network socket acceleration based on RDMA compatible network adaptors. https://github.com/Mellanox/libvma.

[32] Mellanox Technologies. 2018. sockperf: Network Benchmarking Utility. https://github.com/Mellanox/sockperf.

[33] Microsoft Brainwave. [n.d.]. Brainwave: a deep learning platform for real-time AI serving in the cloud. https://www.microsoft.com/en-us/research/project/project-brainwave/.

[34] Microsoft Catapult. [n.d.]. Microsoft Catapult: Transforming cloud computing by augmenting CPUs with an interconnected and configurable compute layer composed of programmable silicon. https://www.microsoft.com/en-us/research/project/project-catapult/.

[35] Nguyen, Khang T. [n.d.]. Introduction to Cache Allocation Technology in the IntelÂő XeonÂő Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[36] NVIDIA. [n.d.]. CUDA Dynamic Parallelism API and Principles. https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/.

[37] NVIDIA. [n.d.]. A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology. https://github.com/NVIDIA/gdrcopy.

[38] NVIDIA. [n.d.]. GPUDirect RDMA: Developing a Linux Kernel Module using GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[39] NVSHMEM. [n.d.]. GPU-side API for remote data access, collectives and synchronization. http://www.openshmem.org/site/sites/default/site_files/SC2017-BOF-NVIDIA.pdf.

[40] Lena Oden and Holger Fröning. 2017. InfiniBand Verbs on GPU: a case study of controlling an InfiniBand network device from the GPU. *IJHPCA* (2017), 274–284. https://doi.org/10.1177/1094342015588142

[41] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodík, and Thomas E. Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. 663–679. https://www.usenix.org/conference/osdi18/presentation/phothilimthana

[42] Sreeram Potluri, Anshuman Goswami, Davide Rossetti, C. J. Newburn, Manjunath Gorentla Venkata, and Neena Imam. 2017. GPU-Centric Communication on NVIDIA GPU Clusters with InfiniBand: A Case Study with OpenSHMEM. In *24th IEEE International Conference on High Performance Computing, HiPC 2017, Jaipur, India, December 18-21, 2017*. 253–262. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8287756

[43] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. 2013. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*. 80–89. https://doi.org/10.1109/ICPP.2013.17

[44] Davide Rossetti and Elena Agostini. [n.d.]. How to make your life easier in the age of exascale computing using NVIDIA GPUDirect technologies. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9653-how-to-make-your-life-easier-in-the-age-of-exascale-computing-using-nvidia-gpudirect-technologies.pdf.

[45] Selectel. 2018. FPGA-accelerators go into the clouds [Russian]. https://blog.selectel.ru/fpga-uskoriteli-uxodyat-v-oblaka/.

[46] Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit, and María Engracia Gómez. 2017. Application Clustering Policies to Address System Fairness with IntelâĂŹs Cache Allocation Technology. *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. https://users.elis.ugent.be/~leeckhou/papers/pact17.pdf

[47] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2014. GPUfs: Integrating a File System with GPUs. *ACM Trans. Comput. Syst.* https://doi.org/10.1145/2553081

[48] TensorFlow Light manual. [n.d.]. TensorFlow Light Delegates. https://www.tensorflow.org/lite/performance/delegates.

[49] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. 2014. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. 321–332.

[50] Amir Watad, Alexander Libov, Ohad Shacham, Edward Bortnikov, and Mark Silberstein. 2019. Achieving scalability in a k-NN multi-GPU network service with Centaur. In *The 28th International Conference on Parallel Architectures and Compilation Techniques Seatltle, WA, USA*.

[51] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: Dynamic Cache Allocation with Partial Sharing. ACM.

[52] Wang Xu. 2018. Hardware Acceleration over NFV in China Mobile. https://wiki.opnfv.org/download/attachments/20745096/opnfv_Acc.pdf?version=1&modificationDate=1528124448000&api=v2.

[53] Yann LeCun. [n.d.]. THE MNIST DATABASE of handwritten digits. http://yann.lecun.com/exdb/mnist/.