# Energy-Efficient Inference Accelerator for Memory-Augmented Neural Networks on an FPGA

Seongsik Park, Jaehee Jang, Seijoon Kim, Sungroh Yoon
Electrical and Computer Engineering, Seoul National University, Seoul, Korea
sryoon@snu.ac.kr

***Abstract*—Memory-augmented neural networks (MANNs) are designed for question-answering tasks. It is difficult to run a MANN effectively on accelerators designed for other neural networks (NNs), in particular on mobile devices, because MANNs require recurrent data paths and various types of operations related to external memory access. We implement an accelerator for MANNs on a field-programmable gate array (FPGA) based on a data flow architecture. Inference times are also reduced by inference thresholding, which is a data-based maximum inner-product search specialized for natural language tasks. Measurements on the bAbI data show that the energy efficiency of the accelerator (FLOPS/kJ) was higher than that of an NVIDIA TITAN V GPU by a factor of about 125, increasing to 140 with inference thresholding.**

***Index Terms*—deep learning, memory-augmented neural networks, inference accelerator, FPGA, data-based maximum-inner product search, question and answer**

## I. Introduction

Deep neural networks (DNNs) require more computing power and storage than most mobile devices can provide. So mobile DNNs are commonly trained and run on remote servers. This limits performance, relies on network availability, and increases maintenance. It motivates the development of on-device inference.

In a dataflow architecture (DFA), data goes directly from one processing element to another, reducing the need for energy-consuming memory accesses [1]. Layer-wise parallelization and recurrent paths can be implemented on DFAs, through the use of fine-grained parallelism. DFAs have therefore been used to realize inference on mobile devices [2]–[4].

Memory-augmented neural networks (MANNs), which include memory networks [5], are recurrent neural networks (RNNs) with external memory to increase learning capacity. MANNs require both recursive and memory operations in each layer, making them difficult to parallelize on CPUs or GPUs.

We propose an accelerator for MANNs based on a field-programmable gate array (FPGA), which uses a DFA to realize energy-efficient inference in the domain of natural language processing (NLP), which is a major application of MANNs. We also introduce a data-based method of maximum inner-product search (MIPS), called inference thresholding, together with an efficient index ordering. This speeds up inference and the operation time of the output layer, which is particularly important in tasks with large classes, such as NLP.

Our implementation outperformed a GPU in terms of energy efficiency (FLOPS/kJ) by a factor of 126 on the bAbI dataset [6], and by 140 when inference thresholding was also used. The contributions of this paper are as follows:

- A streaming-based inference architecture for MANNs, which we believe is the first.
- Fast inference on this hardware using inference thresholding.
- Implementation and validation of this approach on an FPGA.

## II. Memory-Augmented Neural Networks

MANNs, which are RNNs with more storage, are designed for question answering (QA) and other NLP tasks [5]. A MANN consists of external memory and a controller, and it learns how to read and write information from and to the memory. The memory operations of a MANN can be divided into three types: addressing, write, and read. Content-based addressing is usually employed in MANNs, and can be expressed as follows:

$$a_i^t = \frac{\exp\{M_{\mathrm{a},i} \cdot k^t\}}{\sum_j^L \exp\{M_{\mathrm{a},j} \cdot k^t\}}, \tag{1}$$

where $a_i^t$ is the read weight of the $i$th memory element at time $t$, $M_a$ is the address memory, $L$ is the number of memory elements, and $k^t$ is a read key.

Each memory element stores an embedded sentence vector as follows:

$$M_i = W_{\mathrm{emb}} S_i = \sum\nolimits_{\mathrm{idx} \in S_i} W_{\mathrm{emb}_{:,\mathrm{idx}}}, \tag{2}$$

where $W_{\mathrm{emb}}$ is a word-embedding weight, and $S_i$ is an input sentence consisting of word indices. A memory read begins with the generation of a read key in the memory controller after the previous write. The read key $k^t$ at time $t$ is found as follows:

$$k^t = \begin{cases} W_{\mathrm{emb\_q}} q & \text{if } t = 1 \\ h^{t\text{-}1} & \text{otherwise,} \end{cases} \tag{3}$$

where $q$ is a question vector, and $h$ is an output vector from the controller, which is described as follows:

$$h^t = r^t + W_{\mathrm{r}} k^t, \tag{4}$$

where $r$ is a read vector, and $W_{\mathrm{r}}$ is the weight of the controller. The read vector for content-based addressing is generated by a content memory as follows:

$$r^t = M_{\mathrm{c}} a^t. \tag{5}$$

The predicted label $\tilde{y}$, produced by inference is given by

$$\tilde{y} = \arg\max_i(z_i) = \arg\max_i(W_{\mathrm{o}_{i,:}} h^t), \tag{6}$$

where $W_{\mathrm{o}}$ is the weight of the output layer, and $z_i$ is a logit with index $i$.

## III. Hardware Architecture

Fig. 1 shows the architecture and data flow of our accelerator, which consists of several modules which receive inference data

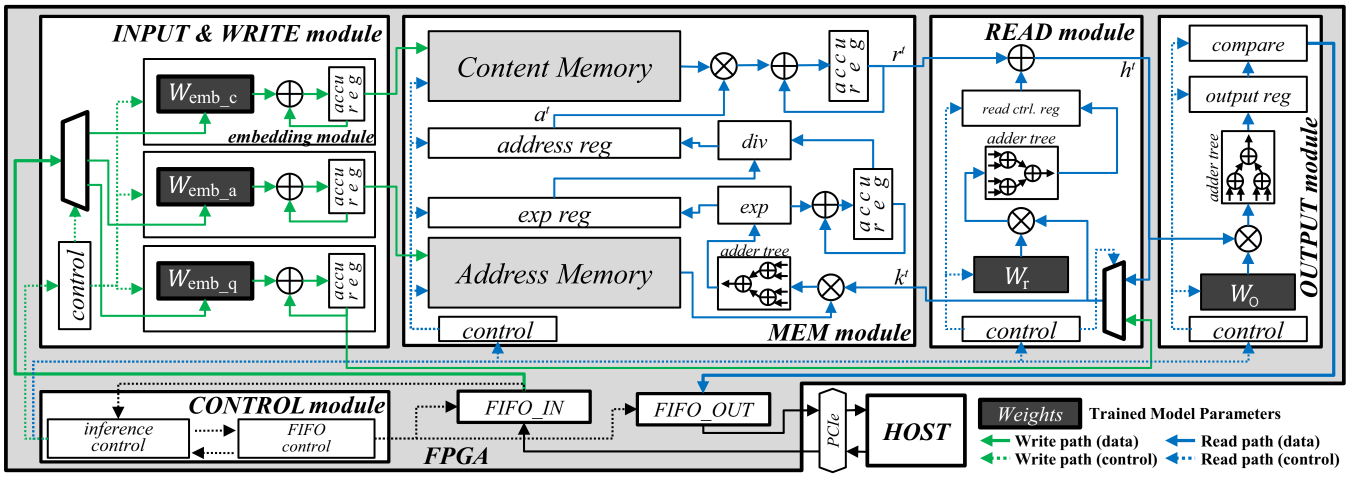

Fig. 1: Proposed architecture of the FPGA-based accelerator for MANNs

and trained models ($W_{\mathrm{emb}}$, $W_{\mathrm{r}}$, and $W_{\mathrm{o}}$) from a host computer in the form of streams through a FIFO queue. A pre-trained model with appropriate data is passed to each module.

Control signals from the host, embedded in the data, pass to the CONTROL module, which has an inference control component that signals other modules. For example, in a QA task, context data in the form of sentences $S$, together with the question $q$, arrive in the input stream (green line in Fig. 1). When this stream is finished, the READ module generates a read key $k^t$, and the MEM module uses this key to read a vector $r^t$ from the context memory. Reads can be recursive because the READ module is composed of an RNN. After all read operations are complete, the OUTPUT module returns the answer to the question through the FIFO queue to the host.

The INPUT & WRITE modules receive input data from the host and write embedded vectors to context and address memory in the MEM module. In an NLP task, a discrete and sparse sentence vector (e.g. a bag-of-words) is converted into a dense embedded vector by the embedding layer. If the input to a MANN includes word indices, then the efficiency of embedding in the INPUT & WRITE module can be improved, as shown in Eq. 2. The embedding module in the INPUT & WRITE module only needs to read the columns of the embedding weight $W_{\mathrm{emb}}$ corresponding to the indices of the input words. This reduces the number of memory accesses needed to read the embedding weights, and the number of multiplications needed to calculate the embedding vector, which lead to improving energy efficiency.

The MEM module consists of the address memory, which is content-addressible (Eq. 1) and context memory, which generates a read vector $r^t$ by soft-addressing based on the attention at obtained from the address memory (Eq. 5). The address and context memory together store the embedded vector from the INPUT & WRITE module. This requires costly operations such as softmax, which incurs an exponentiation and a division, which cannot be parallelized on an FPGA. The MEM module is therefore implemented with element-wise sequential operations which can exploit fine-grained parallelism.

The READ module is an RNN, and the OUTPUT module is a fully connected neural network. The READ module generates the read key $k^t$ which is used to calculate the attention at in the MEM module, and receives a read vector $r^t$ from the MEM module (Eqs. 3 and 5). The blue line in the READ module in Fig. 1 shows how a recurrent READ path can be implemented efficiently.

The OUTPUT module predicts the label $\tilde{y}$ based on the read vector, by multiplying the vector $h^t$ and the weight matrix of the output layer $W_{\mathrm{o}}$, as shown in Eq. 6. Matrix multiplication is implemented as a series of dot products because the hardware is insufficient to parallelize it directly. In the OUTPUT module the logit $z_i$ of each index is sequentially calculated to find the maximum logit; this takes up a lot of the inference time.

## IV. Fast Inference Method

### A. Inference Thresholding

A MANN implemented as a DFA can exploit fine-grained parallelism in each layer. However, in an NLP task the dimension of the output $|I|$ is much larger than that of the embedding $|E|$, making it difficult to parallelize operations in the output layer [7]. Thus, when calculating a logit $z$ in the output layer, we must sequentially calculate the dot product of the input vector $h$ and the row of the weight matrix corresponding to the index $W_{\mathrm{o}i,:}$ in the output module, as shown in Fig. 2-(a). Because the operation time of the output layer is $O(|I|)$, the inference time increases with $|I|$.

We implement the output layer sequentially, but limit the computation required by introducing inference thresholding (Algo. 1). We approximate the MIPS by speculating that, given $z_i$, the index $i$ will be the predicted label $\tilde{y}$. If we can conjecture the maximum logit for index $i$ with sufficient confidence, then we need not compare the remaining logits.

Inference thresholding was motivated by observing logit distributions in a trained model in which the logits $z$ are fitted to the mixture models, as shown in Fig. 2-(b). To predict whether logit $z_i$ is the maximum value of all logits $z$, we consider two distributions: in one, $z_i$ is the maximum, and in the other it is not.

On this basis we can estimate conditional probability density functions (PDFs) $p(z_i|y = i)$ for the training label $y$ by kernel density estimation (Step 1 in Algo. 1). The PDFs of the inference dataset can be approximated by those obtained from the training dataset. By applying Bayes' theorem to the

**Algorithm 1:** Inference Thresholding

```
Input   : training dataset $\mathcal{D} = \{x_n, y_n\}_{n=1}^{N}$,
          inference data $\tilde{x}$
Output  : prediction label $\tilde{y}$
Notations: $z$: vector of logits, $z_i$: logit value at $i$th index,
  $M$: pre-trained model, $I$: dimension of output vector,
  $\rho$: thresholding constant,
  $HG_i$: histogram of $z_i$ when $i = \arg\max_i z_i$,
  $HG_{\bar{i}}$: histogram of $z_i$ when $i \neq \arg\max_i z_i$
Step 1: Estimate logit distributions
for $(x_n, y_n)$ in $\mathcal{D}$:
    $z \leftarrow$ Do forward pass $M(x_n)$, $y \leftarrow \arg\max_i z_i$
    if $y == y_n$:
        for i in $1 : \mathrm{I}$:
            if $i == y$:
                Update $HG_i \leftarrow z_i$
            else:
                Update $HG_{\bar{i}} \leftarrow z_i$
for $i$ in $1 : \mathrm{I}$:
    Estimate $p(z_i|y = i)$ from $HG_i$
Step 2: Set the inference thresholds
$p(y = i|z_i) \leftarrow p(z_i|y = i)p(y = i)$
for $i$ in $1 : \mathrm{I}$:
    $\theta_i \leftarrow \min(\{z_i | p(y = i|z_i) \geq \rho\})$
Step 3: Set the efficient index order
for $i$ in $1 : \mathrm{I}$:
    $S_i \leftarrow$ avg. silhouette coefficient of $HG_i$
$A \leftarrow$ indices sorted by $S_\mathrm{i}$ in descending order
Step 4: Inference thresholding
$h \leftarrow$ Do forward pass $M(\tilde{x})$ until output layer
for i in $1 : \mathrm{I}$:
    $a \leftarrow A_i$
    $z_a \leftarrow W_{o_{a,:}} h$
    if $z_a > \theta_a$:
        return $\tilde{y} \leftarrow a$
return $\tilde{y} \leftarrow \arg\max_i z_i$
```

approximated PDFs, we can obtain the posteriors of the logits for the inference dataset $p(\tilde{y} = i|z_i)$ as follows:

$$p(\tilde{y} = i|z_i) \approx p(y = i|z_i) \propto p(z_i|y = i)p(y = i), \qquad (7)$$

where $P(y = i)$ is the probability that the index $i$ is a training label $y$.

To apply estimated probabilities to the inference process in the output layer, we compare each logit $z_i$ with a threshold $\theta_i$, which is the the smallest value of those logits of which the estimated posterior probability $p(y = i|z_i)$ is larger than $\rho$:

$$\theta_i := \min(\{z_i | p(y = i|z_i) \geq \rho\}), \qquad (8)$$

where $\rho$ is a thresholding constant (Step 2 in Algo. 1). This yields a speculative value for the label.

### B. *Efficient Index Order for Inference Thresholding*

Inference thresholding is quicker if we order the logits so that those for which thresholding is most effective come first (Fig. 2). This can be seen as determining whether the logit belongs to the class $y = i$. From this perspective, inference thresholding will be more effective for a logit with a long inter-class distance and a short intra-class distance. We therefore sort the indices into descending order of silhouette coefficient [8] (Step 3 in Algo. 1).

The effect of inference thresholding and index ordering is depicted in Fig. 3. As the threshold constant $\rho$ decreases, MIPS

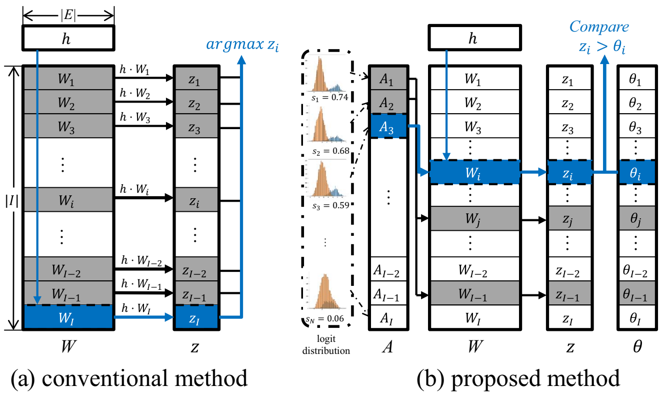

Fig. 2: MIPS in the OUTPUT module: (a) the conventional method needs to compare all logits; (b) inference thresholding stops the comparison if $z_i > \theta_i$.

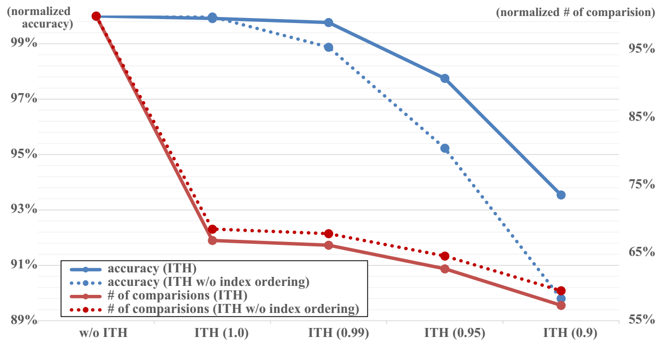

Fig. 3: Evaluation of the effect of inference thresholding and index ordering: in terms of the accuracy and number of comparisons required in the MIPS against threshold constant $\rho$, on the bAbI dataset (ITH = inference thresholding).

requires fewer comparisons but accuracy declines. Ordering improves both accuracy and speed.

## V. Experimental Results

We implemented the accelerator and measured the inference time and power consumption on an Intel Core i9-7900X CPU, and on an NVIDIA TITAN V GPU, and a Xilinx Virtex UltraScale VCU107 FPGA linked to the same CPU.

Time and power measurement were made for 20 tasks from the bAbI QA dataset [6]. Timings, which included transmission of the pre-trained model and inference data to the GPU and FPGA, were repeated 100 times; power measurements were made over five minutes. We ran the FPGA at 25, 50, 75, and 100 MHz to evaluate the effect of the host-FPGA interface. We set the thresholding constant $\rho$ to 1.0, which reduced accuracy by less than 0.1%.

Averaged timings and power measurements are listed in Table I. Running on the FPGA, the accelerator took less time at higher frequencies, as we would expect: but the improvement was not linear. Inference thresholding reduced timings by 6-18%, depending on frequency. The accelerator ran between 5.2 and 7.5 times faster than the GPU, and between 5.6 and 8.0 times faster than the CPU. The GPU used most power, and the FPGA running at 25MHz used least. The CPU used 1.7 times less energy than the GPU, and the FPGA used 74 times less, or 140 times less using inference thresholding.

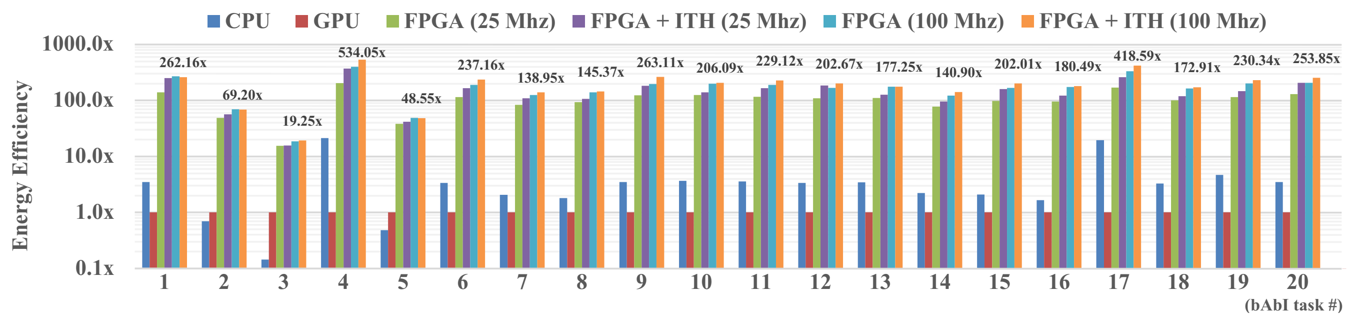

Fig. 4: Energy efficiency of inference on the bAbI dataset on various configurations compared with the GPU (ITH = inference thresholding).

TABLE I: Average measurement results, speedup, and energy-efficiency of inference on the bAbI dataset

| Configurations | Time (s) | Power (W) | Speedup[a] | FLOPS/kJ[a] |
|---|---|---|---|---|
| CPU | 242.77 | 23.28 | 0.94 | 1.70 |
| GPU | 226.90 | 45.36 | 1.00 | 1.00 |
| FPGA | | | | |
| 25 Mhz | 43.54 | **14.71** | 5.21 | 83.74 |
| 50 Mhz | 34.95 | 17.53 | 6.49 | 109.06 |
| 75 Mhz | 31.96 | 19.02 | 7.10 | 120.24 |
| 100 Mhz | 30.28 | 20.10 | 7.49 | 126.72 |
| FPGA + Inference thresholding | | | | |
| 25 Mhz | 35.36 | 17.36 | 6.42 | 107.61 |
| 50 Mhz | 30.81 | 20.11 | 7.36 | 122.35 |
| 75 Mhz | 29.18 | 20.18 | 7.78 | 135.87 |
| 100 Mhz | **28.53** | 20.53 | **7.95** | **139.75** |

[a] normalized to the result on the GPU

Results on individual tasks are shown in Fig. 4, again normalized to the performance of the GPU. The FPGA implementation was the most energy-efficient across all tasks, and inference thresholding increased the margin.

Inference thresholding is more beneficial at low operating frequencies. As the frequency increases, inference time is dominated by the interface between the host and the FPGA. If this were not the case, we estimate that our approach would use 162 times less energy than the GPU.

Inference thresholding did not have a significant effect on the inference time of our accelerator running on the CPU or GPU. On the CPU, the output layer only represents a small part of the computation; and the GPU can process the output layer in parallel.

## VI. Related Work

### A. DNN Inference Accelerator

Hardware matrix multiplications can reduce inference times for CNN models [2], [9]. Several architectures [2]–[4] have been introduced for different types of RNN, such as LSTMs and GRUs. These accelerators save energy, but are not readily extensible to the memory operations required in MANNs. A method of accelerating inference of MANNs has been studied [10], but it has not been implemented in hardware.

### B. Maximum Inner-Product Search

In applications with large search spaces, including NLP, MIPS takes a long time. Hence, approximations using hashing [11], or clustering [12] have been proposed. Some of these approaches, including sparse access memory [13] and hierarchical memory networks [14], have also been used to accelerate memory reads and writes in MANNs. However these techniques may be too slow to be used in the output layer of a DNN in resource-limited environments.

## VII. Conclusion

We believe that the DFA-based approach, and its implementation on an FPGA, which are reported in this paper, represent the first attempt at energy-efficient inference specifically for MANNs. We also introduce a method of speculation about the inference results which avoids computations which are difficult to parallelize. This reduces computation times and saves energy at an extremely small cost in accuracy. We believe that this work shows how inference tasks such as QA may be preformed in mobile devices. We also expect that our data-based MIPS will find applications in large-class inference.

## Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) [2016M3A7B4911115, 2018R1A2B3001628], the Strategic Initiative for Microbiomes in Agriculture and Food (Ministry of Agriculture, Food and Rural Affairs) [918013-4], and the Brain Korea 21 Plus Project in 2018.

## References

[1] M. Horowitz, “1.1 Computing’s energy problem (and what we can do about it),” in *ISSCC*, 2014.
[2] S. Han *et al.*, “EIE: Efficient Inference Engine on Compressed Deep Neural Network,” in *ISCA*, 2016.
[3] V. Rybalkin *et al.*, “Hardware architecture of Bidirectional Long Short-Term Memory Neural Network for Optical Character Recognition,” in *DATE*, 2017.
[4] S. Han *et al.*, “ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA,” in *FPGA*. ACM, 2017.
[5] S. Sukhbaatar *et al.*, “End-To-End Memory Networks,” in *NIPS*, 2015.
[6] J. Weston *et al.*, “Towards ai-complete question answering: A set of prerequisite toy tasks,” *arXiv preprint arXiv:1502.05698*, 2015.
[7] S. Li *et al.*, “FPGA Acceleration of Recurrent Neural Network Based Language Model,” in *FCCM*, 2015.
[8] P. Rousseeuw, “Silhouettes: A graphical aid to the interpretation and validation of cluster analysis,” *Journal of computational and applied mathematics*, 1987.
[9] Y. Chen *et al.*, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks,” *IEEE Journal of Solid-State Circuits*, 2017.
[10] S. Park *et al.*, “Quantized Memory-Augmented Neural Networks,” in *AAAI*, 2018.
[11] A. Shrivastava and P. Li, “Asymmetric LSH (ALSH) for Sublinear Time Maximum Inner Product Search (MIPS),” in *NIPS*, 2014.
[12] A. Auvolat *et al.*, “Clustering is Efficient for Approximate Maximum Inner Product Search,” *arXiv preprint arXiv:1507.05910*, 2015.
[13] J. Rae *et al.*, “Scaling Memory-Augmented Neural Networks with Sparse Reads and Writes,” in *NIPS*, 2016.
[14] S. Chandar *et al.*, “Hierarchical Memory Networks,” *arXiv preprint arXiv:1605.07427*, 2016.