CPAD 2019
December 8, 2019

Future DAQ Concepts
Edge ML For High Rate Detectors

Ryan Herbst
Department Head, Advanced Electronics Systems

(rherbst@slac.stanford.edu)

SLAC TID-AIR
Technology Innovation Directorate
Advanced Instrumentation for Research Division

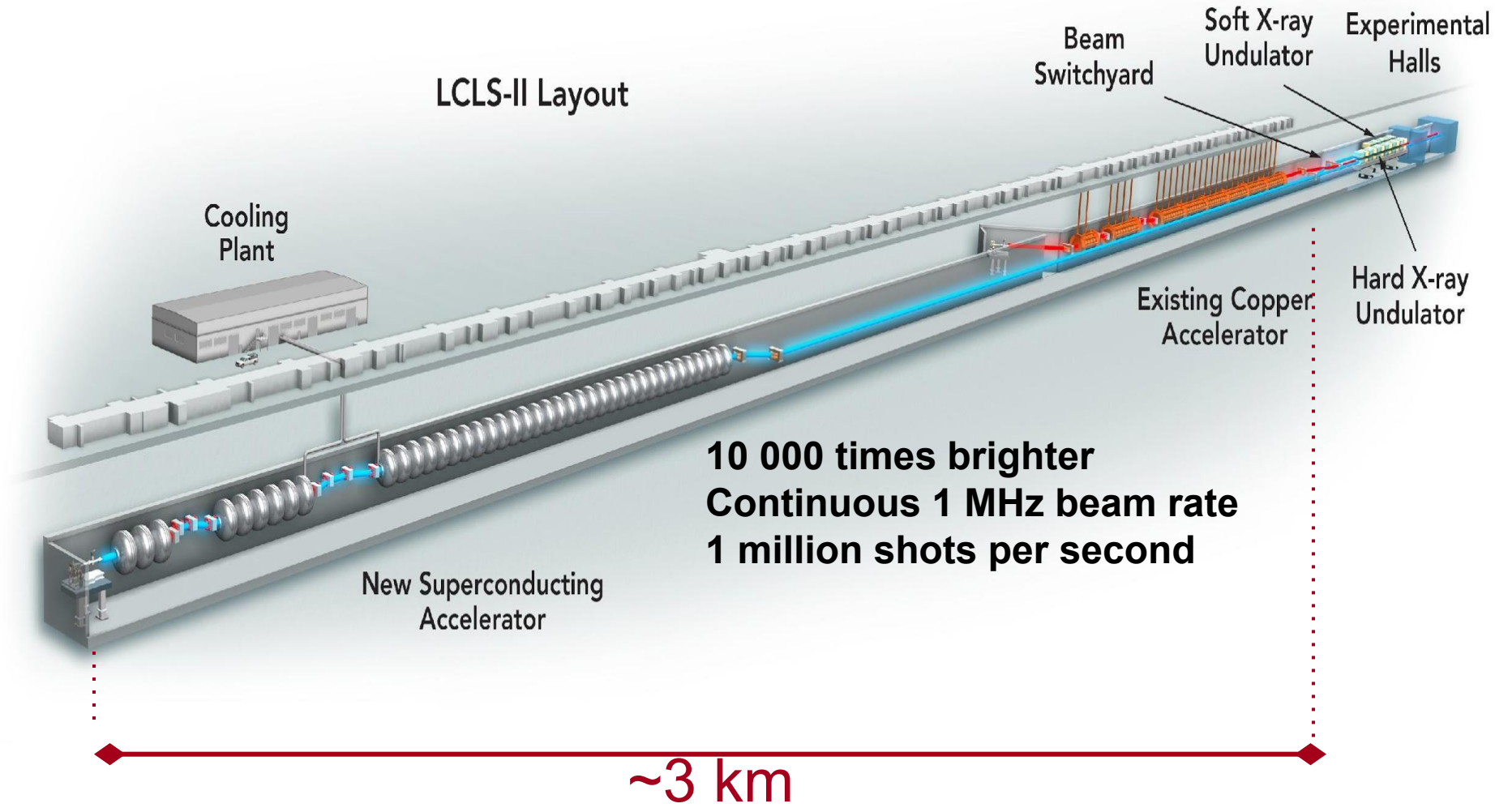
U.S. DEPARTMENT OF ENERGY
Office of Science

SLAC NATIONAL ACCELERATOR LABORATORY

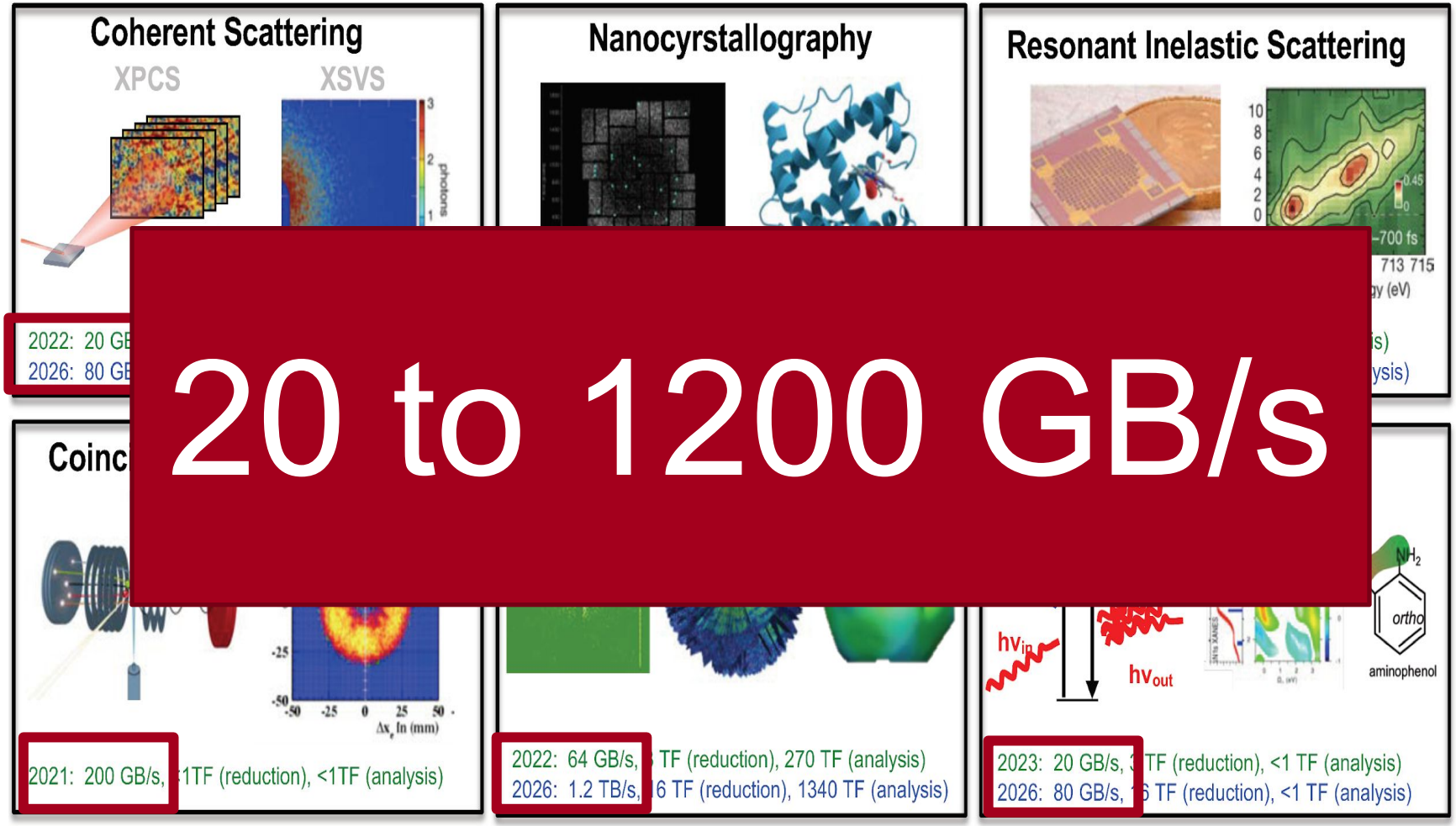
Overview

- Describe Data Reduction & Processing Challenges
- Overview of VHDL based inference framework
  - Example network
  - Usage model
- Targeted usage in LCLS-2 beamlines (CookieBox)
- Observations on current framework
  - Possible enhancements

LINAC Coherent Light Source - II



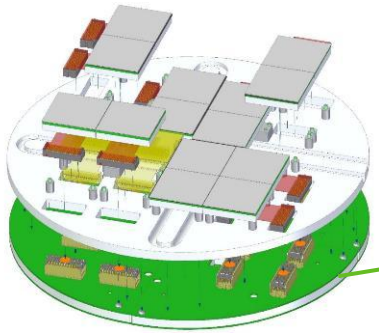
LCLS-II Detector Raw Data Rates



20 to 1200 GB/s

Image courtesy of Jana Thayer, Mike Dunne

Data Processing Techniques At Different System Levels



EDGE Computing on camera

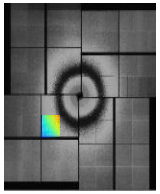
Data System

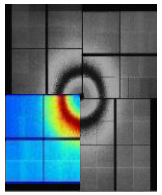
ASIC Level

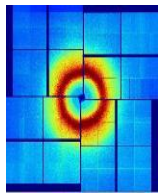
FPGA Level

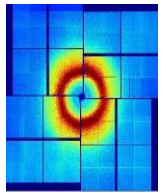
Farm of FPGAs

CPUs/GPUs









Versatility

Rate reduction

- Application specific
- Limited number of techniques:
  - Sparsification
  - Event driven trigger
  - Back-end zero suppression
  - Region of Interest (RoI)
- Algorithms can be tailored
- Limited number of techniques:
  - Back-end zero suppression
  - Region of Interest (RoI)
- Algorithms can be tailored to different applications (Possibility to use ML)
- Fast feedback to the detector (trigger generation)
- Vetoing
- Large number of lossless techniques
- Calibration

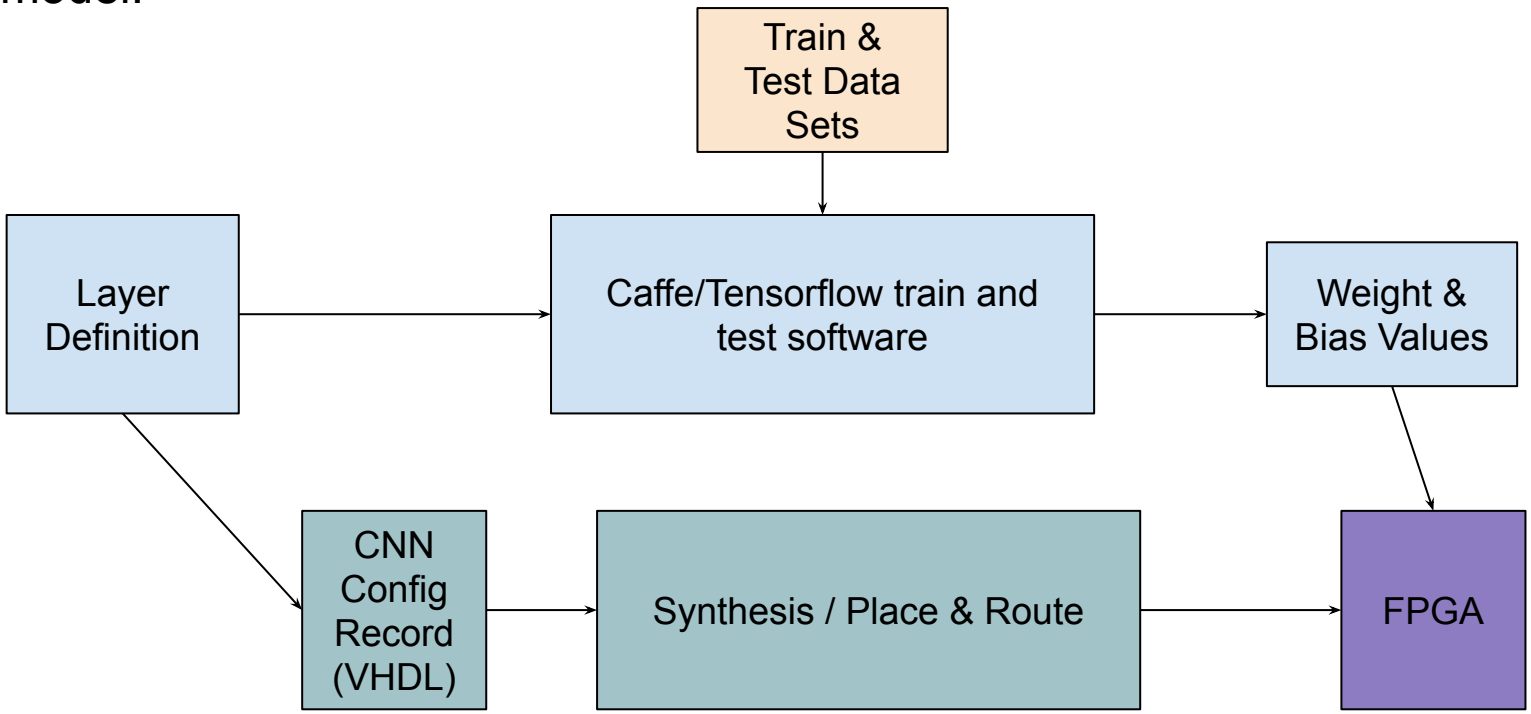
Image courtesy of Jana Thayer, Mike Dunne

General Requirements & Applications For ML In Detector Systems

- Target latency < 100uS
  - > 100uS better suited towards to software & GPU processing
  - Specific latency target depends on buffer capabilities of the cameras
    - Typically in the 1uS - 50uS range
- Frame rate of 1Mhz
  - Early detectors will run at 10Khz - 100Khz
- Support fast retraining and deployment of new weights and biases
  - Limits synthesis optimization around zero weights
  - The beamline science and algorithms will evolve
  - Large investment into fast re-training infrastructure
- Target applications:
  - Camera protection against beam misteer or sample icing
  - Region of interest identification
  - Zero suppression
  - Convert raw data to structured data

One Possible Approach
VHDL Based ML Framework

- Framework provides a configurable VHDL based implementation to deploy inference engines in an FPGA
  - Layer types supported: Convolution, Pool & Full
- Developed as a proof of concept with limit resources
- Design flow for deploying neural networks in FPGA from Caffe or Tensorflow model:



Synthesis, Configuration & Input/Output Data

- Library consists of generic layer modules with input and output dimensions auto inferred during synthesis based upon input configuration and each layer configuration.

- Configuration map is determined by the computational element dimensions along with the input configuration
  - For each computation element there is a single bias value and a weight for each of the connected inputs

- Input and output interfaces are Axi-Stream types, containing values scanned in the following order:

```
for (srcX=0; srcX < inXCnt; srcX++) {
  for (srcY=0; srcY < inYCnt; srcY++) {
    for (srcZ=0; srcZ < inZCnt; srcZ++) {
```

- Auto generated structures does not take weights and biases into considering and assumes the values will be dynamic (no pruning).

Generating The Firmware: LeNET Example

- Configure the input data stream:

```
constant DIN_CONFIG_C : CnnDataConfigType := genCnnDataConfig ( 28, 28, 1 ); // x, y, z
```

- Configure the network:

```
constant CNN_LENET_C : CnnLayerConfigArray(5 downto 0) := (

        0 => genCnnConvLayer (strideX   => 1,  strideY   => 1,
                              kernSizeX => 5,  kernSizeY => 5,
                              filterCnt => 20,
                              padX      => 0,  padY      => 0,
                              chanCnt   => 10, rectEn    => false),

        1 => genCnnPoolLayer (strideX   => 2, strideY   => 2, kernSizeX => 2, kernSizeY => 2),

        2 => genCnnConvLayer (strideX   => 1,  strideY   => 1,
                              kernSizeX => 5,  kernSizeY => 5,
                              filterCnt => 50,
                              padX      => 0,  padY      => 0,
                              chanCnt   => 50, rectEn    => false),

        3 => genCnnPoolLayer (strideX   => 2, strideY   => 2, kernSizeX => 2, kernSizeY => 2),

        4 => genCnnFullLayer ( numOutputs => 500, chanCnt => 50, rectEn => true ),

        5 => genCnnFullLayer ( numOutputs => 10, chanCnt => 1, rectEn => false ));
```

Generating The Code

- Generate connected configuration of all of the layers + input:

```
constant LAYER_CONFIG_C : CnnLayerConfigArray := connectCnnLayers(DIN_CONFIG_C, CNN_LENET_C);
```

- Instantiate the CNN module:

```
U_CNN: entity work.CnnCore
   generic map (
     LAYER_CONFIG_G     => LAYER_CONFIG_C)          -- CNN Layer configuration
   port map (
     cnnClk          => cnnClk,
     cnnRst          => cnnRst,

     -- Input data stream
     sAxisMaster     => cnnObMaster,
     sAxisSlave      => cnnObSlave,

     -- Output data stream
     mAxisMaster     => cnnIbMaster,
     mAxisSlave      => cnnIbSlave,

     -- AXI bus for weights & biases
     axilClk         => axilClk,
     axilRst         => axilRst,
     axilReadMaster  => axilReadMaster,
     axilReadSlave   => axilReadSlave,
     axilWriteMaster => axilWriteMaster,
     axilWriteSlave  => axilWriteSlave);
```

Convolution Layer Configuration Parameters

- strideX: number of input points to slide the filters in the X axis
- strideY: number of input points to slide the filters in the Y axis
- kernSizeX: kernel size in the X axis (number of inputs per filter in X)
- kernSizeY: kernel size in the Y axis (number of inputs per filter in Y)
- filterCount: number of filters in the Z direction
- padX: pad size in the X axis
- padY: pad size in the Y axis
- rectEn: flag to enable application of a rectification function on the outputs
- chanCount: number of computation channels to allocate (Z direction)

Computations:

outXCount = ((inXCnt - kernSizeX + 2*padX) / strideX) + 1
outYCount = ((inYCnt - kernSizeY + 2*padY) / strideY) + 1
outZCount = filterCount

Current implementation limits parallelization to elements in the Z direction due to the way the input data is iterated over.

Pool Layer Configuration Parameters

- strideX: number of input points to slide the filters in the X axis
- strideY: number of input points to slide the filters in the Y axis
- kernSizeX: kernel size in the X axis (number of inputs per filter in X)
- kernSizeY: kernel size in the Y axis (number of inputs per filter in Y)

Computations:

outXCount = ((inXCnt - kernSizeX) / strideX) + 1
outYCount = ((inYCnt - kernSizeY) / strideY) + 1
outZCount = inZCount

Pool layer does not support parallelization.

Full Layer Configuration Parameters

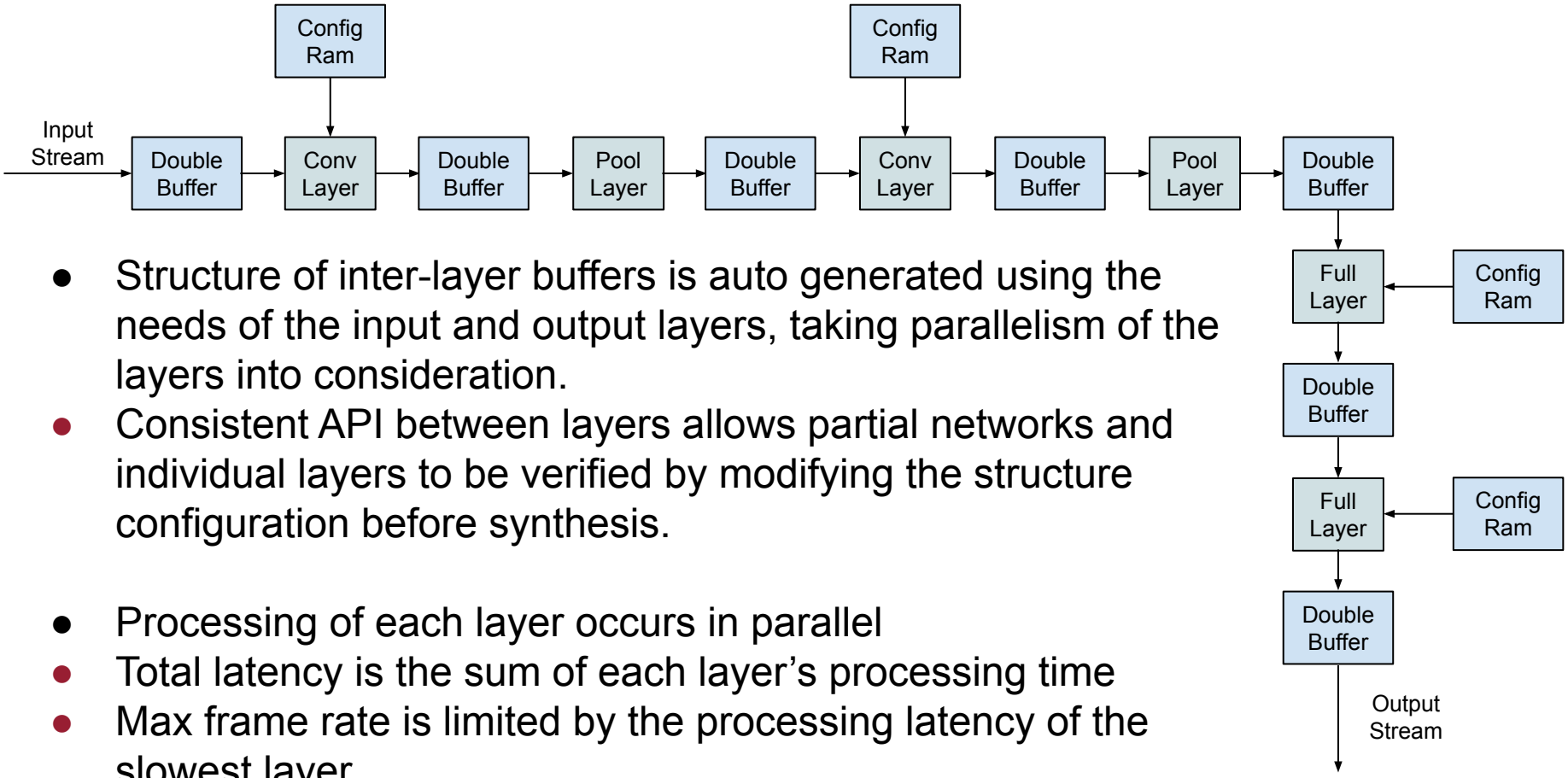
- numOutputs: number of output filters
- chanCount: number of computation channels to allocate
- rectEn: flag to enable application of a rectification function on the outputs

Computations:

outXCount = numOutputs
outYCount = 1
outZCount = 1

Full layer can support between 1 and numOutputs computation channels

Current implementation: Generated Structure For LeNet-4



- Structure of inter-layer buffers is auto generated using the needs of the input and output layers, taking parallelism of the layers into consideration.
- Consistent API between layers allows partial networks and individual layers to be verified by modifying the structure configuration before synthesis.

- Processing of each layer occurs in parallel
- Total latency is the sum of each layer's processing time
- Max frame rate is limited by the processing latency of the slowest layer
  - Each layer is flow controlled with full handshaking between layers

Current implementation: Convolution Layer Processing

- Iterate through each of the computational elements in the x & y dimension

```
for (filtX = 0; filtX < outXCount; filtX++) {
   for (filtY = 0; filtY < outYCount; filtY++) {
```

- Iterate through each of the computational elements in the Z direction, process chanCount z-dimension elements in parallel:

```
for (filtZ = 0; filtZ < outZCount/chanCount; filtZ++) {
```

- For each computational element, iterate over its connected inputs while performing multiply and accumulate, with one extra clock for bias value.

```
for (srcX=0; srcX < kernSizeX; srcX++) {
  for (srcY=0; srcY < kernSizeY; srcY++) {
    for (srcZ=0; srcZ < inZCount; srcZ++) {
```

latency(clock cycles) = (outXCount * outYCount * (outZCount / chanCount))
(kernSizeX * kernSizeY * inZCount + 1) *

Current implementation: Pool Layer Processing

- Iterate through each of the computational elements in the x, y & z dimension

```
for (filtX = 0; filtX < outXCount; filtX++) {
   for (filtY = 0; filtY < outYCount; filtY++) {
      for (filtZ = 0; filtZ < outZCount; filtZ++) {
```

- For each computational element, iterate over its connected inputs finding max value, index of input Z element = index of output Z element.

```
for (srcX=0; srcX < kernSizeX; srcX++) {
   for (srcY=0; srcY < kernSizeY; srcY++) {
```

```
latency = (kernSizeX * kernSizeY) *
                (outXCount * outYCount * outZCount)
```

Current implementation: Full Layer Processing

- Full layer has a single dimension X.
- Iterate through each of the computational elements in the X direction, process chanCount x-dimension elements in parallel:

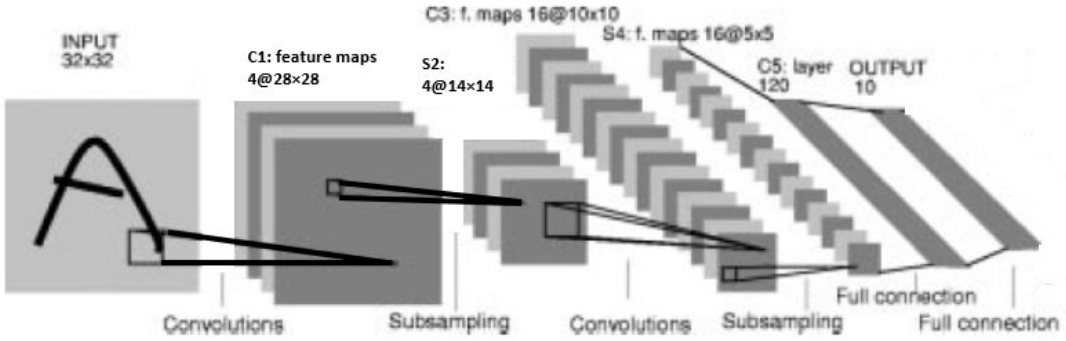
```
for (filtX = 0; filtX < outXCount/chanCount; filtX++) {
```

- For each computational element, iterate over its connected inputs while performing multiply and accumulate, with one extra clock for bias value.

```
for (srcX=0; srcX < inXCnt; srcX++) {
  for (srcY=0; srcY < inYCnt; srcY++) {
    for (srcZ=0; srcZ < inZCnt; srcZ++) {
```

latency = (inXCnt * inYCnt * inZCnt + 1) * (outXCount / chanCount)

LeNet-4 Fpga Utilization

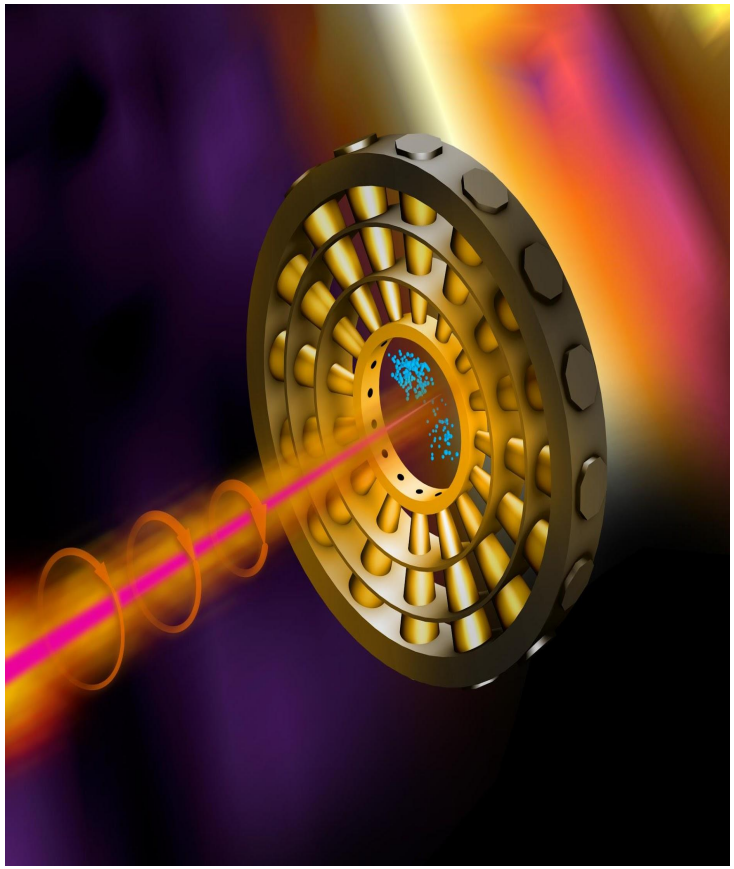


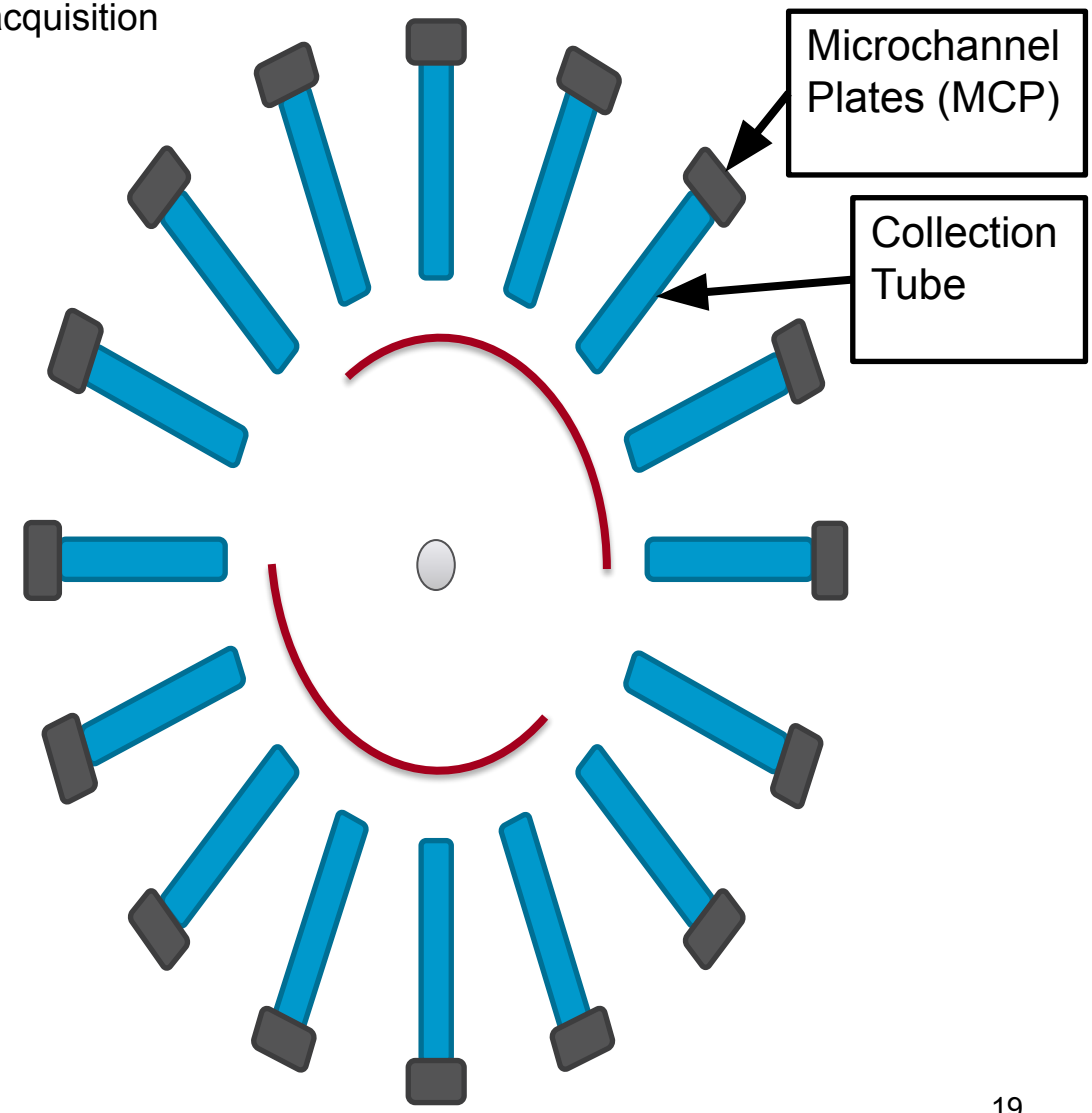
Xilinx XCKU115

Resource	Total	Available	PCT
CLB LUTs	116110	663360	17.5%
CLB Regs	33949	1326720	3%
Block ram	951	2160	44%
DSPs	333	2160	15.4%

CookieBox – Angular Streaking Detector Beam Qualification For Image Selection

- Detector is used to veto LCLS2 detector acquisition based upon detected beam parameters

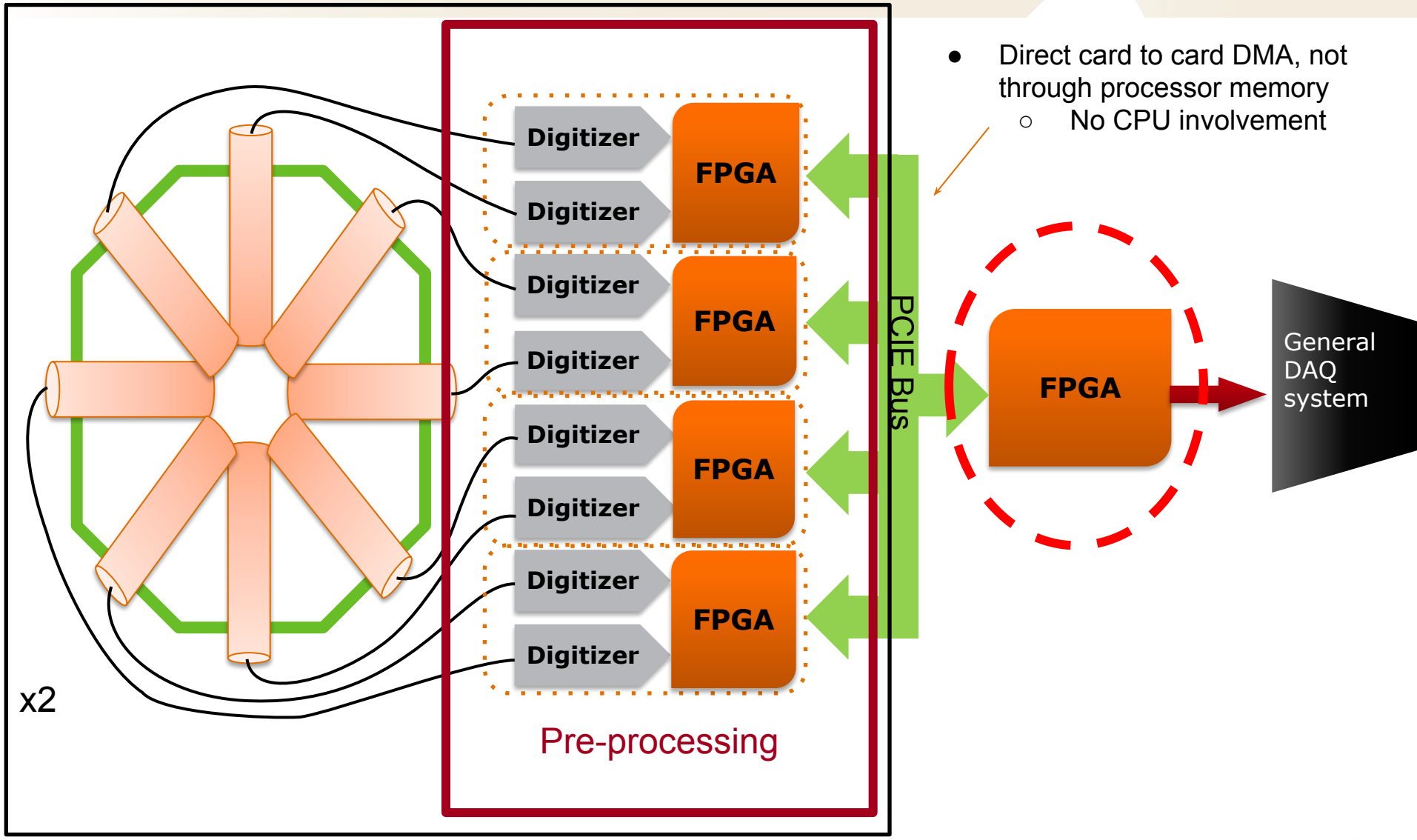




Hartmann, N. et al., Nature photonics, 2018
Siqi, Li et al. Optics express, 2018

Slides from A. Therrian

DAQ Chain Overview



- Direct card to card DMA, not through processor memory
  - No CPU involvement

Slides from A. Therrian

CookieNet Layer Configuration & Utilization


```
-- Input data config
constant DIN_CONFIG_C : CNNDataConfigType := genCnnDataConfig (800, 1, 1);

-- Network Config
constant NN_COOKIE_C : CnnLayerConfigArray(2 downto 0) := (
        0 => genCnnFullLayer (numOutputs => 200, chanCnt => 200, rectEn => true),
        1 => genCnnFullLayer (numOutputs => 100, chanCnt => 100, rectEn => true),
        2 => genCnnFullLayer (numOutputs => 5, chanCnt => 5, rectEn => true)));
```

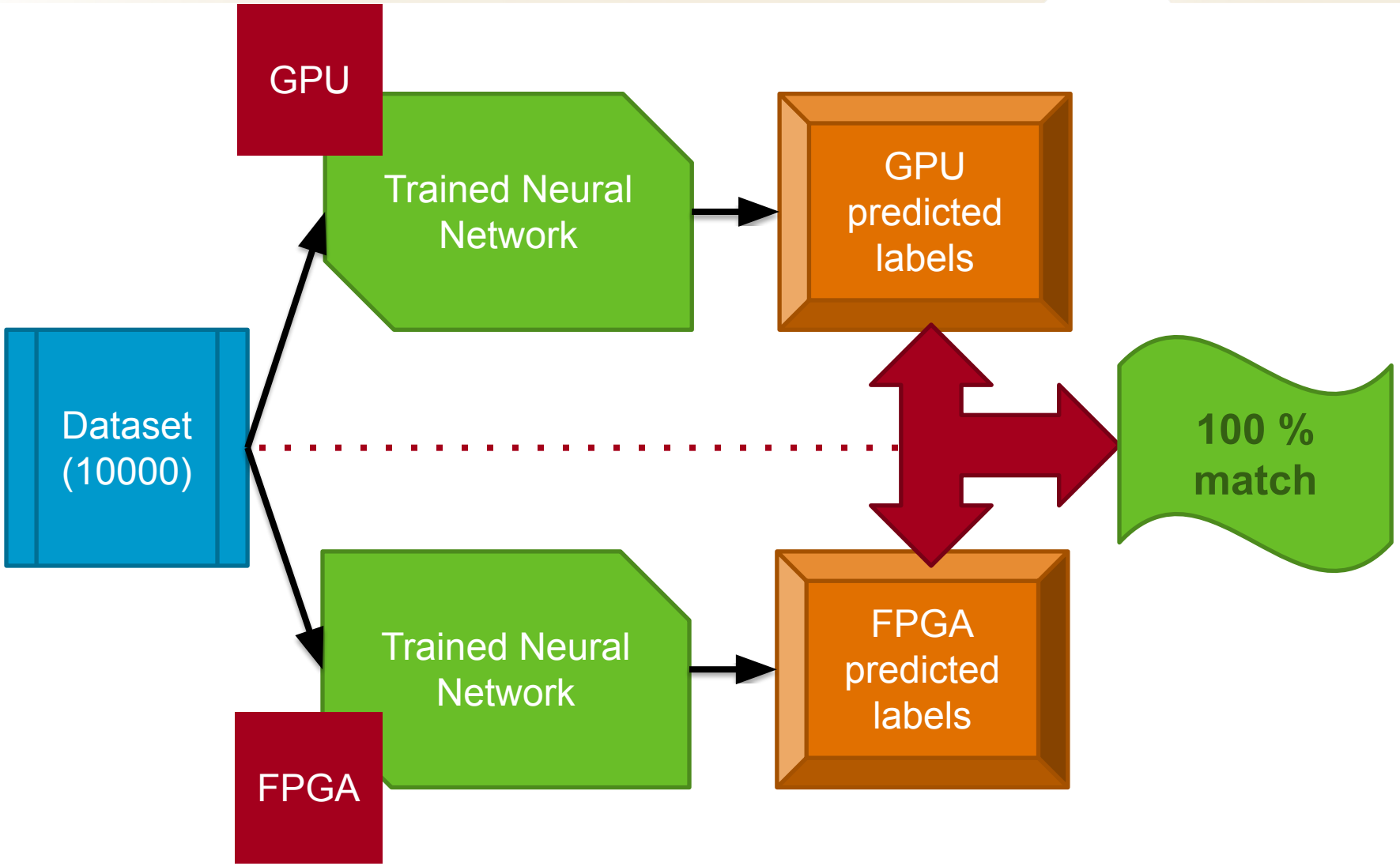
- Input array = 800 x 1 x 1
- Layer 1 = Full with 200 outputs, fully parallel
- Layer 2 = Full with 100 outputs, fully parallel
- Layer 3 = Full with 5 outputs, fully parallel

UTILIZATION OF RESOURCES ON THE XCKU115-2FLVB2104E

CLB - Look-up tables	55.77 %
CLB - Registers	1.06 %
Block RAM	55.16 %
DSP Slices	17.92 %
Gigabit transceivers	12.5 %
PCIe	16.67 %

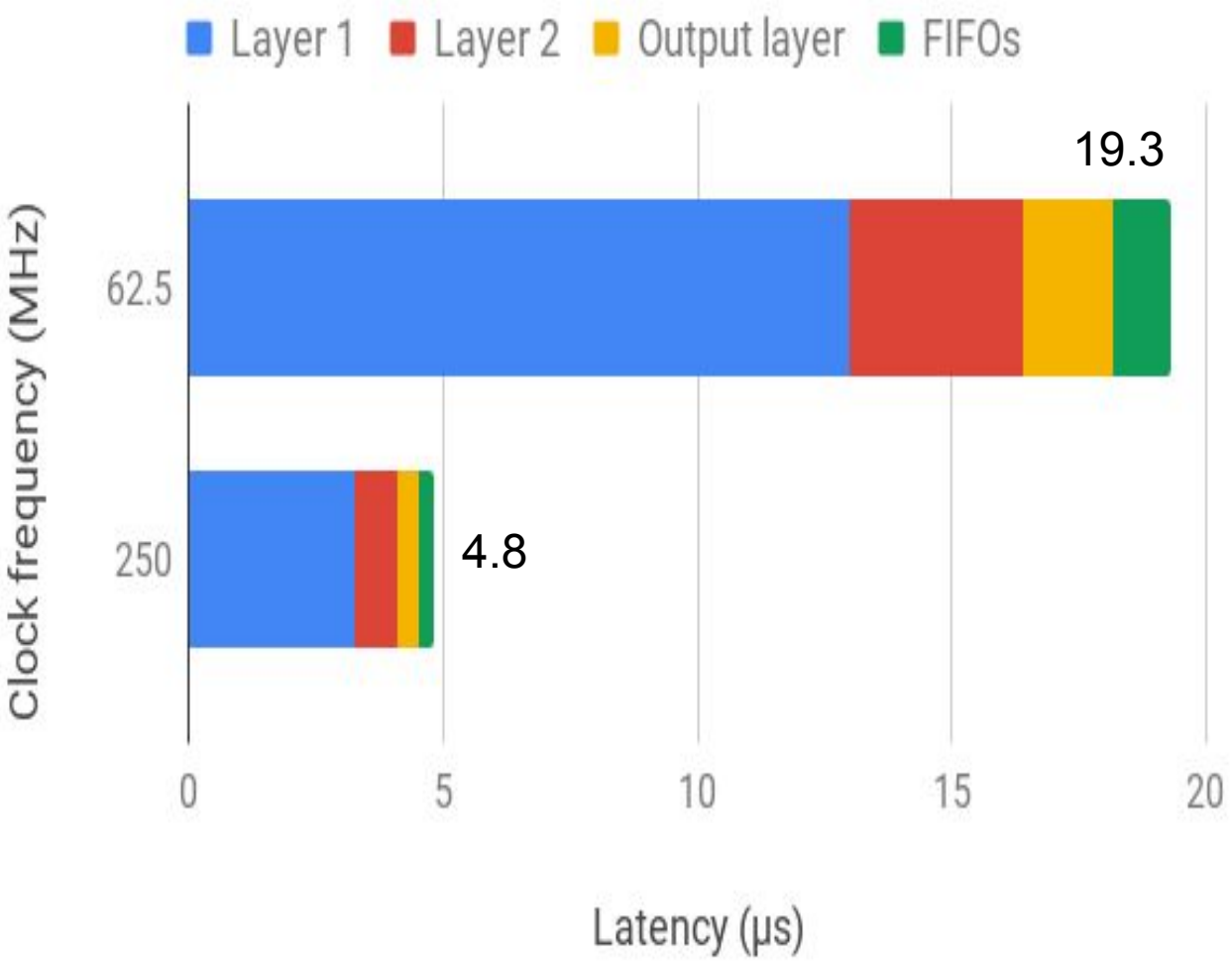
Slides from A. Therrian

Functionality Test



Slides from A. Therrian

Latency – Measured



Layer 1 : 800 inputs
Layer 2 : 200 inputs
Output Layer : 100 inputs

Maximum theoretical throughput R :

$$R = \frac{1}{MAX\ (layer\ latency)}$$

R (62,5 MHz) = 77 kHz
R (250 MHz) = 308 kHz

Slides from A. Therrian

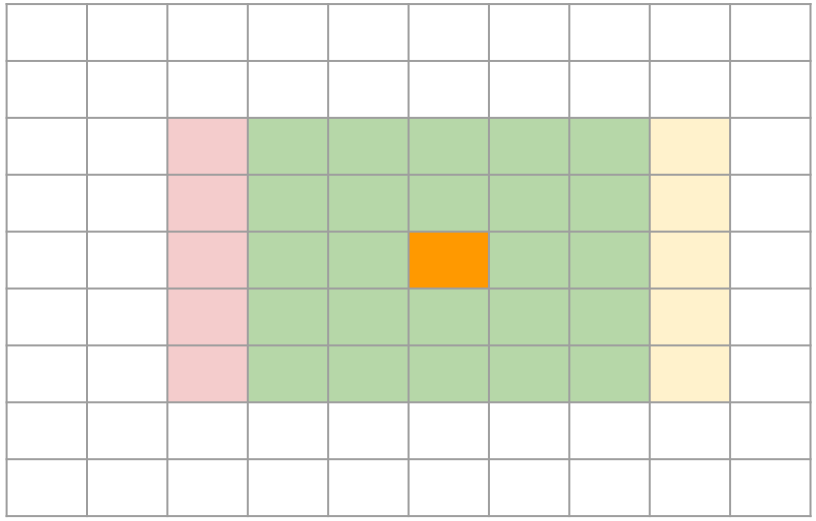
Current Implementation Observations: Full Layer

- Good utilization of DSP elements as 100% of layer can be operated in parallel
  - All elements active each clock cycle
  - All weight and bias configuration memories are active each clock
- Input buffer arrangement is decent as input array is iterated over sequentially
- Output buffering is not consistent with block ram as the output values are all written during the final clock.
  - Current generic block ram model results in wasted ram resources when parallelism is increased.
  - Cascaded full layers generates muxes with a large number of inputs in the following layer, creating large combinatorial latencies
  - Easy to address with proper pipelining and inter layer buffer restructuring
- Layer latency is dominated by the number of inputs
  - Width of input memory buffer could be increased to output multiple input pixels per clock.
    - Width of 128 bits = 4 x 32-bit values
    - Latency for largest layer decreases from 800 clocks to 200 clocks

Current Implementation Observations: Convolution & Pool Layers

- Latency is driven by the repeated scan of relevant inputs for each computational element as they are iterated over
  - Parallelism is only available in the z-dimension of computational elements due to the way the inputs are scanned and accessed.
  - Allocated DSP elements are idle during most of the clock cycles.

- Better approach would be to scan once over input data, passing data to a reusable processor, caching state & configuration data as necessary
  - Latency further reduced by passing input values in parallel



- Large block ram utilization for storing weights and measures
  - Most values not needed each clock cycle
  - An enhancement would be to stream weight and bias configuration from DRAM, aligned to input data, or to cache configuration as needed from external DRAM

Summary

- Proof of concept framework is viable for deploying inference networks in FPGAs
  - Framework provides ability to trade off latency for resource usage
  - Fixed network structure with fully configurable weight and bias configuration allows for fast re-training and rapid network re-deployment

- Framework has plenty of opportunities for optimization and enhancement
  - Continue work requires partnerships with funded projects and real world applications for testing
    - LCLS2 detector projects are an opportunity
    - Possible interest for HEP projects @ SLAC

- Other areas under investigation:
  - HLS based layer processing cores with data movement coordinated by lower level VHDL
    - Smaller units for debug and simulation, greater visibility into data movements
    - Cores can be dynamically swapped in based upon data patterns (partial reconfiguration)
  - Keep an eye on Xilinx offerings
    - Xilinx is heavily invested in higher level languages for FPGA based co-processing
    - DPU cores and other hard core processing may be interesting.
      - They are geared towards co-processing, it may be possible to drive them purely from firmware
  - General purpose ASIC offerings
  - DirectDMA to GPUs: Custom fiber card with inter-card DMA capability at ~80Gbps

The End