OpenExpress: Fully Hardware Automated Open Research Framework for Future Fast NVMe Devices

Myoungsoo Jung, *KAIST*

https://www.usenix.org/conference/atc20/presentation/jung

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

OpenExpress: Fully Hardware Automated Open Research Framework for Future Fast NVMe Devices

Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory
Korea Advanced Institute of Science and Technology (KAIST)
http://camelab.org

Abstract

NVMe is widely used by diverse types of storage and non-volatile memories subsystems as a de-facto fast I/O communication interface. Industries secure their own intellectual property (IP) for high-speed NVMe controllers and explore challenges of software stack with future fast NVMe storage cards. Unfortunately, such NVMe controller IPs are often inaccessible to academia. The research community, however, requires an open-source hardware framework to build new storage stack and controllers for the fast NVMe devices.

In this work, we present OpenExpress, a fully hardware automated framework that has no software intervention to process concurrent NVMe requests while supporting scalable data submission, rich outstanding I/O command queues, and submission/completion queue management. OpenExpress is available to download and offers a maximum bandwidth of around 7GB/s without a silicon fabrication.

1 Introduction

NVM Express (NVMe) is successful in bringing diverse solid state drive (SSD) and non-volatile memory (NVM) technologies closer to CPUs. NVMe supports many current I/O submissions and completions for higher throughput compared to traditional block storage interfaces. The storage-level firmware and host-side software can synchronize their communication over a flexible queueing method [26]. NVMe also provides a data transfer mechanism that allows the underlying NVMe devices to leverage abundant system memory resources of the host via physical region page (PRP) lists.

As NVMe devices have been widely used in a broad spectrum of computing domains [10, 17, 21], storage vendors secure their own *intellectual property* (IP) cores to build high-speed NVMe controllers. The NVMe specification stipulates these NVMe controller IP cores to be implemented in either firmware or hardware [26]. Considering the design flexibility of the underlying SSD architecture, many storage vendors and communities typically implement the controller IP cores as firmware [20,31,39]. Even though firmware-based controllers are sufficient to manage flash, their firmware execution can be a critical performance bottleneck when used in conjunction with faster NVM media, such as phase change memory (PRAM) and magnetoresistive memory (MRAM) [8,23]. Since the latency of the emerging fast NVMs is shorter than that of flash by an order of magnitude, CPU cycles to execute the firmware cannot be hidden behind or overlapped with I/O burst times of the underlying NVM backend.

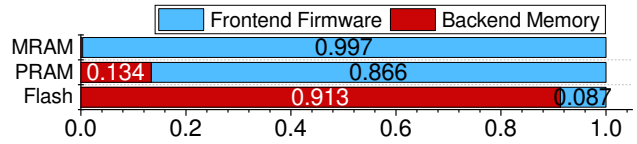


Figure 1: Decomposition comparison for execution cycles with different non-volatile memories.

To illustrate this overhead, we perform an emulation study to decompose the latency of an NVMe SSD into two parts: i) NVMe firmware execution time (*Frontend*) and ii) NVM backend access latency (*Backend*). In this study, we intermix read and write requests (1:1) and issue them to our PCIe hardware platform; this platform executes the NVMe firmware on a 1 GHz processor and emulates three different types of NVM backends, each employing 16 flash chips [14], and 16 PRAM [23] chips, and 16 MRAM chips [8], respectively. As shown in Figure 1, the access latency of flash-based backend contributes 91% of the total NVMe service time, which makes it possible to hide the front-end firmware latency by interleaving with flash I/O bursts. In contrast, the firmware execution cannot be overlapped with the I/O burst times of PRAMs and MRAMs as its execution time accounts for 86% and 99% of the total I/O latency. To address this challenge, industries have begun to employ hardware automated NVMe controller IP cores [9,24]. For example, recent state-of-the-art prototypes [9] automate NVMe control logic in an SSD to achieve 1 million I/O per second (IOPS).

Unfortunately, such NVMe hardware IP cores are prohibited to access and redesign from academia. There exist a few third-party vendors that do hardware IP business [7, 15, 16]. However, their IP cores require expensive "per-month" unitary prices, ranging from \$30K $\sim$ \$45K; the price of single-use source code is around \$100K. There are even restrictions on modifying the hardware IP and architecture, which is inadmissible in research communities. Because of such unapproachability and high costs of NVMe IP cores, many recent studies that redesign the existing storage stack, such as bypass kernel designs or NVM file systems implemented atop a fast NVMe device, are mostly performed by a simulation [11, 12, 19, 34] or an emulation [22, 37, 40]. In addition, most academic studies related to SSDs such as firmware algorithm developments [4, 6, 25, 29] and cross-layer optimizations [5,18,38], are all performed by the simulation/emulation.

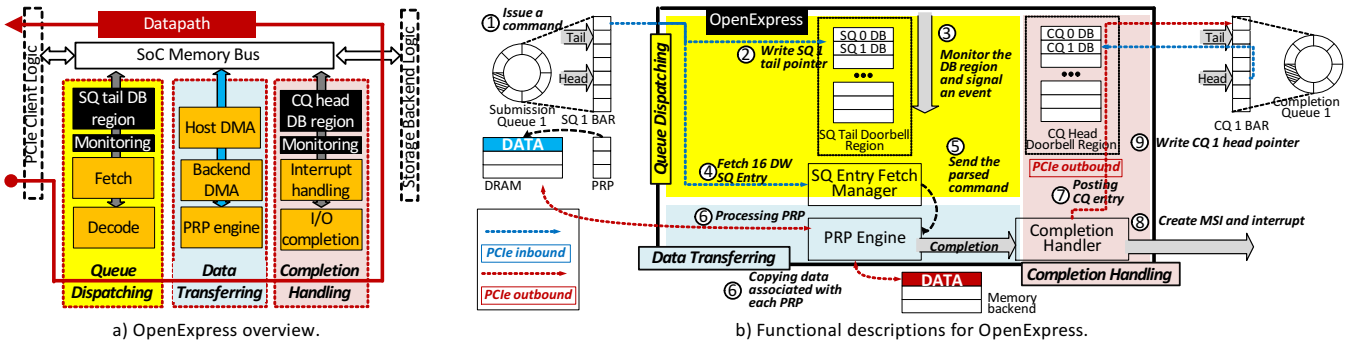


Figure 2: Overview and hardware details of the proposed OpenExpress.

While there are benefits for the simulation/emulation-based studies (e.g., fast prototyping and validation), many researchers still require real NVMe devices that they can modify to develop full-system prototypes and perform end-to-end performance evaluation with no strings attached. In this work, we propose *OpenExpress*, a framework that fully automates NVMe control logic in hardware to make it possible to build customizable NVMe devices. To the best of our knowledge, OpenExpress is only the open research framework that realizes NVMe hardware accelerator IPs [1], which does not require any software intervention to process concurrent read and write NVMe requests. Considering diverse demands of the academic research, OpenExpress leverages multiple DRAM channels and modules as storage backend and supports scalable data submission, rich outstanding NVMe commands and submission/completion queue management.

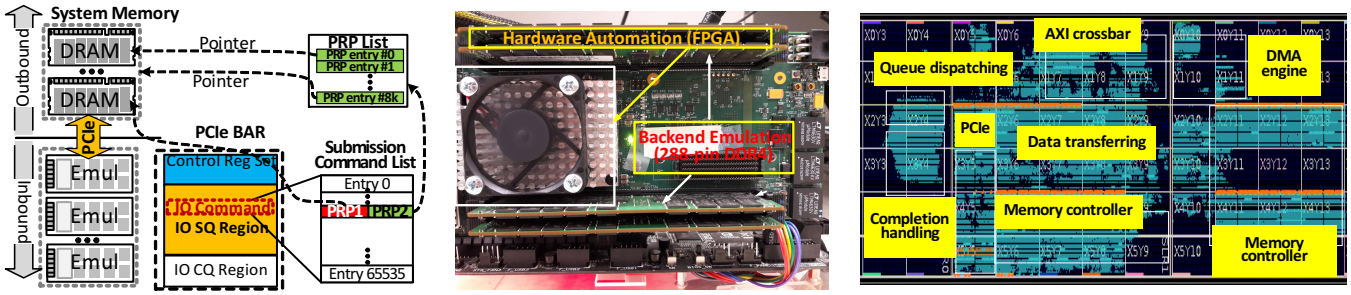
We prototype OpenExpress with a commercially available Xilinx FPGA board [3] and optimize all its logic components to operate at a high frequency. Even though it is a well-known fact that FPGAs are slower than CPUs, we demonstrate that OpenExpress can expose the true performance of backend memory modules to users over PCIe. At its peak, OpenExpress provides an access bandwidth of 7GB/s. Our evaluation also shows that OpenExpress can, on average, exhibit 76.3% better performance than an Optane SSD [13], which is one of the fast NVMe devices in the market.

2 Module Design of OpenExpress

Overview. Figure 2a illustrates a high-level view of OpenExpress, which is composed of three groups of hardware modules: i) *queue dispatching*, ii) *data transferring*, and iii) *completion handling*. At the beginning of OpenExpress's datapath, queue dispatching hardware modules fetch and decode all incoming I/O requests by monitoring multiple NVMe *submission queues* (SQs). Data transferring hardware modules then check the host system memory over *physical region page* (PRP) lists, and perform DMA for both host and backend memories. Once the data is successfully served by the backend, OpenExpress's completion handling modules manage interrupts and NVMe *completion queues* (CQs) over a set of registers (*doorbells*) mapped to a system-on-chip (SoC) internal memory bus. The details of the three different hardware module groups are explained in Figure 2b.

Queue dispatching modules. [① ∼ ⑤] of Figure 2b describe the queue dispatching functions that OpenExpress implements over hardware, including a set of procedures for automatic NVMe doorbell monitoring, command fetches, and command decodes. Specifically, when the host issues an NVMe command by pushing an SQ entry into the per-core queue [①], it should also write (i.e., ring) the corresponding entry of the doorbell (DB). Thus, OpenExpress creates a DB entry per queue by grouping two sets of DBs, each keeping track of a *tail* pointer per SQ and a *head* pointer per CQ. We map these sets of DBs, called *DB region*, to the internal SoC memory bus address space, and in parallel, we expose it to a PCIe's *base address register* (BAR) [②]. Since our fetch and decode modules can detect any DB updates (made by the host) through the memory-mapped DB region, OpenExpress is able to synchronize the host NVMe queue states with its internal NVMe queue states. Note that ringing a DB at the host side is the same as writing a PCIe packet (i.e., PCIe inbound) in NVMe. Therefore, OpenExpress can simply catch each ringing event by monitoring the memory bus address space mapped to the DB region for changes. Whenever OpenExpress observes an update of a DB that exists in the SQ-tail DB region and CQ-head DB region, it generates an event signal [③]. This event signal includes information relevant to the target SQ, and is passed to the underlying SQ entry *fetch manager*. Since multiple event signals can arise simultaneously, our fetch manager arbitrates between different NVMe queues in a round robin manner as specified by NVMe 1.4 [26]. The fetch manager then checks the delivered information and copies the 16-double-word (DW) NVMe command, associated with the target SQ entry, from the host-side system memory. After fetching the command, OpenExpress can see all relevant request information, such as the operation code (opcode), logical block address (LBA), block size, and PRPs [④]. Our fetch manager finally parses

[1] The intellectual property (IP) cores and/or register transfer level (RTL) code our OpenExpress is available to download for non-commercial and academic uses from http://openexpress.camelab.org.



a) Data transferring (PRP). b) Reference board for our prototype. c) Floorplan of OpenExpress.

Figure 3: Implementation View of OpenExpress.

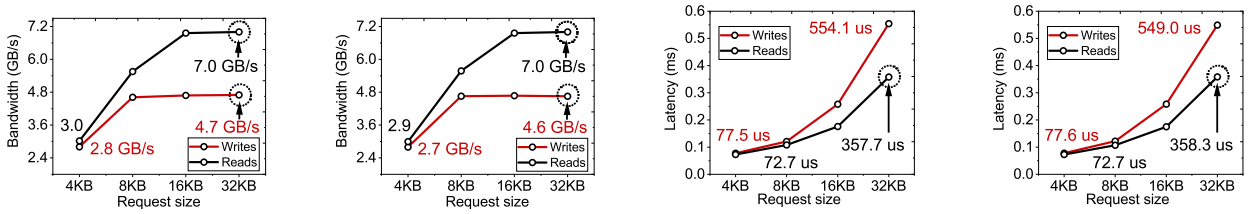
the request information and forwards it to the underlying PRP engine. To enhance responsiveness of the storage frontend (i.e., queue dispatching), the fetch manager pipelines its processing of other remaining requests with other operations of data transferring and completion handling modules [⑤].

Data transferring modules. To handle page-aligned data, our PRP engine accesses the location of the host-side system memory via a PRP, which in turn generates an out-bound PCIe packet. The PRP engine then initiates data transfer by setting the source address (i.e., host-side system memory) and the destination address (i.e., OpenExpress's memory) of our backend DMA engine [⑥]. The PRP engine copies the data from the host DRAM to our DMA engine for each 512B chunk to make device memory compatible with sector-based block I/O services. On the other hand, the DMA engine employs multiple DMA IP cores (as many as FPGA's backend memory channels) to fully parallelize data transfers. Specifically, the DMA engine splits the data of a single request into multiple small-sized data chunks and spreads them across the multiple DMA engine IP cores of the target backend. Doing so will help shorten the I/O latency of underlying NVM emulation memories. Note that the payload size of a PCIe packet is 4KB. Thus, if the request offset is not aligned to a 4KB boundary, the target data can exist in two different system memory pages of the host-side DRAM. To manage this case, as shown in Figure 3a, the PRP engine fetches the target data from two different host locations, referred by *PRP1* and *PRP2*, respectively. Specifically, if the request size is greater than a page, the PRP engine brings a page and the PRP list, indicated by PRP1 and PRP2, respectively, from the host DRAM and parses each entry of the list. The PRP engine then traverses all entries of the PRP list and transfers the data of each entry from the host system memory pages to the underlying memories. Once the data transfers are completed by the DMA engine, the PRP engine signals the completion handler, which in turn creates a completion request (CQ entry), and manages an *message-signaled interrupt* (MSI) packet corresponding to the submission request.

Completion handling modules. The functions of completion handling that we implement include I/O completion (CQ management) and interrupt processing. Based on the NVMe specification states, an NVMe queue is logically composed by a pair of an SQ and a CQ. Thus, to make the NVMe queues easy to manage, one may locate the SQ-tail DB region with the CQ-head DB region, side-by-side. However, as in-flow velocities of NVMe submissions and NVMe completions are asymmetric, a physical integration of the SQ-tail DB region and CQ-head DB region close together is not ideal from both performance and implementation angles. Thus, OpenExpress completely separates the SQ-tail DB region and CQ-head DB region. To pair OpenExpress's CQ and SQ status in an appropriate manner, we introduce a completion handler that automatically detects which CQ is engaged with the I/O request finished to process by the data transferring modules. Specifically, the detection is performed by checking the corresponding request information, which is forwarded from the SQ entry fetch manager of the queue dispatching modules. Once the completion handler finds out the appropriate CQ, the handler posts the target CQ entry by directly writing it to the corresponding host-side system memory (BAR) [⑦]. The completion handler then interrupts the host by pushing (writing) a PCIe packet to an MSI region that the host driver manages to inform the host of the I/O service completion [⑧]. The host driver later invokes its own interrupt service routine, which will notify the completion to the user who requested such I/O service. The host finally terminates the I/O service and synchronizes its CQ state with OpenExpress by updating the CQ-head DB region [⑨]. Note that queue synchronization means that the completion handler releases the target SQ and CQ entry pair from the corresponding NVMe queue. In this way, the internal CQ and SQ states of our OpenExpress can be consistent with the host-side SQ and CQ states of NVMe driver(s).

3 Hardware Prototyping

We implement OpenExpress on an Xilinx FPGA board [3] that employs an UltraScale [36] chip and PCIe Gen3 interface, which are shown in Figure 3b. We use four 288-pin DDR4 dual in-line memory modules (DIMMs) for storage-backend emulation. In our implementation, the FPGA logic modules are classified into two: frontend and backend automa-



(a) Sequential bandwidth. (b) Random bandwidth. (c) Sequential latency. (d) Random latency.

Figure 4: Bandwidth and latency performance for multi-block I/Os.

tions. The frontend automation contains most logic modules for queue dispatching and completion handling. It also exposes PCIe BAR registers and maps them into the internal advanced extensible interface (AXI) memory crossbar [1, 35]. The queue dispatching IP cores fetch and decode NVMe commands issued by the host while completion handling logic manages all aspects of I/O request completion, including interrupt handling. The completion handling logic also maintains all the contexts to automatically pair different SQs and CQs by collaborating with queue dispatching cores. On the other hand, the backend automation consists of data transferring IP cores, DDR4 memory controllers, and a DMA engine. The data transferring IP cores traverse all PRP entries and migrate the corresponding data between the host's system memory and FPGA memory modules through the DDR4 memory controllers and DMA engine.

Frequency tuning. To satisfy diverse demands of NVMe-related academic research, employing reconfigurable hardware such as FPGA is essential. However, its slow clock speed is one of challenges for our prototyping to overcome. We therefore minimize the number of connections among all logic modules to make OpenExpress operate with the highest frequency of our FPGA platform. Specifically, in our implementation, the logic modules are directly engaged one-to-one, and their input and output ports are connected in a undirectional manner. The target logic can directly retrieve any necessary information that is not forwarded from a source module by accessing the memory address space of the AXI crossbar. Note that all these hardware IP cores process different parts of I/O request(s) in a pipelined manner to improve I/O bandwidth.

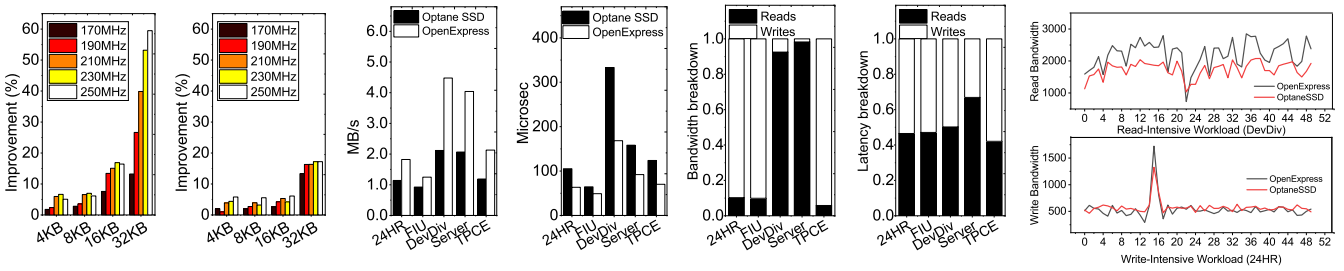
Our frequency tuning process involves gradual trial-and-error attempts to reduce the amount of route delay in a critical path. Since the delay is not straightly related to logic design, it cannot be reduced by revamping only the design itself. Instead, we perform multiple iterations of the FPGA implementation process (i.e., physical design layout decision), including a translate, map, and place-and-route (PAR) for different high-clock frequencies, ranging from 50MHz to 250 MHz. Typically, applying a higher clock frequency introduces different timing constraint violations. Unfortunately, fixing the timing violation of one core has an impact on other cores' timing characteristics. Thus, we separate the memory controller IPs from OpenExpress logic modules to the extent permitted by each each module's timing constraint, and then, we group the logic together by considering a boundary of super logic region (SLR) that contains multiple fabric modules and IP cores. The result of our design is a physical layout that is able to successfully operate at the highest frequency of the target FPGA platform as shown in Figure 3c. While all the logic modules of OpenExpress are located around PCIe and AXI crossbar, the modules associated with data transferring and DMA engine, including the memory controller IP cores, occupy only two SLRs. Lastly, the memory backend is emulated as NVMe's low-level storage complex over four channel FPGA memory IP cores. Note that, while the current prototype uses DRAM memory channels and components for the storage backend, OpenExpress IP cores can be integrated with other persistent memories.

4 Evaluation

The host system that we tested employs 8-core, 3.3GHz Intel Skylake-X processor (i9-9820X) with 32GB DDR4. Our OpenExpress prototype is attached to the host over PCIe Gen3 [30]. In this evaluation, we mainly focus on demonstrating that OpenExpress can be adopted in a real system as an open-source hardware research platform.We characterize the performance behaviors of OpenExpress (Section 4.1) and then show the performance enhancement by tuning the FPGA clock frequency (Section 4.2). We also compare the performance of OpenExpress with that of an Optane SSD [13] under diverse real workloads [32]. Note that we do not claim that OpenExpress can be faster than other fast NVMe devices. Instead, the evaluation results show that, despite the slow clock-frequency, an FPGA-based design and implementation for NVMe IP cores can offer good performance to make it a viable candidate for use in storage systems research. We use FIO [2] and execute diverse types of real workloads [32] for the user-level. We observe that the computation of a single I/O thread is insufficient to extract the maximum performance of OpenExpress. Thus, we execute FIO with a ten I/O threads, each having its own NVMe queue and I/O workload execution.

4.1 OpenExpress Characterization

Bandwidth. Figures 4a and 4b show the bandwidth trends of sequential and random I/O requests whose size varies ranging from 4 KB to 32KB, respectively. We observe that all device tests exhibit the best performance at the queue depth 8, and thus, we present only the results at such queue depth. There



(a) Read imprv. (b) Write imprv. (c) Bandwidth. (d) Latency. (e) BW. brkdown. (f) Lat. brkdown. (g) Random bandwidth.

Figure 5: Performance improvement analysis and in-depth performance comparison analysis with real workloads

is no much bandwidth difference between reads and writes at the page-size (4KB) I/O service; 3.0 GB/s and 2.8 GB/s for sequential reads and writes, respectively. As the I/O request size increases, the read and write performance improve. Note that OpenExpress reaches the maximum bandwidth with 16KB-sized I/O services; the write and read maximum bandwidth for both random and sequential are 4.6∼4.7GB/s and 7.0 GB/s, respectively. Interestingly, as the request size increases, the read bandwidth gets better than the write bandwidth. This is because of PCIe packet management issues, impacted by what we tuned for a high FPGA clock frequency, which will be analyzed in details in Section 4.2. Generally speaking, the data and command movement of reads are not congested as much as that of writes; all doorbell register updates, and command/write data payloads come together through PCIe inbound links.

Latency. Figures 4c and 4d present the latency of sequential and random I/O requests, respectively. Like the bandwidth behaviors of OpenExpress, the trend of latency curve with varying request sizes for sequential and random is similar to each other; for both reads and writes, their latency increases as the request size grows; the average latency of page-sized reads and writes is 72.7 us and 77.5 us, respectively. This is because OpenExpress exhibits the same execution latency for data transferring at a same queue depth condition. In addition, the latency of backend memory accesses with a bulk of pages does not depend on their access patterns. Even though write latency characteristics are similar to those of reads for 4∼8 KB sized requests, as increasing the I/O sizes, the latency disparity becomes more distinguishable. Specifically, the read and write latency of 32KB-sized requests is 358 us and 551.5 us, on average, respectively; for both sequential and random patterns, the writes are 54% slower than reads, on average.

Note that the latency of page-sized requests is observed with high queue depths to achieve the best bandwidth. While bandwidth is the matter on block storage, OpenExpress can reduce the latency by 62% with single queue depth operations (by sacrificing the bandwidth). Specifically, the latency of 4KB-sized requests (72.7us ∼ 77.5us) are evaluated with 8 queue depths to achieve the max bandwidth that OpenExpress can offer. Optane SSD's latency for 4KB-sized requests is 120 ∼ 150us with those queue depths. The user-level latency of 4KB-sized requests for OpenExpress is 27us ∼ 30us with a single queue-depth, which is similar to the latency that Optane SSD provides.

4.2 Frequency Tuning and Real Workload

FPGA clock frequency optimization. Figure 5a and 5b show the percentage of bandwidth improvement (as representative) by tuning the FPGA clock frequency, ranging from 150 MHz to 250 MHz. We normalize all the results to the bandwidth that we can achieve by executing OpenExpress at 150 MHz. As shown in Figure 5b, the write bandwidth improvement is 8.6%, on average. Our frequency tuning exhibits higher bandwidth improvements in cases of larger size requests. This is related to the amount of data (per request) that OpenExpress should move between the host and underlying memory backend. The small requests (4 KB∼16 KB) have a few system addresses that OpenExpress needs to be involved in parsing the PPR entries and perform DMA, which implies that there are more operations related to NVMe protocol management of SQ, CQ, doorbell, etc. To handle the large requests (32 KB), data transferring modules successfully remove the software intervention related to all PRP traversing, address parsing and DMA control. They are also well operated with other hardware automation modules in a pipeline manner, which can improve the overall performance by 17.1%. We observe that the performance improvement of reads are more significant (Figure 5a). The read bandwidth improvement for 4 KB, 8 KB, 16 KB and 32 KB-sized requests is 5%, 6.1%, 16.4% and 59.4%, on average, respectively. All NVMe-related traffic for the reads are not congested as much as that of the writes thanks to the dual-simplex design of PCIe. Specifically, as the operations of queue dispatching modules are mostly engaged with PCIe inbound links, data transferring modules can write up data to the host system memory in serving the read requests via PCIe outbound links, in parallel. Because of this, our hardware automation becomes more promising to serve the large block-sized read requests than the writes.

Real workload. We compare the bandwidth and latency of OpenExpress with those of Optane SSD by executing diverse

workloads, and the results are shown in Figures 5c and 5d, respectively. For better understanding, we decompose the bandwidth and latency for each workload, which are shown in Figures 5e and 5f, respectively. The bandwidth and latency of OpenExpress are better than that of Optane SSD by 76.3% and 68.6%, on average, respectively. Since our frequency turning gives better performance on reads, the performance on read-intensive workloads (DevDiv and Server) is better than write-intensive workloads (24HR, FIU and TPCE). In particular, OpenExpress achieves more than 4 GB/s on DevDiv workload execution, which is 111.5% higher than the performance that Optane SSD provides (2.1 GB/s).

To see each of device bandwidth behaviors, Figure 5g shows the time series analysis of read-only and write-only bandwidth for DevDiv and 24HR workloads in a certain amount of I/O processing periods. As reads are dominant on DevDiv, the read-only bandwidth of OpenExpress is 27.1% better than Optane SSD. In contrast, the write-only bandwidth of OpenExpress for 24HR shows a similar or a little bit worse characteristic, compared to Optane SSD. We conjecture that Optane SSD buffers write requests into its internal DRAM, and thus, FPGA-based automation makes the performance difference between OpenExpress and Optane SSD (all logic is built on an ASIC). For both read and write bandwidth, the trends of dynamic performance curve for OpenExpress and Optane SSD are similar to each other.

5 Discussion, Related Work and Future Work

Related work. There are only a handful of frameworks that enable NVM-related storage systems research. OpenSSD [27, 28] provides flash software, which can be modified by users based on their research demands. We believe that OpenSSD offers yet an excellent opportunity to examine the diversity in flash management, but it only supports low speed SATA and non-standard interfaces. Unfortunately, their low performance (<350MB/s [33]) is not suitable for conducting research related with high-speed NVM technologies. In addition, OpenSSD does not allow users to modify hardware design and architecture. Note that OpenSSD is also not a cost-free open-source platform; Jasmine and Cosmos require around $2K and $3.5K per-device license fees, respectively.

Dragon fire card (DFC) implements a customized FPGA design, but it is only used for back-end flash controllers. All other functionalities are managed by software and a heavy operating system, which executes on multiple CPU cores. DFC is not in a public domain yet and unfortunately available for only a small, closed research community. Note that, in addition to the lack of open licensing and limitations with respect to support for fast, NVM devices, all the above frameworks require software to manage the host communication and NVMe protocol in contrast to OpenExpress.

Limits. Our hardware automated logic can remove software involvement for NVMe management, thus, providing performance inline with the expected user case – a fast PCIe storage card for NVM research. However, the automated hardware modules may require more technical efforts than what software-based NVMe utilities need to pay for a change, in cases where one needs to either apply the hardware logic to different types of FPGA technologies, or employ additional hardware/software logic. While OpenExpress implements most critical modules on both read and write paths, it requires users to modify the hardware logic to add new NVMe features and other commands for good measure. Even with these limits, we believe that OpenExpress is worthwhile to explore, as third-part NVMe IP cores are hard nut to crack to access for research purposes because of their expensive per-month unitary prices and restrictions to modify.

Future work and use cases. As an on-going research, we are updating the existing Linux storage stack and memory subsystem modules to exploit the true performance, exposed by different types of emerging NVMs. We are also fully updating OpenExpress to have a real NVM storage backend rather than DRAM emulation for a device-level study. This project requires a set of extra RTL designs and modifications to remove all firmware executions from the internal I/O path of an existing SSD. It also needs new types of NVM controllers the corresponding and physical layers to correctly handle the NVM backend complex with a low power. Lastly, we leverage a part of OpenExpress to i) build disaggregated pooled memory over a cache coherent interconnect for accelerators and ii) process data near storage class memory, which requires different computational IP cores for such new interface and in-storage processing acceleration.

6 Acknowledge

The author thanks Raja Appuswamy and Heiner Litz for shepherding this paper. The author also thanks anonymous reviewers for their constructive feedback as well as MemRay for their sample donations and technical support. This work is supported by MemRay grant (G01190170) and performed as a part of its open NVMe interface research project.

7 Conclusion

We propose an open-source hardware research platform, called OpenExpress. OpenExpress is an NVMe host accelerator IP for the integration with an easy-to-access FPGA design. We prototype OpenExpress on a commercially available Xilinx FPGA board and optimize all the logic modules to operate a high frequency. Using a thorough evaluation, we demonstrate that OpenExpress, with a maximum bandwidth of 7 GB/s, is an open research platform that can further storage system research related to fast NVM devices.

References

[1] ARM. AMBA AXI and ACE protocol specification. https://static.docs.arm.com/ihi0022/d/IHI0022D_amba_axi_protocol_spec.pdf.

[2] AXBOE, J. Flexible I/O tester. https://github.com/axboe/fio.

[3] BITTWARE. Xilinx ultrascale 3/4-length pcie board. http://www.bittware.com/wp-content/uploads/datasheets/ds-xusp3r.pdf.

[4] CHOI, W., JUNG, M., AND KANDEMIR, M. Parallelizing garbage collection with I/O to improve flash resource utilization. In *the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2018).

[5] DAYAN, N., KJÆR SVENDSEN, M., BJØRLING, M., BONNET, P., AND BOUGANIM, L. Eagletree: exploring the design space of ssd-based algorithms. In *Proceedings of the VLDB Endowment* (2013), vol. 6, ACM, pp. 1290–1293.

[6] ELYASI, N., ARJOMAND, M., SIVASUBRAMANIAM, A., KANDEMIR, M. T., DAS, C. R., AND JUNG, M. Exploiting intra-request slack to improve ssd performance. In *the 27th International Conference on Architectural Support for Programming anguages and Operating Systems (ASPLOS)* (2017).

[7] EPOSTAR-ELECTRONICS. Meissa nvme. http://www.epostar-elec.com/products_IP_Cores_Meissa.html.

[8] EVERSPIN. MR2A16A. https://www.everspin.com/supportdocs/all.

[9] FADU. Fadu annapurna ssd controller. http://fadutec.com/wp-content/uploads/2019/08/FADU-Annapurna-Brief-073019-WEB.pdf.

[10] FLASHMEMORYSUMMIT. PCIe/NVMe in mobile devices. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2015/20150811_S101C_Baram.pdf.

[11] GOUK, D., KWON, M., ZHANG, J., KOH, S., CHOI, W., KIM, N. S., KANDEMIR, M., AND JUNG, M. Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *51th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018).

[12] HU, Y., JIANG, H., FENG, D., TIAN, L., LUO, H., AND ZHANG, S. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing (ICS)* (2011).

[13] INTEL. INTEL® OPTANE™ SSD DC P4800X SERIES. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-p4801x-brief.pdf.

[14] INTEL. PF29F64G08LCMFS.

[15] INTELLIPROP. NVMe target core. http://intelliprop.com/hardware-storage-design/ip-cores/nvme-target-ip-core-IPC-NV163-DT.htm.

[16] IP-MAKER. NVMe product overview. https://www.ip-maker.com/index.php?option=com_phocadownload&view=category&download=34:nvme_product_overview&id=6:nvm.

[17] JIN, Y. T., AHN, S., AND LEE, S. Performance analysis of nvme ssd-based all-flash array systems. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2018), IEEE.

[18] JUNG, M., AND KANDEMIR, M. Middleware - firmware cooperation for high-speed solid state drives. In *Middleware '12* (2012).

[19] JUNG, M., ZHANG, J., ABULILA, A., KWON, M., SHAHIDI, N., SHALF, J., KIM, N. S., AND KANDEMIR, M. Simplessd: modeling solid state drives for holistic system simulation. *IEEE Computer Architecture Letters 17*, 1 (2017), 37–41.

[20] KIM, J., LEE, D., AND NOH, S. H. Towards SLO complying ssds through OPS isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), USENIX Association.

[21] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Reflex: Remote flash = local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17.

[22] LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., BJØRLING, M., AND GUNAWI, H. S. The case of FEMU: cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)* (2018).

[23] MICRON. P8P parallel phase change memory. https://media.digikey.com/pdf/Data%20Sheets/Micron%20Technology%20Inc%20PDFs/NP8P128Ax60E_Rev_K.pdf.

[24] MICROSEMI. Flashtec NVMe Controllers. https://www.microsemi.com/product-directory/storage/3687-flashtec-nvme-controllers.

[25] MURUGAN, M., AND DU, D. Rejuvenator: A static wear leveling algorithm for nand flash memory with minimized overhead. In *27th Symposium on Mass Storage Systems and Technologies (MSST)* (2011).

[26] NVM EXPRESS INC. NVM express base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf, 2017.

[27] OPENSSDTEAM. Cosmos openssd platforms. http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Platform.

[28] OPENSSDTEAM. Jasmine openssd platforms. http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform.

[29] PARK, J. K., LEE, J.-Y., AND NOH, S. H. Divided disk cache and ssd ftl for improving performance in storage. *JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE 17*, 1 (2017), 15–22.

[30] PCI-SIG. PCI express base specification revision 3.1a, 2015.

[31] SK HYNIX. PCIe NVMe controller firmware and drivers. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2015/20150813_FJ31_Parepalli.pdf.

[32] SNIA. Block I/O traces. http://iotta.snia.org/traces/list/BlockIO.

[33] SONG, Y. H., JUNG, S., LEE, S.-W., AND KIM, J.-S. A pcie-based open source ssd platform. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2014/20140807_301B_Song.pdf.

[34] TAVAKKOL, A., GOMEZ-LUNA, J., SADROSADATI, M., GHOSE, S., AND MUTLU, O. MQSim: A framework for enabling realistic studies of modern multi-queue ssd devices. In *16th USENIX Conference on File and Storage Technologies (FAST)* (2018).

[35] XILINX. AXI Bridge for PCI Express. https://www.xilinx.com/support/documentation/ip_documentation/axi_pcie3/v3_0/pg194-axi-bridge-pcie-gen3.pdf.

[36] XILINX. Ultrascale architecture and product data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[37] YOO, J., WON, Y., HWANG, J., KANG, S., CHOI, J., YOON, S., AND CHA, J. Vssim: Virtual machine based ssd simulator. In *Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)* (2013).

[38] ZHANG, J., KWON, M., GOUK, D., KOH, S., LEE, C., ALIAN, M., CHUN, M., KANDEMIR, M. T., KIM, N. S., KIM, J., AND JUNG, M. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[39] ZHANG, J., KWON, M., SWIFT, M., AND JUNG, M. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (2020), USENIX Association.

[40] ZHANG, Y., PRASATH ARULRAJ, L., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)* (2012).