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

***Abstract*—Model compression techniques such as quantization and pruning have shown great promise in drastically reducing model size without degrading model effectiveness. Quantization of model parameters when combined with parameter pruning results in a significantly reduced model size. However, such sparse neural networks have irregular structures. As such the forward pass (inference step) of such networks cannot be executed efficiently by processing hardware like GPUs. FPGA's offer a flexible platform to process irregular sparse networks. However, in order to fully realize the efficiency gains promised by the FPGA architecture, it is essential to minimize or completely eliminate off-chip memory accesses. Accommodating a large model completely on the FPGA fabric is restricted by the scarcity of available high-speed on-chip RAM, forcing a fraction of model weights to be stored in off-chip DRAM. We propose a method to accommodate very wide and very deep hypersparse feed forward networks (FFNs) completely on the FPGA fabric by compressing data structures in addition to quantizing the network parameters. Our method makes it possible to fit large FFNs completely on the FPGA fabric, resulting in inference performance almost 1000x higher than that of the state-of-the-art.**

***Index Terms*—Deep Neural Networks, FPGA Neural Network Acceleration, Sparse Networks, High Performance Computing.**

I. Introduction

Feed Forward Networks (FFNs) have proven to be versatile architectures for many machine learning tasks, including image classification, recommender systems, and time series predictions. For some problems, it is essential that we obtain these predictions in real-time. In the medical domain, especially in the field of radiology, FFNs play an integral role in electro-cardiogram (ECG) de-noising [1]. They have also been extensively used as an effective means for data compression [2] to allow efficient use of bandwidth between communication endpoints. FFNs have also become an integral part of modern cognitive networks such as CNN's [3], [4] and Transformer-based LLMs [5], [6].

In this project, we focus on very deep and very wide sparse FFNs and how they can be deployed in the embedded domain, both as stand-alone and as backbone networks for more complex cognitive tasks. The flexible and versatile hardware architecture and low power requirements of FPGA's make them a promising solution for processing large FFNs efficiently, especially at the edge.

The processing cost of FFNs can be significantly reduced by pruning away unimportant weights, also known as sparsification, and weight quantization [7]. Recently, the availability of powerful sub-billion parameter LLMs and advances in model compression techniques has led to an era of Small Language Models (SLMs) [8]. SLMs are ideal candidates for deploying LLM capabilities at the edge. The unique hardware architecture of FPGAs is uniquely suited for hosting SLM's either completely on their own, or as a part of a System On Chip (SOC) solution. The irregular network structures resulting from the application of aggressive model compression techniques to SLMs, make SOCs with large FPGA a highly viable platform to deploy SLMs. Our work endeavors to develop highly efficient FPGA accelerators for sparse FFN's, which can be used as sub systems of larger SLM/CNN accelerators, or as stand-alone FFN accelerators. We focus on the following challenges which must be overcome to fully realize the benefits of the FPGA architecture to enable highly efficient inference at the edge:

- Minimizing off-chip memory accesses. FPGAs are primarily used as custom processors, relying on off-chip DRAM for data.
- Optimizing usage of scarce BRAM and URAM.
- Parallelizing the custom processing logic.

II. Background

Commodity inference hardware, such as GPUs, supports structured sparsity at the block level. Structured sparsity refers to the systematic pruning of neural network weights, where certain patterns of weights (e.g., entire blocks or groups) are set to zero, enabling hardware optimizations. For instance, the NVIDIA Ampere Architecture implements 2:4 sparsity, meaning that half of the weights in the network are set to zero in dense subgroups. Although structured sparsity offers computational benefits, it can lead to accuracy degradation [9]. As sparsity becomes more fine-grained, transitioning from block-level to global sparsity (e.g., 2:8, 2:16, etc.), the overhead for identifying indices of non-zero values (NNZs) increases. With globally pruned sparse networks, maintaining the absolute indices of NNZs becomes critical.

Compressed Sparse Row (CSR) is a widely used data structure for storing weight matrices in globally sparse networks.

CSR represents a sparse matrix using three arrays. The 'values' array stores all non-zero weights contiguously. The 'indices' array holds the row positions of these weights and matches the size of the 'values' array. Together, these arrays enable efficient representation and retrieval of sparse matrix data.

FPGAs are particularly well suited for performing Sparse Matrix Vector Multiply (SPMV) operations when sparse matrices are stored in the CSR format and the corresponding vectors are stored in dense format. These operations dominate the computational workload during the inference phase of feed-forward networks (FFNs). Although FPGAs can parallelize dot product computations due to their flexible architecture, achieving high throughput requires that all necessary data for the forward pass be stored on the FPGA fabric. A common bottleneck arises from frequent accesses to weights stored in DRAM and the subsequent writing of results back to DRAM.

Quantizing weights is an effective approach to reduce the total data size required for the forward pass. In sparse networks stored in the CSR format, quantization significantly reduces the number of bits needed to store the 'values' array. Recent studies demonstrate that weights can be quantized to very low bit-widths without compromising the effectiveness of FFNs [10]. However, for extremely wide and deep quantized sparse FFNs, the memory demand for storing the indices of NNZs can surpass that required for storing quantized weights. Table I shows the proportion of total memory occupied by indices arrays at different quantization levels for very wide FFNs. In this work, we present an algorithm to reduce the number of bits required to store indices information, enabling the accommodation of very large FFNs completely on the FPGA fabric.

TABLE I
PROPORTION OF MEMORY OCCUPIED BY INDICES ARRAYS

	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%

III. Related Work

The reduction of model size through parameter pruning [11], [12] has been extensively studied. The primary pruning approaches include Structured Pruning, Unstructured Pruning, and Pre-Pruning. Structured Pruning [13], [14] is a coarse-grained method that removes entire network components such as filters, layers, and channels. This approach enables efficient processing on commodity hardware like GPUs. However, structured pruning often results in a loss of model effectiveness. This degradation arises because block-sparse structures require a minimum number of non-zero elements (NNZs) within blocks, which may include globally insignificant weights. Structured pruning has been extensively explored in FPGA settings. Wang et al. proposed an FPGA accelerator for LSTM networks pruned via structured pruning [15]. Their method prunes entire columns of the weight matrix during training, removing columns whose cumulative weight magnitudes fall below a threshold. The resulting sparse matrix is processed using a bit mask that indicates pruned ('0') and unpruned ('1') indices. Similarly, Han et al. introduced load-balance-aware pruning, maintaining a constant pruning ratio across all submatrices to ensure equal work distribution among Processing Elements (PEs) [16]. Shi et al. proposed a hybrid approach that combines the benefits of structured and unstructured pruning [17]. By applying unstructured pruning within predefined structured matrix blocks, they limit the processing complexity associated with fully unstructured matrices while retaining unstructured pruning's advantages within the blocks.

Unstructured pruning [9], [12], [18] is a fine-grained approach where individual weights are pruned based on their significance. Magnitude-based pruning, the simplest form of unstructured pruning, ranks all weights in the network by magnitude and removes a percentage of the lowest-ranked weights, regardless of their position. This method minimizes the degradation of model effectiveness compared to structured pruning. However, unstructured pruning creates irregular network structures that are challenging to process efficiently on hardware like GPUs. Huang et al. proposed an FPGA accelerator that mitigates this issue for very wide and deep sparse FFNs [19]. Their accelerator divides sparse FFN matrices into configurable tiles, assigning these tiles to PEs on the FPGA. The tile size and corresponding PEs are adjustable to fit the target device's capacity.

Unstructured pruning typically begins with a dense reference network. The final pruned subnetwork is achieved after several iterations of training, followed by the removal of a fraction of the least important parameters in each round. This iterative process is computationally intensive and time-consuming. Pre-Pruning [20]–[22], a variation of unstructured pruning, eliminates the need for a dense reference network. Instead, it starts with a network initialized with a sparse topology, often selected randomly, and trains the network to convergence. This approach reduces computational resources and time. The Lottery Ticket Hypothesis (LTH) [23] is one of the most effective pre-pruning techniques. LTH involves pruning a trained network by removing a percentage of weights based on their magnitudes. The pruned network is then reinitialized with its original weights, and the process is repeated to produce progressively sparser networks.

IV. Method

Our method focuses on reducing the total memory footprint of very wide and very deep sparse FFNs represented using the CSR data structure. The primary objective is to fit the entire network onto the FPGA fabric, thereby eliminating all off-chip memory accesses. The memory footprint of model parameters can be reduced significantly by quantizing them and has been studied extensively. Unlike network parameters, indices arrays of sparse FFNs represented using CSR data structures cannot be quantized to reduce their memory footprint.

Fetching model parameters and indices of NNZs from off-chip DRAM to the PEs synthesized on the FPGA fabric can

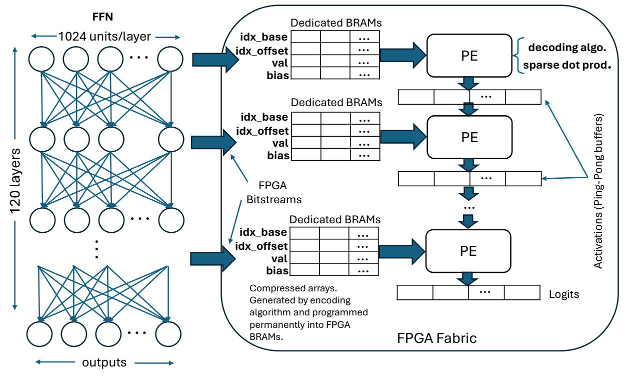


Fig. 1. Overall System Architecture

take orders of magnitude longer than fetching data from on-chip Block Random Access Memory (BRAM). Data from BRAM can typically be accessed in a single clock cycle, whereas accessing off-chip DRAM may require several hundred clock cycles. In very wide sparse FFNs, the indices arrays constitute the majority of the data required for inference. To address this bottleneck, we propose a method to encode the indices arrays in a highly memory-efficient format.

Although our method is applicable for any FFNs, we specifically focus on FFNs pruned using RadiX-Net pre-pruning [24]. In the following subsections, we detail our approach. We begin with an overview of the accelerator design, followed by a discussion on the properties of RadiX-Net pre-pruning. Next, we describe how RadiX-Net pre-pruning facilitates encoding the indices arrays into a compressed format. Finally, we explain the decoding algorithm used to reconstruct the absolute index values from their compressed representations.

A. Accelerator Architecture

Fig. 1 depicts the overall architecture of our sparse FFN accelerator. The primary objective of this accelerator is to accommodate as large a sparse FFN as possible entirely on the FPGA fabric and to efficiently execute its inference step. To achieve this, we focus on three key objectives: eliminating off-chip memory accesses, minimizing on-chip resource usage, and parallelizing layer processing. By compressing both the model parameters and the indices arrays, our method eliminates round-trips to the DRAM and minimizes on-chip resource usage. These optimizations ensure that all data required for inference is embedded within the bitstream used to configure the FPGA. Consequently, the only data exchanged with the environment comprises the input and output vectors.

Our design incorporates a high degree of parallelism by having a dedicated PE for each network layer. Each PE acts as a pipeline stage in a Dataflow Pipeline. The availability of a large number of BRAMs on the FPGA fabric enables assignment of dedicated sets of BRAMs to each PE, allowing all PEs to run in parallel. Each PE reads four arrays to perform a sparse dot product. The idx_base and idx_offset arrays are used to reconstruct the indices of non-zero weights present in the layer. The val and $bias$ arrays contain the quantized weights and biases, respectively. The dense activations between PEs are implemented as ping-pong buffers.

B. RadiX-Nets Pre-Pruning

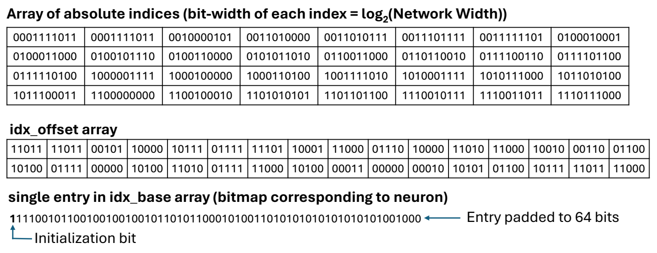
RadiX-Nets [24] are pre-pruned networks initialized as sparse networks without reference to any dense parent network. These topologies are based on mixed-radix number systems, which result in network structures where all nodes in a layer connect to a fixed number of nodes in the subsequent layer. The use of mixed-radix numeral systems ensures symmetry, a property that guarantees path connectedness and reduces training bias. These characteristics make RadiX-Nets as expressive as their dense counterparts [25], even at very high levels of sparsity.

RadiX-Net sparsity exhibits traits of both unstructured and structured (N:M) sparsity. As $N \ll M$, the distribution of NNZs becomes incompatible with hardware designed to process structured sparsity where N and M are closer in value. In terms of model effectiveness, RadiX-Nets are more robust than structured sparse networks with low N:M ratios and are comparable to globally unstructured pruned networks at identical pruning levels. The NNZ distribution in very sparse and hyper-sparse RadiX-Nets suggests that techniques designed for globally unstructured pruning could be applied. However, careful analysis of their overall sparsity patterns reveals opportunities for significantly more efficient processing compared to fully unstructured counterparts. Thus, RadiX-Net pre-pruning offers a way to retain the robust characteristics of unstructured pruning without incurring the full processing cost associated with it.

C. Encoding Algorithm

The encoding algorithm, described in Algorithm 1, is a one-time pre-processing step in the accelerator design process. It transforms the array of indices from the CSR format into a more memory-efficient representation. The method assumes that the FFN's width (M) is a power of 2 (e.g., 1024, 2048, 4096, ...) and considers the weight matrix as a single block with structured sparsity in the form $N{:}M$, where $N \ll M$. This sparsity pattern, referred to as quasi-unstructured sparsity, allows for a compact representation of indices.

The algorithm outputs two arrays, idx_base and idx_offset, which together require significantly less memory than the original array of absolute indices. Fig. 2 demonstrates the encoding process for a single neuron with 32:1024 sparsity. The inputs to the algorithm are the array of absolute indices of all NNZs in single neuron, sorted in ascending order, the number of NNZs per neuron, denoted by N, and the width of the network M. The algorithm initializes a bit vector for the neuron with a leading '1' bit, and sets $currentbase$ and the counter idx to 0. Then it scans the sorted array of absolute indices sequentially. If the difference between the current absolute index and $currentbase$ is less than $baseinc$, the algorithm appends a '0' bit to the bit vector and stores the difference $(current\ absolute\ index - currentbase)$ in the idx_offset



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

Fig. 2. Encoding all NNZs of a single neuron. The neuron in this example exhibits a sparsity of 32:1024

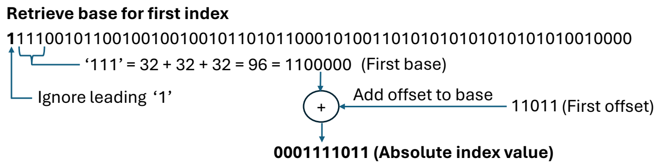


Fig. 3. Decoding a single absolute index

array. Otherwise, $currentbase$ is incremented by $baseinc$ until the difference between the current absolute index and $currentbase$ is less than $baseinc$. Each increment adds a '1' bit to the bit vector. Once all N indices have been processed, the resulting bit vector and corresponding offset values are appended to the idx_base and idx_offset arrays, respectively.

The length of the bit vector is bounded by $2N$ bits, ensuring a compact representation. In the illustrated example with $M = 1024$ and $N = 32$, this results in a maximum bit vector length of 64 bits (including the initialization bit). This compression significantly reduces the memory footprint while preserving all the information contained in the indices array.

```
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
```

Algorithm 1: Encoding Algorithm

D. Decoding Algorithm

The decoding algorithm (Algorithm 2) is implemented as custom processing logic within the PE of each layer, and is invoked at each inference pass. It reconstructs the absolute values of indices from the idx_base and idx_offset arrays created by the encoding algorithm.

For each neuron the algorithm reads in its corresponding bitmap from the idx_base array. This is followed by initializing the variable $currentbase$ to zero. It then scans the bit vector starting from its second most significant bit (MSB) until all absolute indices for the given neuron have been reconstructed.

When a '0' bit is encountered, the next offset value is read from the idx_offset array and added to $currentbase$ to reconstruct the absolute index. This index is used to fetch the corresponding activation value from the prior layer's activation vector, which is multiplied by the corresponding weight value in the $values$ array, and the result is accumulated. After each multiplication, the pointer to the $values$ array is incremented by one.

When a '1' bit is encountered, the value of $currentbase$ is incremented by $baseinc$, which updates the base value used for subsequent index reconstructions.

The process continues until all NNZ indices for the neuron have been recovered. Although a variable number of bits may need to be scanned to construct each base value, the total number of bits required for all base values of a neuron is bounded by $2N$. This bounded requirement allows for the use of fixed-length data types. In this implementation, the bit vector for each neuron is stored using a 64-bit (long int) data type.

The decoding algorithm operates concurrently for all layers, ensuring efficient and high-throughput execution. Fig. 3 depicts an example of decoding an index value.

```
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
```

Algorithm 2: Decoding Algorithm

V. Experiment Design

A. Dataset and Metrics

We used the MNIST dataset to evaluate our method, with 60,000 training images and 10,000 testing images, each of size 32×32. Images are categorized into 10 classes depending on the digit they represent. From the training set, we segregated 15,000 images as the validation set. We use accuracy as the primary effectiveness metric for each model.

B. Networks

We created pre-pruned FFNs with relatively fewer $(2, 3, 4, 6)$ and larger $(10, 20, 30 \ldots, 120)$ number of layers using Radix-Net pre-pruning [24]. Each layer except the output layer had exactly 1024 units in all networks. The output layer had 10 units, as specified for the multiclass classification task for which the model is trained, and was left unpruned. The bit-widths of all network weights were restricted to signed 4-bit values. The activations were restricted to 4-bit unsigned values, and the biases were restricted to 8-bit signed values. Quantization was performed on a per-layer basis.

C. Model training

All networks were trained with Pytorch 2.4.1 on a Supermicro SYS-420GP-TNAR+ system with NVIDIA HGX A100 8-way GPUs[1], using only one GPU for all training. We used the Brevitas 0.11.0 library for Quantization Aware Training (QAT) and the Adam optimizer with a learning rate of $10e-3$, batch size of 256, and cross entropy loss. Most networks converged to optimal levels after 2-3 training iterations, with each iteration lasting 100 epochs.

D. FPGA Implementation

The FPGA accelerator was designed entirely using High Level Synthesis (HLS), in the AMD Vitis 2024.2 development environment. A python script was used to extract the val, idx_base, idx_offset and $bias$ arrays, for each layer from the trained models (.pth files), and instantiated as CONST arrays in the HLS design, ensuring all parameters become part of the bitstream and are synthesized as BRAMs. The Quantize-Dequantize-Quantize (QDQ) approach was used to implement the inference step. The required scale factors were extracted from the Open Neural Network Exchange (ONNX) representation of the trained models. The scale factors were stored using fixed-point data types, enabling efficient computations.

E. Measurement of efficiency

The clock period (CP) achieved after the implementation phase (place and route) was used to calculate the latency and performance of the design. The latency was calculated as $CP \times cycles/layer \times \#layers$. The throughput was calculated as $CP \times cycles/layer$.

VI. Results and Discussion

We compared our method against that of Huang et al. [19] on the Xilinx VC 709 board, which is also used by the baseline against which we compare our method. The selection of this board assures a fair comparison of our method with the baseline. Our results are composed of two parts. The first part of the results compares the efficiency of our method with that of the baseline, both in terms of inference time and resource utilization. The second part of the results demonstrate that model effectiveness is retained after the optimizations performed by our method.

A. Efficiency Results

The efficiency results are divided into two parts. First, we compared the inference efficiency achieved by our method with that of the baseline, in terms of inference time. Then, we compared the two methods in terms of FPGA resources utilized by each.

To maintain a fair comparison, we compare the inference efficiency of our method with the baseline under the following constraints:

- The baseline method uses several configurations. We compare our results with the best performing configuration reported by Huang et al.
- The baseline reports the inference performance for a single image, which translates to the latency of the inference accelerator. Our results also report latency, averaged across inference of all images in the test set using a 120-layer FFN.
- We also report on the throughput of our accelerator for the interested reader, along with the inference performance of the baseline MATLAB code.

The 'Performance' section of table II summarizes the efficiency of our method w.r.t latency (lower is better) and throughput. The latency achieved by our method is orders of magnitude lower (better) compared to the SOTA baseline. Our method's lower latency is a direct consequence of its ability to fit the entire network in the FPGA's fabric and avoid costly value look-ups in the DRAM. The second part of the efficiency results summarized under the 'Resource Usage' section of Table II compares the efficiency of the data paths in our method with that of the baseline. The four main hardware resource types used to construct a data path in an FPGA are considered, namely Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAMs (BRAMs), and Digital Signal Processing Units (DSPs). The superior timing performance of our method comes at the expense of higher BRAM usage. However, it is important to note that the data path of the baseline is invoked several times during each inference pass, and the number of such invocations is a function of the configuration used in the baseline. Each inference pass in our method invokes the data path only once.

B. Effectiveness results

Our effectiveness results show our method does not result in material degradation of model effectiveness. Our effectiveness studies comprised comparing effectiveness of models of varying depths, compressed using our method with their fully connected counterparts. The results showed only a slight degradation ($< 1\%$) in accuracy. For certain model depths our method improved effectiveness.

VII. Conclusion

To the best of our knowledge, our work is the very first attempt to deploy a very large pre-pruned FFN, which is pruned using Radix-Net pre-pruning, completely on the FPGA fabric. We achieved this objective by significantly compressing the data structures associated with sparse networks, i.e., the indices and values arrays of the CSR matrix. For models with quantized parameters, the indices arrays of the CSR format can account for a majority of the total memory requirements of a pruned model. Radix-Net pre-pruning is almost as robust as unstructured pruning when it comes to retaining model effectiveness. However, compared to the network structures resulting from unstructured pruning, the networks resulting from RadiX-Net pre-pruning possess some structure. We used this key observation to develop algorithms to compress and

[1]This research was made possible through resources provided by Supermicro and NVIDIA. Code for our project is available at https://github.com/davidanastasiu/fast-ffn.

TABLE II
EFFICIENCY RESULTS (120 LAYER NETWORK)

Method	Performance (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

decompress the sparse data structures associated with RadiX-Nets in a deterministic way. By combining the compression of sparse data structures and quantizing model parameters, we were able to accommodate a very wide and very deep sparse FFN completely on the FPGA fabric. This led to gains in performance which were orders of magnitude greater when compared to the SOTA. We further demonstrated the robustness of our method by performing a model effectiveness study, which demonstrated the networks compressed using our method are just as effective as their fully connected uncompressed counterparts.

REFERENCES

[1] C. Pravin and V. Ojha, "A novel ecg signal denoising filter selection algorithm based on conventional neural networks," in *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2020, pp. 1094–1100.

[2] A. Berezkin, A. Slepnev, R. Kirichek, D. Kukunin, and D. Matveev, "Data compression methods based on neural networks," in *Proceedings of the 5th International Conference on Future Networks and Distributed Systems*, ser. ICFNDS '21. New York, NY, USA: Association for Computing Machinery, 2022, p. 511–515. [Online]. Available: https://doi.org/10.1145/3508072.3508177

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[5] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[6] L. Basyal and M. Sanghvi, "Text summarization using large language models: A comparative study of mpt-7b-instruct, falcon-7b-instruct, and openai chat-gpt models," 2023. [Online]. Available: https://arxiv.org/abs/2310.10449

[7] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[8] C. V. Nguyen, X. Shen, R. Aponte, Y. Xia, S. Basu, Z. Hu, J. Chen, M. Parmar, S. Kunapuli, J. Barrow, J. Wu, A. Singh, Y. Wang, J. Gu, F. Dernoncourt, N. K. Ahmed, N. Lipka, R. Zhang, X. Chen, T. Yu, S. Kim, H. Deilamsalehy, N. Park, M. Rimer, Z. Zhang, H. Yang, R. A. Rossi, and T. H. Nguyen, "A survey of small language models," 2024. [Online]. Available: https://arxiv.org/abs/2410.20011

[9] E. Frantar and D. Alistarh, "Sparsegpt: Massive language models can be accurately pruned in one-shot," 2023. [Online]. Available: https://arxiv.org/abs/2301.00774

[10] X. Sun, N. Wang, C.-y. Chen, J.-m. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. Srinivasan, and K. Gopalakrishnan, "Ultra-low precision 4-bit training of deep neural networks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[11] B. Hassibi, D. Stork, and G. Wolff, "Optimal brain surgeon and general network pruning," in *IEEE International Conference on Neural Networks*, 1993, pp. 293–299 vol.1.

[12] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016. [Online]. Available: https://arxiv.org/abs/1510.00149

[13] S. Ashkboos, M. L. Croci, M. G. do Nascimento, T. Hoefler, and J. Hensman, "Slicegpt: Compress large language models by deleting rows and columns," 2024. [Online]. Available: https://arxiv.org/abs/2401.15024

[14] X. Ma, G. Fang, and X. Wang, "Llm-pruner: On the structural pruning of large language models," 2023. [Online]. Available: https://arxiv.org/abs/2305.11627

[15] S. Wang, P. Lin, R. Hu, H. Wang, J. He, Q. Huang, and S. Chang, "Acceleration of lstm with structured pruning method on fpga," *IEEE Access*, vol. 7, pp. 62 930–62 937, 2019.

[16] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, "Ese: Efficient speech recognition engine with sparse lstm on fpga," 2017. [Online]. Available: https://arxiv.org/abs/1612.00694

[17] R. Shi, P. Dong, T. Geng, Y. Ding, X. Ma, H. K.-H. So, M. Herbordt, A. Li, and Y. Wang, "Csb-rnn: a faster-than-realtime rnn acceleration framework with compressed structured blocks," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. ACM, Jun. 2020, p. 1–12. [Online]. Available: http://dx.doi.org/10.1145/3392717.3392749

[18] M. Sun, Z. Liu, A. Bair, and J. Z. Kolter, "A simple and effective pruning approach for large language models," 2024. [Online]. Available: https://arxiv.org/abs/2306.11695

[19] S. Huang, C. Pearson, R. Nagi, J. Xiong, D. Chen, and W.-m. Hwu, "Accelerating sparse deep neural networks on fpgas," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.

[20] C. Wang, G. Zhang, and R. Grosse, "Picking winning tickets before training by preserving gradient flow," 2020. [Online]. Available: https://arxiv.org/abs/2002.07376

[21] H. Tanaka, D. Kunin, D. L. K. Yamins, and S. Ganguli, "Pruning neural networks without any data by iteratively conserving synaptic flow," 2020. [Online]. Available: https://arxiv.org/abs/2006.05467

[22] N. Lee, T. Ajanthan, and P. H. S. Torr, "Snip: Single-shot network pruning based on connection sensitivity," 2019. [Online]. Available: https://arxiv.org/abs/1810.02340

[23] B. Liu, Z. Zhang, P. He, Z. Wang, Y. Xiao, R. Ye, Y. Zhou, W.-S. Ku, and B. Hui, "A survey of lottery ticket hypothesis," 2024. [Online]. Available: https://arxiv.org/abs/2403.04861

[24] J. Kepner and R. Robinett, "Radix-net: Structured sparse matrices for deep neural networks," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2019, p. 268–274. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2019.00051

[25] K. Kwak, Z. West, H. Jananthan, and J. Kepner, "Testing radix-nets: Advances in viable sparse topologies," 2023. [Online]. Available: https://arxiv.org/abs/2311.03609