SANTA CLARA UNIVERSITY
SCHOOL OF ENGINEERING

Efficient Deployment of Very Wide and Very Deep Hypersparse FFNs on FPGA

Paramdeep Singh
Computer Science and Engineering
Santa Clara University
Santa Clara, CA, USA
psingh7@scu.edu

David C. Anastasiu
Computer Science and Engineering
Santa Clara University
Santa Clara, CA, USA
danastasiu@scu.edu

Santa Clara University

Introduction

- Feed Forward Networks (FFNs) are an integral part of many deep neural networks.
- FFNs account for up to two-thirds of the processing cost of LLMs.
- Compressing FFNs can lead to substantial reduction in overall processing cost of many deep neural network architectures, including LLMs.
- FFN compression techniques like pruning and quantization yield irregular structures well-suited or FPGA processing.

Problems solved by this paper

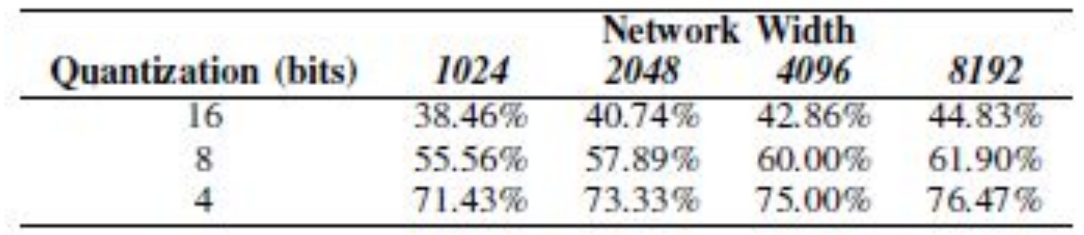
- Minimizing off-chip memory accesses. FPGAs are primarily used as custom processors, relying on off-chip DRAM for data.

- Optimizing usage of scarce Block Ram (BRAM) and UltraRAM (URAM).

- Parallelizing the custom processing logic.

Background

- Unstructured sparsity is the most robust form of sparsity.
- GPUs support structured sparsity (N:M)* only.
- FPGAs uniquely suited to process inference of FFNs with unstructured sparsity.
- Data structures used to support structured sparsity (indices arrays) can consume the bulk of scarce BRAM on the FPGA fabric.



	Network Width			
Quantization (bits)	*1024*	*2048*	*4096*	*8192*
16	38.46%	40.74%	42.86%	44.83%
8	55.56%	57.89%	60.00%	61.90%
4	71.43%	73.33%	75.00%	76.47%

*N:M, sparsity means that in a contiguous block of M elements within the tensor, exactly 2 elements are allowed to be non-zero, and the other 6 are forced to be zero.

Our Solution

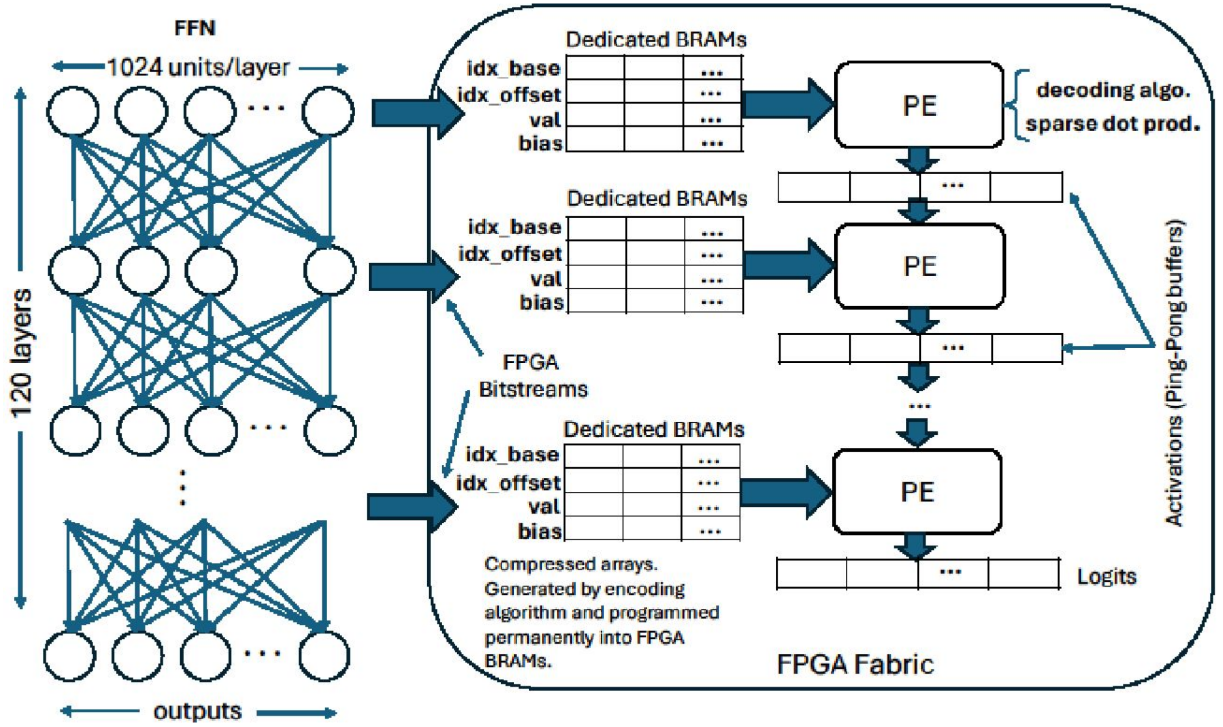
- Use a sparse architecture with robustness of unstructured sparsity, sans all the processing overhead associated with it.

- RadixNet pre-pruning fits the bill.

- **Key observation**: Fan-out of all neurons in a RadixNet layer is identical, and a factor of the network width.
  - Develop deterministic compression and decompression algorithms for indices arrays.
  - Achieve drastic reduction in BRAM required to store indices arrays.
  - Eliminate off-chip DRAM accesses.
  - 1000X improvement in inference performance.

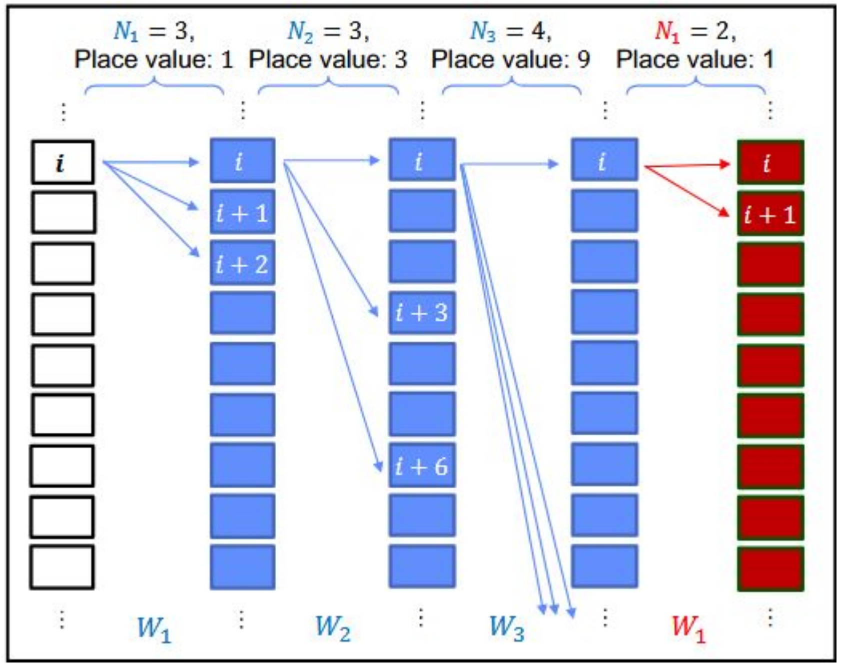
Method – Accelerator Architecture

- All data required for forward pass on FPGA fabric.
- Dedicated Processing Element (PE) for each network layer (pipeline stage)
- Dedicated BRAMs for each PE.
- Sparse, compressed weights (4-bit).
- Compressed indices arrays.
- Dense activations.



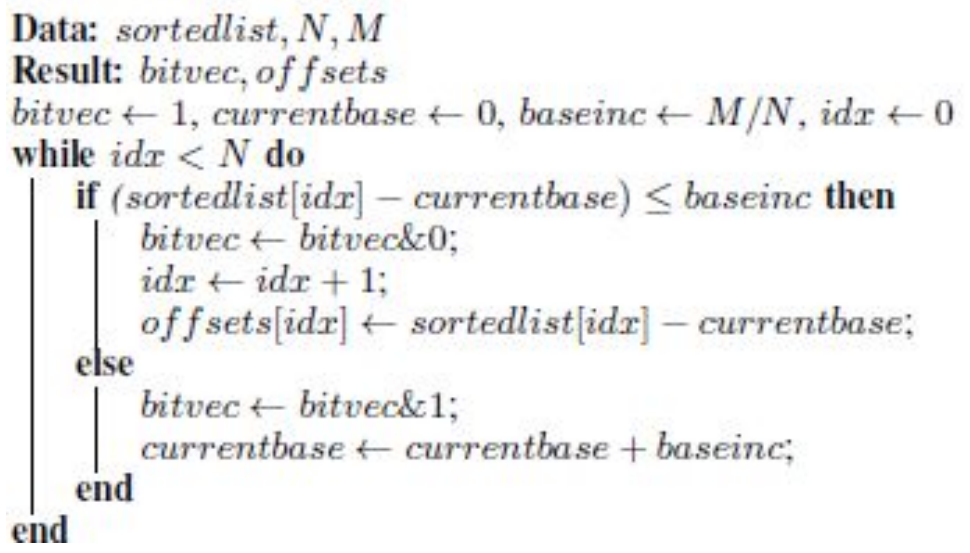
Method – Radix-Nets Pre-Pruning

- Hypersparsity with complete path connectedness. (each output connected to all inputs)
- Performance at par with unstructured pruning.
- Flexibility in architecture design (variable units per layer, total number of layers, concatenation, etc.)
- Training Challenges


$N_1 = 3$, Place value: 1
$N_2 = 3$, Place value: 3
$N_3 = 4$, Place value: 9
$N_1 = 2$, Place value: 1
i
$i+1$
$i+2$
$i+3$
$i+6$
W_1
W_2
W_3
W_1

Method – Encoding Algorithm

- Pre-Processing Step.
- Considers the weight matrix as a single block with structured sparsity in the form N:M , where N ≪ M .
- Compresses indices array of CSR representation into bases and offset arrays.
- Bases of all NNZs in a neuron packed into single bit vector.
- The length of the bit vector is bounded by **2N** bits.



```
Data: $sortedlist, N, M$
Result: $bitvec, offsets$
$bitvec \leftarrow 1$, $currentbase \leftarrow 0$, $baseinc \leftarrow M/N$, $idx \leftarrow 0$
while $idx < N$ do
    if $(sortedlist[idx] - currentbase) \leq baseinc$ then
        $bitvec \leftarrow bitvec\&0$;
        $idx \leftarrow idx + 1$;
        $offsets[idx] \leftarrow sortedlist[idx] - currentbase$;
    else
        $bitvec \leftarrow bitvec\&1$;
        $currentbase \leftarrow currentbase + baseinc$;
    end
end
```

Method – Encoding Algorithm (Example)

Array of absolute indices (bit-width of each index = $\log_2$(Network Width))

0001111011	0001111011	0010000101	0011010000	0011010111	0011101111	0011111101	0100010001
0100011000	0100101110	0100110000	0101011010	0110011000	0110110010	0111100110	0111101100
0111110100	1000001111	1000100000	1000110100	1001111010	1010001111	1010111000	1011010100
1011100011	1100000000	1100100010	1101010101	1101101100	1110010111	1110011011	1110111000

idx_offset array

11011	11011	00101	10000	10111	01111	11101	10001	11000	01110	10000	11010	11000	10010	00110	01100
10100	01111	00000	10100	11010	01111	11000	10100	00011	00000	00010	10101	01100	10111	11011	11000

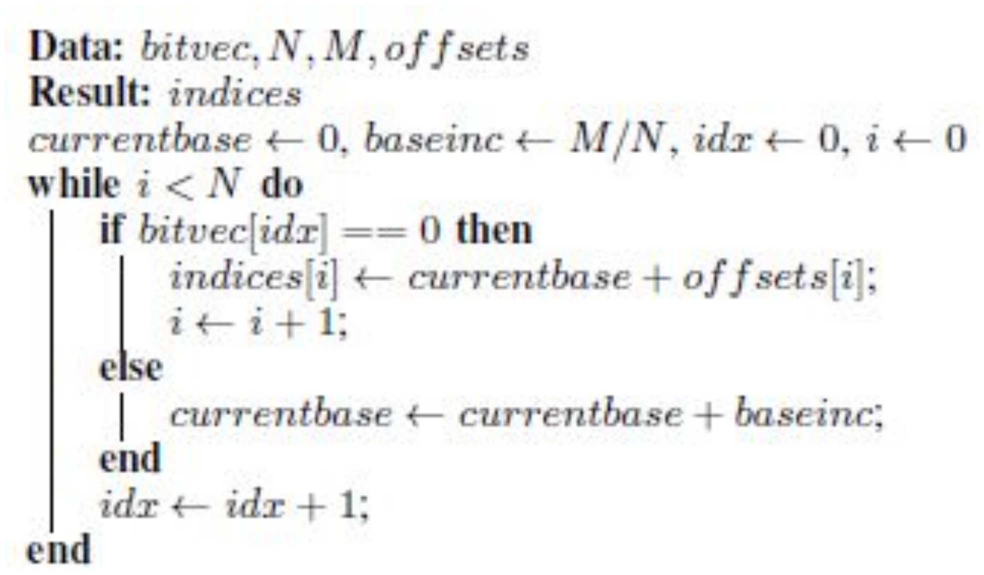
single entry in idx_base array (bitmap corresponding to neuron)

1111001011001001001001011010110001010011010101010101010101001000 ← Entry padded to 64 bits

↑ Initialization bit

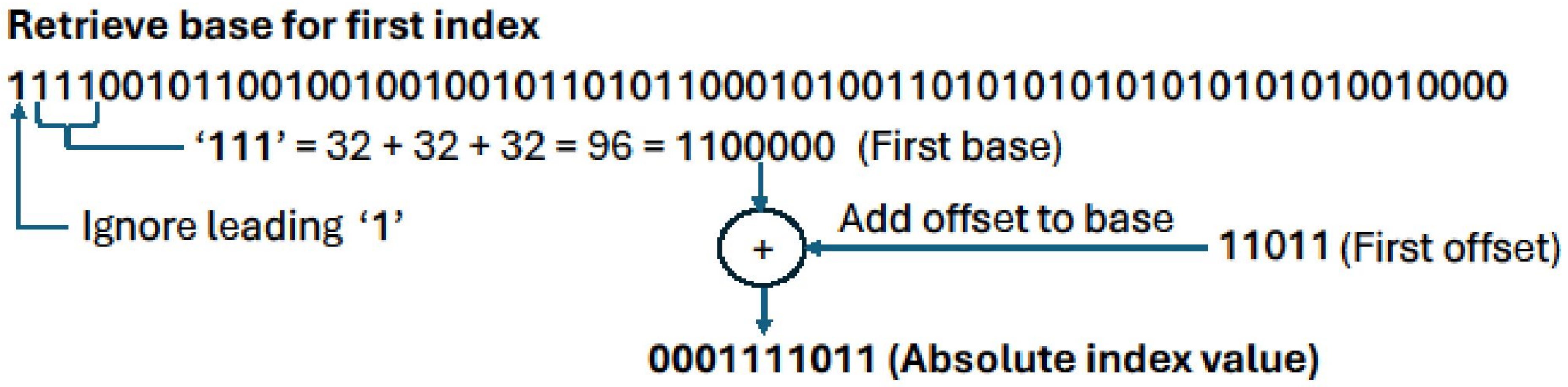
Method – Decoding Algorithm

- Implemented as custom logic in each PE.
- Scans bit vector created by encoding algorithm to recover bases of all NNZs.
- Combines bases with offsets to recover indices if NNZs.



```
Data: $bitvec, N, M, offsets$
Result: $indices$
$currentbase \leftarrow 0$, $baseinc \leftarrow M/N$, $idx \leftarrow 0$, $i \leftarrow 0$
while $i < N$ do
    if $bitvec[idx] == 0$ then
        $indices[i] \leftarrow currentbase + offsets[i]$;
        $i \leftarrow i + 1$;
    else
        $currentbase \leftarrow currentbase + baseinc$;
    end
    $idx \leftarrow idx + 1$;
end
```

Method – Decoding Algorithm (Example)



Experiment Design - Datasets

- MNIST Dataset (32 X 32 images)
- 60K training images, 10K test images, 15 K validation images
- Primary effectiveness metric – accuracy

Experiment Design – Networks

- Networks pre-pruned using RadixNet pre-pruning.
- Deep Networks (2,3,4,6 layers)
- Very deep networks (10, 20, 30 . . . , 120 layers). Constructed by concatenating deep networks.
- 1024 units per layer in all networks
- 10 units in output layer.
- 4-bit weights and activations. 8-bit biases. (Layer-wise quantization)

Experiment Design – Model Training

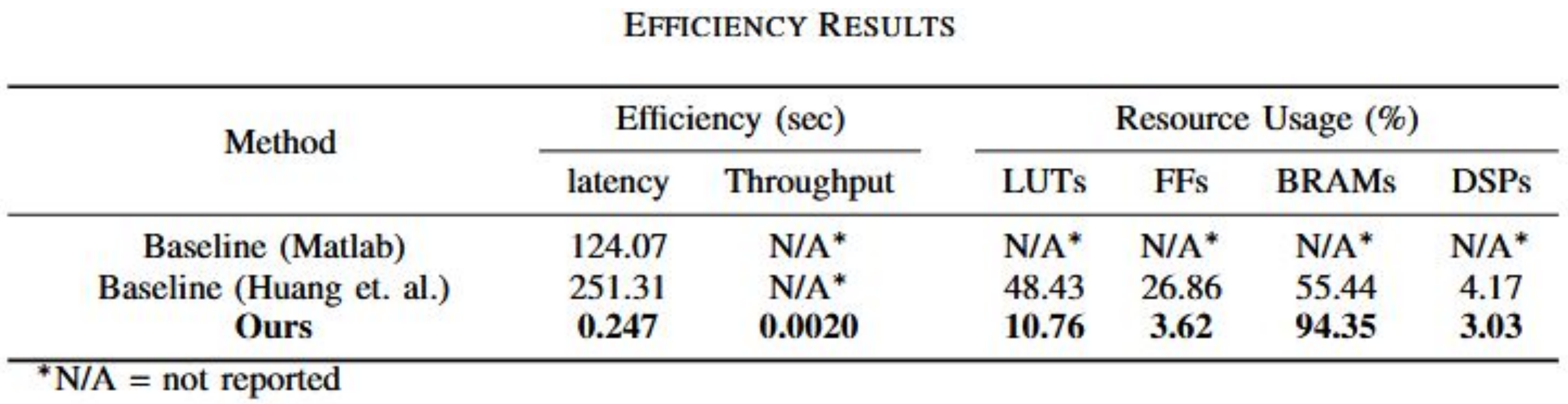
- All networks were trained with Pytorch 2.4.1 on a Super-micro SYS-420GP-TNAR+ system with NVIDIA HGX A100. (Only 1 GPU used)
- Quantization performed using Brevitas library.
- Adam Optimizer with learning rate = 10^{-3}
- Batch Size = 256, Loss = CCE
- Pre-pruned networks may need 2-3 training iterations for optimal results.

Experiment Design – FPGA Implementation

- Accelerator code generated in 100% HLS.
- Vitis-HLS 2024.2 environment.
- Code generator written in python.
- All arrays initialized as CONST to enable bitstream injection.
- Inference step implemented using Quantize-Dequantize-Quantize (int->float->int) approach.
- Scale factors extracted from ONNX representation of trained models.

Results - Efficiency

- Inference efficiency.
- FPGA resource usage



EFFICIENCY RESULTS

Method	Efficiency (sec)		Resource Usage (%)			
	latency	Throughput	LUTs	FFs	BRAMs	DSPs
Baseline (Matlab)	124.07	N/A*	N/A*	N/A*	N/A*	N/A*
Baseline (Huang et. al.)	251.31	N/A*	48.43	26.86	55.44	4.17
Ours	**0.247**	**0.0020**	**10.76**	**3.62**	**94.35**	**3.03**

*N/A = not reported

Results - Effectiveness

- Our method does not result in significant degradation in model accuracy.
- Limited degradation in accuracy (< 1%) between networks compressed using our method and their fully connected counterparts.

Q & A

Research Partners:

NVIDIA

SUPERMICRO