XILINX

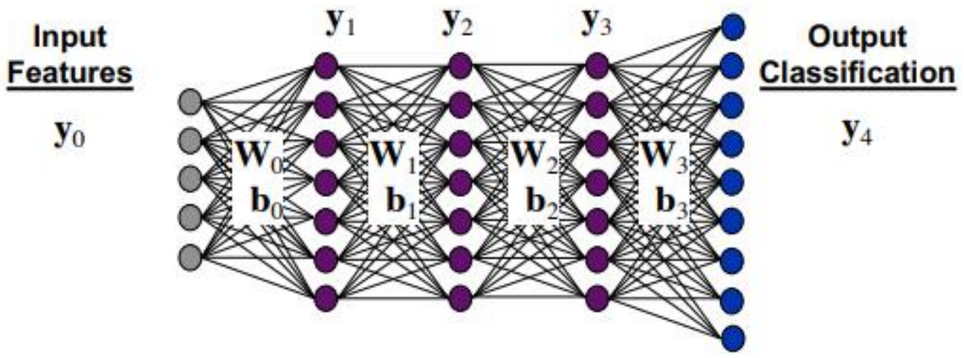
Sparse Deep Neural Network Acceleration on HBM-Enabled FPGA Platform

Abhishek K Jain, Sharan Kumar, Aashish Tripathi, Dinesh Gaitonde
Xilinx Inc., San Jose, CA, United States

September 23, 2021
Graph Challenge Special, IEEE High Performance Extreme Computing (HPEC) Virtual Conference 2021

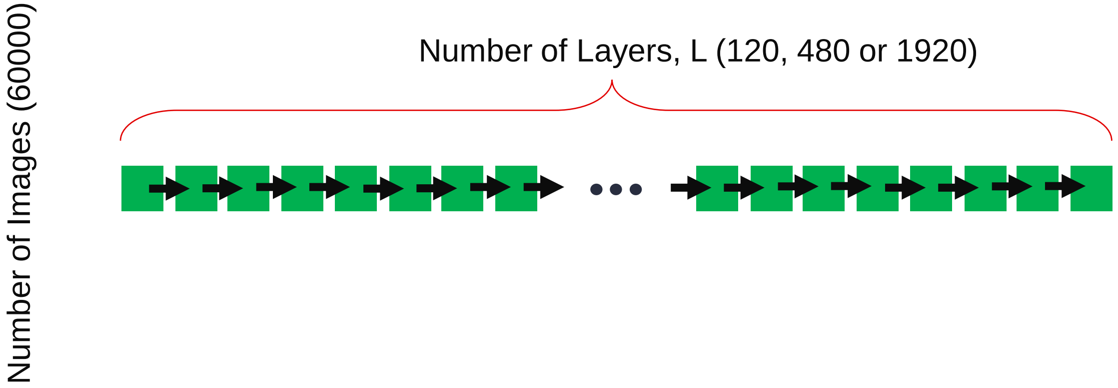
Sparse Deep Neural Network (SDNN) Challenge

- Bunch of sequential Layers
  - each layer has a weight matrix with very high sparsity (> 97%)
  - multiply input feature vector with weight matrix, followed by bias addition and ReLU
  - example: $y_1 = y_0 \times W_0 + b_0$
  - or compose as Sparse Matrix by Vector (SpMV) product: $y_1^T = W_0^T \times y_0^T + b_0^T$
  - neurons per layer ranges from 1K to 64K
  - number of layers (L) ranges from 120 to 1920
  - challenge contains 60,000 input feature vectors



- Mainly two approaches in previous submissions:

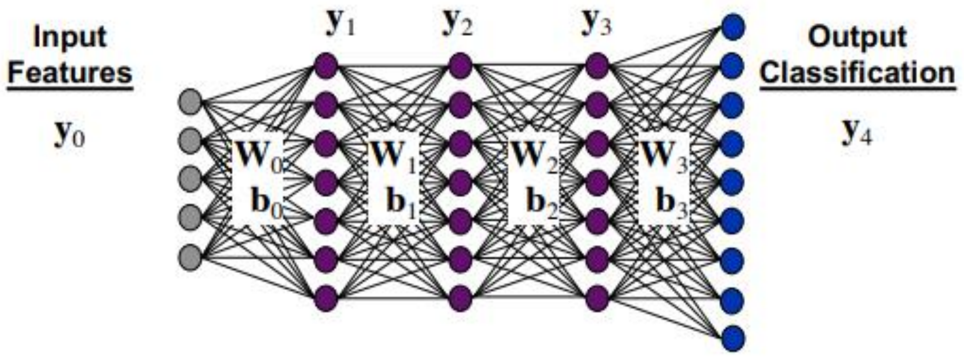
	Sparse Matrix by Matrix (SpMM)	Sparse Matrix by Vector (SpMV)
Inputs	Use all input feature vectors at once → large matrix (batch size of 60000)	One feature vector processing at a time → one vector (batch size of 1)
For 120-layer network	120 x 1 = 120 SpMM calls	120 x 60K = 7.2 million SpMV calls



XILINX

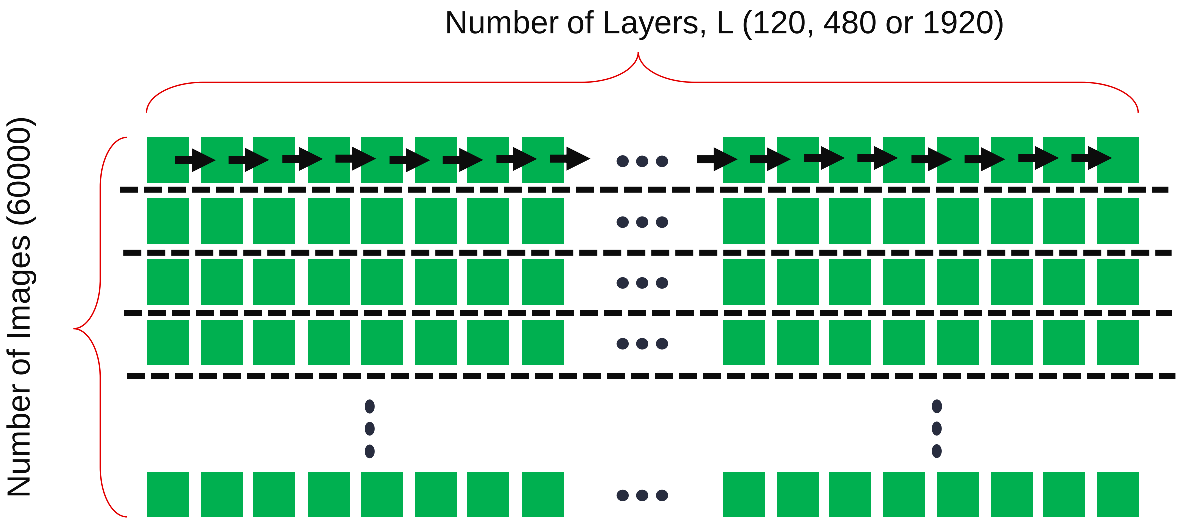
Sparse Deep Neural Network (SDNN) Challenge

- Bunch of sequential Layers
  - each layer has a weight matrix with very high sparsity (> 97%)
  - multiply input feature vector with weight matrix, followed by bias addition and ReLU
  - example: $y_1 = y_0 \times W_0 + b_0$
  - or compose as Sparse Matrix by Vector (SpMV) product: $y_1^T = W_0^T \times y_0^T + b_0^T$
  - neurons per layer ranges from 1K to 64K
  - number of layers (L) ranges from 120 to 1920
  - challenge contains 60,000 input feature vectors



- Mainly two approaches in previous submissions:

	Sparse Matrix by Matrix (SpMM)	Sparse Matrix by Vector (SpMV)
Inputs	Use all input feature vectors at once → large matrix (batch size of 60000)	One feature vector processing at a time → one vector (batch size of 1)
For 120-layer network	120 x 1 = 120 SpMM calls	120 x 60K = 7.2 million SpMV calls



XILINX

Sparse Matrix Vector Multiplication (SpMV)

- Key primitive for a wide range of ML, HPC and Graph problems
  - examples: sparse neural nets, conjugate gradient, pagerank etc.
- Traditional CPU/GPU platforms do not perform well for SpMV workload:
  - due to highly irregular and random memory access pattern (very high cache miss rate)

$$\begin{bmatrix} 1.0 & -- & 1.0 & -- \\ -- & -- & -- & -- \\ -- & -- & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix} * \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} (1.0)(1.0) + (1.0)(1.0) \\ 0.0 \\ (1.0)(1.0) + (1.0)(1.0) \\ (1.0)(1.0) + (1.0)(1.0) + (1.0)(1.0) + (1.0)(1.0) \end{bmatrix}$$

sparse matrix **A** | dense vector *x* | dense vector *y*

Table: Matrix A encoded in COO format

data	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
row	0	0	2	2	3	3	3	3
col	0	2	2	3	0	1	2	3

```
for(i = 0; i < NNZs; i++){
    y[row[i]] += data[i] * x[col[i]];
}
```

Note: NNZs → Number of non-zeros

Sparse Matrix Vector Multiplication (SpMV)

- Key primitive for a wide range of ML, HPC and Graph problems
  - examples: sparse neural nets, conjugate gradient, pagerank etc.
- Traditional CPU/GPU platforms do not perform well for SpMV workload:
  - due to highly irregular and random memory access pattern (very high cache miss rate)
- FPGA platforms are attractive for SpMV due to:
  - the use of many block memories (BRAMs/URAMs) to hold ***x*** and ***y*** vectors on-chip
  - the ability to avoid off-chip random memory access
  - streaming multiple non-zeros (NZs) in parallel from off-chip DRAM

$$\begin{bmatrix} 1.0 & -- & 1.0 & -- \\ -- & -- & -- & -- \\ -- & -- & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix} * \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} (1.0)(1.0) + (1.0)(1.0) \\ 0.0 \\ (1.0)(1.0) + (1.0)(1.0) \\ (1.0)(1.0) + (1.0)(1.0) + (1.0)(1.0) + (1.0)(1.0) \end{bmatrix}$$

sparse matrix **A** — dense vector *x* — dense vector *y*

Table: Matrix A encoded in COO format

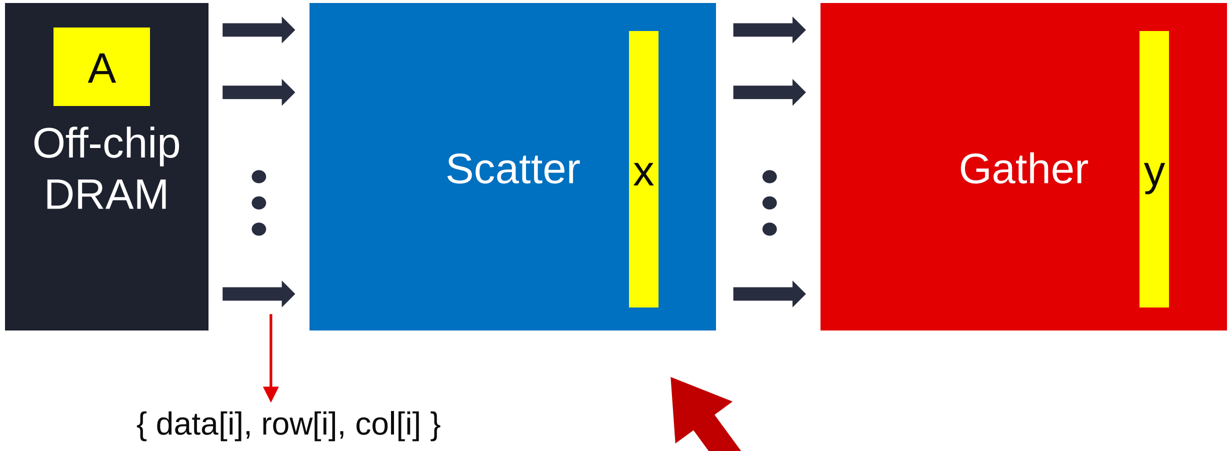
data	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
row	0	0	2	2	3	3	3	3
col	0	2	2	3	0	1	2	3

```
for(i = 0; i < NNZs; i++){
    y[row[i]] += data[i] * x[col[i]];
}
```

Note: NNZs → Number of non-zeros

Streaming Dataflow Pipeline

- Common theme of Streaming Dataflow Pipeline
- Two stages: Scatter and Gather
- Scatter
  - a multi-ported buffer for storing x
  - perform N reads in parallel, multiply with corresponding data
- Gather
  - a multi-ported buffer for storing y
  - perform N reads in parallel,
  - add with corresponding data and write back in y

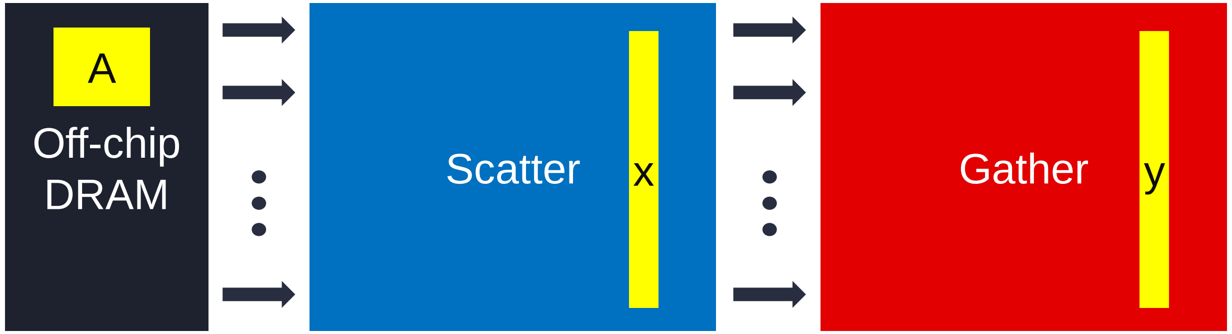


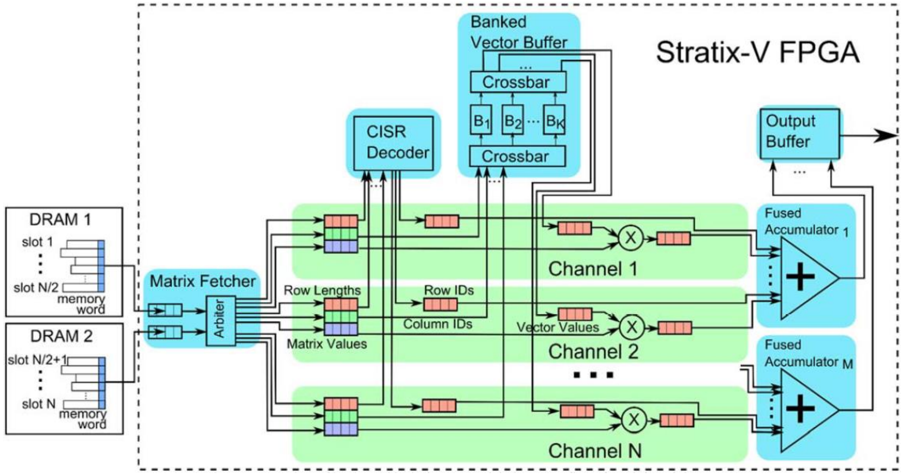

```
for(i = 0; i < NNZs; i++){
    y[row[i]] += data[i] * x[col[i]];
}
```

```
for(i = 0; i < NNZs; i=i+N){
    y[row[i]]       += data[i]     * x[col[i]];
    y[row[i+1]]     += data[i+1]   * x[col[i+1]];
    y[row[i+2]]     += data[i+2]   * x[col[i+2]];
    y[row[i+3]]     += data[i+3]   * x[col[i+3]];
    y[row[i+4]]     += data[i+4]   * x[col[i+4]];
                  ..
                  ..
                  ..
    y[row[i+N-1]]   += data[i+N-1] * x[col[i+N-1]];
}
```

Streaming Dataflow Pipeline

- Banked Vector Buffer (BVB) based SpMV [1]
  - pipeline for streaming 32 non-zeros
  - scatter → 32 banks of block memories + two crossbars

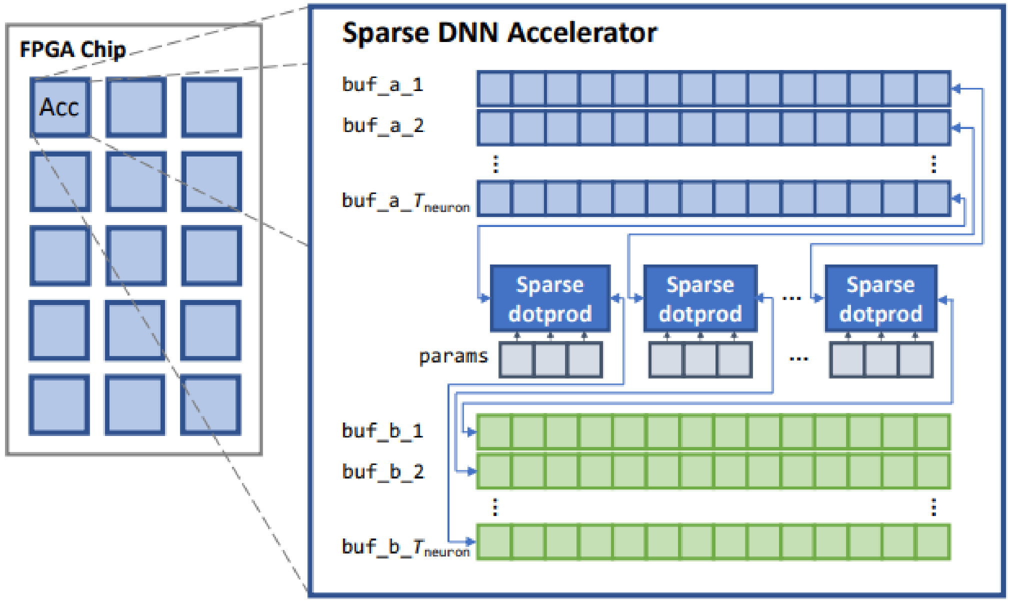




1. Fowers, Jeremy, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication." FCCM 2014

Sparse DNN Accelerator around FPGA based SpMV

- Sparse DNN accelerator[1]
  - SpMV as a building block
  - 15 blocks on the device → 15 parallel SpMV calls
  - 5x higher energy efficiency compared to the CPU baseline
  - rely on low DRAM bandwidth (30 GB/s) on FPGA board (VC709)
- Scaling limitations and Performance bottlenecks
  - require input vector replication (16 buffers)
  - limited on-chip memory capacity → only small-scale networks (1K neuron)

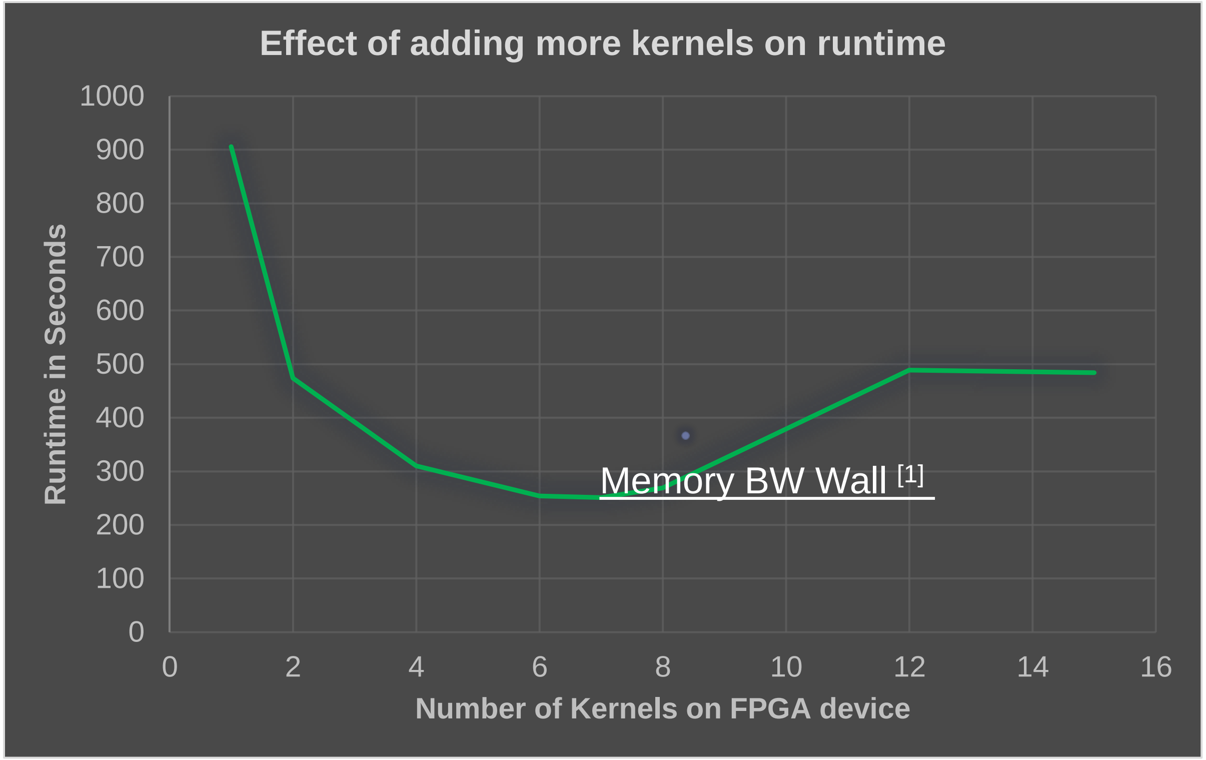


FPGA based Sparse DNN Accelerator from HPEC 2019[1]

1. Huang, Sitao, et al. "Accelerating sparse deep neural networks on fpgas." IEEE High Performance Extreme Computing Conference (HPEC), 2019.

Sparse DNN Accelerator around FPGA based SpMV

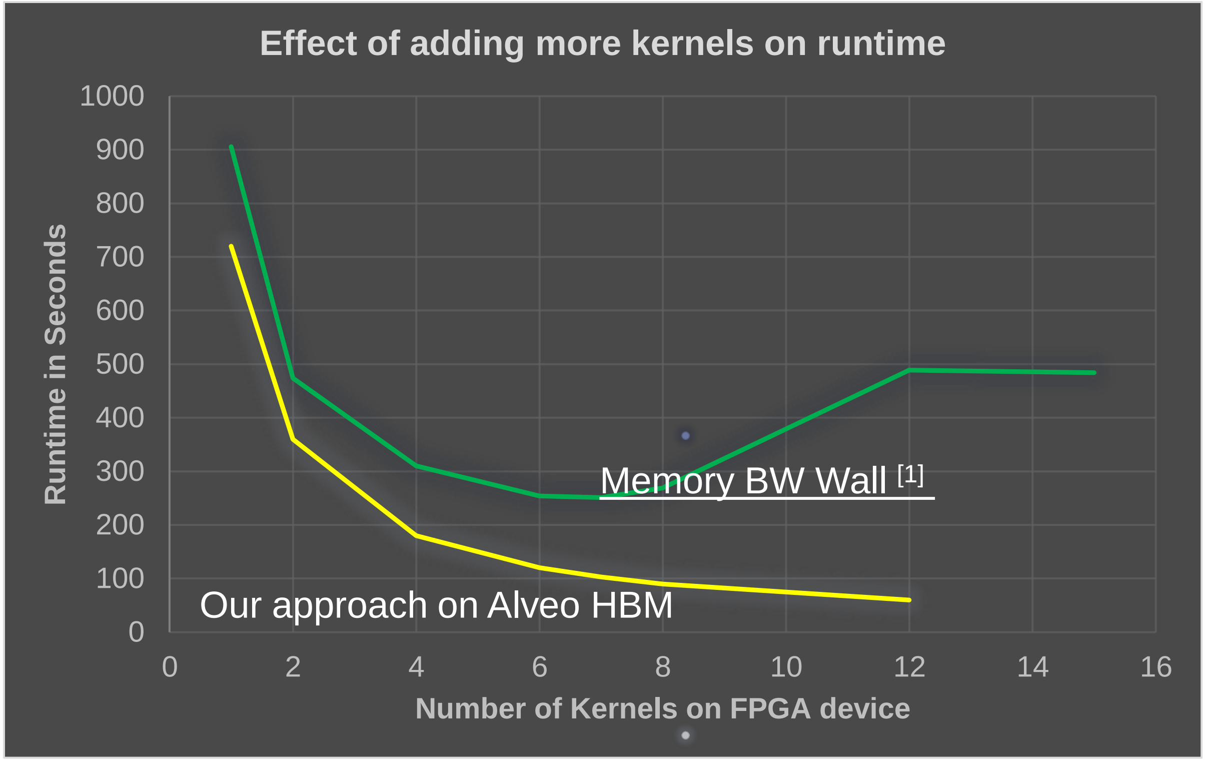
- Scaling limitations and Performance bottlenecks
  - multiple blocks share the bandwidth of a DRAM channel
  - bandwidth utilization and performance improves when activating up-to 7 blocks
  - performance and parallelization is limited by DRAM memory bandwidth



1. Huang, Sitao, et al. "Accelerating sparse deep neural networks on fpgas." IEEE High Performance Extreme Computing Conference (HPEC), 2019.

Sparse DNN Accelerator around FPGA based SpMV

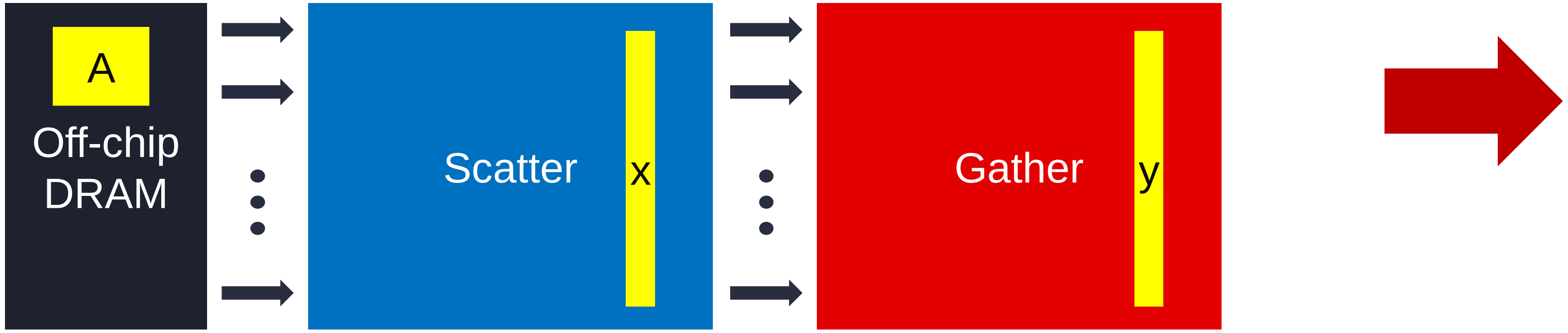
- Scaling limitations and Performance bottlenecks
  - multiple blocks share the bandwidth of a DRAM channel
  - bandwidth utilization and performance improves when activating up-to 7 blocks
  - performance and parallelization is limited by DRAM memory bandwidth
- Our approach
  - avoid replication of input feature vector → implement all networks (1K, 4K, 16K and 64K neurons)
  - uses multi-ported multi-banked buffer (based on URAMs and NoC)
  - Each block has dedicated HBM channels → No need of sharing
  - memory bandwidth → no longer the performance bottleneck
  - supports floating point FP32 arithmetic
  - supports completely unstructured sparse matrices (standard encoding format: COO)

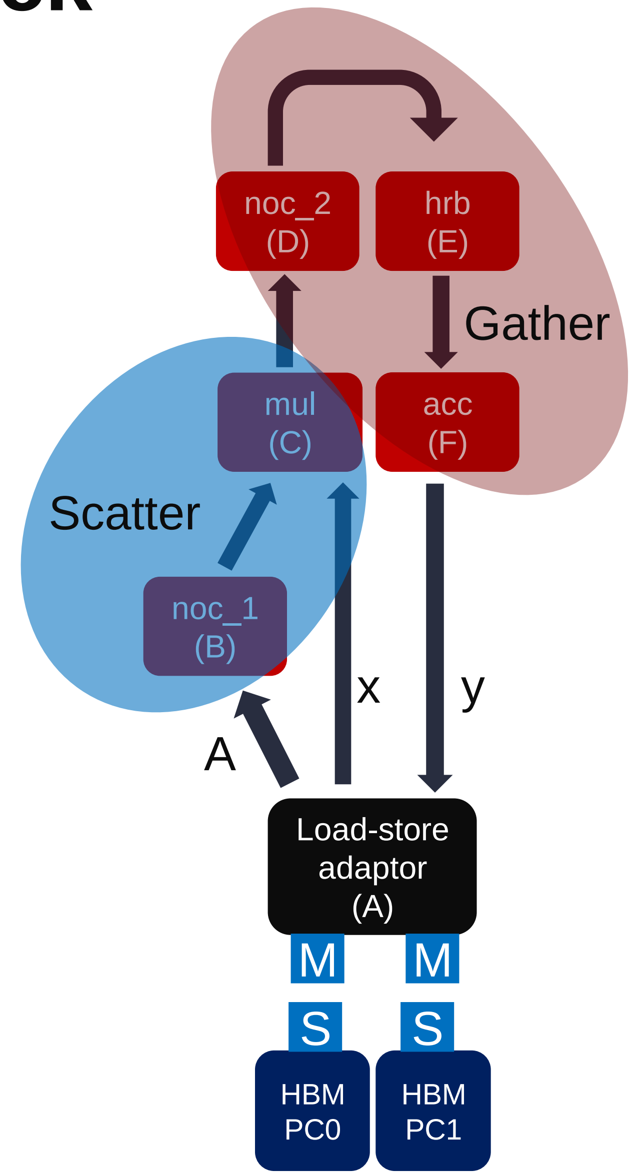


1. Huang, Sitao, et al. "Accelerating sparse deep neural networks on fpgas." IEEE High Performance Extreme Computing Conference (HPEC), 2019.

Streaming Dataflow Pipeline for SpMV Block

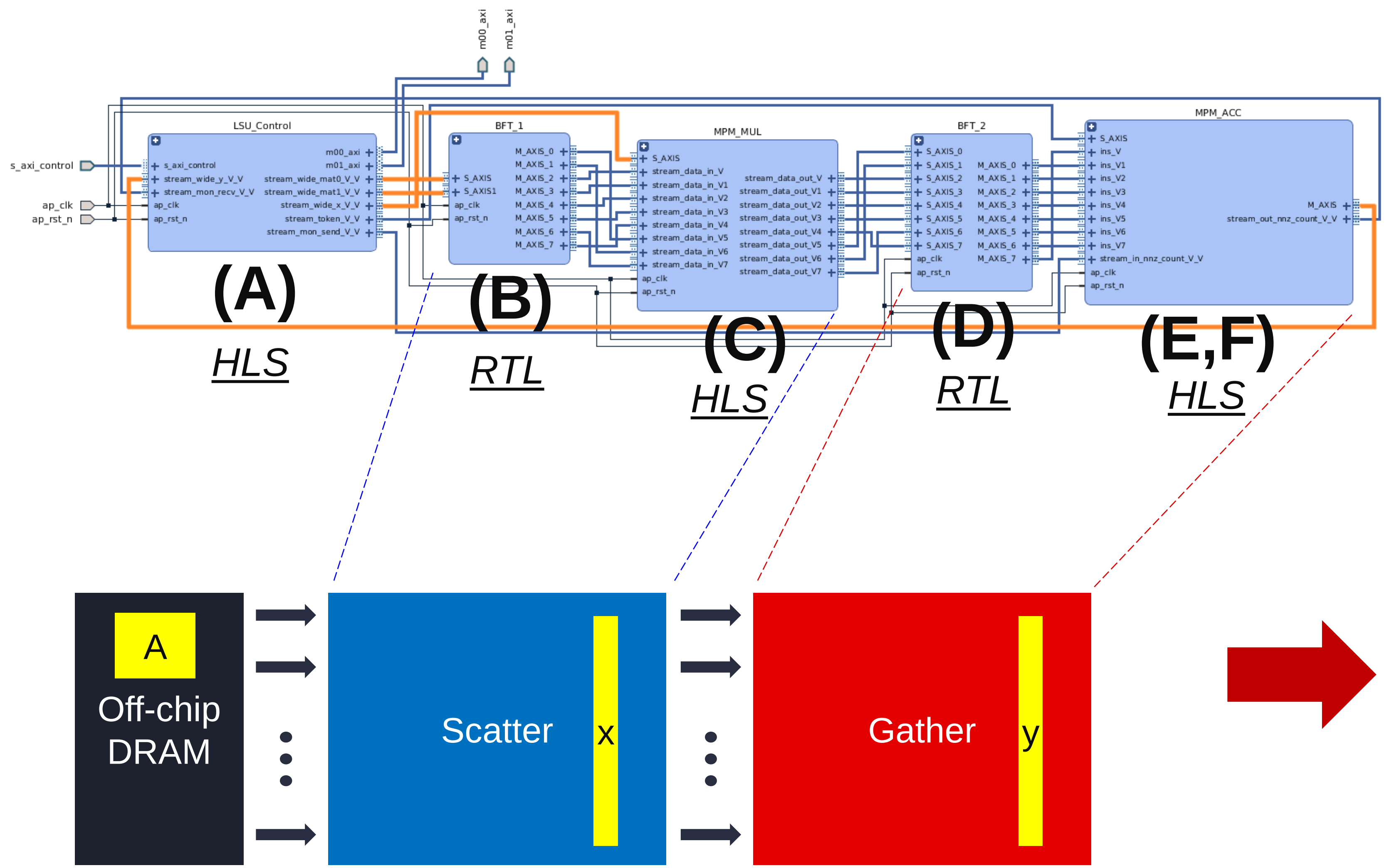
- Load-store adaptor to supply data in and out of compute pipeline
  - Each HBM PC → 32-Byte (32B) interface → packs 4 non-zeros
  - Each non-zero → 8B packed tuple → {4B FP32 value, 2B row id, 2B col id}
- Operations for SpMV:
  - load vector x from 32B memory interface PC1 (8 FP32 entries in parallel)
  - stream matrix A from PC0 and PC1 (8 non-zero in parallel, 4 from PC0 and 4 from PC1)
  - store vector y to PC1 (8 FP32 entries in parallel)
- Streaming dataflow pipeline built using
  - FPGA-optimized NoC RTL (B and D) and HLS-generated building blocks (A, C and F)

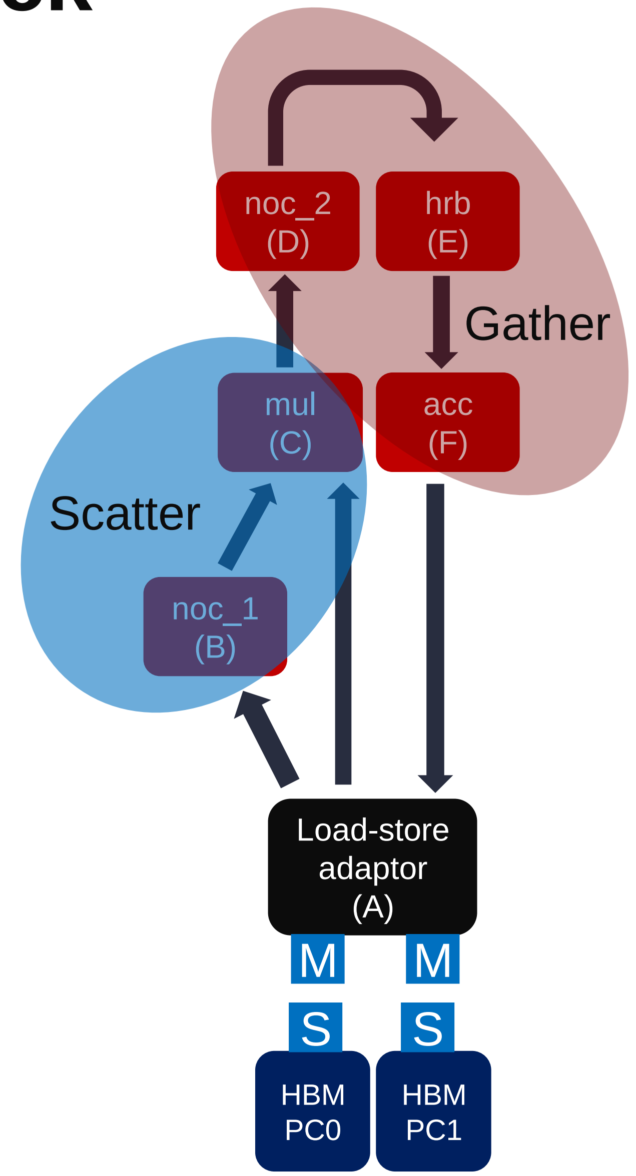




XILINX

Streaming Dataflow Pipeline for SpMV Block





XILINX

Streaming Dataflow Pipeline for SpMV Block

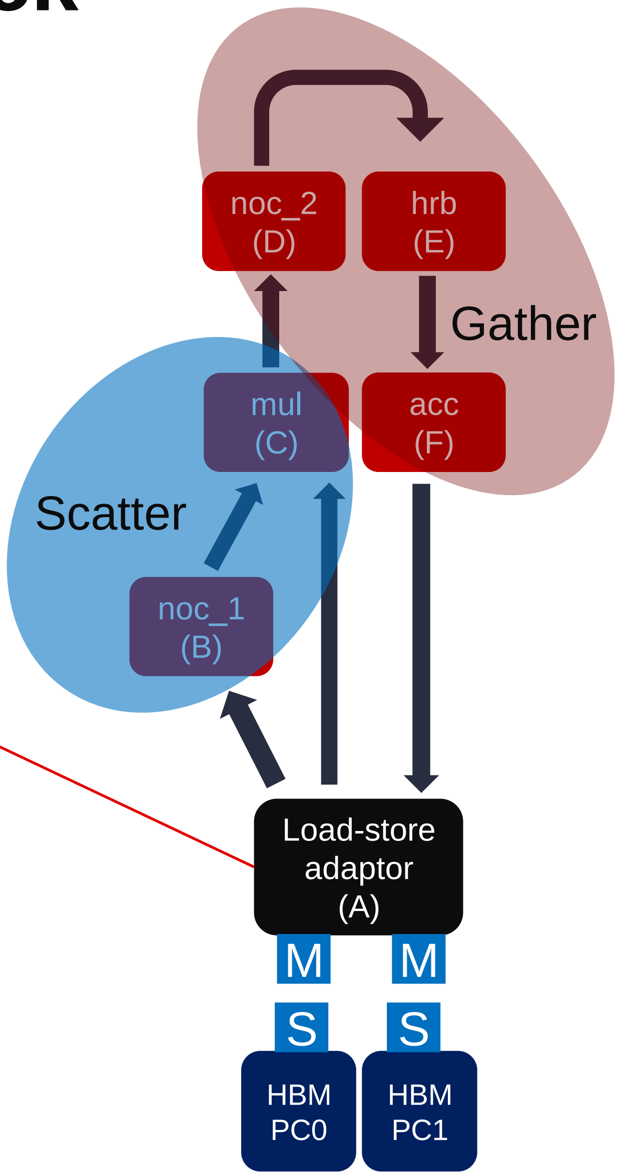
```
void example(wide_mat_dt *matrix, wide_vec_dt *x, wide_vec_dt *y,
             stream_wide_mat_dt &stream_wide_mat, stream_wide_vec_dt &stream_wide_x,
             stream_token_dt &stream_token, stream_wide_vec_dt &stream_wide_y,
             int mat_size, int vec_size)
{
#pragma HLS INTERFACE m_axi port=matrix offset=slave bundle=gmem0 num_read_outstanding=8 max_read_burst_length=16 depth=256
#pragma HLS INTERFACE m_axi port=x offset=slave bundle=gmem1 num_read_outstanding=8 max_read_burst_length=16 depth=256
#pragma HLS INTERFACE m_axi port=y offset=slave bundle=gmem1 num_read_outstanding=8 max_read_burst_length=16 depth=256

#pragma HLS INTERFACE axis register both port=stream_wide_mat
#pragma HLS INTERFACE axis register both port=stream_wide_x
#pragma HLS INTERFACE axis register both port=stream_token
#pragma HLS INTERFACE axis register both port=stream_wide_y

#pragma HLS INTERFACE s_axilite port=matrix bundle=control
#pragma HLS INTERFACE s_axilite port=x bundle=control
#pragma HLS INTERFACE s_axilite port=y bundle=control
#pragma HLS INTERFACE s_axilite port=mat_size bundle=control
#pragma HLS INTERFACE s_axilite port=vec_size bundle=control
#pragma HLS interface s_axilite port=return bundle=control

    wide_vec_dt temp_x;
    for(int i = 0; i < vec_size; i++) {
    #pragma HLS PIPELINE II=1
        temp_x = x[i];
        stream_wide_x.write(temp_x);
    }

    wide_mat_dt temp;
    for(int i = 0; i < mat_size; i++) {
    #pragma HLS PIPELINE II=1
        temp = matrix[i];
        stream_wide_mat.write(temp);
    }

    token_dt token = vec_size*2 + 1;
    stream_token.write(token);

    wide_vec_dt temp_y;
    for(int i = 0; i < vec_size; i++) {
    #pragma HLS PIPELINE II=1
        temp_y = stream_wide_y.read();
        y[i] = temp_y;
    }
}
```

Load-store adaptor (A)

Load vector x

Stream matrix A

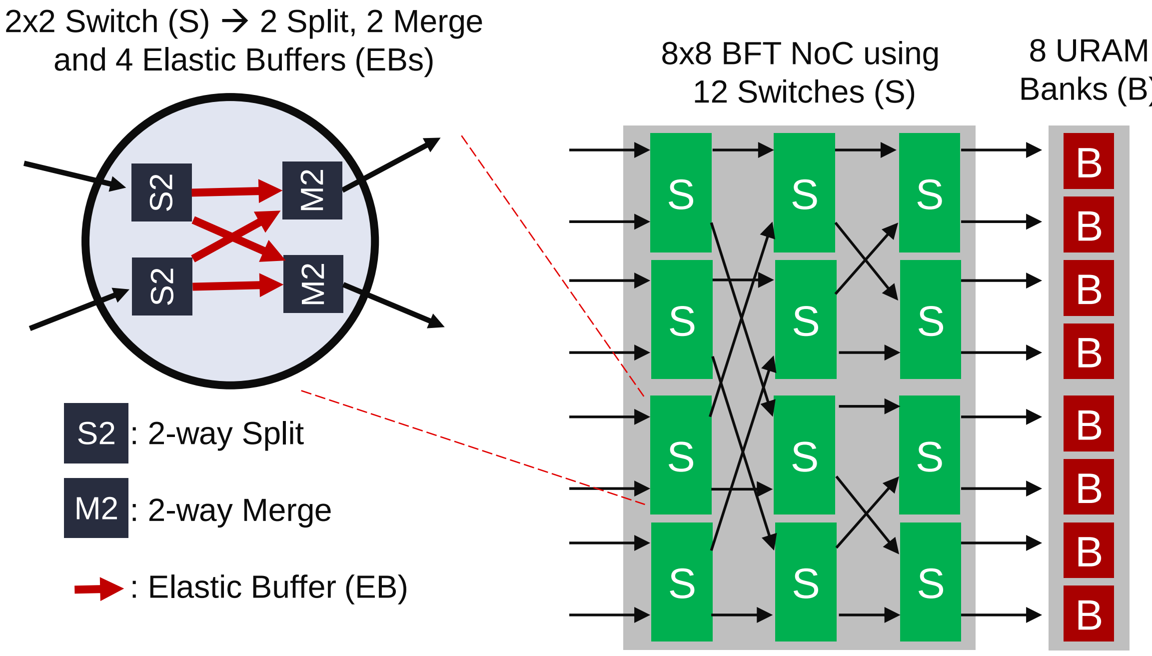
Store vector y

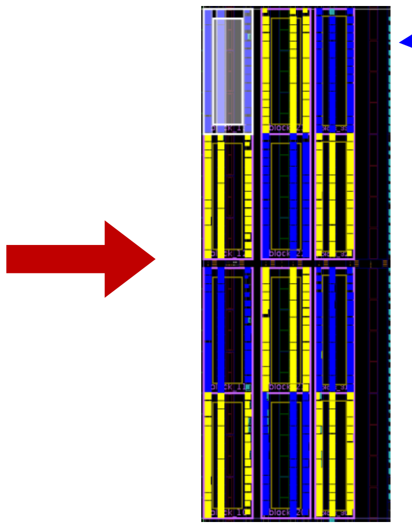


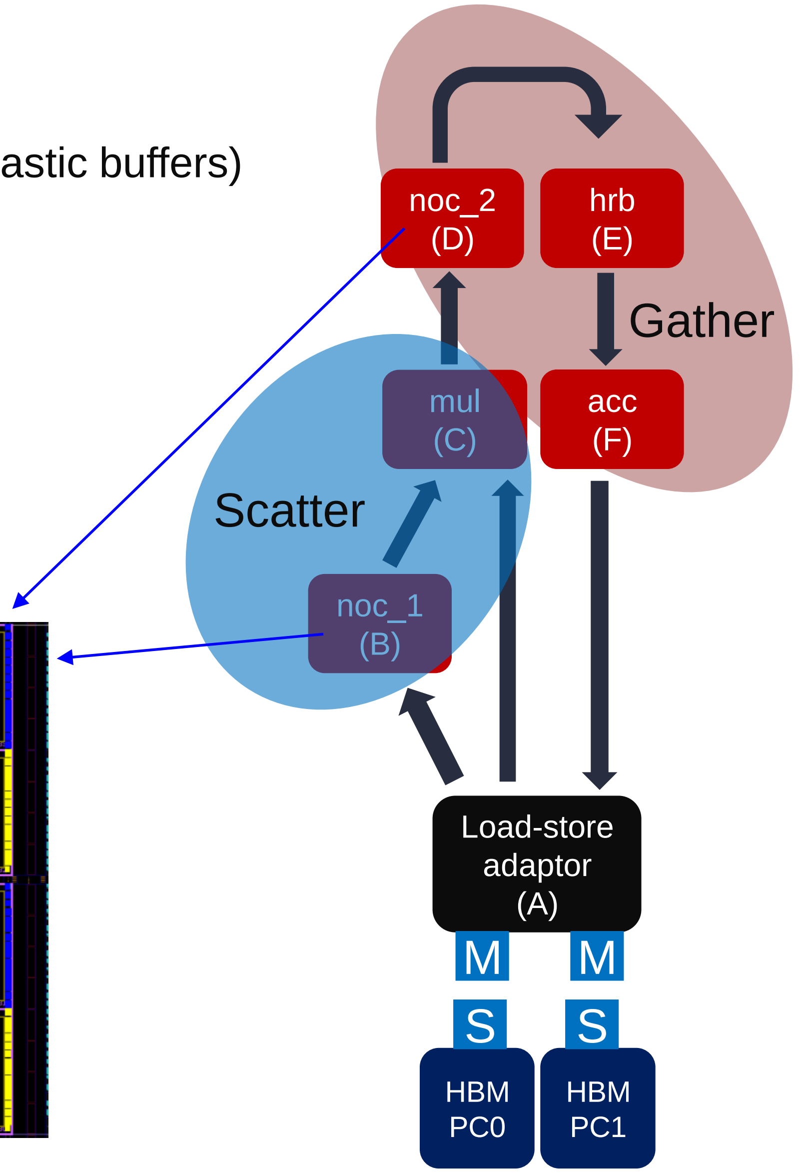
XILINX

Butterfly Fat Tree (BFT) NoC

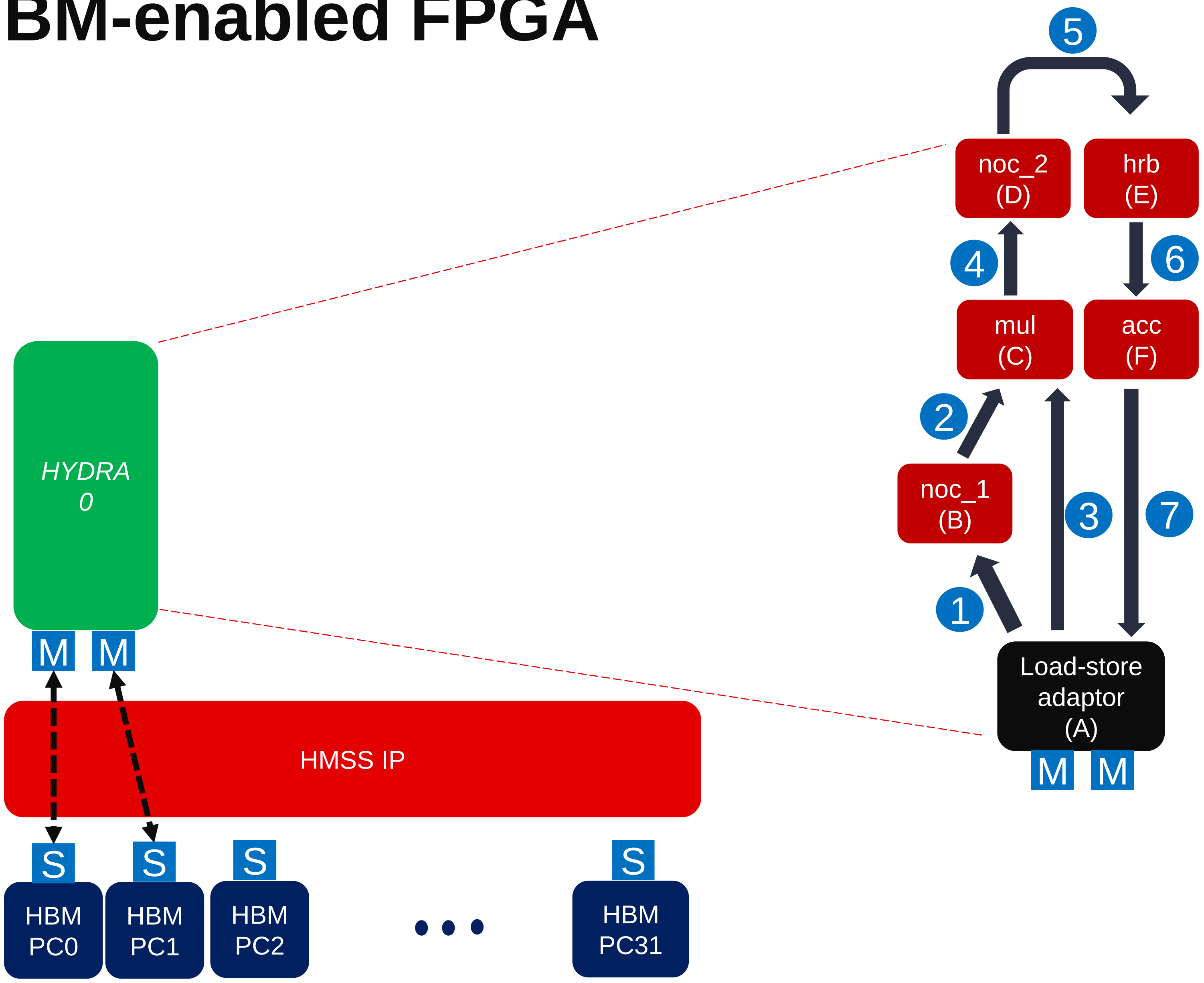
- FPGA-optimized 2x2 switches (S) built around dataflow units (split, merge and elastic buffers)
- Flow-control using ready-valid handshake
- 8x8 BFT NoC using multi-stage switching network (12 switches)





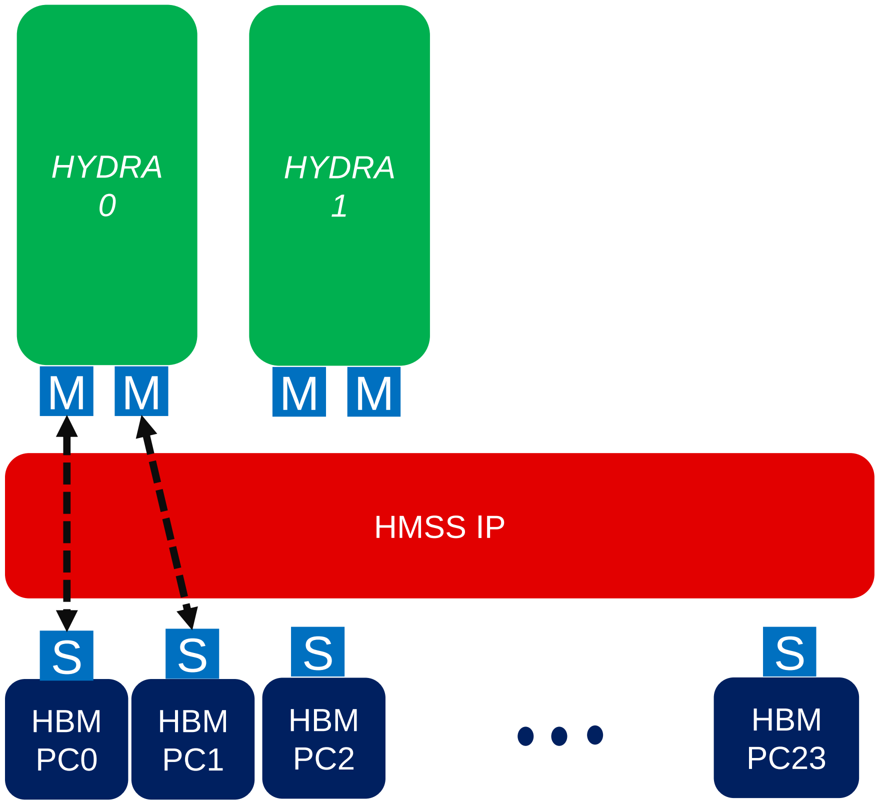


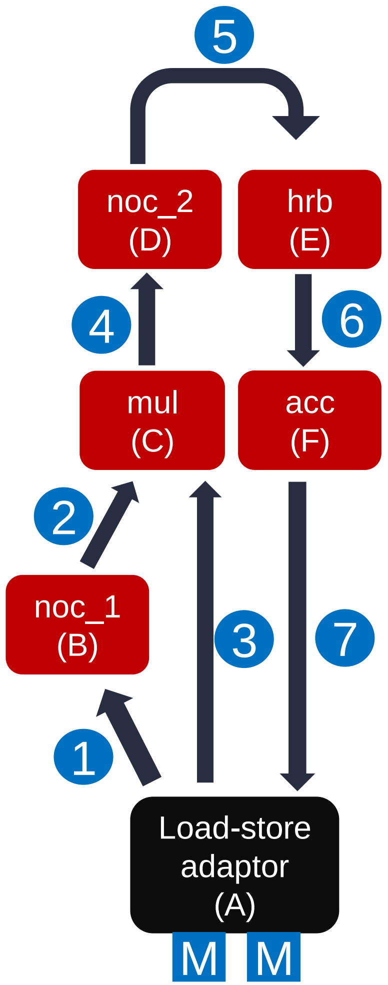
SpMV appliance on HBM-enabled FPGA



XILINX

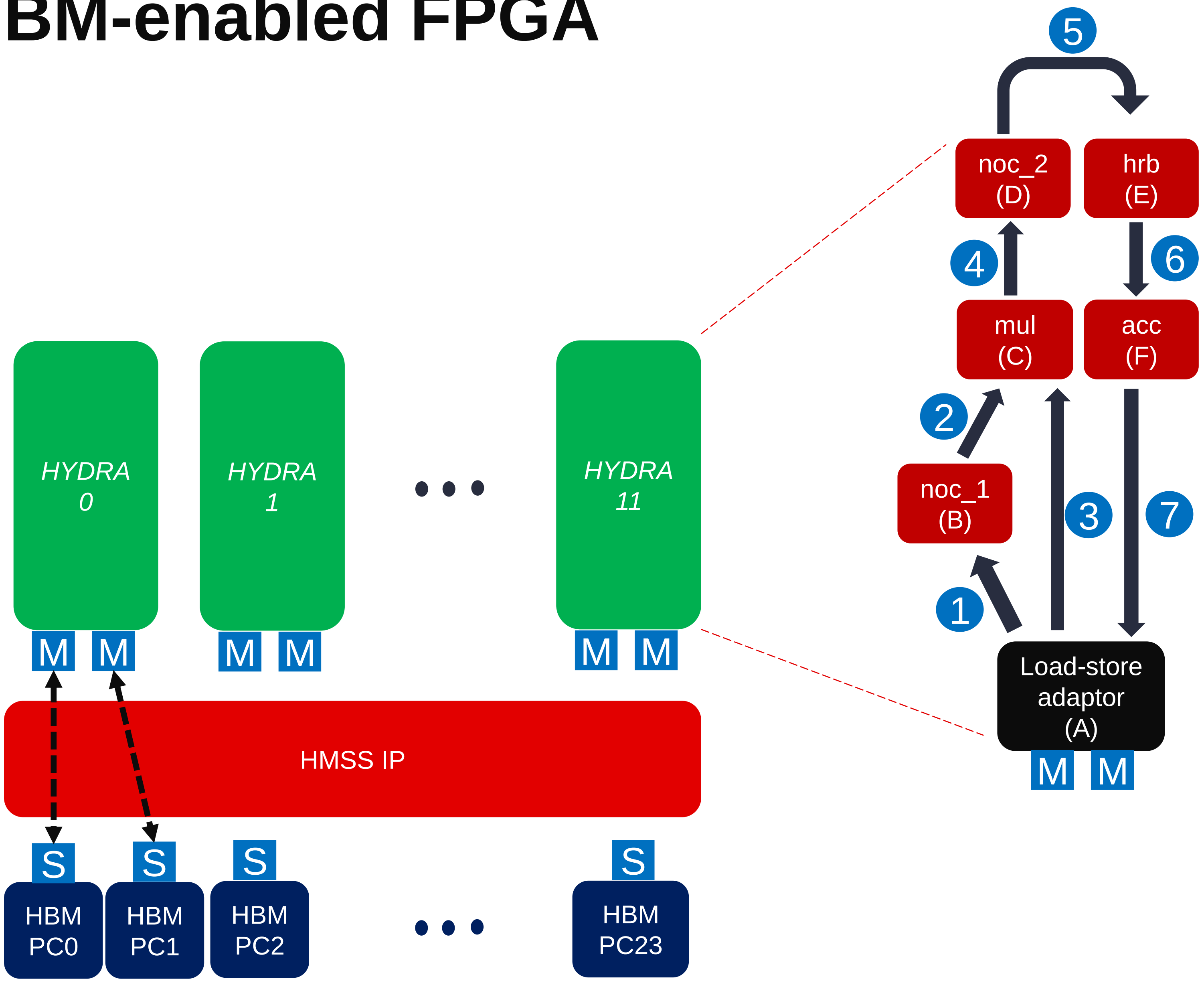
SpMV appliance on HBM-enabled FPGA



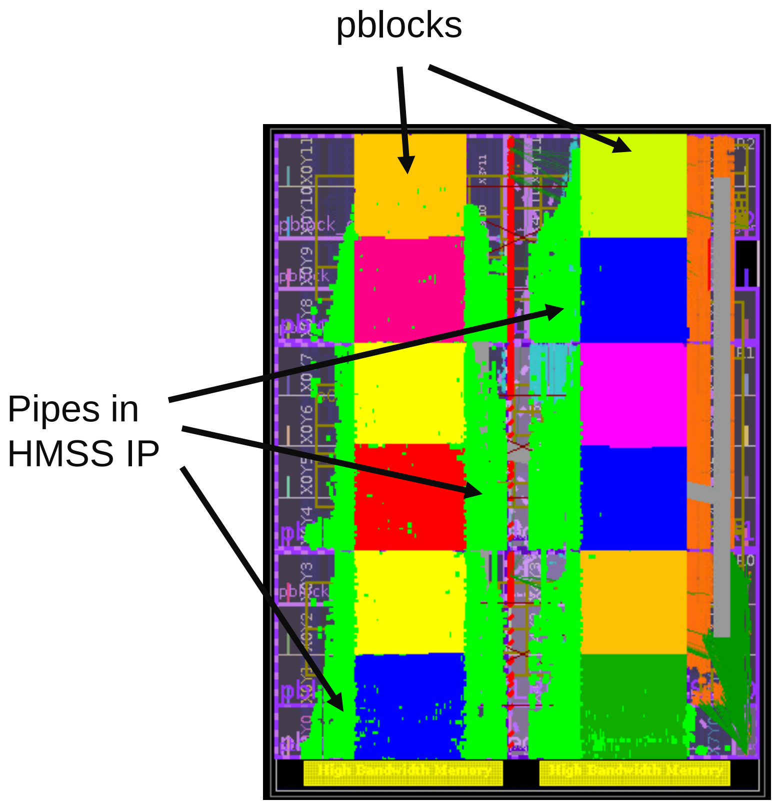


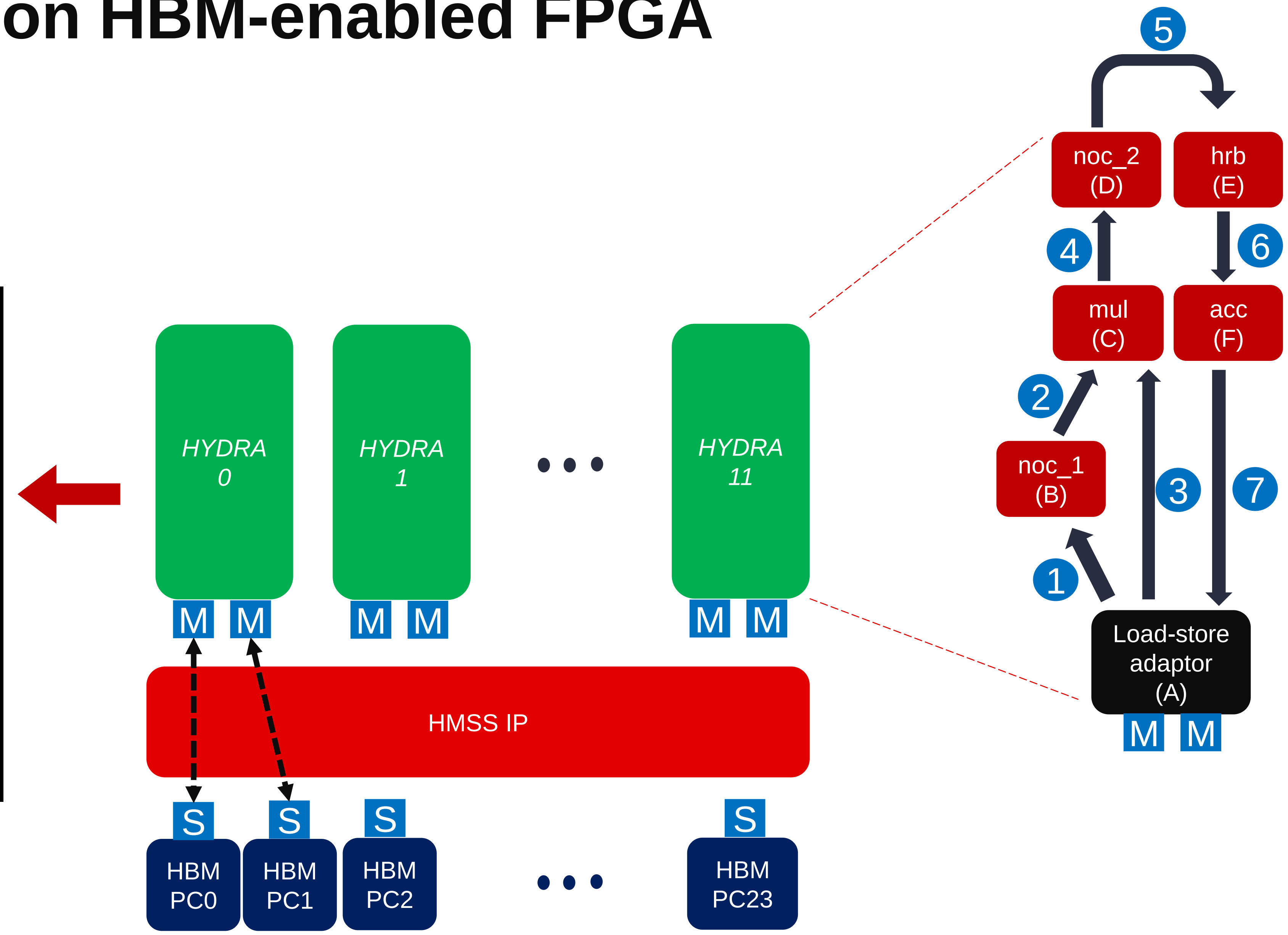
SpMV appliance on HBM-enabled FPGA

- SpMV appliance using HYDRA:
  - One block is able to saturate two HBM channels
  - AXI master 32B interface: M
  - AXI slave 32B interface: S
  - ***AXI stream interfaces connecting six sub-blocks (A,B,C,D,E and F) via AXI stream channels (1,2,3,4,5,6 and 7)***
  - ***1 and 2 are 8B wide 8 parallel channels***
  - ***3 4B wide 8 parallel channels***
  - ***4, 5 and 6 are 6B wide 8 parallel channels***
  - ***7 32B wide single channel***
  - Our implementation for SDNN challenge uses 12 HYDRA blocks and 24 HBM PCs



SpMV appliance on HBM-enabled FPGA

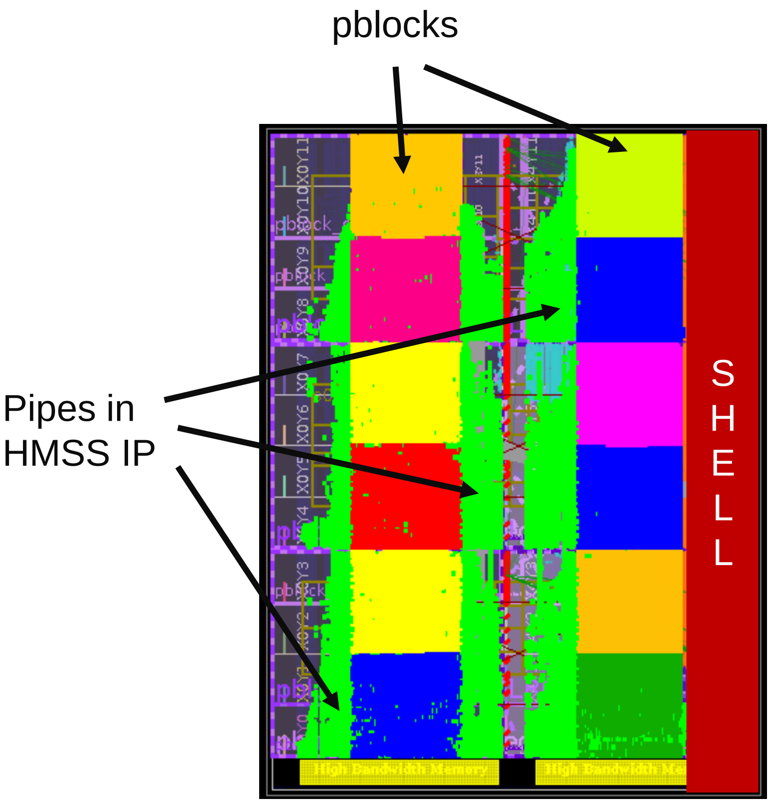




XILINX

Implementation on HBM-enabled FPGA

- SpMV appliance implemented on Alveo U280 (Vivado 2020.1)
  - host app (OpenCL) for managing the entire appliance and data movement between host and appliance
  - host communicates with U280 using PCIe Gen 3x16 (matrix A and vectors x,y)
  - sparse matrix: COO-encoded, with FP32 data and 2-Byte indexes
- Timing closed at 275 MHz for 12 block design
  - manual floorplan of each *HYDRA* block
  - each block uses < 3.33% of device resources
  - bias and ReLU adds extra eight FP32 adders and ReLU operators
  - throughput of each block is up-to 8 non-zero (or edges) every cycle @ 275 MHz

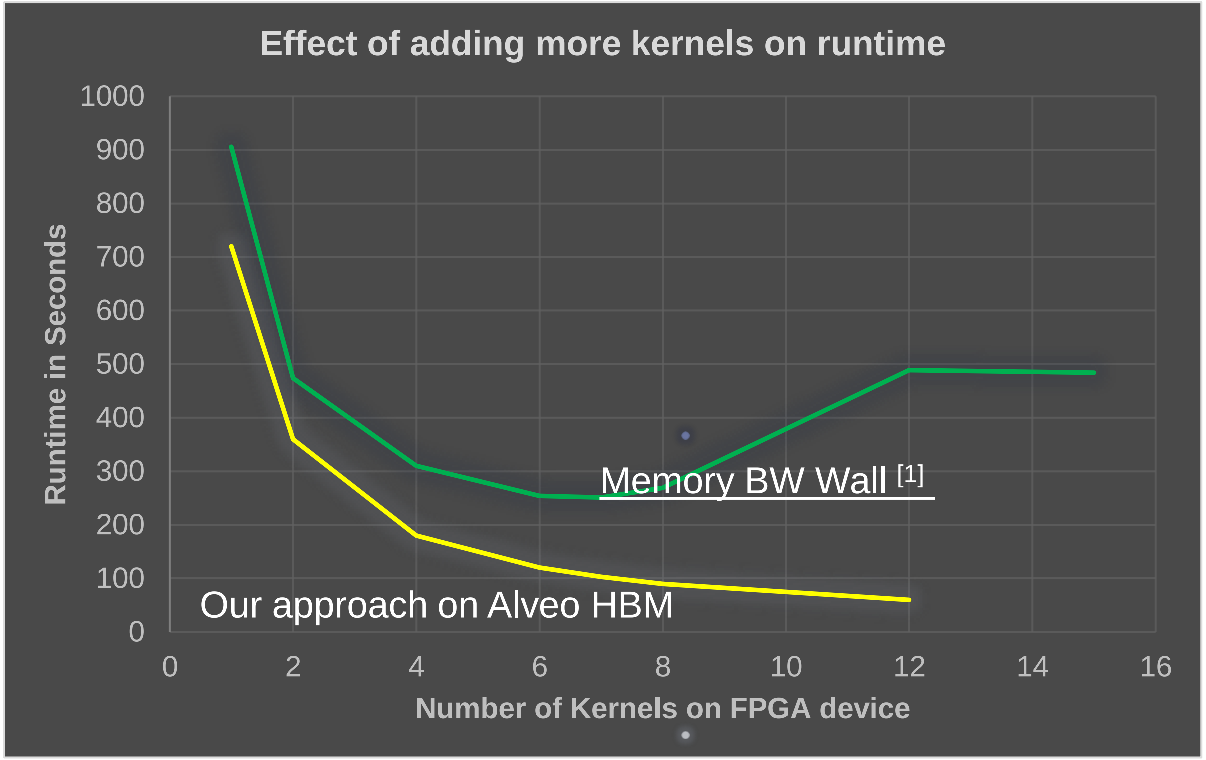


Resources	FFs	LUTs	DSPs	BRAMs	URAMs
HYDRA block	50K (2%)	25K (2%)	40 (0.5%)	24 (1.2%)	32 (3.33%)

Challenge Results: 120-layer network (1024 neurons)

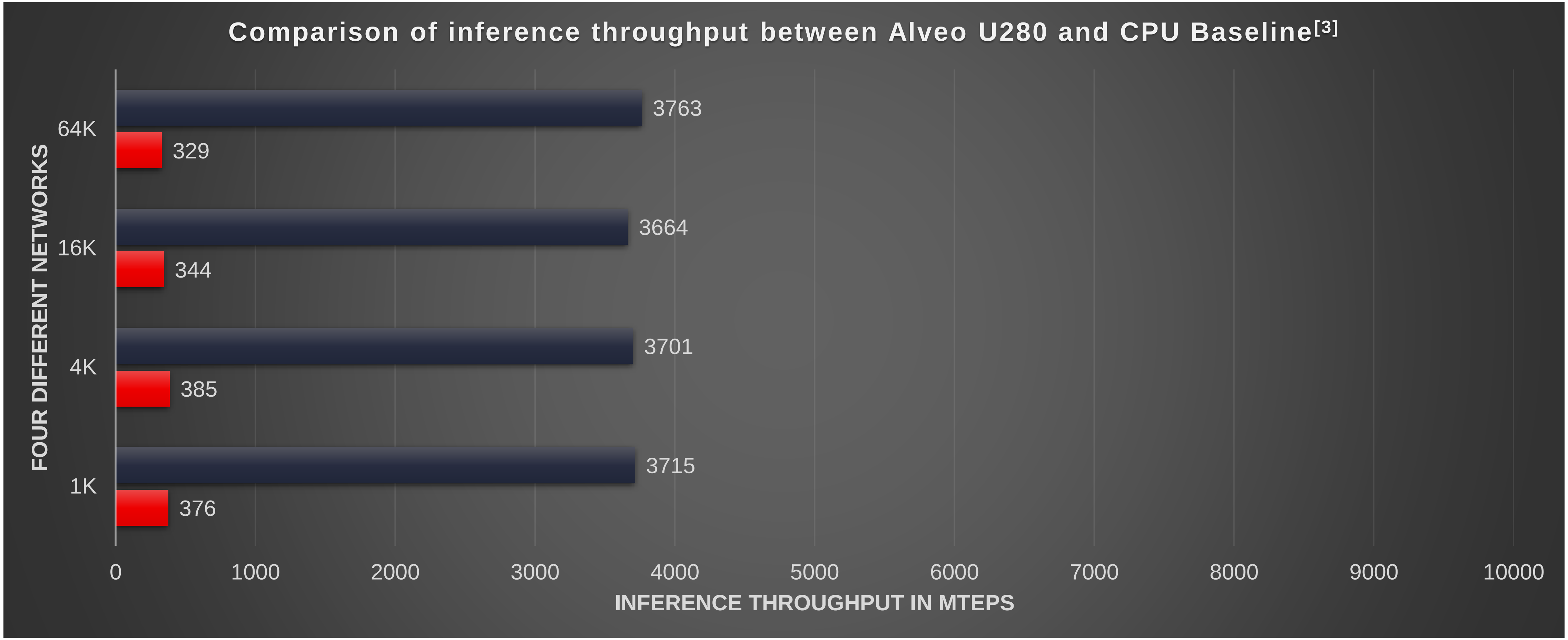
- Previous FPGA implementation[1] hits a wall at 7 blocks → low memory bandwidth available on the platform (only 30 GB/s)
- In our case: run time is scaling linearly by providing more blocks and more HBM channels

	Previous FPGA Implementation [1]	Ours on Alveo U280 with 12 *HYDRA* blocks
Best run time	252 seconds	63 seconds (4x faster)
Feature vector storage	Replicate input vectors to expose more read ports	No need to replicate (multi-ported multi-banked buffer)
Neurons allowed	Only 1K neurons (due to replication on vectors)	All the neurons in the challenge (1K, 4K, 16K, 64K)
Flexibility	Require different bitstream for each network	Single bitstream for all networks in SDNN challenge
Arithmetic	16-bit Fixed point	Floating point FP32
Implementation	Memory bound on VC709 FPGA Board	No longer memory bound on Alveo U280 HBM platform



1. Huang, Sitao, et al. "Accelerating sparse deep neural networks on fpgas." IEEE High Performance Extreme Computing Conference (HPEC), 2019.

Challenge Results: 120-layer networks for all neuron size



[3] Kepner, Jeremy, et al. "Sparse deep neural network graph challenge." IEEE High Performance Extreme Computing Conference (HPEC) 2019.
[9] Huang, Sitao, et al. "Accelerating sparse deep neural networks on fpgas." IEEE High Performance Extreme Computing Conference (HPEC), 2019.

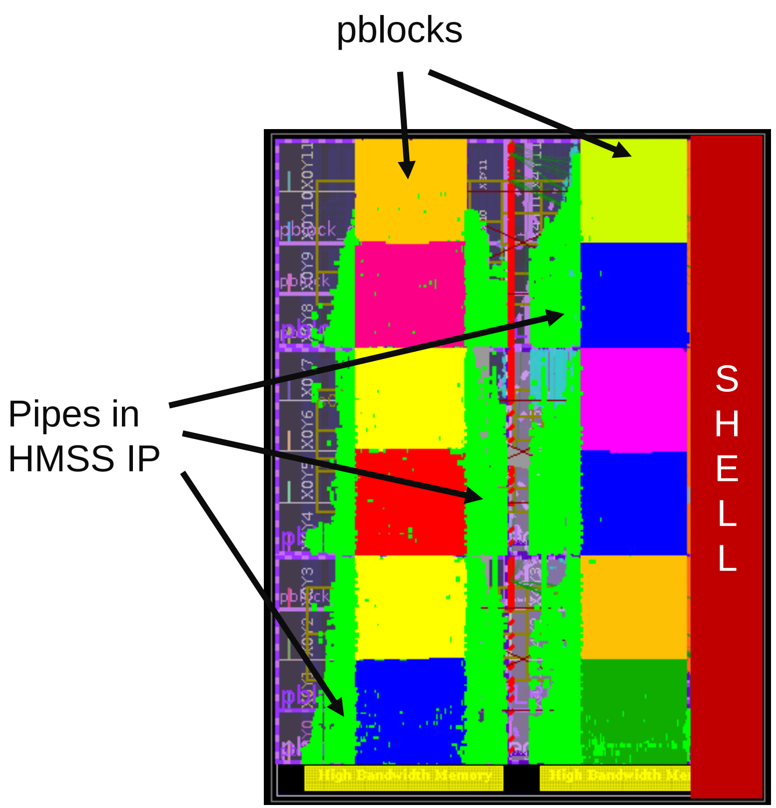
Challenge Results: 10x speedup over CPU baseline

Neurons per Layer	Layers	Connections (edges)	Time (Seconds)	Inference Rate	Time (seconds)	Inference Rate	Time (seconds)	Inference Rate
			CPU baseline [3]		[9]		Ours	
1024	120	3932160	626	376×10^6	251	940×10^6	63.5	3715×10^6
1024	480	15728640	2440	386×10^6	–	–	251	3759×10^6
1024	1920	62914560	9760	386×10^6	–	–	997	3786×10^6
4096	120	15728640	2446	385×10^6	–	–	255	3701×10^6
4096	480	62914560	10229	369×10^6	–	–	985	3832×10^6
4096	1920	251658240	40245	375×10^6	–	–	3917	3854×10^6
16384	120	62914560	10956	344×10^6	–	–	1030	3664×10^6
16384	480	251658240	45268	333×10^6	–	–	3916	3855×10^6
16384	1920	1006632960	179401	336×10^6	–	–	15664	3856×10^6
65536	120	251658240	45813	329×10^6	–	–	4012	3763×10^6
65536	480	1006632960	202393	299×10^6	–	–	16327	3699×10^6
65536	1920	4026531840	–	–	–	–	63478	3699×10^6

[3] Kepner, Jeremy, et al. "Sparse deep neural network graph challenge." IEEE High Performance Extreme Computing Conference (HPEC) 2019.
[9] Huang, Sitao, et al. "Accelerating sparse deep neural networks on fpgas." IEEE High Performance Extreme Computing Conference (HPEC), 2019.

Conclusions and Future Work

- Presented a high-performance SpMV block, *HYDRA,* which supports
  - completely unstructured sparse matrices
  - floating point FP32 arithmetic
- Used *HYDRA* for constructing an appliance which can be
  - used for sparse data processing
  - adopted in both edge and data center (cloud) scenarios
- Demonstrated that the SpMV appliance can be used for DNN workloads
  - by running a variety of sparse neural net workloads given as part of the SDNN challenge
  - linear scaling in inference throughput performance as we activate up-to 12 blocks
  - 3.7 billion edges per second inference throughput on Alveo U280 platform
  - 10x faster execution compared to challenge CPU baseline
- Planning to extend this work
  - to support Sparse Matrix by Sparse Matrix (SpMSpM) multiplication in *HYDRA* block
  - to support reuse of streaming weight matrix across multiple feature vectors instead of just one



XILINX

Thank You