BEAM

Dune NVMe Storage Manual

Project	DuneNvme
Date	2020-08-17
Reference	DuneNvmeStorageManual
Version	1.0.2
Author	Dr Terry Barnaby

Table of Contents

1. Introduction

This document covers the use of the Dune NVMe Storage Fpga core and the testing thereof. There is also a design level manual, DuneNvmeStorageDesign which covers detailed implementation information and general notes.

2. NvmeStorage core overview

The Dune NvmeStorage FPGA core provides the ability to store a data stream of 4 kByte blocks to one or two Nvme storage devices working in parallel at a rate of 4 GBytes/sec or greater. It then allows the stored data to be read out at a slower rate to a host computer or FPGA fabric.

The NvmeStorage core is designed to be controlled by a set of registers from a CPU. An AXI4-lite bus interface provides this ability. It's data input is designed to come from a raw FPGA binary bit stream using an 256 bit AXI4 stream interface. The read data is sent over a 128 bit AXI4 stream encoded in PCIe TLP type packets.

As well as providing data storage, the module also provides direct access to the Nvme devices from a host computer so that statistics and other housekeeping information can be obtained and the Nvme's controlled. This interface is also useful for test purposes. Two AXI4 streams, one sending a form of PCIe TLP packets to the module from the host and one sending packets to the host from the module provide this ability. It is expected that these AXI4 streams would be sent to a host's CPU over DMA stream interfaces, possibly multiplexed with packets from other FPGA systems.

Ultimately the write performance is limited by the NVMe device in use. Larger NVMe's will generally have faster write rates due to internal parallelism and higher endurance but will cost more. The NVMe's internal controller and its firmware used will be a significant factor including write latency timings.

Although the NvmeStorage core will support most Nvme drives, to simplify the NvmeStorage design, it uses a few fixed parameters that need to match the devices used. These are set as parameters to the NvmeStorage core. See the section NVMe Devices for more information on these parameters.

3. Dune System Overview

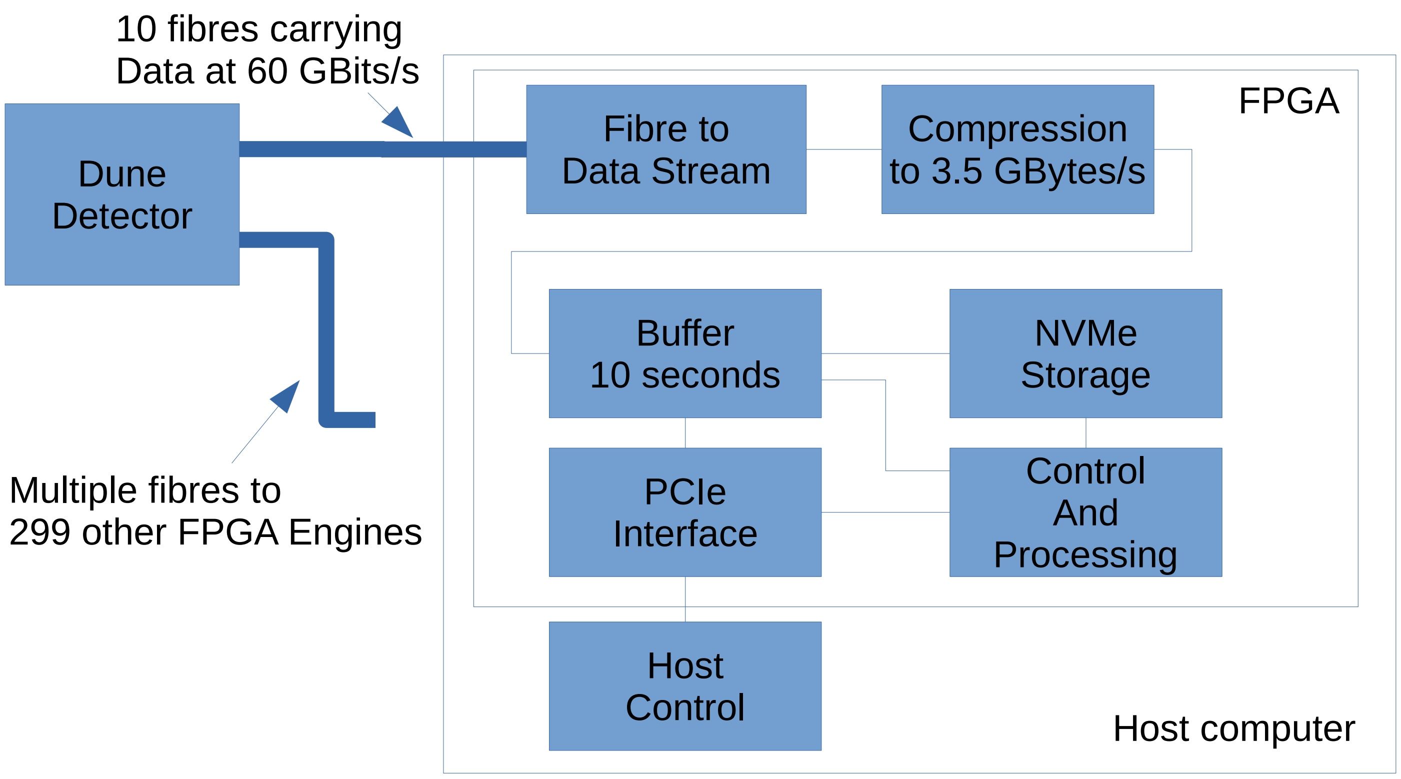
The Dune Neutrino experiment has two neutrino detection chambers. Each chamber has a matrix of wires and electronics to amplify and digitise signals captured from neutrino interactions in the chambers. The digitised data is passed through optical fibres to 300 FPGA data processing engines installed in conventional x86 computer servers. The fibres to each FPGA carry around 60 GBits/s of data plus extra bits for encoding overheads.

The Dune project will be designing a custom FPGA board to be installed into the multiple host server computers. This FPGA board will convert the optical data streams to electrical signals, compress and process this data. Currently a simple compression algorithm is proposed that provides around a 2:1 compression ratio. The resulting 3.5 GBytes/s, approx, data stream will be stored in a DDR4 RAM based buffer that can store up to 10 seconds worth of data.

The overall requirement of the NvmeStorage core is to store, real-time, the raw data stream from the Dune system and be able to, with a slower access rate, read that data. The full system is able to store 120 TeraBytes of incoming data at a rate of 600 GigaBytes per second across its multiple FPGA based data processing engines.

For each FPGA engine the source data consists of fixed sized chunks (bursts) of data that are from 20 to 200 GBytes in size. These data chunks are stored, round robin fashion, into the NVMe devices with a new chunk overwriting the oldest chunk written.

The host server computer controls the overall operation of the system via a PCIe interface. It is able to



lock a particular chunk in NVMe storage against subsequent writes if it contains data of interest. The data from this chunk can then be read out by the host computer.

3.1. Dune Raw Data

For information, the raw data from the Dune sensors consist of 2560 wires sampled with 12 bits resolution. A super packet containing 64 sets of this data will likely be generated, that includes a start time field and other control information. This will yield data packets of a size near 128 kBytes after compression.

The NvmeStorage core will store data with a block size granularity of 4 kBytes. If an NVMe error occurs, maybe due to a large block write latency or other error, the system will continue storing data, if it can, keeping the block numbers consistent. This means that there could be a set of blocks with in-valid data. Unwritten data blocks should have there contents set to 0, as long as the Nvme trim/deallocate function works correctly in the Nvme's used. So some method of determining if a packet of data is corrupt is needed. The compression algorithm used will provide variable length blocks. We suggest using a variable length packet with CRC checksum padded to a 4 kByte boundary. So a data packet for 64 * 2560 x 12 bit wire samples could look line:

Item	Length	Notes
Length	32 bits	The compressed data length in bytes (packet length is this field + timestamp + misc header length + CRC length).
Timestamp	64 bits	Timestamp for data
Misc	?	Extra header information as needed
Compressed samples	~122 kBytes	The compressed data for 64 samples of 2560 wires at 12 bit resolution.
CRC	32 bits	CRC32 checksum
Padding	?	Padding to next 4/8 kByte boundary

The actual packet format is irrelevant to the NVMe storage system. It simply stores data in blocks to a 4 kByte boundary. However, due to possible latency issues, the NVMe may have to drop data and this needs to be handled in a way that allows the host software to use the resulting data stream and understand when data has been lost. To do this the data stream needs to be blocked into some sized packet with a suitable header and dropped at that chunk size.

Two possible methods are:

1. Store and drop at the super packet (~128 kByte) level.
2. Store and drop at the storage packet 4 kBytes level, the same as the underlying NvmeStorage block size. Ideally this 4 kByte block would have some header to allow the software to understand if some of these 4 k packets are missing. This could be two 16bit values: the super packet number and the 4k packet number within the super packet.

When a 4k block is sent across the AXI4-Stream, the tlast signal will need to be set in the last clock cycle of the block.

4. NvmeStorage core operation

On the reset hardware line driven high or the reset bit in the control register set high the NvmeStorage system will reset its internal state machines and reset the external Nvme drives.

After this it is necessary to configure the Nvme drives for operation. Either the core or, if wanted, the host computer will then need to configure the NVMe devices for operation. The core can be configured to perform the configuration automatically after reset if the **UseConfigure** parameter is set to True.

If the host is to perform this task, it can instruct the core to perform this using the **Initialise** bit in the control register. Alternatively it can communicate directly with the Nvme drives to accomplish this task over the PCIe stream interface.

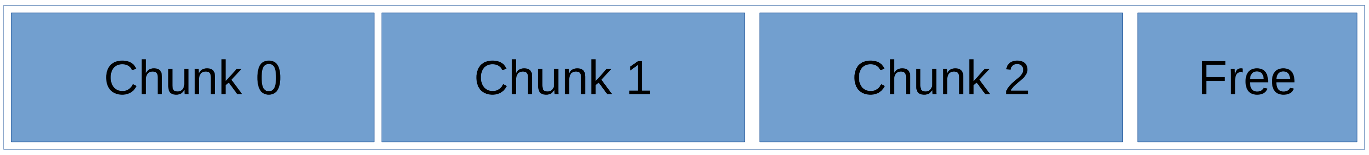
Once initialised the core is ready to accept commands from the software running on the host computer and the host can perform Nvme reads and writes.

A new data chunk capture and store session is started by setting the **DataChunkStart** and **DataChunkSize** registers to the starting block number and the number of blocks to store. Once set the **DataEnable** register bit in the **Control** register is set high to begin writing the data stream to the Nvme devices. Data will be written to the NVMe devices until either the **DataEnable** register bit is reset or the number of data blocks written equals the programmed **DataChunkSize**.

The NvmeStorage core will respond to data read requests as commanded by the host CPU writing to the read control registers or directly to the Nvme device. Note that whilst a data capture process is in progress the read process will be stalled.

4.1. NVMe Storage Access

The NvmeStorage uses a block size of 4 kBytes. The NVMe storage use a simple integer block number to



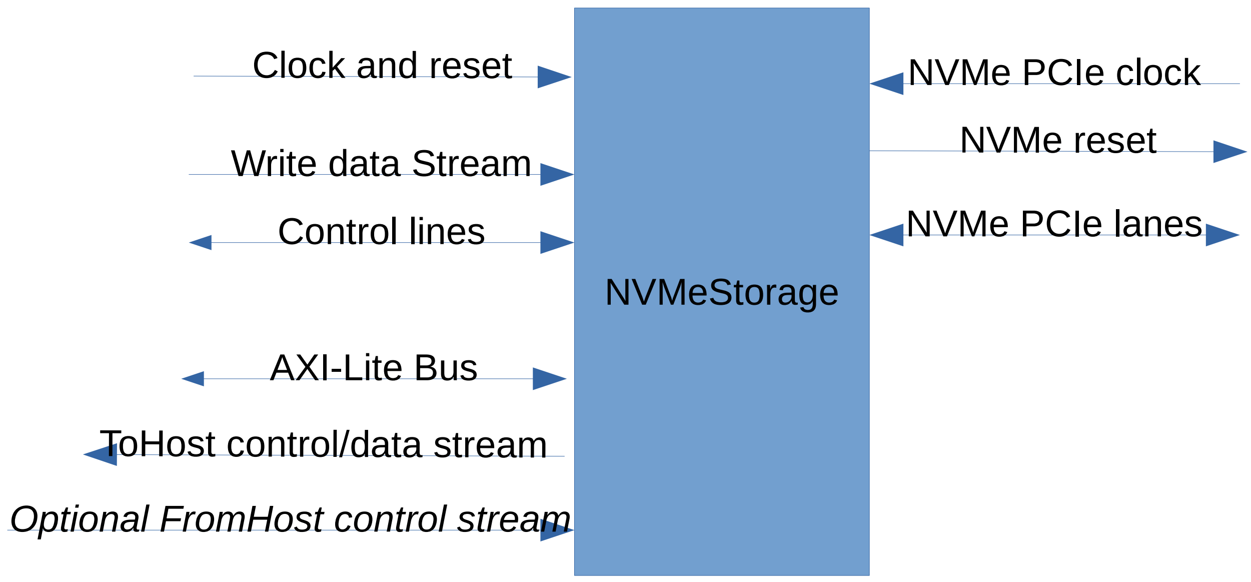
address the individually stored blocks. The host's software will likely split the data storage area into DataChunk areas of between 20 and 200 GBytes in size. It is recommended to leave a certain amount of storage at the end of the NVMe reserved for additional free blocks to assist with the NVMe's wear bad blocks system and probably reduce write latency. The NVMe devices manage the actual physical "blocks" used for each logical block and having a percentage that have not been written to provides it with a greater number of free blocks to use. Note that these blocks should not be written to or they should be trimmed/deallocated so that the NVMe drives are free to use them.

The data chunks will align to a 4 kByte boundary. It might be good to align them to an NVMe erase block boundary (possibly 1 MByte depending on NVMe), we will determine this later in the project.

The NvmeStorage modules registers are used to indicate the starting block and how many blocks to write.

5. NvmeStorage core interfaces

The NvmeStorage core has the following interfaces:



5.1. Hardware interface

The system will use two NVMe devices operating in parallel to achieve the data rate required. It may be possible in the future to use a single NVMe device operating over a Gen4 PCIe interface if Ultrascale+ or Versal FPGA's are used and a very high performance Nvme is available in the future. The overall design concept can handle that, but the state machines in the code would need changing to support 256 bit data widths.

The NVMe PCIe devices require a low jitter PCIe clock for operation. The design can use either a single shared external 100 MHz PCIe clock driving both NVMe devices or separate external 100 MHz PCIe clocks. The FPGA's differential reference clock inputs need to be located near to the GTH/GTY transceivers that will be used for NVMe PCIe communications using pins that match the requirements of the particular FPGA used. On an Ultrascale FPGA's an IBUFDS_GTE3 unit should be used to accept the differential clocks and feed the PCIe hard blocks nvme_clk and nvme_gt clock inputs. An IBUFDS_GTE4 should be used on Ultrascale+ FPGA's.

The GTH/GTY transceivers will have to be located near the PCIe hard blocks used for NVMe communications in the FPGA so they can be used by the PCIe core. See the Xilinx information on the particular FPGA that will be used. For the xcku040-ffva1156-2-e, as used on the XKU105 development board, the pins to be used are fixed and described in the “Test FPGA resources” section of the DuneNvmeStorageDesign document. Note that on the XKU105 development board the physical PCIe lanes for one of the NVMe's do not match the expected MGT ordering.

The PCIe Gen3 hard block is instantiated within the NvmeStorage module for simplicity of usage. We provide the **Platform** parameter that can be used to introduce different PCIe interfaces for different platforms/architectures or you can provide a Pcie_nvme0 and Pcie_nvme1 component to match that of the Xilinx PCIe Gen3 core.

Note that we need to be able to drive the nvme_reset_n line to the NVMe devices in order to be able to reset them as some NVMe's do not have the ability to do this over their communication interfaces.

The pin interfaces will thus be:

Item	Bits	Notes
nvme_reset_n	1	Output driving external NVMe's reset
nvme0_clk	1	From 100 MHz PCIe clock
nvme0_clk_gt	1	From 100 MHz PCIe clock for GT's
nvme0_exp_txp	4	PCIe TX lanes plus
nvme0_exp_txn	4	PCIe TX lanes neg
nvme0_exp_rxp	4	PCIe RX lanes plus
nvme0_exp_rxn	4	PCIe RX lanes neg
nvme1_clk	1	From 100 MHz PCIe clock
nvme1_clk_gt	1	From 100 MHz PCIe clock for GT's
nvme1_exp_txp	4	PCIe TX lanes plus
nvme1_exp_txn	4	PCIe TX lanes neg
nvme1_exp_rxp	4	PCIe RX lanes plus
nvme1_exp_rxn	4	PCIe RX lanes neg

The PCIe lanes will be operating at 8 GT/s speeds, so track impedance and routing will be critical.

5.2. Clocks and reset

The NvmeStorage block accepts a single 250 MHz clock that all of its control and data interfaces is synchronised to. The individual NvmeStorageUnit modules have most of their operation driven by a separate 250 MHz clock derived from the NVMe PCIe external clock input. The NvmeStorage core handles the clock domain crossings.

There is a single, active high, asynchronous reset signal that is used to reset the core and the external NVMe devices.

Item	Bits	Notes
clk	1	250 MHz clock input for interfaces
reset	1	Active high reset input

The reset line should be set high for at least 100 ms to match PCIe reset requirements. This would normally be sourced, via the PCIe interface to the hosts PERST line.

5.3. Axi4-Lite bus interface

The Axi4-lite bus interface provides host access to a set of control and status registers. It has the following ports all synchronised to the main data clock:

Item	Bits	Notes
axilOut.awready	1	Write address ready input
axilIn.awvalid	1	Write address valid input
axilIn.awaddr	32 bits, 10 bits used	Write address, address input
axilOut.wready	1	Write data ready output
axilIn.wvalid	1	Write data valid input
axilIn.wdata	32 bits	Write data, data input
axilIn.wstrb	N/A	Not used
axilIn.bready	1	Write response ready input
axilOut.bvalid	1	Write response valid output
axilOut.bresp	2	Write response, “00” for Ok is output.
axilOut.arready	1	Read address ready output
axilIn.arvalid	1	Read address valid input
axilIn.araddr	32 bits, 10 bits used	Read address, address input
axilIn.rready	1	Read data ready input
axilOut.rvalid	1	Read data valid output
axilOut.rdata	32 bits	Read data, data output
axilOut.rresp	2	Read response, “00” for Ok is output.

When reading data there will be a 8 clock cycle delay before the axilOut.rvalid is set. The delay handles the clock domain crossing latencies. See the section “NvmeStorage Alternate Bus Interface” for alternative bus implementations

5.4. NVMe write data stream

This accepts the Dune data on an AXI4-Stream interface at up to nearly 8 GBytes/s. Its specification is:

Item	Info	Notes

Clock	250 MHz	
Bus width	256 bits	8 GBytes per second interface (actually peak of 250e6 * 32 Bytes/s)
Data length	Multiple of 4 kBytes	The NVMe will expect a stream of data which is a multiple of 4 kBytes, the “last” signal being set on the last data transfer in a 4 kByte block.

The core has a degree of data buffering of the data streams, but this will not be sufficient to handle all NVMe latencies. The dataIn_ready line will be low when the buffers are full. It is expected that the FPGA’s large system RAM buffer will have sufficient space to buffer the data for about 1 second to handle this latency.

If this external buffer is getting too full, some means of dropping data needs to be provided to handle extreme Nvme latencies. We are unsure as yet what the peak latencies of the Nvme drives used will be.

The core provides the user with two methods of achieving flow control: an externally controlled method and an internal to the NvmeStorage core method.

If handled by the external logic, this can drop 4 kByte blocks based on the state of its buffer. The external logic would set the **dataDropBlocks** input low. It would likely be best to drop complete super packets to ease data integrity issues and reduce the number of super packets with data loss.

If handled, by the NvmeStorage module, then the external logic would set the **dataDropBlocks** input high when the external logic’s buffer is getting too full. When this occurs the NvmeStorage core will drop complete 4 kByte blocks from the data input FIFO’s and increment its packets dropped register. The **NumBlocksDrop** parameter defines how many blocks to drop at a time. It would likely be best to make this an even number so that blocks are dropped from both Nvme’s equally. The **BlocksLost** register can then be directly read by the host’s software at the end of a capture cycle.

We don’t expect this dropping of packets to be used unless there are significant NVMe latencies for some reason. The NVMe data will have missing packets in this case. It is assumed that the data reading software can detect this from the packet headers/trailers in the 4 kByte blocks.

Item	**Bits**	**Notes**
dataEnabledOut	out 1	This is set high when the NvmeStorage unit is enabled to save data. Useful to start and reset test signal generators etc.
dataDropBlocks	in 1	Set to 1 if the external logics buffer is getting too full. The NvmeStorage block will drop complete input blocks (two at a time) and store the number of dropped blocks in its registers while this is high. If external logic is handing the dropping of blocks this can be tied to 0.
AXI4-Stream		
dataIn_ready	out 1	Ready for data output.
dataIn.valid	in 1	Data is valid input

dataIn.last	in 1	Last word of data packet input. This is used to indicate the end of a 4 kByte block.
dataIn.data	in 256 bits	The data input

5.5. NVMe ToHost Control/Data stream

This AXI4 stream is used to send the Nvme stored data blocks to the host computer when reading the data. It is also used to return Nvme responses to requests issued on the FromHost control stream if this is used. It is a AXI4-Stream interface at up to 4 GBytes/s. Its specification is:

Item	Info	Notes
Clock	250 MHz	
Bus width	128 bits	4 GBytes per second interface (actually 250e6 * 16 Bytes/s)
Data length	128 Byte packets	The raw PCIe write data stream will be sent with the Xilinx PCIe blocks data headers

Item	Bits	Notes
hostRecv_ready	in 1	Ready for data, input.
hostRecv.valid	out 1	Data is valid output
hostRecv.last	out 1	Last word of packet. Set at the end of each data packet.
hostRecv.data	out 128 bits	The data output (bits 0-31 Dword0, bits 32-63 Dword 1)
hostRecv.keep	out 4 bits	One bit for each 32bit word that is present in the 128 bits being transferred.

The stream passes a form of PCIe TLP packets from the NVMe devices to the host. These are likely to contain 128 or 256 bytes of data depending on the PCIe max payload size in use on the NVMe interfaces. The packet structure is based on the Xilinx Pcie Gen3 hard blocks “TLP” packets.

See the section on the Communication with the Nvme for more information.

5.6. NVMe FromHost Control/Data stream

This is an optional stream to send PCIe request packets to the Nvme devices. Requests should match the Xilinx Pcie Gen3 hard blocks “TLP” request packets. Responses will be sent on the ToHost Control/Data stream with appropriate headers. This interface can be used to enquire NVMe status information and configure the NVMe devices. It is a AXI4-Stream interface at up to 4 GBytes/s. Its specification is:

Item	Info	Notes
Clock	250 MHz	
Bus width	128 bits	4 GBytes per second interface (actually 250e6 * 16 Bytes/s)

Data length	128 Byte packets	The raw PCIe write data stream needs to be sent with the Xilinx PCIe blocks data headers

Item	Bits	Notes
hostSend_ready	out 1	Ready for data, output.
hostSend.valid	in 1	Data is valid input. Set to 0 if the control interface is not used.
hostSend.last	in 1	Set to 1 in the last word of the packet.
hostSend.data	in 128 bits	The data input (bits 0-31 Dword0, bits 32-63 Dword 1)
hostSend.keep	in 4 bits	One bit for each 32bit word that is present in the 128 bits being transferred.

Please see the section “Communication with the Nvme” for more details on how to use the fromHost and toHost interfaces to communicate with the Nvme devices.

6. Control and status registers

The NvmeStorage core is controlled using a set of registers likely accessed from a host computer over the modules AXI4-Lite bus interface. It is designed for host computer configuration and control of the NVMe storage system. The interface implements the following registers.

Item	Address	Bits	Notes
ID register	0x0000	32 bits	The NvmeStorage storage core type (0x56, Top 8 bits) and the version number (8.8.8 bits)
Control	0x0004	32 bits	The control register
Status	0x0008	32 bits	The status register
TotalBlocks	0x000C	32 bits	The total number of blocks available. This just returns the value of the NvmeTotalBlocks parameter.
BlocksLost	0x0010	32 bits	The number of blocks lost if the DataDropBlocks functionality is in use.
DataChunkStart	0x0040	32 bits	The data chunk’s starting block number
DataChunkSize	0x0044	32 bits	The data chunk size in 4k blocks. Note that the capture will be limited to the total number of blocks on the device for safety as set by the NvmeTotalBlocks parameter.
Error	0x0048	32 bits	Information on any error that occurred
NumBlocks	0x004C	32 bits	The number of blocks written
TimeUs	0x0050	32 bits	The time in microseconds since the start of the

			capture
PeakLatencyUs	0x0054	32 bits	The peak block write latency in microseconds for a block write request
ReadControl	0x0080	32 bits	Read data control
ReadStatus	0x0084	32 bits	Read data status
ReadBlock	0x0088	32 bits	The starting 4 kByte read block number
ReadNumBlocks	0x008C	32 bits	The number of blocks to read
ReadError	0x0090	32 bits	Information on any error that occurred
ReadBlocksProc	0x0094	32 bits	Current NVMe read bock
ReadBlocksDone	0x0098	32 bits	NVMe read bock number completed
Nvme0 Registers. Same as main set above			
ID register	0x0100	32 bits	The NvmeStorage storage core type (0x56, Top 8 bits) and the version number (8.8.8 bits)
...			
Nvme1 Registers. Same as main set above			
ID register	0x0200	32 bits	The NvmeStorage storage core type (0x56, Top 8 bits) and the version number (8.8.8 bits)
...			

The Nvme0 and Nvme1 sets of registers directly access the individual NvmeStorageUnit engines of the two Nvme devices. The standard set of registers, starting at address 0, provide access to both Nvme0 and Nvme1 simultaneously when written too. The read values of these will be from Nvme0 only.

6.1. Control Register

The control register controls the overall system.

Item	Bits	Notes
Reset	0	Set high to reset/initialise the system
Initialise	1	Software set to 1 to start initialisation (Normally initialised by Reset)
DataEnable	2	Enable data input and writing to the NVMe's when set high. When reset to 0 data input and writing will be stopped.

The Reset bit, when set high, will generate a 100ms reset pulse that will instigate a reset of the module and the external Nvme devices.

If the core is configured to initialise on reset, then it will itself configure the Nvme devices just after reset. The Initialise bit will be set high when it has completed this process. Do not communicate with the Nvme's before this bit is set high. If the core is not configured to initialise the Nvme's on reset, then the software can either manually configure the Nvme's or set the Initialise bit high to instigate this process.

6.2. Status Register

Item	Bits	Notes
Reset	0	When high reset is in progress
Initialised	1	When high the system has completed initialisation
DataEnabled	2	Data capture is enabled
Completed	3	Data capture has completed
PciePhyReady	30	The PCIe PHY is ready
PcieLinkUp	31	The PCIe link is up

6.3. The Error register

If no errors occur the Error register (either the write or read engines error register) will be set to 0. If the Nvme device returns an error status the botton 16 bits will be set to the Nvme's error status. Please see the Nvme 1.3 documentation for more details on the error values and what they mean.

6.4. Read Blocks System

The read block system will send the block data over a AXI4-Stream interface to a host computer. The AXI4-Stream would typically be connected to a Xilinx PCIe DMA core and be read using the Linux XDMA or other such driver driver.

In order to read data blocks the host computer should set the ReadBlock register to the starting block number and the ReadNumBlocks to how many blocks to read. When the ReadControl register's ReadEnable bit is set the NvmeStorage core will read the data from the Nvme devices and send the resulting data as PCIe "TLP" packets to the host over the ToHost AXI4 stream.

The top 4 bits of the address field in the Pcie packets header can be used to determine which Nvme unit the block is from. This will be 0 or 1.

It address bits 20 to 23 are set to 0xF this indicates that the packet block data. The bottom 20 bits of the address field in the Pcie packets header can be used to determine that the block's data is in the correct sequence. (The system could be changed to use 64bit addressing if needed at the expense of more logic).

The PCIe packets data length will be 4096 bytes if the NvmeRead engine has been configured to collate the Nvme's 128 Byte PCIe packets into 4096 Byte PCIe packets otherwise they will be 128 Bytes in length. Having the larger packet size reduces software processing overheads and thus is faster and more efficient.

Note that the data blocks from one Nvme will always arrive in block number order. However, the blocks from one Nvme drive may be delayed from that of the other Nvme drive depending on the processing work on-going in the individual Nvme devices.

This module also allows an NVMe block trim/deallocate function to be performed. The host computer should set the ReadBlock register to the starting block number and the ReadNumBlocks to how many blocks to trim/deallocate. The PerformTrim bit in the control register should be set and the Enable bit set. When complete the status register’s complete bit will be set.

6.4.1. Read Control Register

Item	Bits	Notes
Enable	0	Set high to start reading or trimming data from the given block number. Set low to stop reading/trimming.
PerformTrim	1	Set high to perform a trim, set low to perform a read

6.4.2. Read Status Register

Item	Bits	Notes
Enabled	0	When high indicates the reading/trimming of data is in progress.
Complete	1	Set high when the read/trim has completed

7. NvmeStorage Alternate Bus Interface

The modules bus interface type can be changed by implementing a different NvmeStorage.vhd module with the appropriate bus interface connections. This could be used to support, say, a Wishbone or IPBus interface.

The NvmeStorage mode primarily implements the bus interface, data stream splitting and NVMe control/reply stream multiplexing/demultiplexing functions for the two NVMe devices. There are only a small set of changes needed for a different bus interface as most of the register read/write and CDC logic is in the NvmeStorageUnit.vhd layer.

The NvmeStorageUnit module has a very simple bus interface comprising:

```
regWrite    : in std_logic;                    --! Enable write to register
regRead     : in std_logic;                    --! Enable read from register
regAddress  : in unsigned(5 downto 0);         --! Register to read/write
regDataIn   : in std_logic_vector(31 downto 0); --! Register write data
regDataOut  : out std_logic_vector(31 downto 0);--! Register contents
```

There are two strobes for control, a regWrite strobe and a regRead strobe, a single address and a data in and data out set of signals.

To write to a register the regAddress and regDataIn values should be set from the bus interface and the regWrite pulsed high for one or more clock cycles. To read a register the regAddress and regDataIn values should be again set from the bus interface and the regRead pulsed high for one clock cycle. In the case of a read the regDataOut will become stable after 8 clock cycles the latency due to the CDC crossing system used.

So the NvmeStorage.vhd logic should handle the following:

1. Set the regAddress and regDataIn values from the bus interface.
2. Pulse the appropriate NvmeStorageUnits regWrite strobe dependent on the write address when the regAddress and regDataIn are stable.
3. Store any writes to any control register as the overal NvmeStorageUnit's reset line is driven by bit 0 of this register.
4. When a read register operation is performed, set the buses appropriate read ready control line at least 8 cycles after the regRead line has been pulsed high.

8. Communication with the Nvme devices

This section describes how to communicate directly with the Nvme devices over the FromHost/ToHost Pcie packet streams. The DuneNvmeStorageDesign document should be consulted along with the Xilinx PCie Gen 3 hard block document PG156 and the Nvme Express Revsion 1.3 specification document. A general knowledge of how PCIe works and its TLP packets is helpful.

In essence the host computer can send Nvme register read/write requests and also send an Nvme admin or read/write data IO queued request over the FromHost AXI4 stream. The NvmeStorage's request/reply queue engine can be used to simplify sending and receiving queued requests and replies. Responses to these request packets are sent back to the host over the ToHost AXI4 Stream.

The stream passes a form of PCIe TLP packets from the NVMe devices to the host. These are likely to contain 128 or 256 bytes of data depending on the PCIe max payload size in use. The packet structure is based on the Xilinx Pcie Gen3 hard blocks "TLP" packets with some alteration to the meaning of some of the bits for usage in the NvmeStorage system. The test software uses data structures to define these packets. The following describing the main fields used in a PCIe request:

DWord (32bits)	Value	Description
0	0xNSAAAAAA	The destination address with destination identifiers. This will have the NVME identified in the topmost 4 bits, the destination stream engine in bits 24-27 and the byte address in the remaining 24 bits. Note that for read data blocks this address will wrap around as the upper address bits, above bit 23, will not be provided.
1	0	Not used
2	0x000SRCCC	S: Source Stream, R: request type and CCC: DWord count (request is 5 bits count is 11 bits)
3	0x000000TT	TT: Request tag
4 -n	...	The following DWords contain the data

When replies to requests are received from the ToHost stream the replies, if there are any, will be the raw PCIe reply packets from the NVMe devices. These are likely to contain 128 or 256 bytes of data depending on the PCIe max payload size in use. These will have the following data structure:

DWord (32 bits)	Value	Description
0	0x0BBBEAAA	B: Byte count, E: Error, A: Lower bits of the address
1	0x000SUCCC	S: To Stream, U: Status, C: DWord count
2	0x800000TT	Reply bit set and T: request tag. Bit 31 is set to 1 for replies and 0 for requests so the type of packet can be determined.
3 -n	...	The following DWords contain the data

Bit 95 of the header (DWord 2 bit 31) is set to 1 for replies and 0 for requests so the type of packet can be determined.

So to read one of the NVMe’s registers you need to send a Pcie read request packet to the appropriate NVMe device. The R read request type is “0”, the N field in the address would be 0 or 1 depending on the NVMe you wish to communicate with, the stream S in word 0 is “0” for the NVMe engines and the AAAA field would be set to the registers address. The source stream “S” in word 2 should be set to “1” (The host) and the CCC count would be set to “1” to read one 32bit value. You will receive a reply packet with the registers contents in the first DWord or data.

To send a queued request to an NVMe, you would send a PCIe write request to the address with S set to 2 (The NvmeQueue engine). The data in this write request will be the 64 Byte NVMe queue entry.

If a queued reply is sent by the NVMe, you will receive a PCIe write request to the address 0x0210xxxx the data contents of which will be the 16 byte NVMe queued reply.

Viewing the test software will help with understanding how this works. In the future a simple API library could be generated to hide the lower level details.

9. Using The NvmeStorage Module

The NvmeStorage system is made up of a number of modules implemented in VHDL source files. The top most module, that the user would interface to, is the NvmeStorage module whose interfaces are described in this document. This is implemented in the NvmeStorage.vhd file. There is also the NvmeStoragePkg.vhd file that defines overall interface’s and parameters for the system that should be included in code using the NvmeStorage module.

The NvmeStorage module has the following configurable parameters:

Parameter	Default	Description
Simulate	False	Generate simulation core for debug

Platform	Ultrascale	The platform name for platform specific code. Currently has support for “Ultrascale” and “Ultrascale+”
ClockPeriod	4 ns (250 MHz)	Clock period for timers. used for statistics information and reset timing
BlockSize	4096	The system block size. Must be a multiple of the physical Nvme block size (typically 512). Max supported size is 4096 at the moment.
NumBlocksDrop	2	The number of input blocks to drop when dataDropBlocks is high. This is triggered on the last signal of odd numbered blocks.
UseConfigure	False	The core will configure the Nvme devices just after reset when this is set to true.
NvmeBlockSize	512	The NVMe's formatted block size
NvmeTotalBlocks	104857600	The total number of 4k blocks available. This is return in the registers to aid the software’s usage of the core and also limits the maximum number of blocks that can be written in a data chunk to protect drives from excessive writes.
NvmeDoorbellStride	4	The doorbell register stride

FPGA modules/files used:

Module	File	Description
	NvmeStoragePkg.vhd	Overall module definitions
NvmeStorage	NvmeStorage.vhd	The modules top level API
	NvmeStorageIntPkg.vhd	Internal definitions for module
NvmeStorageUnit	NvmeStorageUnit.vhd	A single NVMe control engine
NvmeConfig	NvmeConfig.vhd	Configure a NVMe engine
NvmeWrite	NvmeWrite.vhd	Write the data to the NVMe with trim
NvmeRead	NvmeRead.vhd	Read data from the NVMe
NvmeQueues	NvmeQueues.vhd	Implement the shared memory NVMe request and reply queues
StreamSwitch	StreamSwitch.vhd	Packet switch between modules
NvmeStreamMux	NvmeStreamMux.vhd	Multiplex PCIe streams between the two NVMe units
PcieStreamMux	PcieStreamMux.vhd	Multiplex PCIe request and reply packets
RegAccessClockConverter	RegAccessClockConvertor.vhd	Register access CDC unit
AxisClockConverter	AxisClockConverter.vhd	AXI4-Stream CDC unit
AxisDataConvertFifo	AxisDataConvertFifo.vhd	256 bit to 128 bit data input FIFO’s
Ram	Ram.vhd	Simple BlockRAM implementation

Fifo	Fifo.vhd	Simple register and BlockRAM based FIFO
IP Blocks		
Pcie_host	Pcie_host.xci	Xilinx PCIe DMA host interface
Pcie_nvme0	Pcie_nvme0.xci	Xilinx PCIe Gen3 hard block 0
Pcie_nvme1	Pcie_nvme1.xci	Xilinx PCIe Gen3 hard block 1
Axis_clock_converter	Axis_clock_converter.xci	Xilinx ACXI4-Stream CDC
Misc for test harness		
DuneNvmeTop	DuneNvmeTop.vhd	Test harness top level. One per board type
	DuneNvmeTop.xdc	Test harness constraints. One per board type
Testdata	TestData.vhd	Test data source. Incrementing 32 bit number
NvmeSim	NvmeSim.vhd	A simple NVMe simulation for test purposes

9.1. Constraints

The DuneNvmeTop.xdc file contains the overall constraints for the test core. The IP blocks provide individual block constraints as needed.

For the PCIe Gen3 hard blocks, the IP modules normally provide the location constraints for the PCIe lane MGT’s. However on the KCU105 evaluation board used the PCIe lanes 1 and 2 (of 0, 1, 2, 3) are swapped relative to the expected Quad layout. So we have overridden the PCIe Blocks location constraints for both blocks to be consistent.

9.2. Building for other platforms

The current source code has support for the Xilinx KCU105 and HTK K800 FPGA boards with the Design Gateway AB17-M2FMC and Ospero NVMe daughter cards. There are separate directories in the source code tree holding the files for these different hardware platforms.

Most of the NvmeStorage module’s code is generic VHDL. However it does use a few Xilinx IP blocks. When building/porting to another platform these will likely need to be changed. The blocks are the PCIe Gen3 hard blocks used for NVMe interfacing and the Axis_clock_converter block. For the test harness the host’s Pcie_host interface IP block may also need changing.

See the NvmeStorageDesign manual for details.

9.3. PCIe Gen3 IP Blocks

The NvmeStorage system makes use of the Ultrascale PCIe Gen3 hard block IP to communicate with the NVMe devices. These have been configured using the Vivado GUI tool and the *.xci files so created saved as source code. These blocks will likely need replacing with slightly different IP on different Xilinx FPGA's.

The PCIe Gen3 hard block is instantiated within the NvmeStorageUnit module for simplicity of usage. We provide the **Platform** parameter that can be used to introduce different PCIe interfaces for different platforms/architectures or you can provide a Pcie_nvme0 and Pcie_nvme1 component to match that of the Xilinx PCIe Gen3 core in separate files during the build.

We only use the PCIe Gen3 hard block for communicating with the NVMe's and thus use very little of the blocks features. We have disabled most of the blocks signal ports to reduce the component's interface pins. Some of the core parameters configured include:

Parameter	Value	Description
Device/Port Type	Root Port … Complex	The PCIe interface behaves as a root port complex
Number of Lanes	4	The number of PCIe lanes used
Link Speed	8.0 GT/s	The Gen3 link speed
AXI-ST Interface width	128 bit	The AXI4-Stream width
AXI-ST Interface freq	250 MHz	The interface clock frequency
Core Clock Freq	500 MHz	The internal core clock frequency
Enable Client Tag	On	The core manages the header tags
PF0 Max Payload Size	1024 Bytes	Could be less as NVMe's often only support up to 256 Bytes. We are only using 128 Bytes at the moment.
PF0 Interrupt Pin	None	No interrupts used
Disable GT Channel Constraint	True	We override the locations of the hard block and transceivers in the overall designs constraint file.
Interface Parameters	All off	All the extra interfaces are turned off

Note that on the KCU105 evaluation board the PCIe lanes for MGT Quad 227 have lanes 1 and 2 (of 0, 1, 2, 3) swapped around from the expected layout. We thus override the Xilinx IP wizards local location constraints for these IP blocks and have added them to our main build constraints file.

9.4. Core Internal Parameters

As well as the main modules parameters provided at the NvmeStorage level, there are some module specific parameters that can be set for debug or testing purposes or for other reasons. Some of these include:

NvmeWrite/DoTrim	If set to True perform a set of trim/deallocate commands for the entire

	region to be written too at the start of a write
NvmeWrite/DoWrite	If set to false don’t actually write any data to the Nvme’s. Can be used for data input stream testing or testing the DoTrim feature.
NvmeWrite/SimWaitReply	Instead of queuing 8 x block write requests, perform one at a time
NvmeWrite/SimDelay	Input data delay after each packet for simulation tests
NvmeRead/SendFullBlock	When set to true collates the small 128 Byte PCIe packets into a 4096 Byte full block PCIe packet. Note that this would not normally pass through a PCIe hardware layer but is fine across a packet stream.

See the design document and the code for more information.

10. Testing the NvmeStorage core

The files for the NvmeStorage system provide a test software and FPGA implementation for a Xilinx KCU105 evaluation board or a HTK K800 board with either an AB17-M2FMC or Ospero dual NVMe FMC adapter board all installed in a suitable PC running Fedora31 Linux. The NvmeStorageDesign manual provides details of this setup including the hardware configuration needed and the NvmeStorageTestSoftware manual provides details of the test software.

10.1. Building the NvmeStorage test FPGA firmware

The FPGA build environment is based around using the simple “make” system to configure and optionally build the FPGA bit file using the Xilinx Vivado 2019.2 toolset. It is also possible to use the Vivado GUI to build and inspect the FPGA code once the make system has generated the initial project file and project directories.

The top level Makefile defines overall build configuration parameters that will be overridden by the parameters in a file named Config.mk if this is present. A Config-template.mk file is provided as a template. Copy this file to Config.mk and edit as needed. The three main parameters are the BOARD_NAME and CARD parameters which defines the platform to build for and thus the FPGA bitfile programming target, the Vivado tools location in VIVADO_PATH and the in VIVADO_TARGET which defines what FPGA programming target to use. There are currently two BOARD_NAME settings supported:

- KCU105: For the Xilinx KCU105 board.
- K800: For the HTK K800 board.

There are two NVMe CARD settings supported:

- AB17-M2FMC: For the Design Gateway AB17-M2FMC NVMe card.
- Opsero: For the Opsero OP47 NVMe card.

The commands to build the firmware when run form the directory containing the DuneNvme code are:

1. Change to the vivado directory: “cd vivado”

2. Create the Vivado project file: “make project”
3. Build the firmware: Either use “make all” or use “make project” followed by using the Vivado GUI with the appropriate *.xpr file so created.
4. Program the FPGA “make program”. This assumes the FPGA’s JTAG USB port is connected to the host and the VIVADO_TARGET parameter is set correctly.

The builds take place in the Projects directory.

10.2. Building the NvmeStorage test software

A simple test program along with a simple Linux FPGA access driver is provided for testing the NvmeStorage system. This is located in the “test” directory. To build the software:

1. Change to the test directory: “cd test”
2. Make sure all of the needed development packages are installed. “make installPackages“ will do this.
3. Build the Linux FPGA driver for the current kernel: “make driver”
4. Build the test program test_nvme: “make”

10.3. Running the test system

Once the hardware platform is setup, the FPGA bit file built and the software built the system can be tested.

One issue with using PCIe express FPGA interfaces in a PC hosted environment is that the BIOS and Linux kernel need to have seen the PCIe device at boot time. There are some ways around this but they depend on the particular PC’s PCIe root complexes hardware, the BIOS and the Linux kernel in use. Some options include booting the FPGA from ROM initially with a suitable PCIe design or using the Xilinx Tandem PCIe system. However on most systems a simple soft reboot of the PC once the FPGA has been loaded with the PCIe firmware suffices. Once the system is up and running with the PCIe core then normally the FPGA can be reloaded without reboot as longs as the kernel module is reloaded and the PCIe device is reset and the PCIe bus is re-scanned. If major changes to the PCIe bus layout are made, a soft reboot will be required.

So the process to perform the test from a running Linux system is:

1. Change to the vivado directory: “cd DuneNvme/vivado”
2. Program the FPGA: “make program” or use the Vivado GUI to do this from the host or a remote system”. You also may use a different method to program the FPGA as needed.
3. Reboot the Linux system when logged in as root: “reboot”
4. The Xilinx PCIe XDMA engine should show in lspci as: “Serial controller: Xilinx Corporation Device 8024”

5. Change to the test directory: “cd DuneNvme/test”
6. Load the FPGA Linux driver as root: “make driver_load”
7. Run the test program: “./test_nvme [options] <test name>”

Note the bfpga driver’s bfpga.rules file can be copied to /etc/udev/rules.d to set the permissions of the device entries for a normal user. Alternatively you can manually use chmod for this if you want to access the drive as a normal user. The device entries are: /dev/bfpga0, /dev/bfpga0-recv0 and /dev/bfpga0-send0. You can also install the driver into the system using the “make install” command in the drivers directory. However having manual driver install is useful for developing and experimenting.

The test_nvme program is a simple test program that was used during initial test work and so has lots of extra test code included. It has documentation in the DuneNvmeStorageTestSoftware manual and in the doxygen software documentation area. Some simple commands include:

- “test_nvme -s 0 -n 262144 capture”: Starts processing the FPGA TestData stream writing 1GByte
- “test_nvme -v -s 0 -n 10 -o data.bin read”: Reads 10 blocks starting at block 0 into the file data.bin validating the data blocks contents and displaying partial block contents (The -v).

11. NVMe Devices

The NvmeStorage core is designed to work with most Nvme drives that conform to the PCIe Gen3 physical interface and the NVMe 1.3 protocol. However to simplify the design some hard coded NVMe drive parameters are used and set as parameters to the block. These are:

Parameter	Default	Description
NvmeBlockSize	512	The NVMe's formatted block size. Most drives appear to support this and are by default formatted like this. Some drives support 4096 Bytes. This may be more efficient on some drives, but on the drives we have tested it doesn’t seem to make much difference when simply streaming data. Using a large block size would allow NvmeWrite to write a larger chunk at a time.
NvmeTotalBlocks	104857600	The total number of 4k blocks available. This is 400 GBytes so would leave 100 GBytes free in each 250 GByte NVMe if two 250 GByte NVMe’s are used or 300 GBytes free in each 500 GByte NVMe if two 500 GByte NVMe’s are used. This restricts the maximum number of NvmeWrite data chunk size, for protection, and is returned to the software in registers for information.
NvmeDoorbellStride	4	The doorbell register stride. Most drives use this value.

11.1. NVMe Write Performance

For the Dune NvmeStorage system we require a sustained write speed of greater than 3500 MBytes/s over 200 GBytes of data. It is difficult to get sustained write performance data, most figures just give the write to RAM cache performance. However from Internet searches it appears that as at 2020-05-22 the fastest commodity PCIe Gen4 drives have a maximum sustained write speed of 2500 MBytes per second whilst the interface can support 4400 MBytes per second. These drives use the Phison E16 controller. So at the moment using two Nvme drives in parallel on a PCIe Gen3 or Gen4 interface looks to be the best system. In the future a single NVMe on a PCIe Gen4 interface could be used.

Some performance figures are listed below.

11.2. NVMe Write latency

When a write to an NVMe block is made the NVMe controller must find an empty storage area within the SSD that it can write the data to. Generally NVMe’s use a large internal erase block size (possibly > 1 MByte) which is the minimum amount of storage area that can be erased at once. Blocks will be written into these erase blocked sized chunks and their locations stored. This makes random block writes tricky to perform and subject to latencies as the NVMe’s internal controller juggles these erase block sized chunks and where the blocks can be written to most efficiently.

The Dune NVMe storage requirements are simpler in that blocks are written sequentially to a fixed sized area of NVMe storage that can be deallocated prior to writing the data. The NvmeStorage system sends trim/deallocate requests to the Nvme for the entire data capture chunk (200 GBytes) at the start of a capture run. This allows the Nvme to get on with erasing the necessary erased blocked sized chunks prior to writing the blocks into them. This aids write performance reducing the latency to erase the blocks on the fly and perhaps juggle the block writes into available areas.

Note that on some NVMe drives, when a block is marked as deallocated it will return 0’s in the block.

However, the NVMe algorithms are unknown and block writes may fail and require reallocation due to SSD wear. This will likely result in increased write latency occasionally and probably more often as the drive ages. Unfortunately peak latency figures are not given or controlled. The NvmeStorage system relies on a large 1 second data buffer and the ability to drop the writing of blocks if a large write latency occurs. The NvmeStorage measures the peak write block latency and provides this in a register.

11.3. NVMe Drive Lifetime

One major issue with SSD storage as used in NVMe devices is that they wear out when writing data to them. The devices are normally specified with an endurance value to define this. With the very fast and large streaming data writes the Dune system uses the NVMe endurance has to be carefully considered.

For a typical commodity Samsung 970 Pro MLC 512 GByte device the wear rating is 600 TByte. Two of these drives provide 1 TByte of storage (2 x the requirement) and should be able to store roughly (2 * 600T) / 200G = 6000 data chunks before failure. So the lifetime of these drives will be approximately:

- One data chunk per day = 16 Years.
- One data chunk an hour = 250 days
- Continuous data storage =6 days

If instead 256 GByte NVMe’s are used (so that 2 x 200 GByte data chunks can be stored) then the NVMe lifetime would be haved from that above.

If instead 1T Byte NVMe's are used (so that 8 x 200 GByte data chunks can be stored) then the NVMe lifetime would be doubled from that above.

As the drives wear out blocks will become unavailable. To extend the lifetime and reduce larger latencies later in life restricting the total number of blocks used on a drive is likely to be beneficial. So say writing just 4 x 100 GByte areas on a 500 GByte drive (4 x 200 GBytes for two 500 GByte drives) thus reserving 100 GByte per drive. We assume here that the drive will use all of the blocks available. Note that the reserved block area should be marked as unused (deallocated/trimmed) which we presume they are from new.

11.4. NVMe Devices Used

The NVMe devices used during the development and testing were (all 500 GByte versions):

Manufacturer	Device	Version	Endurance	Notes
Samsung	970 Pro, 500 GByte	Gen3,1.3	600 TB	2.3 GBytes/s, 5.2W, Samsung Phoenix controller MaxPayloadSize: 256 Bytes PcieCapabilities: 0x70 DoorbellStride: 4 Num 512 Blocks: 1000215216 Actual write performance about: 2200 MBytes/s Note only supports 512 Byte blocks.
Seagate	FireCuda 520, 500 GByte	Gen4,1.3	850 TB	2.5 GBytes/s, Phison E16 Controller MaxPayloadSize: 256 Bytes PcieCapabilities: 0x80 DoorbellStride: 4 Num 512 Blocks: 976773168 Actual write performance: about 2300 MBytes/s. Has issues writing > 100GByte when partially full. Speed drops to 1000 or even 500 MBytes/s
WD	Black SN750, 500 GByte	Gen3,1.3	300 TB	2.6 GBytes/s, WD NVMe Architecture Controller MaxPayloadSize: 512 Bytes PcieCapabilities: 0xC0 DoorbellStride: 4 Num 512 Blocks: 976773168 Actual write performance: about: 1024 MBytes/s
Others				
Seagate	FireCuda 520, 1TByte	Gen4,1.3	1800 TB	4.4 GBytes/s, Phison E16 Controller
Samsung	970 EVO Plus	Gen3,1.3	300 TB	3.2 GByes/s, 9W, Samsung Phoenix controller

Seagate	BarraCuda 510	Gen3,1.3	320 TB	2.1 GBytes/s, 4.6W, Phison E12 Controller
Corsair	Force MP510	Gen3,1.3	800 TB	2 GBytes/s, 4.8W, Phison E12 Controller
Intel	Optane SSD 905P			

List of some other devices: https://en.wikipedia.org/wiki/List_of_Intel_SSDs

Notes on the devices:

- **Black SN750:** This only had around 1024 MBytes/s write performance.
- **Seagate FireCuda 520:** This has performance issues when writing > 100 GBytes to a partially full drive. Deallocating all of the space helps but keeping at least 100 – 200 GBytes free on a 500 GByte device is needed to keep write performance up.
- **Samsung 970 Pro:** This drive worked best for the Dune usage and so was used in the performance tests.

During testing work we noticed write performance issues with the **Seagate FireCuda 520** drives. After investigation and communications with the manufacturer we found out the following (rough description that may be wrong as details are hard to get hold of):

- The Seagate FireCuda 520 uses TLC NAND FLASH (3 bits per cell) rather than the MLC NAND FLASH (2 bits per cell) of the Samsung 970 Pro.
- Writing to TLC is slower than writing to MLC (maybe 500 MBytes/s vs 2 GBytes/s ?).
- To improve write performance the FireCuda dynamically configures 1/3rd of its free space as SLC cache (1 bit per cell) which can store 1/3rd of the amount as if it was configured as TLC (3 bits per cell). The write performance to the SLC cache is > 2 GBytes/s.
- As the blocks are written to the SLC cache they are copied/converted to TLC storage. This will carry on after the write has completed.
- So with a 500 GByte drive with 100 GByte previously written and in TLC it now has (400/3) = 130 GBytes of SLC cache to write to. (Not sure if this is actually sized as (400/8) = 50 GBytes ?). As the write progresses it will, over time, copy/convert this to 100 GBytes in TLC.

As new NVMe devices are developed it is expected more will move from using MLC to TLC type storage and hence probably use a similar SLC/TLC caching system. The Dune project, requiring fast real-time write performance over a large chunk of data, will thus need to be careful with the actual NVMe to use. This also may mean using much larger devices than actually needed for the raw storage.

Note also that SLC devices are faster and last longer than MLC devices which in turn are faster and last longer than TLC devices. So there is a cost vs type vs longevity trade off.

12. Performance Figures

These tests were carried out using two Samsung 570 Pro 500 GByte drives and two Seagate FireCuda 520, 500 GByte NVMe drives fitted to an Xilinx KCU104 FPGA board. The NvmeStorage firmware and test software was the pre-release 0.9.1 version and carried out on 2020-06-12 with GitId: bc89250c1fd1e0b448a26b6fa373fbabdf8f3352.

Data capture and write performance writing 200 GBytes to a deallocated NVMe device. The drives have first had all blocks deallocated and the NVMe's left for 5 minutes to completely complete erasing of the blocks. So this gives the fastest possible capture and write performance. The block size was NvmeStorage 4k for these tests (512 Byte Nvme block size).

```
Seagate FireCuda 520, 500 GByte initial performance (Issues after these writes)
Write FPGA data stream to Nvme devices. nvme: 2 startBlock: 0x00000000 numBlocks: 52428800
20:38:38.282: StartBlock:        0, DataRate: 4530.139 MBytes/s, PeakLatency:    10535 us
20:39:41.226: StartBlock: 52428800, DataRate: 4582.337 MBytes/s, PeakLatency:     3764 us


Samsung 970 Pro, 500 GByte
Simple capture test loop: 200 GByte no trim
NvmeTrim: nvme: 2 startBlock: 0 numBlocks: 52428800
NvmeTrim: nvme: 2 startBlock: 52428800 numBlocks: 52428800

nvmeCapture: Write FPGA data stream to Nvme devices. nvme: 2 startBlock: 0 numBlocks: 52428800
08:49:51.070: StartBlock:        0, DataRate: 4410.940 MBytes/s, PeakLatency:     6927 us
08:50:48.251: StartBlock: 52428800, DataRate: 4366.160 MBytes/s, PeakLatency:     9734 us
08:51:45.407: StartBlock:        0, DataRate: 4365.319 MBytes/s, PeakLatency:     9507 us
08:52:42.688: StartBlock: 52428800, DataRate: 4787.418 MBytes/s, PeakLatency:     9964 us
08:53:39.844: StartBlock:        0, DataRate: 4367.115 MBytes/s, PeakLatency:     4290 us
08:54:37.024: StartBlock: 52428800, DataRate: 4365.441 MBytes/s, PeakLatency:     3920 us
08:55:34.281: StartBlock:        0, DataRate: 4359.005 MBytes/s, PeakLatency:     7085 us
08:56:31.561: StartBlock: 52428800, DataRate: 4359.715 MBytes/s, PeakLatency:     7650 us
08:57:28.817: StartBlock:        0, DataRate: 4359.584 MBytes/s, PeakLatency:     5288 us
08:58:26.098: StartBlock: 52428800, DataRate: 4358.331 MBytes/s, PeakLatency:     7907 us
08:59:23.354: StartBlock:        0, DataRate: 4356.805 MBytes/s, PeakLatency:    10047 us
09:00:20.534: StartBlock: 52428800, DataRate: 4362.351 MBytes/s, PeakLatency:     5001 us
09:01:17.790: StartBlock:        0, DataRate: 4361.404 MBytes/s, PeakLatency:     9720 us
```

12.1. Nvme Read Performance Figures

The Nvme read performance is primarily dictated by the FPGA core to Linux software performance. The performance figures below are using the simple test software with the simple bfpga Linux driver and the Xilinx xdma core. The bfpga Linux drive in particular is simple and performs double buffering with a memory copy so is not that efficient.

```
NvmeRead: nvme: 2 startBlock: 0 numBlocks: 5242880
Read complete at: 5242880 blocks
NvmeRead: rate: 476.061220 MBytes/s
NvmeRead: nvme: 2 startBlock: 0 numBlocks: 5242880
Read complete at: 5242880 blocks
NvmeRead: rate: 467.960966 MBytes/s
NvmeRead: nvme: 2 startBlock: 0 numBlocks: 5242880
Read complete at: 5242880 blocks
NvmeRead: rate: 476.618854 MBytes/s
NvmeRead: nvme: 2 startBlock: 0 numBlocks: 5242880
Read complete at: 5242880 blocks
NvmeRead: rate: 474.601276 MBytes/s
NvmeRead: nvme: 2 startBlock: 0 numBlocks: 5242880
Read complete at: 5242880 blocks
NvmeRead: rate: 468.334678 MBytes/s
NvmeRead: nvme: 2 startBlock: 0 numBlocks: 5242880
```

12.2. Nvme Deallocation Performance

The NVMe block deallocation/trim command takes very little time to operate and return its status. Two commands are available the Nvme Write 0’s command and the Nvme Dataset management command. However, it takes significantly longer for the NVMe drive to actually erase these deallocated blocks and block erasure is necessary for a high sequential write performance. The different device types behaved in different ways. We used a simple test script: test_deallocate.sh to experiment with this. test_deallocate.sh simulates the NVMe usage in the Dune system. What this does is:

1. Either: Initially capture and write data to 800GBytes of the total 900 GBytes available on both of the the NVMe drives combined. This fills the drives with data. or deallocate all of the drives data.
2. Run a loop with an after trim delay. This loop will:
    - Trim the first 200GByte area
    - Sleep for the delay time
    - Capture and write 200 GBytes into the first 200 GByte area
    - Trim the second 200GByte area
    - Sleep for the delay time
    - Capture and write 200 GBytes into the second 200 GByte area
3. Repeat the loop increasing the delay each time.

The Seagate FireCuda 520 drives proved problematic and seem to have firmware or controller issues when writing large chunks of data. We observed:

1. If the drives were first filled with data and the above algorithm was run the write performance went down to a stable 1000 or 2000 MBytes per second and it seemed to be a binary swap between these speeds. The amount of delay after the deallocate did not seem to make a difference.
2. If the drives were first completely deallocated the performance was generally high.
3. Testing one drive in a Linux system showed that writing more than 100G to a single drive appeared to show the slow 500/1000 MBytes per second performance.
4. The usage of deallocate did not seem to make much difference while in the processing loop.
5. The drive does not update the “nuse” namespace block usage register.

The Samsung 750 Pro drives behaved much better. We observed:

1. Apart from deallocating the unused space, the deallocate command did not seem to effect performance much (4369→ 4467 MBytes/s, note includes a larger delay in the tests). We assume the drive is efficient at erasing the blocks and can do this at a rate close to the write block time.
2. It was noted that if the 200 GBye writes were made directly after each other performance dropped to around the 3729 MBytes/s level. If a 10 second delay was placed between 200 GByte captures

this did not happen. We assume the block erasing process is slightly slower than the block write process or some other housekeeping tasks are on-going.

3. The drive does update the “nuse” namespace block usage register. It can be seen decrementing after a deallocate command over a few minutes.

At this point in time we believe that using the Samsung 750 Pro 500 GByte drives seem like the best option and no deallocate commands will be necessary for normal Dune usage although it would likely be beneficial to use the NvmeRead engine to trim/deallocate the 200 GByte region after the data has been read.

It would be good to do a long term capture test to see how the performance degrades with usage and drive wear.

13. Notes

1. When running at full data rate these device could use around 25 Watt’s of heat, so cooling systems might need to be implemented. However if run for one 200 GByte sample once per day this should not be a problem.