SNAcc: An Open-Source Framework for Streaming-based Network-to-Storage Accelerators

David Volz
volz@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Torben Kalkhof
kalkhof@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Andreas Koch
koch@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

ABSTRACT

Network accessible databases are a common use case in modern data centers, often paired with pre-processing before storing results for later use. However, general purpose CPUs struggle to keep up with current Ethernet line speeds. Furthermore, in such a compute pipeline, the CPU is mostly used to manage storage accesses, wasting compute resources and communication bandwidth.

Due to their wide data path, FPGAs are very suitable for network applications. Hence, we propose an open-source framework for the seamless high-performance integration of custom FPGA-based network-to-storage accelerators. Our solution leverages the flexible communication interfaces of FPGAs, namely Ethernet and PCIe for direct access to NVMe storage, without host CPU interaction. We are able to saturate the bandwidth of both 100G Ethernet and state-of-the-art SSDs, and demonstrate our implementation in a case study performing DNN-based classification on an image stream.

CCS CONCEPTS

• **Hardware** → *Buses and high-speed links*; **External storage**; Networking hardware.

KEYWORDS

FPGA, NVMe, Network, Streaming-based acceleration, Heterogeneous computing

ACM Reference Format:
David Volz, Torben Kalkhof, and Andreas Koch. 2025. SNAcc: An Open-Source Framework for Streaming-based Network-to-Storage Accelerators . In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3731599.3767412

1 INTRODUCTION

Modern data centers handle vast amounts of data, which places high demands on bandwidth and storage capacity, particularly in databases. To meet these requirements, NVMe-based storage devices are increasingly being utilized [1]. They are attached to the PCIe bus, allowing for high-bandwidth access within a single node. For inter-node storage access, fast network links such as 100 G Ethernet are employed.

Although NVMe devices are directly accessible over the shared PCIe bus by other devices like hardware accelerators, storage access is typically managed exclusively by the host CPU. Relying on the CPU solely to transfer data from storage to accelerators is inefficient, as it underutilizes computing power. Therefore, enabling direct access from accelerators to NVMe storage is advantageous. Furthermore, CPUs often struggle to keep pace with the line rates of high-speed network connections, which can limit the bandwidth available for remote storage access. In contrast, FPGAs are well-suited for processing incoming data over the network, thanks to their wide data paths.

To address these issues, we propose *SNAcc*, a framework for custom FPGA-based network-to-storage accelerators. SNAcc facilitates direct access to NVMe devices from the FPGA without the need for host interaction. It provides NVMe access to user-defined accelerators through standard AXI4 Stream interfaces, aligning with the streaming-oriented structure of many FPGA-based hardware designs. By integrating SNAcc into the open-source TaPaSCo framework [5], we can leverage its networking capabilities, which we enhance further by implementing flow control for 100 G Ethernet.

We present three implementations within SNAcc that utilize URAM, FPGA on-board DRAM, and host DRAM to exchange data with the NVMe device. We evaluate these implementations against SPDK [22], a state-of-the-art framework, demonstrating that our fastest implementation can fully saturate the available bandwidth of current SSDs. In our case study, we use SNAcc to perform image classification on an incoming stream over 100 G Ethernet, storing the images and their classifications directly in a database on an NVMe device.

This work is structured as follows: First, we provide the necessary background on NVMe and TaPaSCo in Section 2, followed by Section 3. We then give a detailed discussion of our SNAcc implementation in Section 4. The evaluation of our cores and comparison with SPDK using synthetic benchmarks is presented in Section 5, followed by our case study in Section 6 showcasing three competitive approaches. We then discuss future extensions of this work in Section 7, before concluding in Section 8.

2 BACKGROUND

In this section, we will provide a brief overview of the TaPaSCo framework, which serves as the foundation for our work, as well as an introduction to the NVMe protocol designed for non-volatile storage devices.

to appear in (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/2025/11…$15.00
https://doi.org/10.1145/3731599.3767412

2.1 TaPaSCo

The open-source TaPaSCo framework [5] facilitates the integration of FPGA-based accelerators into heterogeneous systems, covering everything from hardware to the application layer. At the hardware level, the user only needs to provide the accelerator, known as the Processing Element (PE) in TaPaSCo, in the form of an High-Level Synthesis (HLS) kernel or a hand-written HDL core in IP-XACT format, complete with standard AXI4 interfaces. The toolchain then automatically generates platform-specific infrastructure, such as an interrupt controller and a DMA engine. The framework supports a wide range of devices, from embedded *ZYNQ* devices to data center accelerator cards like the *Alveo U280* and *VCK5000*.

On the application level, the user interacts with the platform-independent API provided by the TaPaSCo runtime and kernel module, which creates an abstraction layer for all hardware interactions. The runtime automatically manages data transfers and PE execution, requiring only a few lines of user code.

Additionally, the toolchain includes a plugin system that allows users to implement new hardware features for the supported platforms. Existing plugins provide functionalities such as 10/100 G Ethernet networking, Shared Virtual Memory [9], and AMD AI Engine [6] support across various devices. We utilize this plugin system to incorporate our proposed support for direct NVMe access in TaPaSCo, and to enhance the existing Ethernet feature with a flow control mechanism.

2.2 NVMe

In recent years, the bandwidth offered by storage devices, particularly SSDs, has increased significantly, making older protocols such as SATA and SAS inadequate. As a result, NVMe [12] has emerged as a widely adopted alternative standard. NVMe utilizes the PCIe bus as its communication medium, promising substantially higher bandwidth and lower latency for storage access.

To achieve this, the NVMe protocol is built around asynchronous communication between host CPU and NVMe controller. The host CPU manages command submission and completion queues, which reside in the host's memory and are accessible to the NVMe SSD via PCIe. The CPU places commands in the submission queue and then notifies the NVMe controller, which autonomously fetches and executes the commands. Once a command is completed, the NVMe controller writes the completion status, including any error codes if necessary, to the completion queue. While there is only one queue designated for administrative commands, multiple queues are available for I/O commands.

The NVMe commands do not contain any payload data by themselves. Instead, they utilize additional memory buffers. The NVMe protocol provides two methods for describing the layout of this data.

Physical Region Pages (PRPs) are the most common way to represent these buffers. Each PRP is a 4 kB page in memory and can serve as a source or destination for the data payload of an I/O command. The NVMe command includes two *PRP entries* that point to the first and second PRP, respectively, allowing for the transfer of up to 8 kB of data directly. However, transferring only 8 kB per I/O command entails significant overhead and is often insufficient to fully utilize the bandwidth of SSDs. To address this, the NVMe protocol offers *PRP lists*, which use a memory page to hold the physical addresses of up to 512 additional PRPs. When transferring more than 8 kB in a single command, e.g. 1 MB), the first PRP entry in the command is still used to point to the initial memory page containing the data payload. However, the second PRP entry instead holds the address of a PRP list, which then contains the addresses of the remaining PRPs. For even larger transfers, PRP lists can be chained together to create a linked list by having the last of the 512 addresses point to the next PRP list.

Scatter-Gather Lists (SGLs) are a more advanced option that expand on PRP lists by allowing the chaining of variable-sized, contiguous buffers rather than being restricted to 4 kB pages. The first SGL entry included in the NVMe command can point to nearly 4 GB of contiguous memory. However, SGLs are not supported by many NVMe drives and therefore are not employed by this work.

3 RELATED WORK

Several works explore the use of NVMe with FPGA hardware acceleration, addressing various aspects of this technology.

The works FastPath [17], FastPath_MP [16], DirectNVM [24], and NoPHAE [23] focus on accelerating software access from the ARM Processing System on ZYNQ devices to NVMe SSDs directly attached to the FPGA board by offloading queue management to the FPGA fabric. In contrast to SNAcc, these systems achieve significantly lower bandwidths, ranging from 0.4 – 3.1 GB/s for sequential accesses, and do not support the direct integration of custom hardware accelerators.

DONGLE [20] offers a HLS library for accessing NVMe through HLS kernels. Like the previous systems, DONGLE requires the NVMe to be directly connected to the FPGA, achieving bandwidths of 3 GB/s for sequential reads and 1.9 GB/s for sequential writes when utilizing its largest version with a 2 MB URAM buffer. Notably, only DONGLE 2.0 [19] introduces support for NVMe SSDs that are not directly connected to the FPGA but are accessible via PCIe P2P in a larger system, which aligns with the use case of SNAcc. Unlike SNAcc, which enables autonomous access to the NVMe device, DONGLE requires the host CPU to initiate each transfer. This process involves polling status registers in the HLS core and then submitting commands to the NVMe device while referencing memory on the FPGA. The resulting synchronization overhead, along with other limitations, results in significantly reduced bandwidths of approximately 1.6 GB/s for sequential reads and 0.8 GB/s for sequential writes, considerably lower than the results presented in this work.

NVMe over Fabrics (NVMe-oF) [13] utilizes a transport protocol, such as TCP/IP or InfiniBand, to provide remote, high-bandwidth, and low-latency access to NVMe SSDs. Sakalley [2] introduced an NVMe-oF solution on an FPGA, achieving bandwidths of up to 7.6 GB/s when using *eight or more* SSDs in parallel. This work has since been expanded to support custom accelerators [21].

OpenExpress [8], FVM [10], and Qiu et al. [15] implement the NVMe controller typically found on NVMe SSDs within the FPGA. While this approach can be beneficial for emulation or virtualization, or research related to near-memory computing, it represents an approach orthogonal to SNAcc.

4 IMPLEMENTATION

In this section, we describe the various components of our implementation. Figure 1 illustrates our overall hardware designs, which utilize URAM, on-board DRAM, or host DRAM to manage the data buffer for NVMe transfers. The numbered arrows visualize the flow of data and commands within the system, which will be explained and referenced as ⓧ throughout this section.

In our setup, both the NVMe device and the FPGA are connected to the PCIe bus of the host CPU. For Direct Peer-to-Peer (P2P) accesses to function properly, permissions must be granted by the IOMMU, enabling communication between the FPGA and the NVMe device.

We start by outlining the interfacing between the user Processing Element (PE) and our NVMe subsystem on the FPGA. Next, we provide detailed insights into our newly introduced *NVMe Streamer*. Following that, we discuss the necessary modifications to TaPaSCo and our host side integration. Finally, we present our enhancement of the 100 G Ethernet interface of TaPaSCo, introducing a flow control mechanism to handle network stalls correctly.

4.1 User PE Interface

As mentioned in Section 1, FPGAs are particularly well-suited for streaming applications. Therefore, we abstract the NVMe access using a total of four standard AXI4 Stream interfaces for commands and responses. For read operations from the NVMe device, the user PE issues a read command by sending the read address and length over one stream (①a), and subsequently receives the data on another stream once the operation is complete (⑥a). For write operations, the first stream beat on the command interface represents the desired write address on the NVMe device, followed by the write data (①b). The length of the data is implicitly indicated by setting the *TLAST* signal for the last data beat. Once the write operation is finished, a token is sent on the write response stream (⑥b).

4.2 NVMe Streamer

Our *NVMe Streamer* IP, highlighted in Figure 1, orchestrates all NVMe accesses in SNAcc. After receiving a read command from the user PE on the respective stream interface (①a), the NVMe Streamer immediately notifies the NVMe controller. The NVMe controller then reads the commands from the submission queue implemented as a FIFO in our IP (②), and executes them by accessing the PRP entries (③) and retrieving or returning the data payload (④). If the read request exceeds the maximum supported read length per command of the NVMe device in use, it must be split into multiple smaller commands. We use a maximum size of 1 MB, which, while not the absolute maximum supported size of our SSD, is sufficient to saturate the available bandwidth and simplifies processing. We only request as much data as can fit in our available data buffer. Afterwards, the respective data buffer space can be reused for the next NVMe read command.

Once finished, the NVMe controller writes to the completion queue (⑤), which is implemented as a reorder buffer containing the necessary information to finalize processing for each command, along with one bit indicating its completion status. While the completion bits may be set *out-of-order*, the NVMe Streamer processes

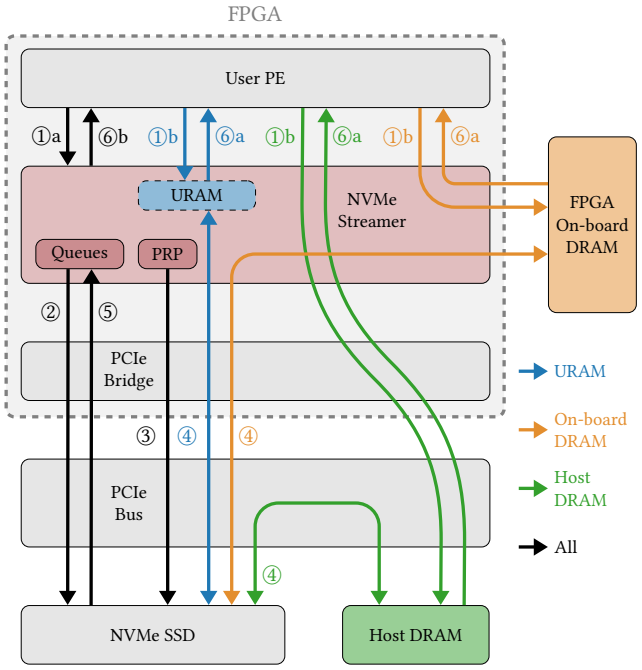


Figure 1: System Architecture of our NVMe implementation on the FPGA using URAM, on-board DRAM, or host DRAM to hold the data buffer for NVMe transfers. The numbered arrows show the command/data flow in the system.

them *in-order*. Once the next in-order command is marked as completed, the NVMe Streamer fetches the read data from the memory buffer and streams it to the user PE (⑥a). Afterwards, the respective data buffer space can be reused for the next NVMe read command.

Write commands, on the other hand, are forwarded to the NVMe device as soon as all data from the user PE has been received and buffered in the appropriate memory (①b). Large write commands are split at each 1 MB boundary into individual commands for the NVMe SSD. The command queue is shared between read and write commands and all commands are retired in the order they are received.

4.3 Buffer Memory

We implement three distinct versions of our NVMe Streamer, primarily differing in the type of memory used for payload data buffers. The first version uses on-die URAM blocks, while the other two place them either in off-chip DRAM on the FPGA board, or in host memory.

URAM. This NVMe Streamer integrates 4 MB of URAM, which are shared between read and write requests instead of using separate buffers, thus reducing usage of limited URAM resources. To simplify control logic, each new read and write command starts at a 4 kB boundary, with a maximum of 1 MB per command.

On-board DRAM. The next NVMe Streamer employs a data buffer located in the on-board DRAM of the FPGA card. This approach conserves resources on the FPGA fabric, allows for larger 64 MB buffers, and enables the complete separation of read and

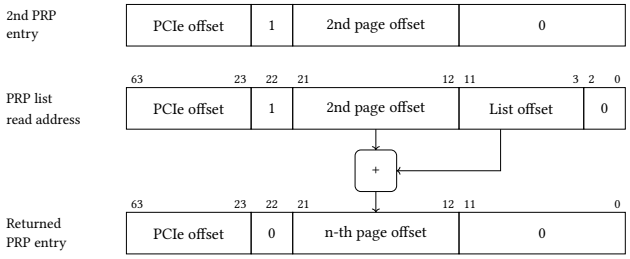


Figure 2: On-the-fly calculation of PRP entries in our URAM-based implementation.

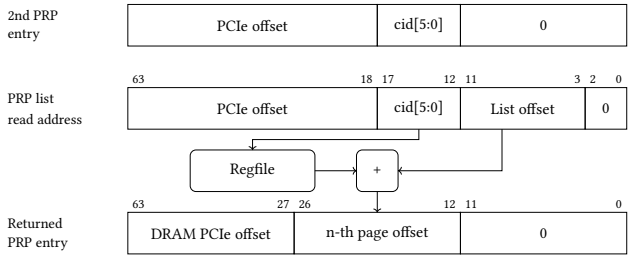


Figure 3: On-the-fly calculation of PRP entries in our DRAM-based solution. The register file holds the base offset of the second data page for each active command.

write channels into distinct buffers. To maximize DRAM bandwidth, we combine smaller memory accesses made by the NVMe controller over PCIe into a joined 4 kB burst access whenever they follow a simple incrementing pattern. Our analysis indicates that the NVMe controller typically reads memory pages contiguously.

Host DRAM. The final NVMe Streamer is nearly identical to the on-board DRAM version, with the DRAM interface connected to the PCIe bus, and thus host memory, instead. This approach conserves bandwidth on the on-board DRAM and minimizes the number of P2P PCIe accesses between NVMe controller and FPGA. The kernel driver is limited to allocating contiguous buffers of 4 MB, which introduces some overhead in address calculations, because we must combine multiple buffers to reach the same 64 MB as with on-board DRAM.

4.4 Physical Region Page Lists

To leverage the full bandwidth of NVMe, our three NVMe Streamer versions issue NVMe commands up to 1 MB in size with 256 4 kB PRPs. As explained in Section 2.2, the command itself is limited to two PRP entries; therefore, we must utilize a PRP list. The first entry in the command points to the initial 4 kB PRP of the buffer, while the second points to an additional page that contains the remaining 255 PRP entries.

A naive implementation might reserve extra memory resources for this PRP list and store the required addresses before submitting the command. However, a closer examination of the generated PRP lists shows that this overhead is unnecessary. Since our Stream Adapter uses a contiguous block of memory and streams data in order, the PRPs are arranged consecutively in memory. Thus, the n-th PRP entry can be easily calculated by adding $n \times 4096$ to the address of the first PRP entry in the list. Thus, instead of storing the PRP lists in memory, we use the address of an incoming PRP request to calculate the corresponding PRP entry on-the-fly.

Figure 2 illustrates this address translation for the URAM version. We double our address space from 4 MB to 8 MB and use the upper half for PRP entries. When issuing an NVMe command, we use the addresses of the first and second PRPs of the buffer as PRP entries, with one modification: bit 22 of the second PRP entry is set to 1, prompting the NVMe controller to read from the upper half of our address space instead. In case of PRP list reads, we can infer the second PRP and the offset within the PRP list from the address. Adding the offset to the second page's address yields the requested PRP entry, which we then return.

Figure 3 shows the address translation for the DRAM version, which introduces an additional register file to avoid doubling the already substantial 128 MB address space. In this case, the PRP lists are located in a separate, smaller address space. Unlike the URAM version, we store the address of the second PRP in a register file, which is indexed by the lower bits of the command ID – a counter that increments each time a new command is submitted. This command ID is used to form the second entry in the NVMe command. When the NVMe controller reads from the PRP list, we can again infer the address of the second PRP and the offset, allowing us to compute the corresponding PRP entry.

4.5 TaPaSCo Integration

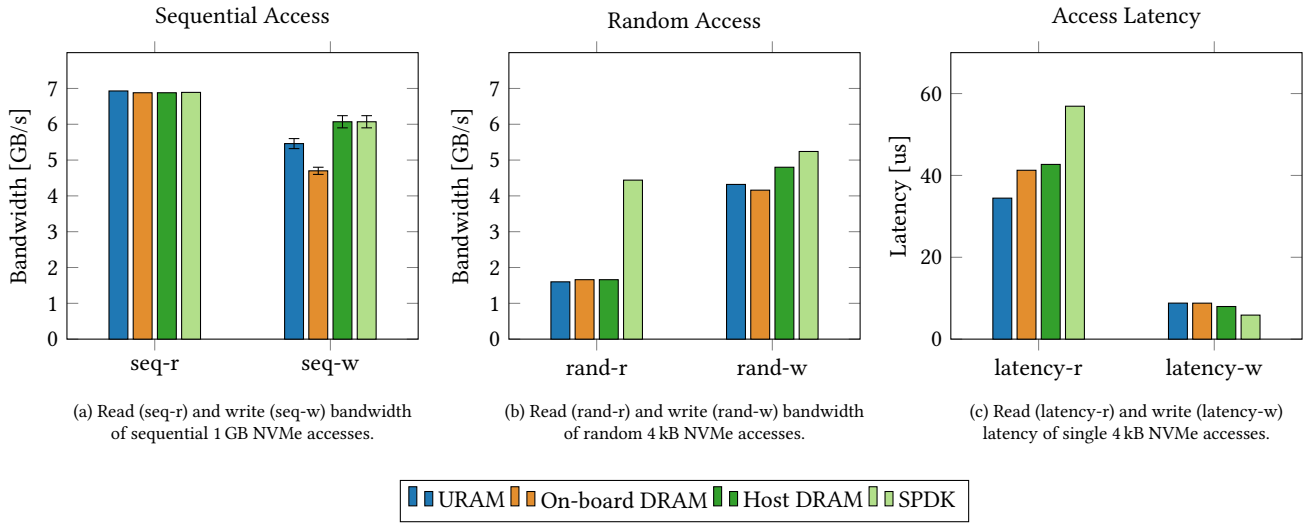
The integration of our NVMe subsystem in TaPaSCo [5] involves two main components. Firstly, we utilize the toolflow's plugin system to incorporate an additional NVMe subsystem into the block design. Currently, this plugin is available only for the *Alveo U280* and *Bittware XUP-VVH* platforms. Our NVMe Streamer operates at the 300 MHz frequency of the memory controller. The PCIe base address for the NVMe device can be specified in the NVMe Streamer but may later be overridden by the host driver. Subsequently, all necessary connections to the user PE, PCIe Bridge IP, and memory controller are established.

TaPaSCo creates a single Base Address Register (BAR) of 64 MB, and the existing address map permits the addition of the required 8 MB address space for our URAM-based implementation. However, if we opt for a variant that uses local on-board DRAM, a second BAR register has to be added once more than 8 MB of memory is utilized.

4.6 Host Side Initialization

While an FPGA only implementation is possible, e.g. by including a soft CPU in the design, we choose not to do so for two reasons: (1) Initialization is not performance-critical and only executed once, hence any hardware spent is mostly wasted; (2) Managing the NVMe admin queue and thus all NVMe configuration on the FPGA side limits system debuggability and NVMe access from other devices. This also precludes us from attaching the SSDs directly to the FPGA.

Hence, our implementation uses the TaPaSCo driver and a custom host side PCIe driver for initialization of the NVMe Streamer IP and NVMe controller after initially loading the FPGA bitstream.



(a) Read (seq-r) and write (seq-w) bandwidth of sequential 1 GB NVMe accesses.

(b) Read (rand-r) and write (rand-w) bandwidth of random 4 kB NVMe accesses.

(c) Read (latency-r) and write (latency-w) latency of single 4 kB NVMe accesses.

Figure 4: Bandwidth and latency of NVMe read and write accesses from the FPGA using SNAcc (URAM, On-board DRAM, and Host DRAM), and from the host CPU using SPDK as reference. (a) shows the bandwidth of sequential accesses with 1 GB length. The stacked bar tops showcase a fluctuating bandwidth for write accesses. (b) illustrates the bandwidth of 1 GB transfer length, but split into 4 kB accesses with randomly generated addresses on the NVMe device. In (c), we measure the latency of a single 4 kB NVMe access.

This includes setting up the NVMe admin queue and using it to create command submission and completion queues, along with dynamically configuring the NVMe Streamer and NVMe controller with the global PCIe addresses of their queues and doorbell registers.

Additionally, the TaPaSCo driver allocates the DMA-capable buffers when using the host DRAM version.

4.7 Ethernet Flow Control

When using network connections for data input or output in a streaming application, flow control is essential to provide the back pressure needed to prevent data loss. TCP is the default choice for standard network applications to achieve this. However, while TCP implementations for 100 G on FPGAs exist, they require a significant amount of FPGA resources and memory bandwidth.

Therefore, we opted for the basic Ethernet-802.3 flow control protocol, which allows an overrun receiver to send a pause packet to the sender. This protocol also works with intermediary switches, which will first pause locally before propagating the pause request further.

Once the transmission of an Ethernet frame starts, it cannot be paused. Hence, we fully buffer the frames on the sender side to prevent incomplete transmission, though this increases latency.

5 EVALUATION

In this section, we benchmark basic metrics of SNAcc, including NVMe access bandwidth and latency as well as resource utilization on the FPGA. We compare our numbers to the state-of-the-art SPDK library for host accesses, detailed in the following section. Our system setup consists of an *AMD EPYC 7302P* 16 core CPU as host, a *Samsung 990 PRO* NVMe SSD with 2 TB of storage, and an *AMD Alveo U280* FPGA.

5.1 Storage Performance Development Kit

We use the state-of-the-art Storage Performance Development Kit (SPDK) [22] library for our reference measurements, which utilizes the host CPU to access the NVMe SSD. SPDK provides high-performance, raw access to NVMe-based SSDs by shifting driver functionality into user space. This approach avoids system calls and enables zero-copy access. All required data buffers are located in pinned memory, making them accessible to both the user space application and the NVMe device through PCIe. Furthermore, SPDK optimizes latency by polling for completions instead of relying on interrupt mechanisms. In a setup with one SSD, it can leverage the full SSD bandwidth running on a single thread. As many *SSD vendors* actually use SPDK for their own benchmarking, we anticipate that SPDK will *achieve the maximum bandwidth* of the NVMe device used in our setup. It thus sets a "gold standard" reference that our hardware implementation should aim to reach.

5.2 Direct FPGA-to-NVMe Bandwidth

Figure 4a illustrates the achieved bandwidth of our URAM and DRAM-based variants, as well as the SPDK reference for sequential read (seq-r) and write (seq-w) accesses. Here, we benchmarked a single large NVMe transfer of 1 GB. In the read direction, all SNAcc variants reach a maximum bandwidth of approximately 6.9 GB/s, which is also confirmed by SPDK measurements. In contrast, only our implementation utilizing host DRAM achieves the same write bandwidth of up to 6.24 GB/s as measured by SPDK.

Interestingly, we found that the write bandwidth alternates between 5.90 GB/s and 6.24 GB/s without any intermediate values.

This behaviour is visualized in Figure 4a, showcased by the error bars. Our URAM-based implementation attains a maximum alternating bandwidth of 5.6 GB/s and 5.32 GB/s. However, this performance seems to be constrained by the PCIe P2P transfers rather than by our NVMe Streamer.

We confirmed this by tracing the DMA interface of the NVMe Streamer using an Integrated Logic Analyzer. The read accesses employed by the NVMe controller to retrieve the data to be written do not occur frequently enough to sustain a higher bandwidth, even though our end responds immediately. Furthermore, disabling the IOMMU had no affect.

However, the write bandwidth for our on-board DRAM implementation is slightly lower, varying between 4.6 GB/s and 4.8 GB/s. We believe this limitation is due to simultaneous read and write accesses to the DRAM. As the NVMe controller reads from DRAM, we are concurrently writing new data into the buffer for the next NVMe write transfers. Although we employ 4 kB bursts whenever feasible, the DRAM controller often has to switch between read and write operations, which introduces latency when responding to NVMe read requests. A potential solution could involve utilizing two DRAM controllers or distinct HBM memory banks on the FPGA. This would allow us to separate local writes from the read requests made by the NVMe controller. However, the current hardware design of TaPaSCo is limited to a single DRAM controller. We further note that the smaller 4 MB URAM buffer poses no limitation on bandwidth compared to the 64 MB DRAM buffer.

To evaluate random access performance, we transfer a total of 1 GB of data using NVMe commands, each with 4 kB payload, and utilizing randomly generated read and write addresses. As shown in Figure 4b, each of our solutions achieves approximately 1.6 GB/s in random read bandwidth (rand-r). In contrast, SPDK can reach up to 4.5 GB/s with the same submission queue depth of 64. Due to the numerous short read commands, the NVMe device heavily relies on out-of-order command execution. However, our NVMe Streamer processes command completions *in-order* to simplify the logic, which compromises performance in this specific scenario. Another limitation is the submission queue size of 64. We observe that SPDK can achieve even higher bandwidth when the submission queue size is increased. Nevertheless, we argue that the typical access pattern of streaming applications – targeted by SNAcc – involves sequential accesses of some length, rather than entirely random patterns.

In contrast, the random write performance (rand-w) of our implementation is notably competitive. When utilizing host memory for the data buffers, we achieve a bandwidth of 4.8 GB/s, compared to SPDK's 5.25 GB/s. In this case, out-of-order execution is less critical because the NVMe controller caches the data in its own DRAM before writing to slower NVM. The bandwidth of the other two SNAcc variants is slightly lower, which aligns with the findings from our sequential write benchmark.

5.3 NVMe Access Latency

Figure 4c shows the the latency of a single 4 kB read or write transfer to a random address. The measurements for our implementation start with the user PE sending the read or write command to the NVMe Streamer and stops after it received its completion. In contrast, the SPDK benchmark transfers data only between the SSD and host memory, and we measure the time from the start of the transfer to its completion. We observe that our implementation achieves significantly lower read latency compared to SPDK. The URAM-based version is the fastest, with a latency of 34 µs in contrast to SPDK's 57 µs. Our DRAM-based implementations show latencies of 41 µs and 43 µs, respectively. After command completion, the DRAM-based versions must read data from DRAM before streaming it to the user PE, which takes longer than directly reading from local URAM. For write accesses, SPDK is slightly faster; however, all four implementations maintain a latency below 9 µs.

Table 1: FPGA resource utilization of SNAcc's NVMe Streamer.

	URAM		On-board DRAM		Host DRAM	
LUT	7260	(0.6 %)	14063	(1.1 %)	12228	(0.9 %)
FF	8388	(0.3 %)	16487	(0.6 %)	13373	(0.5 %)
BRAM	-		24	(1.2 %)	17.5	(0.9 %)
URAM	4 MB	(13.3 %)	-		-	
DRAM	-		128 MB		128 MB*	

*pinned host memory

5.4 FPGA Resource Utilization

Table 1 presents the FPGA resource utilization of our three different NVMe Streamer implementations, highlighting the general trade-offs among various resources. At this point, we only compare the utilization of this specific IP implementing the main functionality of our NVMe connection. In terms of Flip-Flop (FF) and Look-Up Table (LUT) utilization, the URAM-based version consumes the least resources. However, it utilizes approximately 13 % of the URAM blocks, which may be significant for accelerators that require substantial scratch pad memory on the FPGA fabric. In comparison, the DRAM-based versions require double to triple the amount of LUTs and FFs, primarily due to the additional AXI interfaces needed for DRAM access and the register file for on-the-fly PRP computation.

These implementations do not use any URAM, but they do utilize a few BRAMs as FIFOs to buffer data for memory burst accesses. The implementation using on-board DRAM requires more BRAMs due to the additional burst logic for NVMe accesses to on-board DRAM. Additionally, we must reserve space in DRAM that cannot be used by the user application, which also necessitates a significant portion of the available bandwidth. When host memory is used, the respective memory range is pinned by the kernel driver throughout the entire execution time.

6 CASE STUDY: IMAGE CLASSIFICATION

We demonstrate the integration of SNAcc's NVMe and Ethernet extensions to TaPaSCo through a case study performing image classification. In this application, we receive image data over Ethernet, perform image classification on the FPGA, and directly write both the original image and classification data to an NVMe SSD. After initialization, the entire application operates autonomously on the FPGA without any host interaction. We view image classification

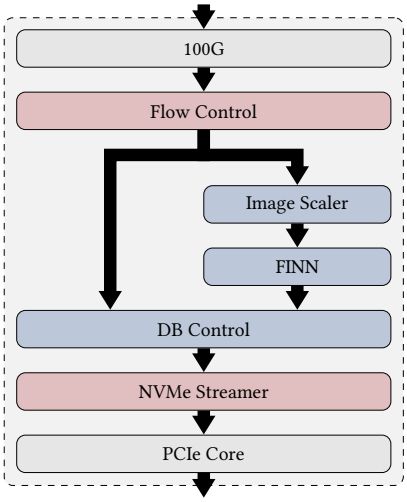


Figure 5: Dataflow on the FPGA in our image classification case study. SNAcc infrastructure IPs are marked in red, and user PEs in blue.

as a good proxy application for many streaming-based network-to-storage use cases, where high-rate input data is received, processed in the FPGA, and the results are directly written to storage.

In the following, we describe the structure of our case study in more detail, along with our reference implementations, before presenting the benchmark results.

6.1 Reference Implementations

We compare five implementations, the first three of which utilize our SNAcc framework.

FPGA. Our hardware design is illustrated in Figure 5. We receive a stream of images over 100 G Ethernet, sent by another FPGA as transmitter in our setup. Our flow control extension, described in Section 4.7, is designed to indicate backpressure to the transmitter. We assume that images are captured at a higher resolution than our classification accelerator can handle, so we scale the images down to 224x224 pixels. The classification is performed using a streaming-based accelerator, based on the MobileNet-V1 [7] network and generated with AMD FINN [18]. While there are more accurate image classification networks, such as ResNet50 [4], we chose MobileNet-V1 due to its high throughput, with the aim to truly stress our infrastructure. Finally, our database controller forwards the original image data stream, bypassing the classification pipeline, alongside with the stream of classifications from the FINN PE to the NVMe peripherals of SNAcc. We include three versions of this design utilizing our different NVMe Streamer implementations.

SPDK. Our first reference implementation is based on SPDK. If we assume direct access to the NVMe device from the FPGA is unavailable, the host will need to manage saving the resulting data. Thus, we maintain the image classification accelerator on the FPGA but transfer the image and classification data to host memory, allowing the host software to handle writing to the NVMe SSD. Again, we employ the SPDK library, as introduced in Section 5.1. To provide a fair comparison with the streaming-based SNAcc solution, we process the incoming data in batches – e.g., 32 images. Using double buffering, this approach enables us to overlap image classification with data transfers from FPGA to host memory and from the host to the NVMe device.

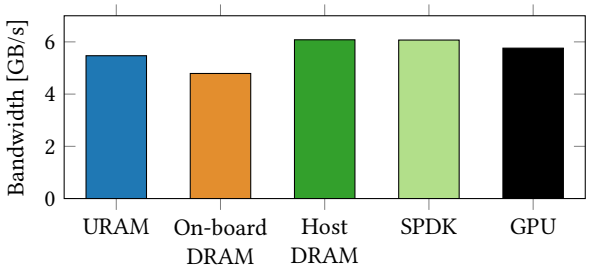


Figure 6: Bandwidth in our image classification case study using SNAcc, SPDK, and GPU as reference.

GPU. In our second reference implementation, we perform image classification on an NVIDIA A100 GPU. We utilize PyTorch [14] to evaluate batches of images on the GPU. Meanwhile, other CPU threads manage data transfers between the NIC, for which we use our FPGA, host DRAM, GPU, and NVMe SSD. To circumvent limitations related to Python's multi-threading and multi-processing capabilities, we implemented these functionalities in C++ and provided a simplified interface for Python. This solution incurs more PCIe traffic since the downscaled images must be transferred to the GPU, and the classifications must be retrieved from it. While NVIDIA offers GPUDirect Storage [11] to enable direct writing from the GPU to NVMe SSDs, we were unable to use this feature in conjunction with PyTorch and therefore reverted to SPDK.

6.2 Evaluation Results

We evaluate our case study using the identical system setup described in Section 5, with the addition of an NVIDIA A100 GPU. We measure the bandwidth by streaming 16384 images, totaling 147 GB, and present the results in Figure 6. All three variants of SNAcc nearly achieve the same bandwidth as measured for sequential writes in Section 5.2. Similarly, the SPDK-based version fully utilizes the available bandwidth of our NVMe SSD, though the CPU must retrieve results from the FPGA before writing to the SSD. The host DRAM and SPDK-based implementations achieve the best performance, reaching about 6.1 GB/s, which is equivalent to 676 frames/s. Clearly, the NVMe bandwidth is still the limiting factor since we are nowhere close to the maximum 12.5 GB/s of our 100 G connection. In comparison, the GPU reference has an overall bandwidth of 5.76 GB/s.

6.3 Considerations Beyond Bandwidth

While our case study indicates that the SPDK and host DRAM implementations achieve the highest bandwidth, this may be misleading, as other system-level factors need to be considered. Both the SPDK and GPU-based variants utilize one CPU thread running at 100 % capacity, doing nothing but moving data around. In contrast, the SNAcc-based variants operate autonomously without putting any load on the CPU after the initial setup, freeing up a CPU core to perform useful work. This impact will most likely become stronger

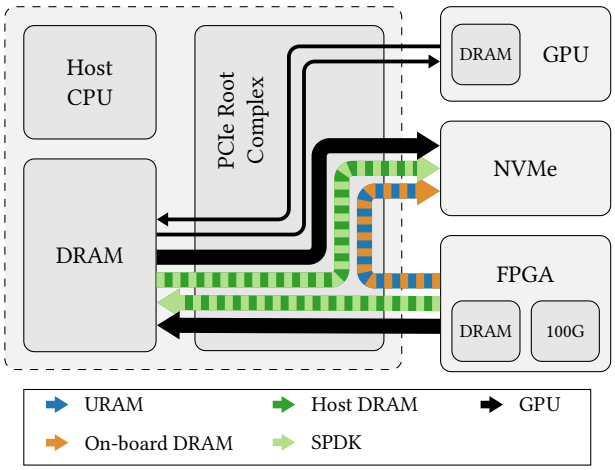


Figure 7: PCIe data transfers for the different configurations in our case study. URAM and on-board DRAM have the fewest transfers compared to GPU, which has the most.

when scaling-up to systems with multiple SSDs, which increase the host CPU load even further.

It may be notable that SPDK implements its driver functionality in user space to enhance performance, which requires root privileges to map the necessary hardware registers into user space and to pin host memory for transfer buffers. In SNAcc, all hardware interactions are encapsulated within the TaPaSCo driver and runtime as well as the provided NVMe driver. However, it is important to note that the host DRAM-based implementation of SNAcc also requires pinned memory, which cannot be utilized by the Linux kernel for other purposes.

Finally, visualizing the data transfers over the PCIe bus in Figure 7 shows that both SNAcc variants, which use memory on the FPGA board for data transfers, result in the least PCIe traffic. This approach avoids unnecessary copies and frees up PCIe bandwidth for other applications on the shared bus.

7 DISCUSSION OF LIMITATIONS AND FUTURE WORK

This work presents a functional implementation of direct FPGA-to-NVMe access, comparing three buffer strategies. Our current implementation already reaches the maximum bandwidth as validated by comparison with SPDK, and does so *without any load* on the host CPU during application runtime, while SPDK occupies one CPU thread at 100 % load. However, our single NVMe cannot keep-up with the 100G network rate, even though the PCIe bus is not fully loaded. We will tackle this in the following enhancements:

PCIe 5.0. Current NVMe SSDs support PCIe Gen5 x4, doubling the bandwidth compared to the SSD used in Sections 5 to 6. Our implementation already can accommodate these SSDs without modifications. Transitioning to newer FPGA generations will further increase PCIe bandwidth, preparing us for even faster SSDs. We will also investigate whether recent SSDs overcome the PCIe P2P traffic limitations observed in our evaluation.

Multi-SSD Support. Our design can easily be extended to access *multiple* SSDs concurrently. To this end, we will establish *separate* submission and completion queues for each SSD, either consolidating them into a single address space or providing distinct stream interfaces. Both approaches will better saturate PCIe bandwidth and mitigate the PCIe P2P limitation by hiding the latency of a single SSD.

HBM. Without using data buffers in host DRAM, *memory* will become a bottleneck in multi-SSD setups. Available URAM resources are limited, and we already observed bandwidth constraints using a single SSD with on-board DRAM. However, we can leverage HBM and distribute data buffers across *different* HBM controllers to maximize parallelism and bandwidth.

Out-of-order Retirement. The low random-read bandwidth could challenge non-streaming applications. This can be improved by increasing the submission queue size, which requires additional resources, or by implementing out-of-order retirement. Our current in-order model allows up to 64 in-flight commands but issues new commands only after the first previous command is completed, potentially leading to idle time. An out-of-order approach must appropriately handle large transfers split across multiple commands while maintaining correct processing order.

8 CONCLUSION

We have introduced the *SNAcc* framework, which enables autonomous, streaming-based access to NVMe devices for custom FPGA-based accelerators directly attached to a 100 G Ethernet connection supporting flow control. In our evaluation and image classification case study, we demonstrated that SNAcc fully utilizes the maximum available bandwidth of our NVMe SSD, as verified using SPDK, achieving 6.9 GB/s for read operations and 6.24 GB/s for write operations. However, the maximum write bandwidth was only attained by using buffers in the host memory to exchange data with the NVMe controller, due to limitations in PCIe P2P transfers and the on-board DRAM controller. To address this bottleneck, we look forward to extending our implementation for PCIe 5.0 devices and multi-NVMe setups.

SNAcc does not require any third-party IP and is publicly available as an easy-to-use extension to TaPaSCo [3].

ACKNOWLEDGMENTS

This research has been partially funded by the German Federal Ministry for Research, Technology and Space (BMRTF) with the funding IDs 01IS21007B (SCAMP) and 01IS18081C (Open6GHub). The authors would like to thank AMD Inc. for supporting their work by donations of hard- and software.

REFERENCES

[1] Mahsa Bayati, Janki Bhimani, Ronald Lee, and Ningfang Mi. 2019. Exploring Benefits of NVMe SSDs for BigData Processing in Enterprise Data Centers. In *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*. 98–106. https://doi.org/10.1109/BIGCOM.2019.00024

[2] Deboleena Sakalley [n. d.]. Using FPGAs to accelerate NVMe-oF based Storage Network. https://files.futurememorystorage.com/proceedings/2017/20170810_FW32_Sakalley.pdf. Accessed: 2025-03-27.

[3] Embedded Systems and Applications Group, TU Darmstadt. 2025. TaPaSCo on Github. https://github.com/esa-tu-darmstadt/tapasco.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385

http://arxiv.org/abs/1512.03385

[5] Carsten Heinz, Jaco Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. 2021. The TaPaSCo Open-Source Toolflow. *Journal of Signal Processing Systems* (02 May 2021). https://doi.org/10.1007/s11265-021-01640-8

[6] Carsten Heinz, Torben Kalkhof, Yannick Lavan, and Andreas Koch. 2024. TaPaSCo-AIE: An Open-Source Framework for Streaming-Based Heterogeneous Acceleration Using AMD AI Engines. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 155–161. https://doi.org/10.1109/IPDPSW63119.2024.00041

[7] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[8] Myoungsoo Jung. 2020. OpenExpress: Fully Hardware Automated Open Research Framework for Future Fast NVMe Devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 649–656. https://www.usenix.org/conference/atc20/presentation/jung

[9] Torben Kalkhof and Andreas Koch. 2022. Direct Device-to-Device Physical Page Migrations in Multi-FPGA Shared Virtual Memory Systems. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 225–234. https://doi.org/10.1109/FPL57034.2022.00043

[10] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. 2020. FVM: FPGA-assisted Virtual Device Emulation for Fast, Scalable, and Flexible Storage Virtualization. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 955–971. https://www.usenix.org/conference/osdi20/presentation/kwon

[11] NVIDIA [n. d.]. GPUDirect Storage. https://docs.nvidia.com/gpudirect-storage/. Accessed: 2025-03-27.

[12] NVMe [n. d.]. NVMe Specifications Overview. https://nvmexpress.org/specifications/. Accessed: 2025-03-27.

[13] NVMe-oF [n. d.]. NVMe over Fabrics Specification. https://nvmexpress.org/specification/nvme-of-specification/. Accessed: 2025-03-27.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR* abs/1912.01703 (2019). arXiv:1912.01703 http://arxiv.org/abs/1912.01703

[15] Yunhui Qiu, Wenbo Yin, and Lingli Wang. 2022. A High-Performance and Scalable NVMe Controller Featuring Hardware Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 5 (2022), 1344–1357. https://doi.org/10.1109/TCAD.2021.3088784

[16] Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, and Mikel Luján. 2020. FastPath_MP: Low Overhead & Energy-efficient FPGA-based Storage Multi-paths. *ACM Trans. Archit. Code Optim.* 17, 4, Article 37 (Nov. 2020), 23 pages. https://doi.org/10.1145/3423134

[17] Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, and Mikel Luján. 2018. FastPath: Towards Wire-Speed NVMe SSDs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 170–1707. https://doi.org/10.1109/FPL.2018.00036

[18] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. 2016. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. *CoRR* abs/1612.07119 (2016). arXiv:1612.07119 http://arxiv.org/abs/1612.07119

[19] Linus Y. Wong, Jialiang Zhang, and Jing Li. 2024. DONGLE 2.0: Direct FPGA-Orchestrated NVMe Storage for HLS. *ACM Trans. Reconfigurable Technol. Syst.* 17, 3, Article 45 (Sept. 2024), 32 pages. https://doi.org/10.1145/3650038

[20] Linus Y. Wong, Jialiang Zhang, and Jing (Jane) Li. 2023. DONGLE: Direct FPGA-Orchestrated NVMe Storage for HLS. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '23)*. Association for Computing Machinery, New York, NY, USA, 3–13. https://doi.org/10.1145/3543622.3573185

[21] Xilinx [n. d.]. Stand alone NVMeOF Acceleration Solution. https://www.xilinx.com/publications/solution-briefs/NVMe-oF%20SolutionBrief%20V5.pdf Accessed: 2025-03-27.

[22] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 154–161. https://doi.org/10.1109/CloudCom.2017.14

[23] Zeng Zhibin, Chen Yu, Qu He, Lou Yongchen, and Bai Lei. 2024. A high performance NVMe host logic engine based on dynamically configurable queues and co-design of NVMe and PCIe. *IEICE Electronics Express* 21, 7 (2024), 20240004–20240004. https://doi.org/10.1587/elex.21.20240004

[24] Yu Zou, Amro Awad, and Mingjie Lin. 2022. DirectNVM: Hardware-accelerated NVMe SSDs for High-performance Embedded Computing. *ACM Trans. Embed. Comput. Syst.* 21, 1, Article 9 (Feb. 2022), 24 pages. https://doi.org/10.1145/3463911