MnnFast: A Fast and Scalable System Architecture for Memory-Augmented Neural Networks

Hanhwi Jang*
Department of Computer Science and Engineering
POSTECH
Pohang, Republic of Korea
hanhwi@postech.ac.kr

Joonsung Kim*
Department of Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
joonsung90@snu.ac.kr

Jae-Eon Jo
Department of Computer Science and Engineering
POSTECH
Pohang, Republic of Korea
jojaeeon@postech.ac.kr

Jaewon Lee
Department of Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
lee.jaewon@snu.ac.kr

Jangwoo Kim†
Department of Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
jangwoo@snu.ac.kr

ABSTRACT

Memory-augmented neural networks are getting more attention from many researchers as they can make an inference with the previous history stored in memory. Especially, among these memory-augmented neural networks, *memory networks* are known for their huge reasoning power and capability to learn from a large number of inputs rather than other networks. As the size of input datasets rapidly grows, the necessity of large-scale memory networks continuously arises. Such large-scale memory networks provide excellent reasoning power; however, the current computer infrastructure cannot achieve scalable performance due to its limited system architecture.

In this paper, we propose MnnFast, a novel system architecture for large-scale memory networks to achieve fast and scalable reasoning performance. We identify the performance problems of the current architecture by conducting extensive performance bottleneck analysis. Our in-depth analysis indicates that the current architecture suffers from three major performance problems: *high memory bandwidth consumption*, *heavy computation*, and *cache contention*. To overcome these performance problems, we propose three novel optimizations. First, to reduce the memory bandwidth consumption, we propose a new *column-based algorithm with streaming* which minimizes the size of data spills and hides most of the off-chip memory accessing overhead. Second, to decrease the high computational overhead, we propose a *zero-skipping* optimization to bypass a large amount of output computation. Lastly, to eliminate the cache contention, we propose an *embedding cache* dedicated to efficiently cache the embedding matrix.

Our evaluations show that MnnFast is significantly effective in various types of hardware: CPU, GPU, and FPGA. MnnFast improves the overall throughput by up to 5.38×, 4.34×, and 2.01× on CPU, GPU, and FPGA respectively. Also, compared to CPU-based MnnFast, our FPGA-based MnnFast achieves 6.54× higher energy efficiency.

*Equal contribution, listed in alphabetical order.
†Corresponding author.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322214

CCS CONCEPTS

• **Computing methodologies → Neural networks**; *Parallel algorithms*; • **Computer systems organization → Neural networks**; Parallel architectures; • **Hardware → Hardware accelerators**; • **Theory of computation** → *Parallel algorithms*.

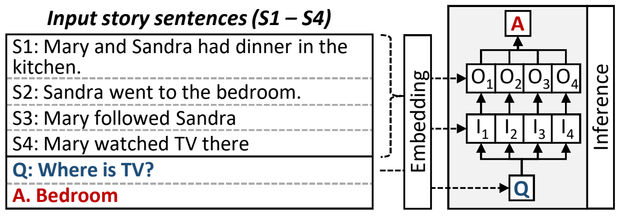
KEYWORDS

Memory Networks, Attention-based Neural Networks, Machine Learning, Parallel Algorithm, Computation/Dataflow Optimization, Accelerator, Algorithm-Hardware Co-Design, Architecture

ACM Reference Format:
Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. 2019. MnnFast: A Fast and Scalable System Architecture for Memory-Augmented Neural Networks. In *The 46th Annual International Symposium on Computer Architecture(ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322214

1 INTRODUCTION

Recently, neural networks have risen as new information processing paradigms in various fields. Among them, newly emerging neural networks called *memory-augmented neural networks* (MemNNs) [69, 78] are getting increasing attention from the researchers, thanks to their powerful context-aware information processing capability. In contrast to feedforward neural networks (e.g., CNNs, DNNs), MemNNs, which exploit their dedicated memory components to process sequences of inputs, are known for their powerful reasoning capability. Different from other neural networks, MemNN can discretely read and write all contents by


Input story sentences (S1 – S4)
S1: Mary and Sandra had dinner in the kitchen.
S2: Sandra went to the bedroom.
S3: Mary followed Sandra
S4: Mary watched TV there
Q: Where is TV?
A. Bedroom
Embedding
Inference
A
O1 O2 O3 O4
I1 I2 I3 I4
Q

(a) Example scenario (b) Memory network

Figure 1: (a) shows an example story with a question. (b) shows where the memory network stores the story, and how it processes the question to derive the answer.

exploiting a flexible mechanism similar to the human's working memory [44].

Figure 1 shows how MemNN processes a story with a question and an answer. In this example, MemNN first receives a four-sentence story and stores it in its memory. Next, it receives a question asking the location of TV, which can be answered only by understanding the story (e.g., the order of sentences, the relation of words across the sentences). To enable such context-aware information processing, MemNN performs an inference by utilizing the information stored in the memory against the question. In this process, the question and input sentences are converted into internal state values (i.e., Q, I_i, O_i), and these values are eventually stored into memory components.

To improve the reasoning power, MemNN needs to increase the size of memory and train the network with a large-scale dataset. In fact, recent studies propose *large-scale memory networks* to support growing demands for large-scale question answering tasks [7].

As the size of memory networks continuously increases, the large-scale Q/A task requires a *fast* and *scalable* computer infrastructure; however, the current system architecture does not provide enough scalability due to the following reasons. First, a large-scale MemNN can suffer from the increasing number of cache misses as the data do not fit into the cache. Second, when MemNN goes through a memory-intensive phase (i.e., the embedding operation), the increased number of DRAM accesses can degrade the overall performance significantly. Third, when MemNN goes through a compute-intensive phase (i.e., the inference operation), the system suffers from the lack of available computing units. Lastly, a large-scale MemNN can suffer from a significant number of cache conflicts when different operations contend for the shared cache (e.g., embedding vs. inference).

In this paper, we propose MnnFast, a novel large-scale MemNN system architecture to achieve fast and scalable reasoning performance. To improve the performance, MnnFast adopts three novel optimizations: *column-based algorithm with streaming*, *zero-skipping*, and *embedding cache*.

First, to reduce memory bandwidth consumption, MnnFast applies a new *column-based algorithm* to minimize the size of data spills and enable more efficient data chunking by transforming a large-scale memory access into many parallelized small-scale memory accesses. With the column-based algorithm, MnnFast can further improve its performance by performing data computation and prefetching the data required for the next calculation in parallel (called *streaming optimization*). In this way, the *column-based algorithm with streaming* effectively hides most of the memory accessing overhead, while still maintaining the operation's integrity.

Second, to reduce the computational overhead, MnnFast applies an optimization (called *zero-skipping*) to bypass computation dealing with zero or near-zero values stored in the memory. Our in-depth analysis indicates that zero-skipping is highly effective during the output memory processing because typically only a small number of words and sentences are correlated with a given question.

Lastly, to solve the cache contention problem, MnnFast makes memory-intensive embedding operations either bypass the cache or utilize a dedicated memory (called *embedding cache*). For the purpose, MnnFast separates two different types of memory requests (i.e., memory-intensive embedding operation vs. compute-intensive inference operation) to eliminate cache-contention effects.

For the evaluation, we implement MnnFast on top of various platforms: CPU, GPU, and FPGA. We measure the impact of the three optimizations (i.e., column-based algorithm with streaming, zero-skipping, embedding cache). To show the effectiveness of each optimization, we first present the results of CPU-based MnnFast with extensive profiling and analysis: memory throttling test and cache statistics. Next, we implement GPU-based MnnFast and show that our optimizations can improve single-GPU performance as well as achieve scalable performance in a multi-GPU environment. Lastly, we build FPGA-based MnnFast with the embedding cache and measure its performance and energy efficiency.

The results show that MnnFast greatly improves the overall performance on CPU, GPU, and FPGA. For CPU, MnnFast achieves 5.38× speedup compared to the baseline using 20 threads (4.02× on average) and shows scalable performance. Our GPU implementation of MnnFast achieves the speedup of 1.33× and 4.34× on one GPU and four GPUs respectively. FPGA-based MnnFast achieves 2.01× speedup compared to the baseline FPGA implementation, and the embedding cache reduces the inference latency by up to 53.1%. In addition, we compare energy efficiency between CPU-based and FPGA-based MnnFast. The results show that FPGA-based MnnFast significantly improves energy efficiency by up to 6.54×.

In summary, we make the following contributions:

- **Problem identification.** We identify critical performance issues in accelerating a large-scale MemNN (i.e., high memory bandwidth consumption, heavy computation, cache contention).
- **High performance from novel solutions.** MnnFast significantly improves the reasoning performance by applying three novel performance optimizations (i.e., column-based algorithm with streaming, zero skipping, embedding cache).
- **Various working prototypes.** We implement MnnFast on three different platforms: CPU, GPU, and FPGA. The CPU/GPU implementations are basically SW-based algorithm optimizations, whereas the FPGA implementation proposes a highly-scalable custom accelerator architecture.

The rest of the paper is organized as follows. Section 2 explains the characteristics of memory networks and motivates our work with its key design goals. Section 3 describes how MnnFast solves the performance problems to achieve the scalable performance.

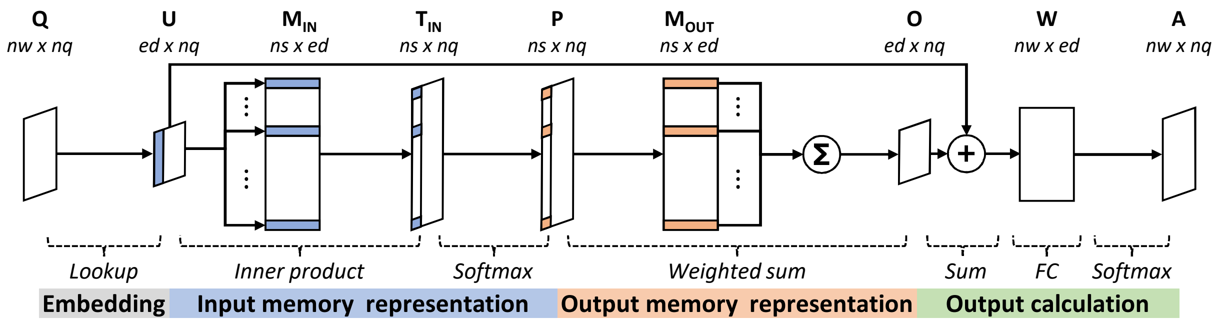


Figure 2: Computational steps of memory networks (MemNN). MemNN consists of embedding, input memory representation, output memory representation and output calculation. *nw* is the maximum number of words in a sentence. *nq* and *ns* are the number of questions and given story sentences, respectively. *ed* is the embedding dimension.

Section 4 shows implementation details of MnnFast on various hardware platforms. We provide evaluation results for MnnFast in Section 5. Finally, Section 6 and Section 7 provide related work and conclusion, respectively.

2 BACKGROUND & MOTIVATION

In this section, we introduce a representative memory-augmented neural network, *memory networks* developed by Facebook [69, 78] (Section 2.1). We then provide challenges and limitations of the state-of-the-art memory networks (Section 2.2) to motivate the design goals (Section 2.3).

2.1 Memory Networks

Neural networks have shown high accuracy comparable to humans on image classification and speech recognition. Recurrent neural networks (RNNs), designed to work on sequence prediction problems, derive an answer to a question from the previous reasonings [34, 48]. However, RNNs cannot memorize the previous history for a long time [5] nor handle a large amount of history due to their small memory [78]. Therefore, they cannot perform sophisticated tasks requiring a large amount of memory (e.g., comprehending a series of books to provide useful information to users).

Memory networks (MemNNs), developed by Facebook, solve the problem of RNNs by augmenting neural networks with external memory [69, 78]. The large-scale external memory allows MemNN to solve the sophisticated tasks. Nowadays, thanks to its huge reasoning power, MemNN is widely used in various fields from simple dialog comprehension to question & answering system using a large-scale dataset (e.g., Wikipedia) [17, 32, 49, 69, 76]

Figure 2 shows the high-level overview of MemNN's computational structure. MemNN consists of two major operations: *embedding* and *inference*. The embedding operation converts a given sentence into an internal state. MemNN first converts story and question sentences into internal states, and these states are stored into input/output memory (M_{IN}/M_{OUT}) and question state memory (U), respectively. The inference operation calculates the answers to each question by going through multiple computational layers of different types: *input memory representation*, *output memory representation*, and *output calculation*. By doing so, MemNN successfully reasons out answers by exploiting the large-scale memory components. The following paragraphs describe each operation and their characteristics in more details.

Embedding operation. The main purpose of the embedding operation is to convert an input sentence into a representative internal state vector (of size $1 \times ed$). First, MemNN extracts features from texts by using a bag-of-words (BoW) model [61]. In the BoW model, MemNN embeds each word by looking up a vector from an embedding matrix (of size $ed \times V$, V is the number of words in a dictionary) and sums the resulting vectors to represent the sentence.[1] The story sentences are translated into internal state vectors with this process, and these vectors are stored into matrices M_{IN} and M_{OUT}. Similarly, question sentences are also embedded in a matrix U (Figure 2).

Inference operation. With the extracted internal state vectors, MemNN calculates an answer during the inference operation which consists of three computational steps: input memory representation, output memory representation, and output calculation.

First, in the input memory representation step, MemNN calculates a probability vector, *p-vector*, which represents the correlation between a question and each story sentence.

$$p = Softmax(u \times M_{IN}) \tag{1}$$

Equation (1) shows how MemNN computes the p-vector. MemNN computes the p-vector by calculating the dot product of the internal state vector of a question (u) and each memory state vector in the input matrix (M_{IN}), and applying a softmax function ($Softmax(x_i) = e^{x_i}/\sum_j e^{x_j}$) to the results of the dot products. By doing so, MemNN can extract the correlation between the question and the story sentences.

Second, in the output memory representation step, MemNN calculates a weighted sum of the internal state vectors (i.e., embedded story sentences) in the output memory (M_{OUT}). Specifically, MemNN computes the sum over these output vectors (m_i^{OUT}) weighted by the probability value (p_i) (Equation (2)).

$$o = \sum_i p_i m_i^{OUT} \tag{2}$$

[1]Some studies multiply position weights to vectors before the sum of all vectors to preserve the order of words in the sentence.

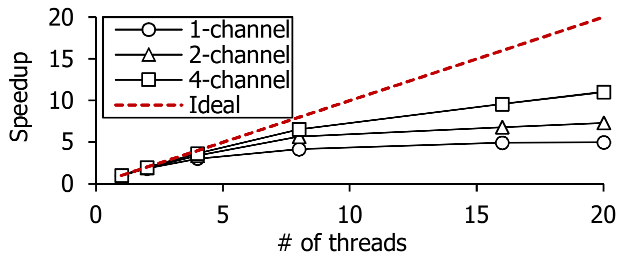


Figure 3: Limited scalability due to memory bandwidth bottleneck. The speedup results of each channel configuration are normalized to the corresponding single-thread results.

The resulting vector o (called response vector) is delivered to the output calculation step to make the final answer.

In the output calculation step, MemNN generates the final prediction for the given question. It computes the sum of the response vector o and the question vector u, and the outcome passes through the fully connected (FC) layer with a weight matrix (W). Depending on the network configuration, the input memory and output memory representation steps iterate over several times for better results, followed by the FC layer and the softmax function.

2.2 Performance Problems in MemNN

Researchers exploit the huge reasoning power of MemNN to solve sophisticated problems such as large-scale question answering tasks. To solve such complex problems, MemNN becomes bigger and turns into a large-scale memory network, demanding larger embedding dimension (*ed*) and more input sentences (*ns*) [7, 8, 17, 32, 41, 49]. The large-scale memory networks require high scalability to handle the increasing computation and memory demands. The current MemNN, however, cannot achieve scalability for three reasons: *high memory bandwidth consumption*, *heavy computation*, and *cache contention*.

In this section, we show the major performance bottlenecks in the state-of-the-art MemNN [69]. We first explain how memory bandwidth affects the overall performance (Section 2.2.1). Next, we provide the characteristics of MemNN computation, which requires huge compute resources (Section 2.2.2). Lastly, we show cache contention between the inference and embedding operations and quantify its performance impacts (Section 2.2.3).

2.2.1 High Memory Bandwidth Consumption. MemNN requires a significant amount of memory bandwidth. For example, during the embedding operation, MemNN looks up the embedding matrix to convert input sentences into internal state vectors. Larger embedding dimension is beneficial for solving complicated questions [7] but incurs higher memory pressure. Not only the embedding operation, but also the inference operation causes high memory traffic (e.g., input/output memory accesses, intermediate data spills). In the inference operation, MemNN has to load the whole input and output memory (M_{IN} and M_{OUT}, respectively) whose size is proportionate to the embedding dimension (*ed*) and the number of story sentences (*ns*). As the networks are getting larger, the size of these in/out memory is rapidly increasing. Furthermore, MemNN spills a large amount of intermediate data between each layer: *Inner product*, *Softmax*, and *Weighted sum* (Figure 2). These data spills are proportionate to *ns* as well. Therefore, their overhead will continuously increase.

Figure 3 shows how available memory bandwidth affects the scalability of MemNN. To prove the high memory bandwidth consumption is one of the key limiting factor, we measure the speedup with increasing number of threads for various memory bandwidth setups (# of memory channels). Here, we provide enough CPU cores so that the computation does not become a performance bottleneck (i.e., Xeon E5-2650 v4 12C/24T 2x). We observe that MemNN quickly reaches a performance saturation point as the available memory bandwidth decreases; in other words, a large amount of memory bandwidth consumption prevents MemNN from achieving scalable performance.

To overcome the memory bandwidth problem, we need more efficient memory management mechanisms to achieve good scalability. We propose a new computation algorithm (called *column-based algorithm*) which minimizes the size of data spills, provides more efficient data chunking, and enables MemNN to hide most memory accessing overhead. We explain the algorithm in Section 3.1 for more details.

2.2.2 Heavy Computation. Analyzing the characteristics and types of computation, we find out that certain phases of MemNN consist of a large number of compute-intensive operations. For example, MemNN requires multiple matrix multiplication operations (i.e., *Inner product*, *Weighted sum*, *FC layer*) known as compute-intensive tasks. Also, the *Softmax* function uses exponentiation requiring a large number of integer multiplications.

Therefore, MemNN is challenging to achieve scalable performance due to its substantial computational overhead. The amount of computation in MemNN superlinearly increases with the size of input data because its time complexity is O(n^a) (where $a >=$ 2.375) [80]. So, even if the number of CPU cores increases, the overall system will show sublinear performance. Also, we cannot easily scale-up CPU performance due to technology constraints; the overall performance will be quickly saturated.

To overcome the high computation problem, we need more powerful computing units (e.g., GPU, FPGA, ASIC) and optimization techniques to reduce the amount of computation. From an in-depth analysis, we find out the high potential for reducing the computation in the output memory representation step. The probability vector p represents the correlation between a question sentence and story sentences, and only a few story sentences are related to the given question; therefore, most of the values in p are close to zero. So, we propose a *zero-skipping* optimization to bypass a large amount of output computation. We explain the optimization in Section 3.2 for more details.

2.2.3 Cache Contention. MemNN consists of the embedding and inference operations, and these two operations exhibit different characteristics: memory-intensive and compute-intensive, respectively. The inference operation has a large amount of computation; therefore, to efficiently handle such a large amount of computes with high throughput, they need to keep their necessary data in a shared cache as much as possible [27]. On the other hand, the embedding operation accesses a large amount of data, which results in polluting the shared cache. Here, as multiple question answering

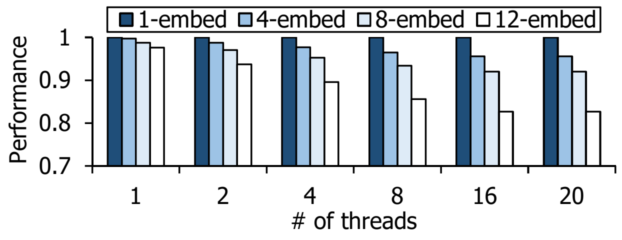


Figure 4: Performance degradation of inference threads due to co-executed embedding threads. The performance results are relative to the corresponding 1-embedding thread cases. Embedding threads contend with inferencing threads for shared memory system, thus reducing the performance of MemNN.

tasks can be executed simultaneously (i.e., assuming multi-tenant setting), two operations contend for the shared cache, which results in the cache contention problem. In turn, cache contention degrades the overall performance of MemNN.

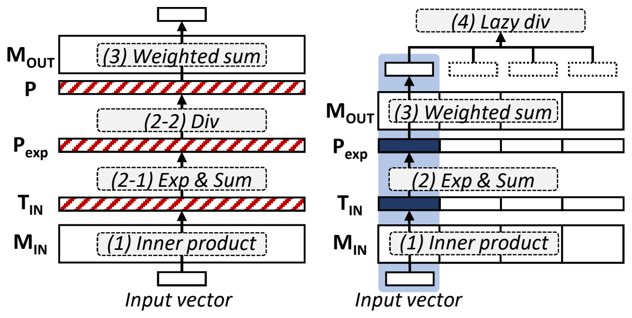
Figure 4 shows the performance degradation due to the cache contention. We measure the cache contention's impact on different scales of MemNN by varying the number of simultaneously-executed embedding operations. The impact increases with the scale of MemNN and the number of embedding operations, which indicates that we cannot simply scale up MemNN to meet the increasing demands due to the contention.

To overcome the cache contention problem, we should isolate memory accesses during the embedding operation from the other memory accesses. We can simply apply cache bypassing techniques; however, it results in high latency overhead to the embedding operation and puts more memory pressure on off-chip DRAM. To minimize such overhead, we propose an *embedding cache* dedicated to efficiently cache the embedding matrix. We describe a caching policy and an architecture of the embedding cache in Section 3.3 for more details.

2.3 Design Goals

Based on the discussion on the performance problems so far, we set our key design goals and provide a brief description of our key ideas to achieve the scalable large-scale MemNN.

- **Efficient memory management algorithm.** It should minimize the memory bandwidth requirements. We propose the *column-based algorithm with streaming* to eliminate the intermediate data spills.
- **Reduction of computation.** It should reduce the amount of computation. We propose the *zero-skipping* optimization to decrease the output computation by skipping output operations of near-zero probability values.
- **Shared cache isolation.** It should avoid cache contention between the embedding and inference operations. We propose the *embedding cache* dedicated for the embedding matrix.



(a) Baseline (b) Column-based algorithm

Figure 5: Dataflow comparison betweeen the baseline and the column-based algorithm.

3 MNNFAST

3.1 Column-Based Algorithm

MemNN suffers from a large amount of off-chip memory bandwidth (Section 2.2.1), which results in poor scalability. The current algorithm (*baseline*) consecutively calculates each layer (i.e., in-memory dot product (*Inner product*), softmax for p-value (*Softmax*), weighted sum with out-memory (*Weighted sum*)), which generates a number of intermediate data spills between each layer. Since shared cache cannot afford to hold these intermediate data, the baseline MemNN necessarily flushes and re-reads those temporary data to and from off-chip DRAM. Not only these frequent intermediate data spills but the baseline MemNN also suffers from inefficient data chunking of current matrix multiplication libraries (e.g., OpenBLAS) as their data chunking mechanisms are not MemNN-friendly.

$$o = \sum_i Softmax(u \times m_i^{IN}) m_i^{OUT} = \sum_i \frac{e^{u \times m_i^{IN}} m_i^{OUT}}{\sum_j e^{u \times m_j^{IN}}} \tag{3}$$

Equation (3) shows how baseline MemNN computes the output vector. The baseline first calculates a probability vector (p) by calculating the dot product between an input vector (u) and each in-memory vector (m_i^{IN}) followed by applying the softmax function. Next, the baseline computes the sum of weighted values multiplying each output-memory vector (m_i^{OUT}) by a corresponding probability value (p_i). Figure 5(a) describes the dataflow of these computational steps. The baseline generates three temporary vectors (i.e., T_{IN}, P_{exp}, P) for each question, and the size of these vectors is proportionate to the number of story sentences (ns) which continuously increases to support more complex question answering tasks. For example, when MemNN uses Wikipedia for training, the number of story sentences is around 200M [79]. In this case, the size of each intermediate vector is 800MB (assuming `float` data type) per each question, which easily exceeds the size of the typical on-chip shared cache (8MB – 40MB). Therefore, these temporary data are spilled to off-chip DRAM, incurring huge memory traffic and exacerbating the overall performance.

To reduce the size of temporary data, we propose a *column-based algorithm* which enables MemNN to partially calculate output vectors. The key idea of the proposed algorithm is a *lazy softmax* *calculation*, which computes the *Softmax*'s division operation at last, not in the middle.

$$o = \frac{1}{\sum_j e^{u \times m_j^{IN}}} \sum_i e^{u \times m_i^{IN}} m_i^{OUT} \quad (4)$$

Equation (4) shows how the column-based algorithm calculates the output vector. Compared to the baseline, the column-based algorithm pulls the sum ($\sum_j e^{u \times m_j^{IN}}$) out of the outer summation ($\sum_i$). Since the sum does not depend on the index i, the column-based algorithm generates the same results as the baseline. By doing so, the column-based MemNN does not need to wait for the sum of entire values in the *Softmax* function and possible to calculate a part of the output vector.

Figure 5(b) describes the computational steps of the column-based algorithm and its dataflow. The column-based MemNN partitions input/output memory into multiple *chunks*, and calculates partial output vectors for each chunk. The column-based MemNN computes dot products of the input vector with each in-memory vector and applies exponential function to the results, similar to the baseline. Here, in contrast to the baseline, the column-based MemNN directly calculates the weighted sum. The column-based MemNN performs the above operations on each chunk and accumulates each weighted sum into the output vector. After processing all chunks, the column-based MemNN divides the output vector by the sum ($\sum_j e^{u \times m_j^{IN}}$) calculated in the second step (*lazy softmax calculation*).

By doing so, the column-based MemNN can successfully reduce the size of temporary data to fit those into the on-chip cache. For example, when the chunk size is 1K, the total size of intermediate vectors (i.e., T_{IN}, P_{exp}) is only 8KB per each chunk calculation; therefore, we can eliminate the entire off-chip DRAM accesses for intermediate vectors. In addition, the column-based MemNN facilitates input/output memory streaming to hide the memory accessing overhead. In contrast to the baseline, which cannot load input/output memory into caches due to their enormous size, the column-based MemNN can load those memory into the cache because it partially loads input/output memory per each chunk processing. We show the performance impacts of both temporary data reduction and input/output memory streaming in Section 5.2.

Also, the column-based MemNN can reduce the amount of computation (i.e., softmax's division operation). In the baseline, the number of division operations is proportionate to the number of story sentences *ns* (step *2-2* in Figure 5(a)). However, the column-based MemNN requires the division operations proportional to the size of the embedding dimension *ed* (step *4* in Figure 5(b)). Since the typical size of *ed* (32 – 256) is much smaller than *ns* (> 100M), the column-based MemNN can significantly reduce the amount of computation.

Lastly, the column-based algorithm enables MemNN to achieve scale-out architecture. As the baseline computes each layer step-by-step, it cannot split each layer into multiple sub-layers due to enormous synchronization overhead. Therefore, to improve the overall performance, the system should be scaled-up. Instead, the column-based MemNN can partition each layer into multiple sub-layers based on the chunk and merge all results at once. Here, synchronization overhead is negligible because the size of output results are proportionate to *ed*. Therefore, the column-based MemNN can distribute these sub-tasks into multiple compute units (e.g., CPU, GPU, FPGA) and fully utilize these resources. We evaluate this scale-out characteristic in Section 5.3.

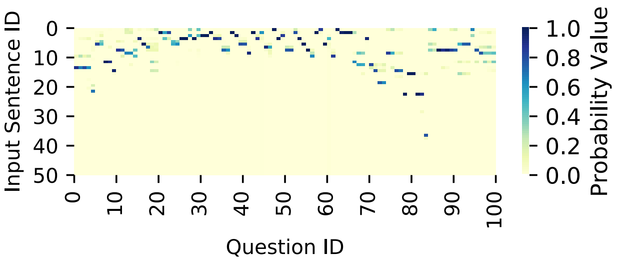


Figure 6: Probability value distribution. Each column represents the probability vector to each question. We use the Facebook bAbi dataset and testset [77].

Algorithm 1: MnnFast's zero-skipping algorithm.

```
input  : The skip threshold $th_{skip}$
input  : The probability vector $P$
input  : The output memory $M_{IN}$
input  : The number of story sentences $ns$
output : The weighted sum $O$
/* Calculate the weighted sum of the output memory with the probability values. */
1 $O = [0]$ /* Initialize the output vector. */
2 foreach $i < ns$ do
      /* $ns$ is the number of story sentences. */
3     if $p_i > th_{skip}$ then
4         $O = O + p_i m_i^{OUT}$
5 end
6 return $O$
```

3.2 Zero Skipping

Increasing demands for large-scale MemNN result in significant computational overhead because its computation algorithm shows super-linear complexity. As described in Section 2.2.2, MemNN's compute-intensive phase (the inference operation) consists of three core computation steps: inner product, softmax, and weighted sum. For inner product and weighted sum, we need to perform matrix multiplications known for super-linear time complexity. In this section, to reduce these computational overhead, we propose a *zero-skipping* technique and show its tradeoff between accuracy loss and the ratio of computation reduction.

The key observation for the zero-skipping technique is that the probability vector, calculated from the inner product between a question vector and in-memory matrix, shows large sparsity. Specifically, only a few values are non-zero and others are close to zero.[2] Figure 6 shows the probability value distribution. We use Facebook bAbi tasks and its dataset [77] and measure probability values to each question. In this evaluation, MemNN gets up to 50 story sentences followed by a question, and we show probability vectors

[2]Note that the sum of all values in the probability vector is one because these values are normalized by the softmax function.

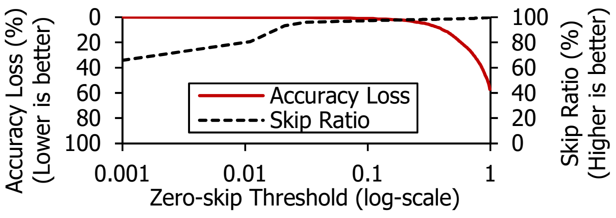


Figure 7: Tradeoffs between accuracy loss (relative loss in accuracy) and computation reduction according to the skip threshold.

for randomly chosen 100 questions. The results show that *only a few probability values are activated and the other values are close to zero.* It is because the probability vector means the correlation of question sentence with story sentences, and only a few story sentences are related to the given question.

By using this characteristic, we propose the *zero-skipping* optimization to bypass a large amount of output computation. Algorithm 1 shows how MnnFast reduces the quantity of output memory computation by using the zero-skipping optimization. In contrast to the baseline which calculates all multiplications between probability values and output memory vectors, the zero-skipping algorithm only computes the multiplication when the probability value is larger than a threshold value (th_{skip}) (line 3 and 4).

The zero-skipping optimization can significantly reduce the amount of output memory computation; however, it also sacrifices the prediction accuracy. If the skip threshold is too high, we may skip too much computation and drastically degrades the prediction accuracy. On the other hand, if the skip threshold is too low, we cannot get enough opportunity to reduce the quantity of computation.

To quantify the tradeoffs between accuracy loss and computation reduction, we measure both the accuracy loss and the ratio of computation reduction according to the different skip thresholds. We use Facebook's bAbi dataset and 20 QA tasks and get an average among the QA tasks. Both accuracy loss and computation reduction are measured by comparing with the baseline's accuracy and the amount of computation respectively.

Figure 7 shows that the zero-skipping optimization can achieve 97% reduction of output computation while sacrificing 0.87% of accuracy when the skip threshold is 0.1. When the skip threshold is 0.01, our optimization shows 81% reduction without any accuracy loss. Note that probability values in MemNN represent the correlation between a question and story sentences, and generally few story sentences are related to the given question. Therefore, the zero-skipping optimization is highly promising for MemNN to reduce the amount of computation.

3.3 Embedding Cache

As described in Section 2.2.3, the large-scale MemNN suffers from huge cache contention between the embedding and inference operations. While the inference operation actively uses the shared cache to maximize CPU utilization, the embedding operation generates a large number of memory requests and pollutes the shared cache. This cache contention dramatically increases the number of cache misses for the inference operation, which results in huge performance degradation. In this section, we provide some techniques (i.e., cache bypassing, embedding cache) to solve the cache contention problem in various hardware architectures.

In the CPU environment, we can simply apply a cache bypassing technique for embedding's memory requests. By using non-temporal memory instructions [1] for the embedding operation, we can achieve the memory isolation between the inference and embedding operations. However, the bypassing techniques has two major drawbacks. First, memory accessing overhead of the embedding operation are limited to DRAM access latencies, which increases the execution latency of the embedding operation significantly. Second, the technique raises the amount of memory pressure as the number of DRAM accesses increases.

To overcome these limitations, we propose an *embedding cache*. The embedding cache is a dedicated cache for storing internal state vectors during the embedding operation. As explained in Section 2.1, during the embedding operation, MemNN looks up a vector from the embedding matrix per each word in a sentence. Here, the embedding cache stores pairs of word ID (represented by the BoW model) and a corresponding internal state vector. Since each access loads the vectors whose size is the embedding dimension, we set the word size of our embedding cache as the embedding dimension. With the embedding cache, we can perfectly eliminate the cache contention.

In addition, the embedding cache reduces the quantity of off-chip DRAM memory requests thanks to the high locality in word usage. Linguistics researchers show that popular and frequently used words exist in both daily conversations and literature [14]. Therefore, most lookup operations load corresponding internal vectors from the embedding cache, which leads to reduce the number of DRAM accesses.

4 IMPLEMENTATION

To show the effectiveness of MnnFast across various platforms, we implement the baseline MemNN and MnnFast on CPU, GPU, and FPGA.

4.1 General-Purpose Architecture

While specialized architectures are getting lots of attention for machine learning, general-purpose CPU and GPU are still popular. Thus, we first validate our idea on CPU and GPU. Note that we implement and evaluate the proposed optimizations except for embedding cache on CPU and GPU as modifying the cache hierarchy of commodity hardware is infeasible. Based on the analysis of the general purpose architecture, we then elaborate on the possible specialized a hardware design.

4.1.1 CPU. We first implement the baseline MemNN [69] and MnnFast on CPU. In Section 5.2, we elaborate on the potential effect of column-based MemNN based on this implementation.

Baseline Implementation (MemNN). We implement MemNN in C++ with open-source BLAS library, OpenBLAS [75, 82]. We implement each operation (described in Section 2.1) as a single function. Our implementation takes three input data: input memory, output memory, and question sentences. The input and output memory are usually provided by the system while users submit the question sentences to the system. For interactive applications, users could provide necessary sentences to build the input and output memory containing a user-specific contextual database (e.g., the contents of a book that a user has read).

We assume that all the input/output memory have already been converted into the internal data format as the data would be prepared from an external database in advance. On the other hand, as questions are generated on-the-fly by users, we assume each question is a raw format (Bag-of-Words) which should be embedded.

First, we convert a given question into an internal representation. The operation consists of *lookup* operations. For each word in the question, we find the corresponding embedding vector from an embedding matrix (an embedding dictionary) and sum up the vectors into a single vector to represent the question. We implement the embedding matrix as an array to access embedding vectors in $O(1)$.

Then, we infer the corresponding answer from the given knowledge database, input and output memory. The remaining computational steps (input and output memory representation and output calculation) consist of a series of *Inner product*, *Softmax*, *Weighted sum*, *Vector sum*, and *FC*. All operations excluding *Softmax* are represented in vector operations (Section 2.1). For example, *Inner product* and *Weighted sum* are implemented as matrix-vector (vector-matrix) multiplications. To implement them, we rely on OpenBLAS for efficient computation.

We parallelize each operation in a lock-step manner. To parallelize the BLAS-based operations, we exploit the multi-threading feature of BLAS library. To parallelize *Softmax*, which does not rely on BLAS operations, we divide it into three steps: (1) applying the natural exponential function on the elements of a vector, (2) calculating the sum of the exponential results, and (3) normalizing the exponential results with the summation. We exploit data-level parallelization of each step with PThread.

Column-based algorithm & zero-skipping. The column-based algorithm simultaneously applies the inference operation on chunks of the given knowledge database. We divide the knowledge database into chunks, each of which contains 1000 sentences, and then make multiple worker threads to process them independently. Unlike MemNN, each thread performs a series of inference operations on only a given chunk, not on the entire data.

To perform inference on chunks in parallel, we use *Partial softmax* proposed in Section 3.1. Except for the softmax, the other operations are implemented in the same way to the baseline MemNN. By doing so, the column-based algorithm enables MemNN to parallelize an entire operation in chunk granularity.

As only a few sentences of the output memory are related to a given question, we apply zero-skipping (Section 3.2) to *Weight sum* operation. Our implementation skips adding up output sentences whose weight (or probability) is lower than 0.1.

4.1.2 GPU. As an intermediate step before moving on to the FPGA implementation, we evaluate the proposed ideas on GPUs. As discussed earlier, the column-based algorithm changes the following three consecutive steps: *inner product*, *softmax*, and *weighted sum*. Therefore, we make these three steps into GPU kernels and perform memory copy from/to GPUs between kernel invocations if needed.

GPU kernel implementation. We use cuBLAS [51] provided with CUDA Toolkit 10.0 to perform matrix-to-matrix multiplications. By using the state-of-the-art GPU BLAS library, we try to

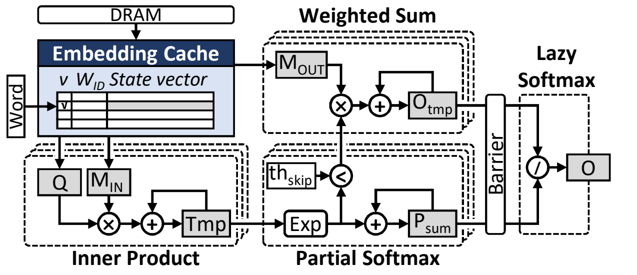


Figure 8: A high-level architecture of FPGA-based MnnFast.

avoid any inefficiency incurred by sub-optimal kernel implementation. Except for the *softmax*, all steps are implemented by simply calling a cuBLAS function. The softmax is implemented as one custom kernel (performing the exponential function on each input value) followed by a cuBLAS function to normalize the exponential results. The other two steps (i.e., inner product, weighted sum) are simply translated into cuBLAS functions. *Inner product* is matrix multiplication between M_{IN} and U. *Weighted sum* consists of two cuBLAS functions; one is matrix multiplication between P^T (transpose of P) and M_{OUT}, and another is multiplication of an all-ones vector and the result matrix of the previous matrix multiplication.

Column-based algorithm. We implement the column-based algorithm by using multiple CUDA streams/GPUs. Each stream/GPU processes chunks consisting of smaller number of sentences. Thanks to the *column-based algorithm*, we can parallelize all steps except for the last function calculating the weighted sum. Fortunately, this last function takes a negligible portion of the entire latency, as it sums up small matrices ($ed \times nq$), and the number of matrices are equals the number of streams/GPUs.

Zero skipping. A pruning scheme like zero skipping is ineffective or even harmful for GPUs [33, 50]. The reason is that a warp cannot complete early unless *all* threads in the warp are zero skipped, which is very unlikely. We can compact a sparse matrix to resolve this poor utilization problem; however, the cost of the transformation is excessive. We port and evaluate the latest matrix compaction scheme [33] for our case and check that the transformation latency is comparable to *weighted sum*'s latency. In addition to the transformation overhead, the matrix multiplication between compacted matrices leads to indirect memory accesses which delays all memory accesses significantly.

4.2 Custom Hardware (FPGA)

We design and implement an FPGA-based accelerator for MnnFast by using Vivado High-Level Synthesis (HLS). We create MnnFast's IP core from Vivado HLS and use the IP core in Xilinx Vivado Design Suite to generate a bitstream file for ZedBoard Zynq-7020 FPGA. We omit the baseline implementation because its design is straightforward.

Column-based algorithm. Figure 8 shows a high-level architecture of our FPGA-based accelerator design. First, each word in a sentence passes through the embedding cache to calculate the corresponding internal state vector. During embedding, MnnFast converts a question and new incoming story sentences into internal state vectors, and the state vectors of story sentences are appended to the input and output memory: M_{IN} and M_{OUT}, respectively.

Table 1: Memory networks configuration for the evaluation.

Entry	CPU	GPU	FPGA
Embedding dimension (# entry)	48	64	25
Database size (# sentences)	100M	100M	1000
Chunk-size (# sentences)	1000	Variable	25

Next, MnnFast partitions a number of story sentences ns into multiple chunks to reduce the size of intermediate data between each computational layer. For each chunk, MnnFast calculates the inner product between question vectors Q and each story vector m_i^{IN}. The resulting vector Tmp, whose size is same as ns, is delivered to the partial softmax.

In contrast to the baseline which is blocked until completely calculating the inner product over all chunks, MnnFast partially computes the softmax function followed by the weighted sum. During the partial softmax, MnnFast applies an exponential function to each value in the vector Tmp and accumulates the exponential results into P_{sum}. The exponential results are also delivered to the weighted sum to calculate the partial output vector O_{tmp}.

After processing all chunks, MnnFast applies a remaining part of the softmax function, *lazy softmax*. MnnFast divides each value in O_{tmp} by P_{sum} and returns the final output vector O.

Zero skipping. To implement the zero-skipping optimization, MnnFast compares the exponential results with the skip threshold th_{skip}. If the skip threshold is larger than an exponential result, MnnFast does not calculate the weighted sum and only accumulates the result into P_{sum}.

As multiple partial softmax units are parallelly executed at runtime, all exponential results may not be lower than the skip threshold. In this case, we calculate the weighted sum with those values although only a few values are higher than the skip threshold. Here, we do not consider the compaction as transformation overhead is significant. By doing so, MnnFast can reduce the amount of output computation significantly.

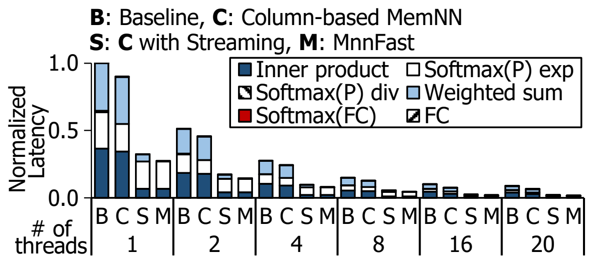
Embedding cache. We design the embedding cache as a direct mapped cache. Each entry in the embedding cache consists of three fields: a valid bit (1 bit), a word ID ($log2$(# words in dictionary) bits), and a state vector (32 * ed bits). By using the embedding cache, MnnFast can isolate the memory used for the inference and the embedding operation and reduce the number of DRAM accesses, which improves the performance of both inference and embedding operation. We show the performance impact of the embedding cache according to different cache sizes in Section 5.4.2.

5 EVALUATION

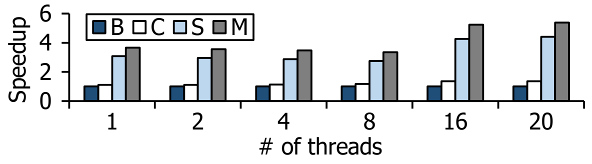
5.1 Experimental Setup

We implement the baseline, the baseline with each optimization (i.e., column-based algorithm, column-based with streaming, zero-skipping, embedding cache), and MnnFast in various hardware platforms: CPU, GPU, and FPGA.

CPU configuration. We compare the baseline with MnnFast on a 24-core dual-socket Xeon CPU system with DDR4-2400MHz 256GB memory. We run our implementation on Ubuntu 16.04 LTS and use OpenBLAS [75, 82] for BLAS operations. For evaluation, we use the network configurations, described in Table 1.



(a) Execution latency breakdown



(b) Performance speedup

Figure 9: Performance of column-based algorithm on CPU.

GPU configuration. We use a SUPERMICRO SuperServer 4028GR-TRT with two Intel Xeon CPU E5-2650 v4 and four Nvidia TITAN Xp GPUs. We measure the performance of GPU-based MnnFast with Linux kernel version (4.4.0-89-generic) and CUDA Toolkit version 10.0.

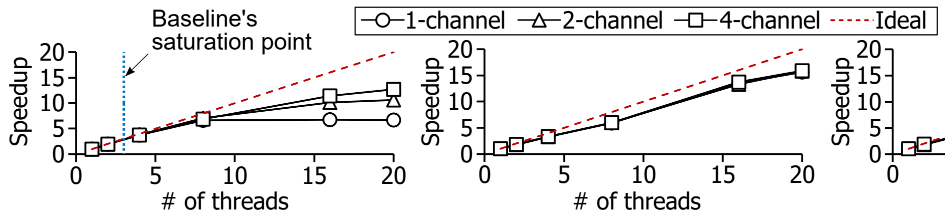
FPGA configuration. We implement FPGA-based MnnFast on ZedBoard featuring Xilinx Zynq-7020 Soc and DDR3 memory by using Vivado HLS. Our implementation on the programmable logic (PL) runs at 100MHz with DDR3 memory operating at 533MHz. The memory has 32-bit effective width. To control MnnFast implemented on the PL, we build a monitoring program executed on the ARM Cortex-A9 processor of Zynq SoC.

Memory network configuration. Table 1 shows the configuration parameters of MemNN for the evaluation. We use a similar configuration for CPU and GPU but scale it down for FPGA due to the lack of available logic cells. The embedding dimension of GPU is different to that of CPU to fully utilize streaming multiprocessors (SMs) in GPUs.

5.2 CPU

This section shows the performance and scalability of MnnFast compared to the baseline MemNN. To validate the key idea of our proposal, we compare the results from 1) the baseline MemNN, 2) Column-based MemNN, 3) Column-based MemNN with data streaming, and 4) MnnFast.

5.2.1 Performance. Figure 9 shows the performance of MnnFast and its comparison targets. Column-based algorithm provides two benefits, 1) improved cache usage from data chunking and 2) data streaming enabled by small-size data chunk. To analyze each benefit, we compare the performance of column-based algorithm without data streaming and with streaming against the baseline. Column-based algorithm achieves 1.21× speedup compared to the baseline. The speedup comes from the efficient data chunking and intermediate data reuse of our algorithm, which reduces the execution latency of Softmax (P) function (Figure 9(a)).



(a) Column (b) Column-S (c) MnnFast

Figure 10: Scalability of column-based algorithm on CPU.

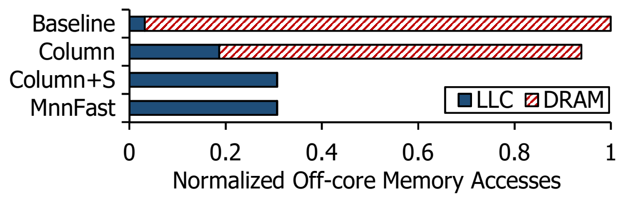


Figure 11: The number of off-chip memory accesses on CPU.

Data streaming significantly improves the performance against the baseline (3.33× on average). As data streaming timely loads all necessary data into computational units, we can hide the majority of cache miss latencies, and in turn improve the performance of *inner product* and *weighted sum* operation.

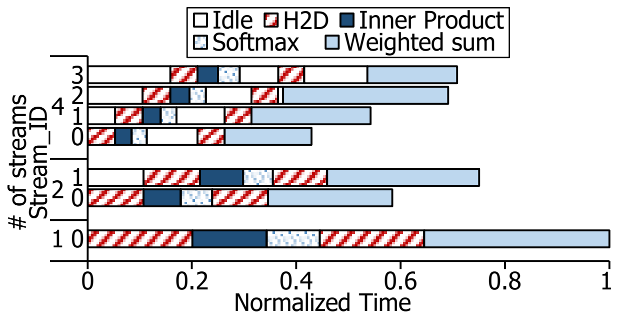
MnnFast achieves 4.02× average speedup thanks to our column-based algorithm with streaming and zero-skipping techniques. As MnnFast improves cache utilization, its effects increase with the number of working threads as shown in Figure 9(b).

5.2.2 *Cache Efficiency.* Figure 10 shows the scalability of MnnFast. Ideally, the performance should be linearly proportionate to the number of threads. We measure the performance improvement from parallelization at different memory systems. Figure 10(a) shows the performance of MemNN using the column-based algorithm is saturated at 10-thread on 4-memory channel system, which is more scalable than the baseline (saturation at around 4-thread). The column-based algorithm improves the scalability, but the performance still suffers from high memory bandwidth consumption.

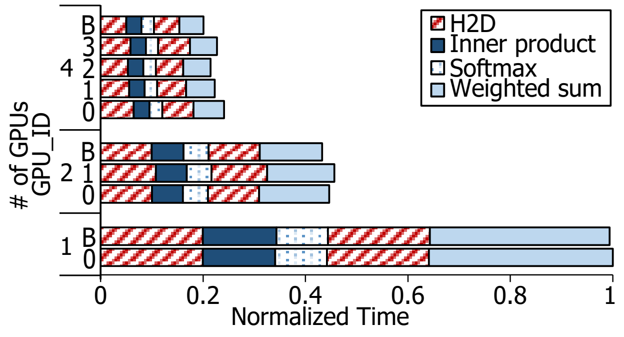
Figure 10(b) and 10(c) show that MnnFast achieves highly scalable performance with data streaming-enabled column-based algorithm, reaching the ideal speedup. Such a scalability comes from the fact that column-based algorithm with data streaming reduces the number of accesses to the shared memory system including shared cache. Figure 11 shows the number of off-chip memory accesses. The counts are normalized to the baseline result. The column-based algorithm makes off-chip DRAM accesses of the baseline hit on on-chip LLC, which in turn improves scalability over the baseline (Figure 10(a)). As the column-based algorithm with data streaming eliminates more than 60% of off-chip memory accesses, MnnFast achieves highly-scalable performance.

5.3 GPU

Figure 12 shows the scalability of MnnFast on GPUs. We use two parallelization schemes: 1) multiple CUDA streams on a single GPU



(a) Multiple CUDA streams



(b) Multiple GPUs (B: best run-alone)

Figure 12: Scalability of column-based algorithm on GPU.

and 2) multiple GPUs. Note that we omit the latency results of the last *sum* function and the device-to-host `memcpy` function, as both take a negligible portion of the entire latency.

Multiple CUDA streams. Figure 12(a) shows the latency results of evaluating multiple CUDA streams on a single GPU. CUDA streams can run concurrently as long as a GPU has enough resources for the concurrent execution. Most notably, memcpy functions cannot be overlapped as each memcpy function uses the full PCI-e bandwidth to transfer data. For this reason, we observe execution overlapped of kernel/kernel and kernel/memcpy, but not memcpy/memcpy. The results show that this simple change gives 1.33× speedup, confirming the importance of data transfer overhead between host memory and accelerator memory [18]. Increasing the number of streams does not reduce the latency much, as memcpy functions form a critical path.

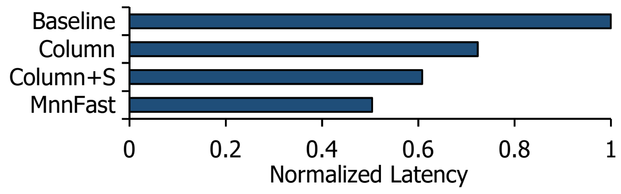


Figure 13: Latency reduction of FPGA-based MnnFast. Each latency is normalized to the baseline.

Multiple GPUs. Figure 12(b) shows the evaluation results of the multi-GPU case. We evenly distribute computation to the multiple GPUs and measure latencies of each GPU (e.g., GPU_ID 0-3). Here, we achieve much better scalability rather than multiple CUDA streams on a single GPU case, as multiple GPUs can overlap between memcpy and memcpy functions.

However, the scalability is still limited as all GPUs share the PCIe bandwidth. Each GPU requires data (i.e., input/output memory) sent from the host (*H2D*), and these memcpy functions for each GPU contend for the PCI-e bandwidth. To measure a performance impact of the PCI-e bandwidth contention, we evaluate an ideal case (B) not containing any effects from the PCI-e bandwidth contention caused by memcpy functions among multiple GPUs. As shown in figure 12(b), H2D latency differences between the worst case and the ideal case are getting larger as the number of GPUs increases.

Fortunately, this problem can be resolved by using multiple nodes to isolate the memory accesses via PCIe. Note that the communication overhead for the synchronization would be negligible, as the size of per-node results (partial weighted sum) is quite small.

5.4 FPGA

5.4.1 Performance. First, we evaluate the effectiveness of each optimization on FPGA. We implement four versions: baseline, column (applying column-based algorithm only), column+S (applying both column-based and streaming), and MnnFast. Figure 13 shows the latency results of the four versions of the FPGA implementation. Each latency result is normalized to the baseline. The results show each optimization gradually reduces the execution latency. Compared to the baseline, the column-based algorithm reduces the latency by 27.6% and by 38.2% with streaming optimization. With all the optimizations (i.e., column-based algorithm, streaming optimization, zero-skipping), MnnFast shows the performance improvement by up to 2.01×.

5.4.2 Effectiveness of Embedding Cache. To evaluate the effectiveness of the embedding cache, we use the word frequency of the Corpus of Contemporary American English dataset (COCA) [14] as input data for the embedding cache. Here, we set the size of the embedding dimension as 256. We measure the latency reduction according to different sizes of the embedding cache: 32KB, 64KB, 128KB, and 256KB. Figure 14 shows the latency reduction from the embedding cache of various sizes. Each latency is normalized to the "No Cache" version. The results show that the embedding cache effectively reduces latency overhead during the embedding operation. For each cache size (32KB, 64KB, 128KB, 256KB), the embedding cache reduces the latency by 34.5%, 41.7%, 47.7%, and 53.1%, respectively. Note that even moderate size of the cache (e.g., 32KB)

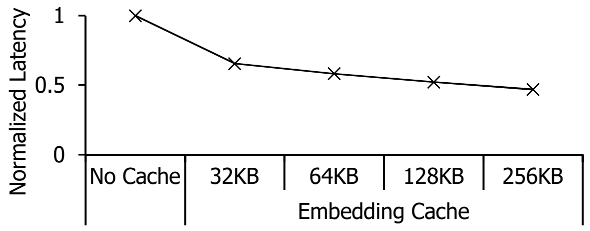


Figure 14: Effectiveness of embedding cache in FPGA-based MnnFast. Each latency result is normalized to the No Cache.

effectively reduces the latency of embedding operation thanks to the high word locality.

5.5 Comparison Between CPU and FPGA

We compare energy efficiency between CPU-based MnnFast and FPGA-based MnnFast. For a fair comparison, we resize the network configuration for both platforms to process the same quantity of question answering tasks. We use `turbostat` to measure energy consumption for CPU-based MnnFast. For FPGA-based MnnFast, we use power stats provided by Xilinx Vivado Design Suite after generating the bitstream. The results show that FPGA-based MnnFast improves energy efficiency by up to 6.54×.

6 RELATED WORK

6.1 Various DNN Accelerators

A DNN is basically composed of a series of linear algebra operations. Variants of existing DNNs usually have different requirements including new operations, which can motivate us to develop new optimized DNN accelerators [11, 15, 22, 39, 43, 64, 83]. However, this optimization is not easy; therefore, researchers propose systematic accelerator performance analysis and optimization methods [4, 23, 46, 58, 59, 63, 67, 68]. In this paper, we address the problems of memory network's huge memory bandwidth consumption and computation.

6.2 Dataflows and Data Reuse

Spatial architecture is optimized for DNNs, as shared values are broadcasted to ALUs, and intermediate results *flow* between ALUs [70]. Different dataflows have different sharing opportunity, which leads to various accelerator designs. Chen et al. [12] classify dataflows into various types, and propose a row stationary type accelerator, Eyeriss. Incrementing the dimensionality of NNs (e.g., image to video) diversifies dataflow configurations (e.g., 3D CNN [30]). High-level reuse opportunities like inter-inference reuse [10, 60] and server-wide reuse [24, 74] are also exploitable.

6.3 Approximation

Fault-tolerance of NNs [71, 72] allows NNs to adopt approximation with small accuracy loss while improving energy efficiency [62]. Fault-tolerance varies by NNs [57]. Some studies help architects properly use the approximation [3, 9, 25]. The excessive approximation can threaten our safety [89], and also degrade performance and energy if subsequent tasks are inefficient on inaccurate data [73, 86, 87].

Network pruning is an effective optimization, especially for constrained environments [13, 26, 55]. Static pruning is usually independent of hardware, while dynamic pruning relies on software-hardware co-designs. Static pruning can greatly reduce NN size with little loss of accuracy [35] and domain adaptation ability [81]. Song et al. [66] improve the accuracy of overly-pruned NNs by incremental updates. Dynamic pruning can reduce the working set size [66, 85] using runtime information. Value prediction [47, 54, 84] can increase pruning candidates [2, 65]. Dynamic pruning can produce a sparse matrix whose irregular data access patterns can negate the pruning benefits; thus, accelerators specialized for sparse matrices [29, 31, 37, 40, 52, 88] are proposed.

Generative adversarial networks (GANs) *populate* zeros as opposed to pruning, but require sparse DNN accelerators [64, 83].

Precision can also be adjusted. Low-voltage SRAM [6] is universally applicable to accelerators. Brandon et al. [56] propose a novel lossy weight encoding scheme. Jain et al. [36] propose layer-specific lossy and lossless encoding schemes for intermediate layer outputs.

Processing in memory (PIM) performs read and computation simultaneously while losing reliability [21], which is appealing features for DNN accelerators [19]. The PIM reliability issue is handled by [20, 21, 42]. Quantization decreases overall bitwidths [28, 45, 55]. Park et al. [53] use two different bitwidths to handle outliers. Ding et al. [16] replace expensive multipliers with cheaper shifters and adders. Kim et al. [38] proposes device-aware quantization by applying different quantization bits into each heterogeneous computation unit.

The previous work searches the applicability of approximation techniques on traditional deep neural networks (e.g., CNNs). Our work, MnnFast studies a new type of neural networks, *memory-augmented networks* and reveals that the effectiveness of approximation remains on them.

7 CONCLUSION

We propose MnnFast, a novel system architecture for large-scale memory networks to achieve fast and scalable reasoning performance. We identify three performance problems (i.e., *high memory bandwidth consumption*, *heavy computation*, *cache contention*) of the current architecture and propose three key optimizations: *column-based algorithm with streaming*, *zero-skipping*, and *embedding cache*. We show MnnFast successfully solves the performance problems and outperforms the baseline on various hardware: CPU, GPU, and FPGA.

ACKNOWLEDGMENTS

This work was partly supported by ICT R&D Program of SK Telecom, Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (NRF-2015M3C4A7065647, NRF-2017R1A2B3011038), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. R0190-15-2012, No. 1711073912), and Creative Pioneering Researchers Program through Seoul National University. We also appreciate the support from Automation and Systems Research Institute (ASRI) and Inter-university Semiconductor Research Center (ISRC) at Seoul National University.

REFERENCES

[1] 2018. *Intel 64 and IA-32 Architectures Software Developer's Manual Vol. 3A.* Intel, Chapter Memory Cache Control.

[2] V Aklaghi, Amir Yazdanbakhsh, Kambiz Samadi, H Esmaeilzadeh, and RK Gupta. 2018. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 662–673. https://doi.org/10.1109/ISCA.2018.00061

[3] Riad Akram and Abdullah Muzahid. 2017. Approximeter: Automatically finding and quantifying code sections for approximation. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*. 116–117. https://doi.org/10.1109/IISWC.2017.8167765

[4] Muhammad Shoaib Bin Altaf and David A. Wood. 2017. LogCA: A High-Level Performance Model for Hardware Accelerators. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 375–388. https://doi.org/10.1145/3079856.3080216

[5] Y. Bengio, P. Simard, and P. Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* 5, 2 (March 1994), 157–166. https://doi.org/10.1109/72.279181

[6] Ramon Bertran, Pradip Bose, David Brooks, Jeff Burns, Alper Buyuktosunoglu, Nandhini Chandramoorthy, Eric Cheng, Martin Cochet, Schuyler Eldridge, Daniel Friedman, Hans Jacobson, Rajiv Joshi, Subhasish Mitra, Robert Montoye, Arun Paidimarri, Pritish Parida, Kevin Skadron, Mircea Stan, Karthik Swaminathan, Augusto Vega, Swagath Venkataramani, Christos Vezyrtzis, Gu-Yeon Wei, John-David Wellman, and Matthew Ziegler. 2017. Very Low Voltage (VLV) Design. In *2017 IEEE International Conference on Computer Design (ICCD)*. 601–604. https://doi.org/10.1109/ICCD.2017.105

[7] Antoine Bordes, Nicolas Usunier, Sumit Chopra, and Jason Weston. 2015. Large-scale Simple Question Answering with Memory Networks. *CoRR* abs/1506.02075 (2015). arXiv:1506.02075 http://arxiv.org/abs/1506.02075

[8] Antoine Bordes and Jason Weston. 2017. Learning End-to-End Goal-Oriented Dialog. In *7th International Conference on Learning Representations (ICLR '17)*.

[9] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability Type Inference for Flexible Approximate Programming. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 470–487. https://doi.org/10.1145/2814270.2814301

[10] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. 2018. EVA^2: Exploiting Temporal Redundancy in Live Computer Vision. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 533–546. https://doi.org/10.1109/ISCA.2018.00051

[11] Ruizhe Cai, Ao Ren, Ning Liu, Caiwen Ding, Luhao Wang, Xuehai Qian, Massoud Pedram, and Yanzhi Wang. 2018. VIBNN: Hardware Acceleration of Bayesian Neural Networks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 476–488. https://doi.org/10.1145/3173162.3173212

[12] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (Jan 2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[13] Ting-Wu Chin, Cha Zhang, and Diana Marculescu. 2018. Layer-compensated pruning for resource-constrained convolutional neural networks. *arXiv preprint arXiv:1810.00518* (2018).

[14] Mark Davies. 2008-. The Corpus of Contemporary American English (COCA): 560 million words, 1990-present. Available online at https://corpus.byu.edu/coca/.

[15] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. 2017. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 561–574. https://doi.org/10.1145/3079856.3080248

[16] Ruizhou Ding, Zeye Liu, Rongye Shi, Diana Marculescu, and R.D. (Shawn) Blanton. 2017. LightNN: Filling the Gap Between Conventional Deep Neural Networks and Binarized Networks. In *Proceedings of the on Great Lakes Symposium on VLSI 2017 (GLSVLSI '17)*. ACM, New York, NY, USA, 35–40. https://doi.org/10.1145/3060403.3060465

[17] Jesse Dodge, Andreea Gane, Xiang Zhang, Antoine Bordes, Sumit Chopra, Alexander H. Miller, Arthur Szlam, and Jason Weston. 2016. Evaluating Prerequisite Qualities for Learning End-to-End Dialog Systems. In *6th International Conference on Learning Representations (ICLR '16)*.

[18] Marco Donato, Brandon Reagen, Lillian Pentecost, Udit Gupta, David Brooks, and Gu-Yeon Wei. 2018. On-chip Deep Neural Network Storage with Multi-level eNVM. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. ACM, New York, NY, USA, Article 169, 6 pages. https://doi.org/10.1145/3195970.3196083

[19] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. 2018. Neural Cache: Bit-serial In-cache Acceleration of Deep Neural Networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 383–396. https://doi.org/10.1109/ISCA.2018.00040

[20] Ben Feinberg, Uday Kumar Reddy Vengalam, Nathan Whitehair, Shibo Wang, and Engin Ipek. 2018. Enabling Scientific Computing on Memristive Accelerators. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 367–382. https://doi.org/10.1109/ISCA.2018.00039

[21] Ben Feinberg, Shibo Wang, and Engin Ipek. 2018. Making Memristive Neural Network Accelerators Reliable. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 52–65. https://doi.org/10.1109/HPCA.2018.00015

[22] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-scale DNN Processor for Real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 1–14. https://doi.org/10.1109/ISCA.2018.00012

[23] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and Jose MF Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (Nov 2018), 1935–1968. https://doi.org/10.1109/JPROC.2018.2873289

[24] Adi Fuchs and David Wentzlaff. 2018. Scaling Datacenter Accelerators with Compute-reuse Architectures. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 353–366. https://doi.org/10.1109/ISCA.2018.00038

[25] Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Christina Silvano. 2018. mARGOt: a Dynamic Autotuning Framework for Self-aware Approximate Computing. *IEEE Trans. Comput.* (2018), 1–1. https://doi.org/10.1109/TC.2018.2883597

[26] Graham Gobieski, Nathan Beckmann, and Brandon Lucia. 2018. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. *ArXiv e-prints*, Article arXiv:1810.07751 (Sept. 2018), arXiv:1810.07751 pages. arXiv:cs.DC/1810.07751

[27] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[28] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168* (2016).

[29] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[30] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W Fletcher. 2018. Morph: Flexible Acceleration for 3D CNN-based Video Understanding. *arXiv preprint arXiv:1810.06807* (2018).

[31] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W Fletcher. 2018. UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 674–687. https://doi.org/10.1109/ISCA.2018.00062

[32] Felix Hill, Antoine Bordes, Sumit Chopra, and Jason Weston. 2016. The Goldilocks Principle: Reading Children's Books with Explicit Memory Representations. In *6th International Conference on Learning Representations (ICLR '16)*.

[33] Parker Hill, Animesh Jain, Mason Hill, Babak Zamirai, Chang-Hong Hsu, Michael A. Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. 2017. DeftNN: Addressing Bottlenecks for DNN Execution on GPUs via Synapse Vector Elimination and Near-compute Data Fission. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 786–799. https://doi.org/10.1145/3123939.3123970

[34] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[35] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

[36] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient Data Encoding for Deep Neural Network Training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 776–789. https://doi.org/10.1109/ISCA.2018.00070

[37] Houxiang Ji, Linghao Song, Li Jiang, Hai Halen Li, and Yiran Chen. 2018. ReCom: An efficient resistive accelerator for compressed deep neural networks. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 237–240. https://doi.org/10.23919/DATE.2018.8342009

[38] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 45.

[39] Jaeha Kung, Yun Long, Duckhwan Kim, and Saibal Mukhopadhyay. 2017. A programmable hardware accelerator for simulating dynamical systems. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 403–415. https://doi.org/10.1145/3079856.3080252

[40] Joo Hwan Lee and Hyesoon Kim. 2018. StaleLearn: Learning Acceleration with Asynchronous Synchronization Between Model Replicas on PIM. *IEEE Trans. Comput.* 67, 6 (June 2018), 861–873. https://doi.org/10.1109/TC.2017.2780237

[41] Jiwei Li, Alexander H. Miller, Sumit Chopra, Marc'Aurelio Ranzato, and Jason Weston. 2016. Dialogue Learning With Human-In-The-Loop. *CoRR* abs/1611.09823 (2016). arXiv:1611.09823 http://arxiv.org/abs/1611.09823

[42] Meng-Yao Lin, Hsiang-Yun Cheng, Wei-Ting Lin, Tzu-Hsien Yang, I-Ching Tseng, Chia-Lin Yang, Han-Wen Hu, Hung-Sheng Chang, Hsiang-Pang Li, and Meng-Fan Chang. 2018. DL-RSIM: A Simulation Framework to Enable Reliable ReRAM-based Accelerators for Deep Learning. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '18)*. ACM, New York, NY, USA, Article 31, 8 pages. https://doi.org/10.1145/3240765.3240800

[43] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. 2018. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 751–766. https://doi.org/10.1145/3173162.3173191

[44] Ying Ma and José C. Príncipe. 2018. A Taxonomy for Neural Memory Networks. *CoRR* abs/1805.00327 (2018). arXiv:1805.00327 http://arxiv.org/abs/1805.00327

[45] Yufei Ma, Naveen Suda, Yu Cao, Jae-sun Seo, and Sarma Vrudhula. 2016. Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2016.7577356

[46] Diana Marculescu, Dimitrios Stamoulis, and Ermao Cai. 2018. Hardware-aware Machine Learning: Modeling and Optimization. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '18)*. ACM, New York, NY, USA, Article 137, 8 pages. https://doi.org/10.1145/3240765.3243479

[47] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load Value Approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 127–139. https://doi.org/10.1109/MICRO.2014.22

[48] Tomas Mikolov, Martin KarafiÃąt, Lukas Burget, Jan CernockÃ¡, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. *Proceedings of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH 2010* 2, 1045–1048.

[49] Alexander Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. 2016. Key-Value Memory Networks for Directly Reading Documents. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 1400–1409. https://doi.org/10.18653/v1/D16-1147

[50] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/3020078.3021740

[51] NVIDIA. [n.d.]. cuBLAS. Retrieved September, 2018 from https://docs.nvidia.com/cuda/archive/10.0/cublas/index.html

[52] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/3079856.3080254

[53] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 688–698. https://doi.org/10.1109/ISCA.2018.00063

[54] Arthur Perais and André Seznec. 2014. Practical data value speculation for future high-end processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 428–439. https://doi.org/10.1109/HPCA.2014.6835952

[55] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. *CoRR* abs/1802.05668 (2018). arXiv:1802.05668 http://arxiv.org/abs/1802.05668

[56] Brandon Reagen, Udit Gupta, Robert Adolf, Michael M Mitzenmacher, Alexander M Rush, Gu-Yeon Wei, and David Brooks. 2017. Weightless: Lossy Weight Encoding For Deep Neural Network Compression. *arXiv preprint arXiv:1711.04686*

(2017).

[57] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. 2018. Ares: A Framework for Quantifying the Resilience of Deep Neural Networks. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. ACM, New York, NY, USA, Article 17, 6 pages. https://doi.org/10.1145/3195970.3195997

[58] Brandon Reagen, José Miguel Hernández-Lobato, Robert Adolf, Michael Gelbart, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2017. A case for efficient accelerator design space exploration via Bayesian optimization. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6. https://doi.org/10.1109/ISLPED.2017.8009208

[59] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-power, Highly-accurate Deep Neural Network Accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 267–278. https://doi.org/10.1109/ISCA.2016.32

[60] Marc Riera, Jose-Maria Arnau, and Antonio González. 2018. Computation Reuse in DNNs by Exploiting Input Similarity. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 57–68. https://doi.org/10.1109/ISCA.2018.00016

[61] Zellig S. Harris. 1954. Distributional Structure. *Word* 10 (08 1954), 146–162. https://doi.org/10.1007/978-94-009-8467-7_1

[62] Adrian Sampson, James Bornholt, and Luis Ceze. 2015. Hardware-Software Co-Design: Not Just a Cliché. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 262–273. https://doi.org/10.4230/LIPIcs.SNAPL.2015.262

[63] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2015. The Aladdin Approach to Accelerator Design and Modeling. *IEEE Micro* 35, 3 (May 2015), 58–70. https://doi.org/10.1109/MM.2015.50

[64] Mingcong Song, Jiaqi Zhang, Huixiang Chen, and Tao Li. 2018. Towards Efficient Microarchitectural Design for Accelerating Unsupervised GAN-Based Deep Learning. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 66–77. https://doi.org/10.1109/HPCA.2018.00016

[65] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. 2018. Prediction Based Execution on Deep Neural Networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 752–763. https://doi.org/10.1109/ISCA.2018.00068

[66] Mingcong Song, Kan Zhong, Jiaqi Zhang, Yang Hu, Duo Liu, Weigong Zhang, Jing Wang, and Tao Li. 2018. In-Situ AI: Towards Autonomous and Incremental Deep Learning for IoT Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 92–103. https://doi.org/10.1109/HPCA.2018.00018

[67] Dimitrios Stamoulis, Ermao Cai, Da-Cheng Juan, and Diana Marculescu. 2018. HyperPower: Power- and memory-constrained hyper-parameter optimization for neural networks. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 19–24. https://doi.org/10.23919/DATE.2018.8341973

[68] Dimitrios Stamoulis, Ting-Wu (Rudy) Chin, Anand Krishnan Prakash, Haocheng Fang, Sribhuvan Sajja, Mitchell Bognar, and Diana Marculescu. 2018. Designing Adaptive Neural Networks for Energy-constrained Image Classification. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '18)*. ACM, New York, NY, USA, Article 23, 8 pages. https://doi.org/10.1145/3240765.3240796

[69] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. 2015. End-to-end memory networks. *CoRR* abs/1503.08895 (2015). arXiv:1503.08895 http://arxiv.org/abs/1503.08895

[70] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (Dec 2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

[71] Hanlin Tang, Chen Yu, Cedric Renggli, Simon Kassing, Ankit Singla, Dan Alistarh, Ji Liu, and Ce Zhang. 2018. Distributed Learning over Unreliable Networks. *arXiv preprint arXiv:1810.07766* (2018).

[72] Olivier Temam. 2012. A Defect-tolerant Accelerator for Emerging High-performance Applications. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 356–367. http://dl.acm.org/citation.cfm?id=2337159.2337200

[73] Radha Venkatagiri, Karthik Swaminathan, Chung-Ching Lin, Liang Wang, Alper Buyuktosunoglu, Pradip Bose, and Sarita Adve. 2018. Impact of Software Approximations on the Resiliency of a Video Summarization System. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 598–609. https://doi.org/10.1109/DSN.2018.00067

[74] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. 2017. ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 13–26. https://doi.org/10.1145/3079856.3080244

[75] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 25, 12 pages. https://doi.org/10.1145/2503210.2503219

[76] Jason Weston. 2016. Dialog-based Language Learning. *CoRR* abs/1604.06045 (2016). arXiv:1604.06045 http://arxiv.org/abs/1604.06045

[77] Jason Weston, Antoine Bordes, Sumit Chopra, and Tomas Mikolov. 2015. Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks. *CoRR* abs/1502.05698 (2015). arXiv:1502.05698 http://arxiv.org/abs/1502.05698

[78] Jason Weston, Sumit Chopra, and Antoine Bordes. 2014. Memory Networks. *CoRR* abs/1410.3916 (2014). arXiv:1410.3916 http://arxiv.org/abs/1410.3916

[79] Wikipedia. [n.d.]. The size of Wikipedia. Retrieved December, 2018 from https://en.wikipedia.org/wiki/Wikipedia:Size_in_volumes

[80] Virginia Vassilevska Williams. 2011. Breaking the Coppersmith-Winograd barrier.

[81] Chunpeng Wu, Wei Wen, Tariq Afzal, Yongmei Zhang, Yiran Chen, and Hai Li. 2017. A Compact DNN: Approaching GoogLeNet-Level Accuracy of Classification and Domain Adaptation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 761–770. https://doi.org/10.1109/CVPR.2017.88

[82] Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS '12)*. IEEE Computer Society, Washington, DC, USA, 684–691. https://doi.org/10.1109/ICPADS.2012.97

[83] Amir Yazdanbakhsh, Hajar Falahati, Philip J Wolfe, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. 2018. GANAX: A Unified MIMD-SIMD Acceleration for Generative Adversarial Networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 650–661. https://doi.org/10.1109/ISCA.2018.00060

[84] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C. Mowry. 2016. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. *ACM Trans. Archit. Code Optim.* 12, 4, Article 62, 26 pages. https://doi.org/10.1145/2836168

[85] Reza Yazdani, Jose-Maria Arnau, and Antonio González. 2017. UNFOLD: A Memory-efficient Speech Recognizer Using On-the-fly WFST Composition. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 69–81. https://doi.org/10.1145/3123939.3124542

[86] Reza Yazdani, Marc Riera, Jose-Maria Arnau, and Antonio González. 2018. The Dark Side of DNN Pruning. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 790–801. https://doi.org/10.1109/ISCA.2018.00071

[87] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 548–560. https://doi.org/10.1145/3079856.3080215

[88] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An Accelerator for Sparse Neural Networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 20, 12 pages. http://dl.acm.org/citation.cfm?id=3195638.3195662

[89] Yuhao Zhu, Vijay Janapa Reddi, Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2017. Cognitive Computing Safety: The New Horizon for Reliability / The Design and Evolution of Deep Learning Workloads. *IEEE Micro* 37, 1 (Jan.-Feb. 2017), 15–21. https://doi.org/10.1109/MM.2017.2