GPU Direct DMA Using The SLAC Open Source DMA Engine

Ryan Herbst For SLAC TID Instrumentation

November 21, 2024

CPAD 2024, Knoxville TN

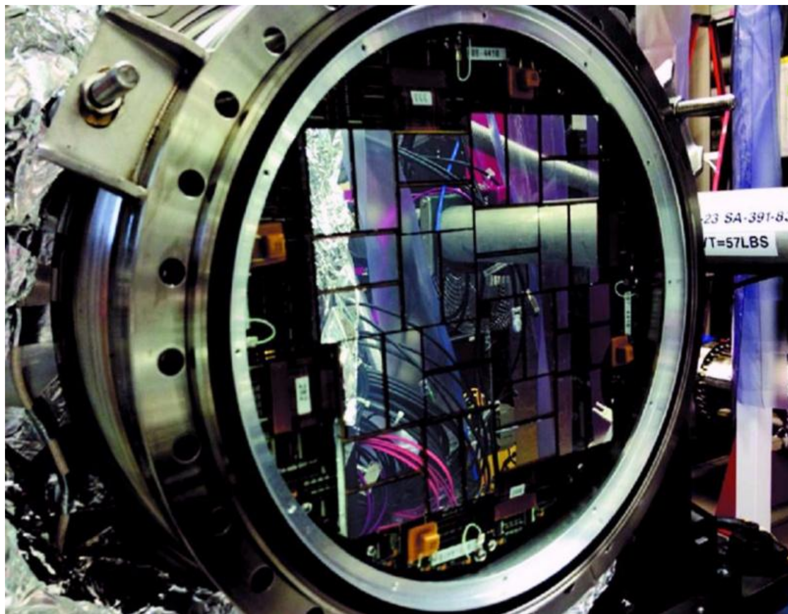
SLAC NATIONAL ACCELERATOR LABORATORY

Stanford University | U.S. DEPARTMENT OF ENERGY

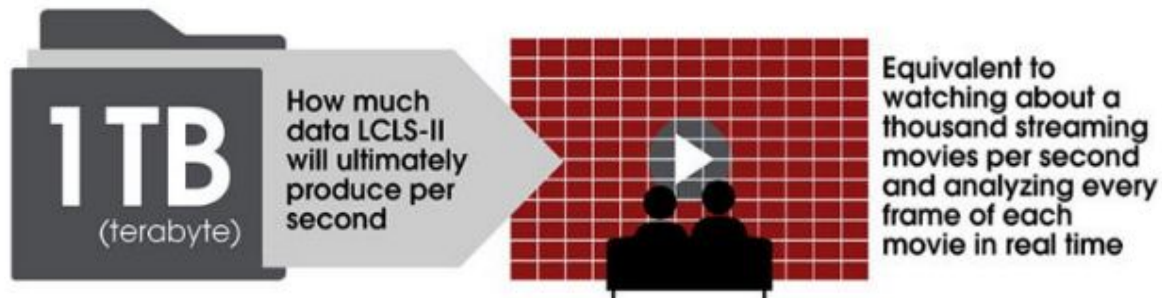
Contents

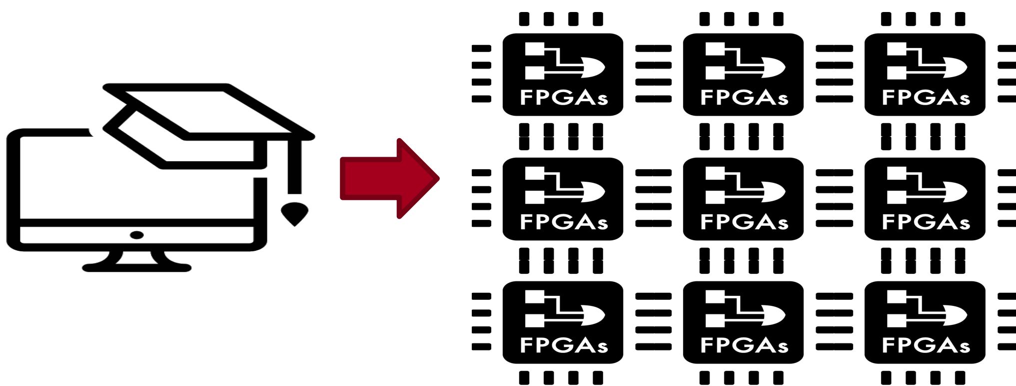
- Data Challenge & Motivation
  - LCLS Data Reduction Pipeline Architecture
- PCI-E FPGA Board Firmware Overview
- SLAC DMA Engine Overview & Direct GPU Firmware Operations
- Kernel Driver and Interaction With CUDA Host Code
- CUDA GPU Code Operation
- High Level Data Flow Options
- Limitations & Conclusions

SLAC LCLS Data Challenge

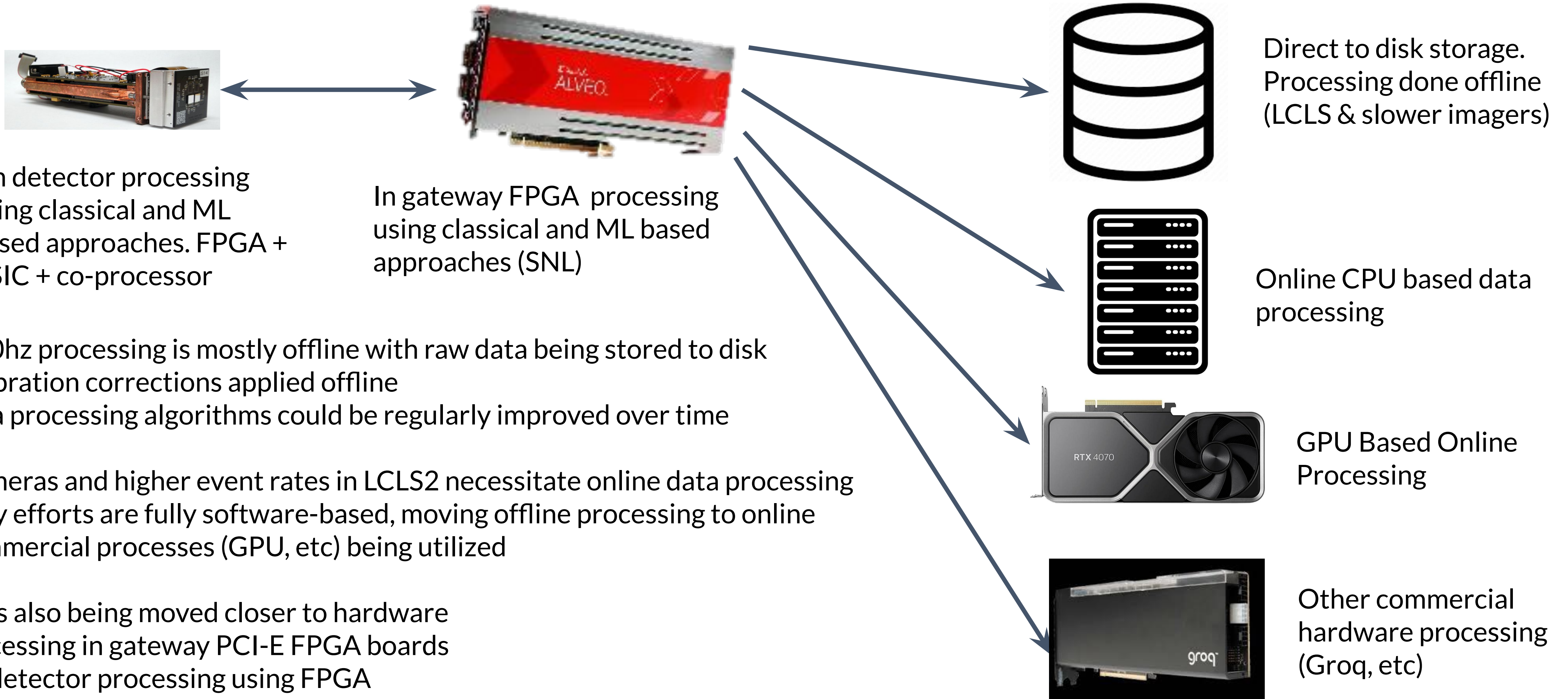


https://www6.slac.stanford.edu/news/2021-02-17-bigger-faster-more-powerful-slacs-new-x-ray-laser-data-system-will-process-million





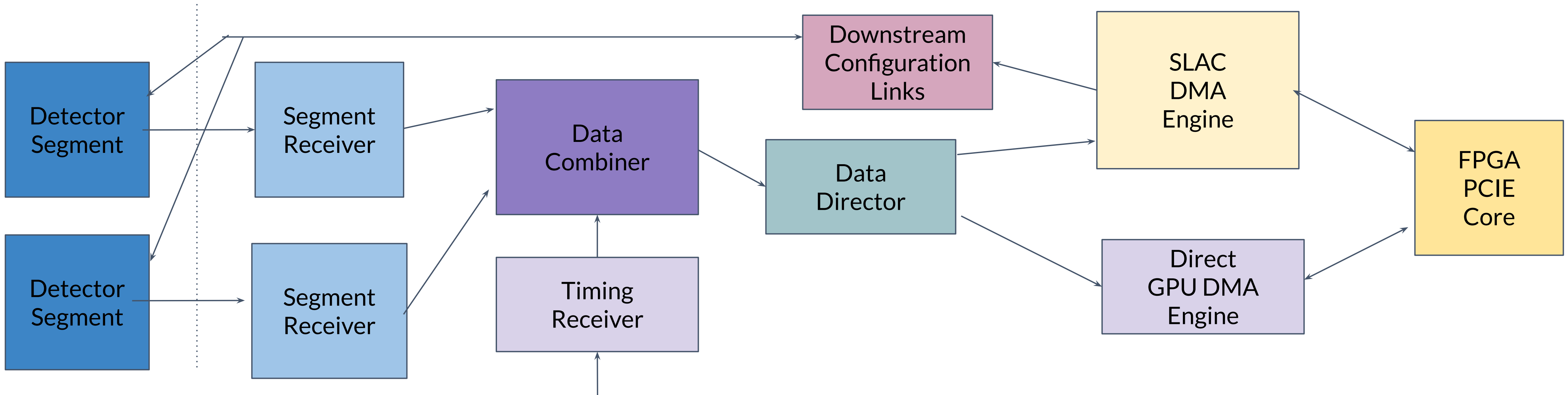
SLAC LCLS Data Reduction Pipeline



- LCLS1 120hz processing is mostly offline with raw data being stored to disk
  - Calibration corrections applied offline
  - Data processing algorithms could be regularly improved over time
- Larger cameras and higher event rates in LCLS2 necessitate online data processing
  - Early efforts are fully software-based, moving offline processing to online
  - Commercial processes (GPU, etc) being utilized
- Algorithms also being moved closer to hardware
  - Processing in gateway PCI-E FPGA boards
  - On detector processing using FPGA
  - eFPGA and direct ML inference in ASICs
- SLAC SNL enables FPGA, eFPGA and ASIC AI Inference Engine deployment

This application described in this talk supports 35Khz events from 5 FPGA boards = 50GB/s processed in a single GPU with an output bandwidth of 5GB/s

LCLS Specific PCI-E FPGA Board



- Data is received from detector segments using SLAC PGP based fiber protocols (PGP2, PGP3, PGP4)
  - Segment receiver performs camera specific data pre-processing

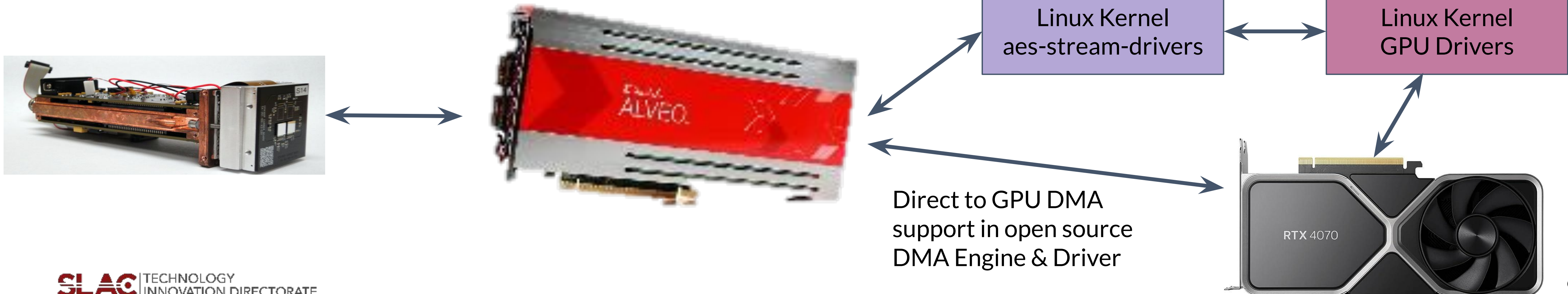
- Data combiner receives data from multiple camera segments and combines them into a single frame
  - Aligned by timestamp attached on camera
  - Timing data added from LCLS/LCLS2 timing receiver

- Data director routes event data either to DMA engine to software or to GPU
  - Configuration data always passed to software
  - Generic DMA engine for software based data reception
  - GPU Direct DMA engine for GPU processing mode

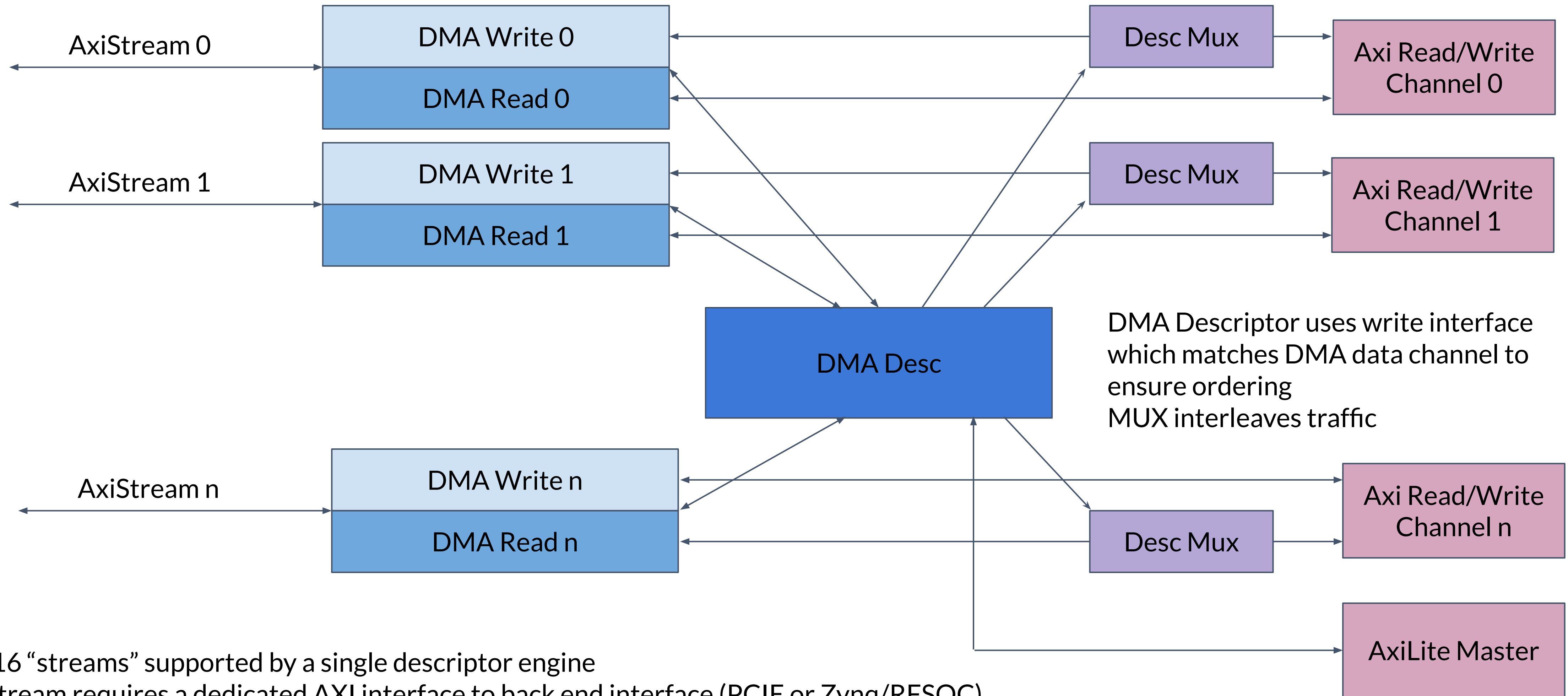
SLAC Open Source DMA Engine Key Features

- Supports multiple hardware lanes
  - Supports interleaved AXI-Stream traffic (tDEST) on each hardware lane
  - Per tID back pressure support
- Supports Zynq /RFSOC (or anything with an AXI interface)
  - ACP interface for automatic cache line
  - Common API for Zynq based and server based applications
- Configurable cache modes
  - Coherent (no caching)
  - Stream (cache with pre-post dma flush
  - Zynq ACP mode (hardware managed caching)
- Supports both interrupt and polling mode operation

- Supports user space memory mapping of DMA buffers
  - Minimizes memory to memory copies
- Supports multiple buffer receive and transmit to/from user space to reduce user to kernel space overhead
- Supports larger frame receive across multiplied DMA buffers
- Supports both high bandwidth and high buffer rate applications
- Open-source firmware and kernel driver
- Read/Write engines can be utilized for direct to/from GPU transfer

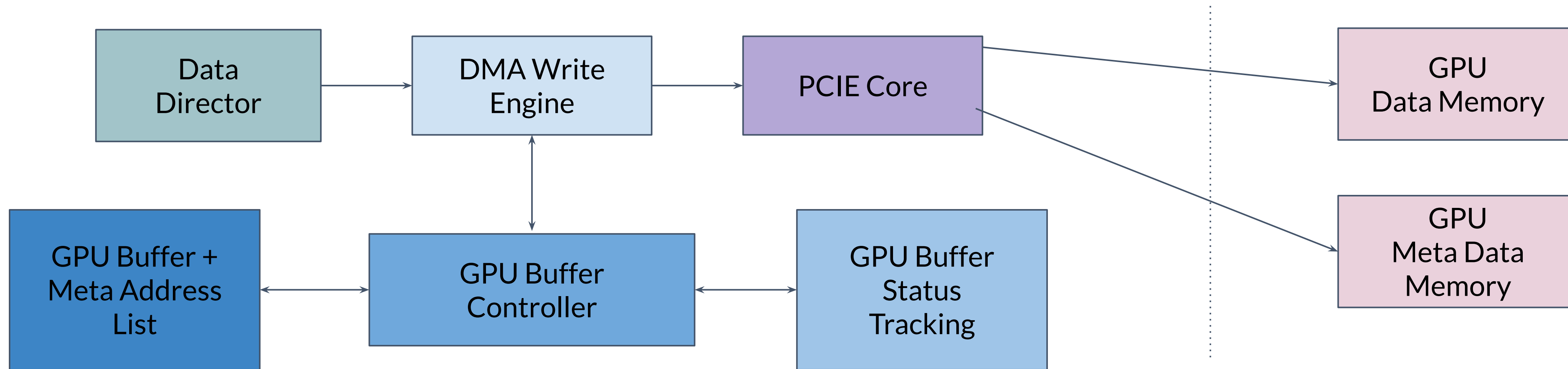


SLAC DMA Engine Overview



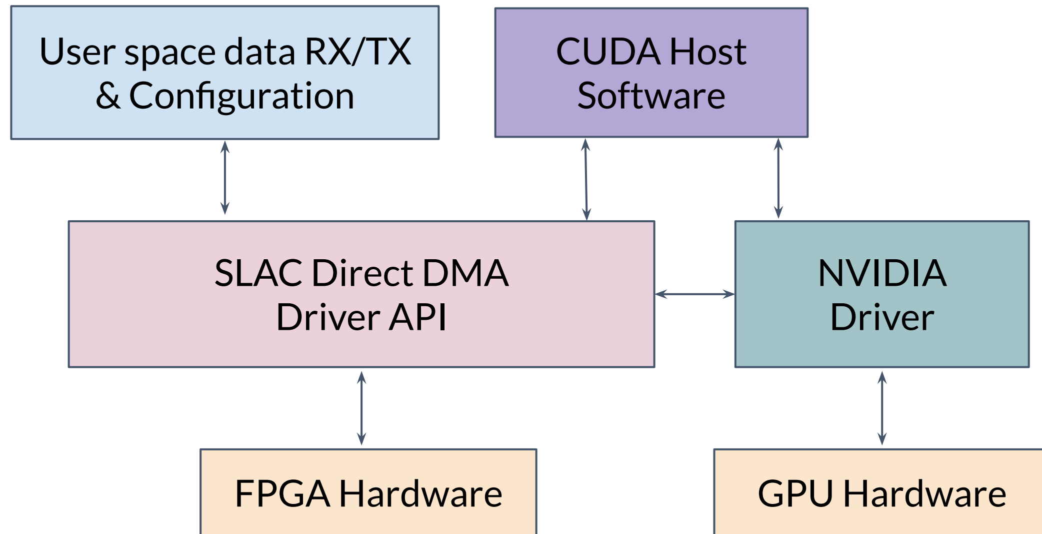
- Up to 16 “streams” supported by a single descriptor engine
- Each Stream requires a dedicated AXI interface to back end interface (PCIE or Zynq/RFSOC)

GPU Direct DMA Firmware



- Current firmware maintains a ordered list of GPU buffer and Meta-Data addresses
  - Buffers are always used in order
  - DMA is paused until next buffer is “enabled”
- Frame reception triggers start of DMA write engine
  - GPU buffer controller provides next buffer address & Meta address if buffer is “enabled”
  - Buffer is then marked disabled
  - DMA transfer is performed
  - Frame data is then written to Meta-Data address
    - Frame size
    - Frame error flags & user flags
- Alternative buffer approaches have been considered, but simple ordered ring of buffers is currently preferred by GPU developers
  - Unordered free-list based approach could easily be implemented if needed

GPU Direct DMA Driver & Host CUDA Software Interaction



- Direct GPU DMA driver is an extension of the standard SLAC DMA driver
  - Supports the same user space interaction for data reception and transmission

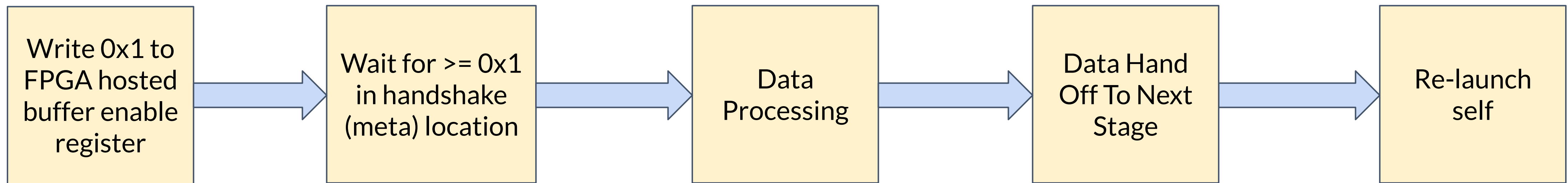
- Adds direct DMA specific registers
  - Write buffer information for n buffers (configured at build)
    - Buffer address, meta address and enable
  - Configuration & monitoring of direct DMA controller

- Generic driver is linked to NVIDIA driver calls required for GPU buffer address translation

- CUDA host software allocates buffers on the GPU, allocates n buffers per source FPGA (baseline = 5 FPGAs per GPU)
  - nvidia_p2p_get_pages/nvidia_p2p_dma_map_pages to map the pages from GPU pointers to physical PCIE bus address
  - Ensures resulting pages are contiguous, only uses contiguous range
  - Mapping results then written to write buffer addresses in FPGA memory

- Current system maps the “meta” register to be the first 64-bit memory location in the allocated GPU buffer
  - Meta data is written to bytes 0-8
  - Event data written to bytes 9+

CUDA GPU Software Operation



(1) Indicates that we're ready to receive data from the FPGA. While enable is 0, the buffer is "owned" by the GPU.

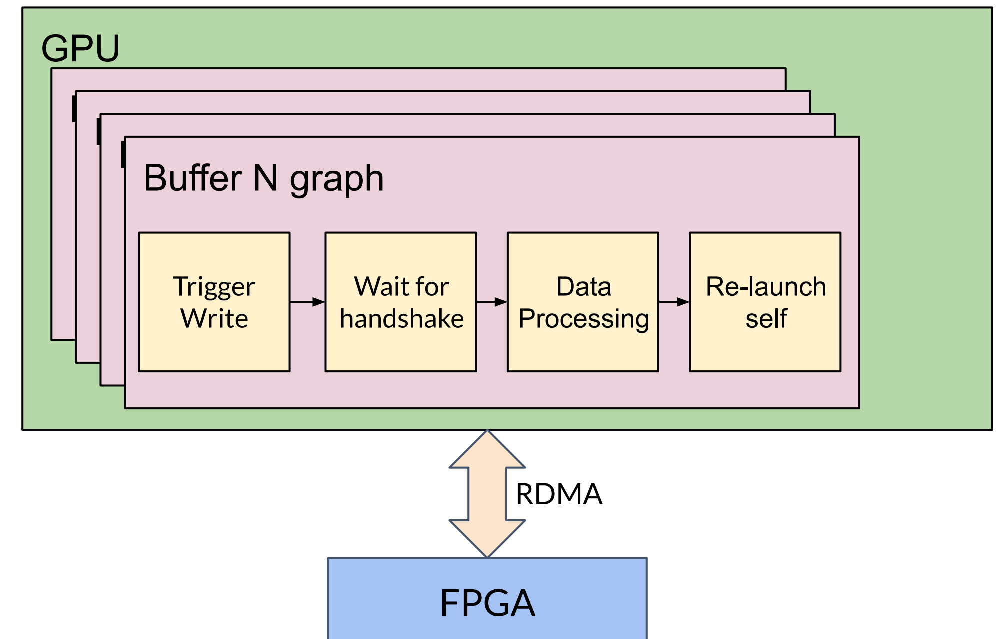
(2) Wait mechanism varies by control flow type. CUDA graphs use a spin-wait kernel. Normal CUDA API calls use cuStreamWaitValue.

For device launched graphs: re-launch using cudaStreamGraphTailLaunch in kernel.

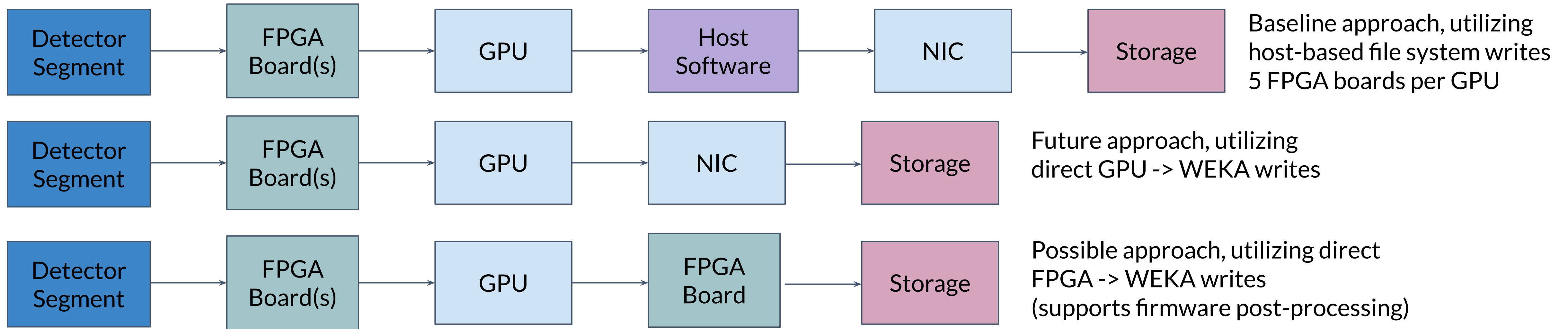
For the host CUDA API: Re-run the same API calls in a loop.

- GPU control flow is designed to be simple and self-contained
  - One buffer per thread or graph (# of FPGAs * buffers/FPGA)
  - Complicated branching/state management not necessary
- Simple control flow allows it to be easily described with CUDA graphs
  - Device-launched graphs further reduce host involvement
  - Reduced latency compared to host API
- CUDA host API
  - cuStream**Write**Value32 to trigger transfer **(1)**
  - cuStream**Wait**Value32 to wait on handshake space **(2)**
- CUDA graphs
  - cudaMemcpyAsync to trigger transfer **(1)**
  - spin-loop kernel to wait on handshake space **(2)**

Optimal execution involves multiple streams running in parallel.



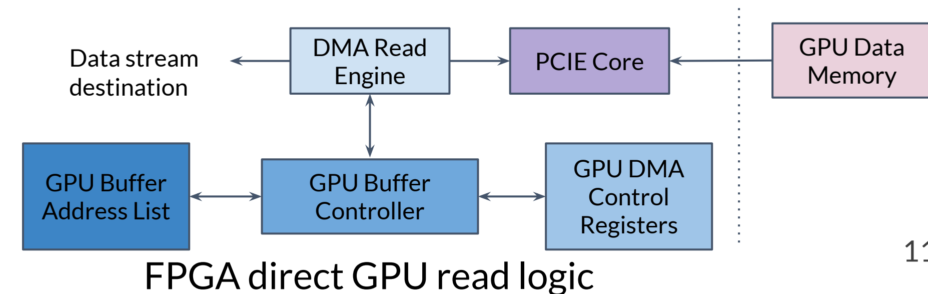
Post GPU Data Flow Options



- Baseline approach involves handing GPU results to host software which executes WEKA file system write
  - Post GPU data volume is greatly reduced, allowing software in the loop processing

- Desired approach is to utilize GPU -> WEKA direct writes through appropriate NIC device
  - Involves direct DMA from GPU to NIC
  - Full support is not quite ready

- SLAC FPGA board also supports DMA pull from GPU using generic read DMA engine
  - GPU kernel can trigger read via write to FPGA handshake register
  - This feature has been tested but not yet fully optimized
  - We could also support FPGA -> WEKA direct writes using integrated NIC



FPGA direct GPU read logic

Limitations & Conclusions

- Current test code requires that the server's iommu be disabled to remove PCIE-PCIE security
  - Access to DRP hosts is limited so there is little concern with this mode of operation
  - We are looking into more targeted configuration which does not require full security disable
  - FPGA boards and GPU must be connected through the same root complex
  - NVIDIA GPU must support direct DMA

- Memory mapped access to FPGA buffer enable registers from CUDA code requires executing CUDA code as root
  - We are looking into ways around this, including using driver-based calls which work for GPU host code.

- Data movement has been demonstrated using both CUDA host API and CUDA graphs
  - Tested with 2 FPGAs feeding data to a GPU
  - Working on scaling up to more FPGA streams to GPU and multiple FPGA boards/streams per GPU
  - Planning on doing performance testing with both dummy and real application code

- Firmware and associated driver are open source and available as part of the TID Instrumentation open-source libraries
  - https://github.com/slaclab/surf
  - https://github.com/slaclab/aes-stream-drivers

- Key contributors:
  - Jeremy Lorelli - Driver development & GPU core software
  - Mudit Mishra - FPGA data flow firmware
  - Larry Ruckman - General firmware & SURF library management
  - Ric Claus - GPU application software