Manna: An Accelerator for Memory-Augmented Neural Networks

Jacob R. Stevens
steven69@purdue.edu
School of ECE
Purdue University

Ashish Ranjan*
aranjan@purdue.edu
School of ECE
Purdue University

Dipankar Das
dipankar.das@intel.com
Parallel Computing Lab
Intel Corporation

Bharat Kaul
bharat.kaul@intel.com
Parallel Computing Lab
Intel Corporation

Anand Raghunathan
raghunathan@purdue.edu
School of ECE
Purdue University

ABSTRACT

Memory-augmented neural networks (MANNs)– which augment a traditional Deep Neural Network (DNN) with an external, differentiable memory– are emerging as a promising direction in machine learning. MANNs have been shown to achieve one-shot learning and complex cognitive capabilities that are well beyond those of classical DNNs. We analyze the computational characteristics of MANNs and observe that they present a unique challenge due to *soft reads and writes* to the differentiable memory, each of which requires access to *all* the memory locations. This results in poor performance of MANNs on modern CPUs, GPUs, and other accelerators. To address this, we present Manna, a specialized hardware inference accelerator for MANNs. Manna is a memory-centric design that focuses on maximizing performance in an extremely low FLOPS/Byte context. The key architectural features from which Manna derives efficiency are: (i) investing most of the die area and power in highly banked on-chip memories that provide ample bandwidth rather than large matrix-multiply units that would be underutilized due to the low reuse (ii) a hardware-assisted transpose mechanism for accommodating the diverse memory access patterns observed in MANNs, (iii) a specialized processing tile that is equipped to handle the nearly-equal mix of MAC and non-MAC computations present in MANNs, and (iv) methods to map MANNs to Manna that minimize data movement while fully exploiting the little reuse present. We evaluate Manna by developing a detailed architectural simulator with timing and power models calibrated by synthesis to the 15 nm Nangate Open Cell library. Across a suite of 10 benchmarks, Manna demonstrates average speedups of 39x with average energy improvements of 122x over an NVIDIA 1080-Ti Pascal GPU and average speedups of 24x with average energy improvements of 86x over a state-of-the-art NVIDIA 2080-Ti Turing GPU.

*Ashish Ranjan is currently a research staff member at IBM T.J. Watson Research Center, Yorktown Heights, NY (ashish.ranjan@ibm.com)

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10.
https://doi.org/10.1145/3352460.3358304

CCS CONCEPTS

• **Computer systems organization → Neural networks**.

KEYWORDS

Memory Networks, Memory-Augmented Neural Networks, Hardware Accelerators, System Architecture

ACM Reference Format:
Jacob R. Stevens, Ashish Ranjan, Dipankar Das, Bharat Kaul, and Anand Raghunathan. 2019. Manna: An Accelerator for Memory-Augmented Neural Networks. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358304

1 INTRODUCTION

Deep Neural Networks (DNNs) have kindled a renewed interest in the field of machine learning over the past decade. Algorithmic advances, availability of larger datasets, and tremendous growth in compute capabilities have led to DNNs achieving state-of-the-art accuracy in a wider variety of perceptual domains, including video, audio, and text processing [19, 20, 41]. Despite the success and widespread deployment of DNNs, many machine learning tasks such as task planning, reasoning *etc.*, as well as one-shot learning (the ability to learn from one or very few examples), remain open challenges for them.

Memory-augmented neural networks (MANNs) promise to address some of these challenges by providing the network with access to dynamic (readable and writeable) state [14, 15, 25, 33]. Unlike recurrent neural networks (RNNs), wherein the dynamic state is intrinsically embodied within the topology of the network itself, in MANNs the dynamic state is decoupled from the neural network. This decoupled dynamic state is realized through a *differentiable external memory* that is accessed by read and write heads using *soft reads and writes*. A soft read is a read that is comprised of a weighted sum over all of the memory locations in the external memory. Similarly, a soft write updates every element in the external memory, with the update to a given location being proportional to that location's similarity to a key vector. Crucially, these soft read and write operations are continuous (in contrast to the discrete nature of reads and writes to single locations), allowing the external memory to be differentiated and the entire MANN to be trained end-to-end with stochastic gradient-descent algorithms. Several variants of

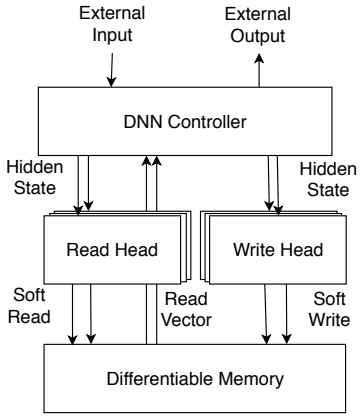

Figure 1: General structure of the Neural Turing Machine (NTM), a Memory-augmented Neural Network (MANN)

MANNs have been proposed, such as the Neural Turing Machine (NTM) [14], Differentiable Neural Computer (DNC) [15], Dynamic Memory Networks [25], and End-to-End Memory Networks [33]. These recent efforts from Google DeepMind, Facebook AI Research, and others have demonstrated the ability of MANNs to solve entirely new classes of problems that are well beyond the capabilities of classical DNNs. For example, DeepMind's Differentiable Neural Computer was trained to efficiently navigate the complex routes that make up the London Underground subway network [15].

The enhanced capabilities of MANNs come at a high computational cost. As mentioned above, the soft nature of the reads and writes results in accesses to *all the memory locations* for each operation in order to keep these operations differentiable. This introduces a serious performance bottleneck, which is exacerbated when dealing with real-world problems requiring thousands to millions of memory locations [15].

Existing neural network accelerators are ill-suited to address this bottleneck for three main reasons. First, existing neural network accelerators are designed primarily for (i) CNNs, which have high FLOPs/Byte ratios and are compute-bound, enabling the use of large arrays of multiply-and-accumulate (MAC) units, or (ii) fully-connected multilayer perceptrons (MLPs) and RNNs, which are memory-bound but whose FLOPs/Byte ratio can be increased considerably by batching inputs and reusing weights across the inputs in a batch.

However, this paradigm of designing for compute-bound networks (*i.e.*, CNNs with high FLOPs/Byte ratios) and using batching to improve the ratios even for memory-bound networks *cannot be applied to MANNs*. The external memory in MANNs represents dynamic state akin to the activations in a traditional DNN and is unique to each input. Therefore, it cannot be shared across a batch, unlike the weights of an MLP or RNN. Thus, an accelerator designed for compute-bound networks, allocating large amounts of die area for compute units, is highly inefficient for memory-bound MANNs. Instead, a MANN accelerator needs to be memory-centric, focusing not on high theoretical throughput but rather on maintaining high utilization of the available compute units through highly banked on-chip memories.

Second, many existing neural network accelerators are also ill-equipped to cater to the heterogeneity of memory access patterns found in MANNs. During inference in DNNs, there is only one access pattern per data structure – for example, a set of activations may be accessed, but not the transpose of those activations. During inference for MANNs, however, the external memory is accessed in both a column-wise as well as a row-wise pattern–that is, both the external memory and the transpose of the external memory are required. Further, the external memory is quite large and can be updated on every time step, making it impractical to efficiently maintain two copies. Therefore, a MANN accelerator must be able to perform efficient transpose operations on-chip.

Finally, all existing neural network accelerators are designed for efficiently executing dot product operations, as matrix-multiply kernels dominate the runtime of current DNNs, including CNNs, MLPs, and RNNs. In MANNs, however, MAC operations are not the dominant operation; non-reductive element-wise addition, subtraction, and multiplication operations make up an equal portion of the relative mix of operations in MANNs. As a result, a MANN accelerator must be designed to perform non-reductive element-wise operations equally as efficiently as dot products.

To address these shortcomings of existing neural network accelerators, we propose `Manna`, an accelerator developed from the ground up to efficiently execute MANNs. `Manna` is a memory-centric design that focuses on maximizing on-chip storage and bandwidth and uses just enough compute to match that bandwidth in order to eschew the die area and power wasted by underutilized compute elements. `Manna` also utilizes a hardware-based transpose mechanism to efficiently perform the row-wise and column-wise memory accesses required. Finally, `Manna` is provisioned with an array of processing tiles, each of which are composed of specialized units (called eMACs) designed to efficiently perform the unique mix of operations present in MANNs.

We also propose a compiler that efficiently maps kernels to `Manna`, exploiting the little available reuse in order to reduce memory transfers and maximize bandwidth utilization.

In summary, we make the following contributions:

- We provide a detailed investigation of the computational characteristics of MANNs, a promising emerging class of DNNs.
- We propose `Manna`, an end-to-end, CMOS-based accelerator that is able to efficiently perform all kernels used in memory-augmented neural networks. We provision `Manna` with a memory-to-compute resource allocation reflective of the low FLOPs/Byte inherent to MANNs.
- We implement a hardware-based transpose mechanism to efficiently execute both vector-matrix and vector-transposed matrix multiplications.
- We provision `Manna` with specialized units (called eMACs), which are designed for the unique mix of operations beyond just MACs that are used in MANNs.
- We develop an ISA and compiler that map MANNs to `Manna` so as to minimize data transfers and maximize the utilization of on-chip bandwidth.
- We develop an architectural simulator and synthesize RTL implementations of key components in order to demonstrate

the benefits of Manna. Our experiments indicate that Manna achieves average speedups of 39x (24x) and average energy improvements of 122x (86x) over an NVIDIA 1080-Ti (2080-Ti) GPU with a Pascal (Turing) architecture.

2 BACKGROUND

In this section, we first provide a brief introduction into the relevant concepts underpinning MANNs. We will then provide a detailed investigation into the computational characteristics of MANNs on CPU and GPU platforms, and highlight opportunities for efficiency improvement.

2.1 Recurrent Neural Networks

Recurrent Neural Networks and their variants such as Long Short-Term Memory networks (LSTMs) [21] or Gated Recurrent Units (GRUs) [10] extend feedforward DNNs by introducing recurrent connections, thereby allowing the network to store dynamic state across iterations or inputs. This introduction of dynamic state has greatly benefited domains such as neural machine translation [2, 38] and speech recognition [13, 19]. However, there exists a fundamental constraint on the use of RNNs for storing dynamic state. The dynamic state is intrinsically coupled to the design and topology of the network. As a result, the amount of information that can be stored in the network cannot be increased without increasing the size of the underlying network, thereby increasing the total number of parameters that must be trained.

2.2 Memory-Augmented Neural Networks

To address this scaling problem, memory-augmented neural networks (MANNs) have been proposed, wherein the dynamic state is explicitly decoupled from the neural network in an external memory. Several variations of MANNs have been proposed [14–16, 25, 33, 37, 39] to tackle a diverse variety of tasks such as question answering, route planning, scene understanding, and language transduction. These MANNs have also been used as building blocks for goal-directed agents in reinforcement learning settings [36].

2.2.1 *Neural Turing Machines.*

While several variants of MANNs exist in the literature, we will focus on DeepMind's Neural Turing Machine (NTM) [14] for our exposition. In this section, we provide a detailed description of NTMs.

As shown in Figure 1, NTMs are composed of three main components: a neural network-based controller, a differentiable external memory, and the read and write heads that control the interaction between the two. To ensure the external memory is differentiable, NTMs use soft read and write operations. These soft operations differ from traditional read and write operations in that the soft operations require access to all the locations in memory. Attention mechanisms are used to determine how the values of all of the different locations are combined (read) or updated (write).

Controller. NTMs employ a standard DNN-based controller that generates both the final output vector as well as the hidden state vector for the read and write heads. This controller network is a major NTM hyper-parameter, and a wide range of DNNs, *viz.*, MLPs of varying depths, RNNs, CNNs, *etc.*, can be utilized. Even with a feedforward network-based controller, NTMs are intrinsically sequential algorithms. At each time step t, the controller network receives as inputs, an external input vector and the read vectors ($\mathbf{r}_h^{t-1}$) from each read head (h) corresponding to the previous time step $t-1$. The controller then produces an output vector that is sampled at the end of execution of the NTM, and a hidden state vector that is used by the read/write heads to interact with the external memory state at the same time step. Throughout this paper, we define $\mathbf{M}^t$ as the differentiable external memory M at time t. $\mathbf{M}^t$ is composed of M_N row vectors, where each row vector $\mathbf{M}^t(i)$ consists of M_M words (dimensions). Thus, $\mathbf{M}^t$ comprises of M_N rows and M_M columns.

Soft Read. A soft read is a weighted summation over all the locations in $\mathbf{M}^t$ at time t, producing one read vector, $\mathbf{r}_h^t$, for each read head h. This can be formulated as a vector-matrix multiplication between the transpose of the weight column vector $\mathbf{w}_h^t$ and $\mathbf{M}^t$ for each read head h, as shown in Equation 1. The column vector w_h^t reflects the contribution of the corresponding rows to the final soft read vector, and is obtained by the attention mechanism described below.

$$\mathbf{r}_h^t = \mathbf{w}_h^{t\,T}\mathbf{M}^t, \forall\ h \in H_r \tag{1}$$

Soft Write. A soft write involves two steps: an *erase* and an *add* operation shown in Equations 2 and 3. In the erase step, an erase row vector $\mathbf{e}_h^t$ produced by the write head h is multiplied with the corresponding scalar from the weight column vector $\mathbf{w}_h^t$. The resulting vector is next subtracted from $\mathbf{1}$ (a vector of length M_M containing all 1s). Finally, the subtracted vector is multiplied in an element-wise fashion with $\mathbf{M}^t(i)$, resulting in a modified external memory $\mathbf{M'}^t$. After erasing, the modified external memory is then updated by adding a similarly weighted add vector $\mathbf{a}_h^t$. The erase and add steps are repeated for each row in $\mathbf{M}$, as well as for each write head h, in order to complete the soft write operation.

$$\mathbf{M'}^t(i) = \mathbf{M}^t(i) \odot [\mathbf{1} - \mathbf{w}_h^t(i) \cdot \mathbf{e}_h^t] \tag{2}$$

$$\mathbf{M}^{t+1}(i) = \mathbf{M'}^t(i) + \mathbf{w}_h^t(i) \cdot \mathbf{a}_h^t, \forall i \in M_N, \forall\ h \in H_w \tag{3}$$

Read/Write Heads. Each read and write head h consists of a weight matrix W_h. In each head, the hidden state from the controller is multiplied by the weight matrix to produce a set of vectors and scalars (*e.g.*, the key vector, erase vector, *etc.*) that are used as inputs to the attention mechanisms present in NTMs.

Attention Mechanism. To determine the weights for soft read and write operations, NTMs use a series of attention mechanisms that consume the output of the read and write heads. First, content-based weighting produces a weight vector $\mathbf{wc}_h^t$ of length M_N for each read and write head h. Each element of the content weight column vector for a given head is based on the similarity between the corresponding row of $\mathbf{M}^t$ and the key row vector $\mathbf{k}_h^t$ emitted by that head (Equation 4).

$$Sim[\mathbf{k}_h^t, \mathbf{M}^t(i)] = \frac{\mathbf{k}_h^t \cdot \mathbf{M}^t(i)}{||\mathbf{k}_h^t|| \cdot ||\mathbf{M}^t(i)||} \tag{4}$$

$$\mathbf{wc}_h^t(i) = \frac{\exp(\beta_h^t Sim[\mathbf{k}_h^t, \mathbf{M}^t(i)])}{\sum_j^{M_N} \exp(\beta_h^t Sim[\mathbf{k}_h^t, \mathbf{M}^t(j)])} \tag{5}$$

After obtaining the similarity for all M_N memory vectors in $\mathbf{M}^t$, the similarities are then amplified (attenuated) by a scalar β_h^t and a softmax function is applied to obtain $\mathbf{wc}_h^t$ (Equation 5). Next, a scalar interpolation gate g_h^t is used to blend the current weighting with the weighting produced at the previous time step of the algorithm (Equation 6). The result of the interpolation is then convolved with a rotation vector $\mathbf{s}_h^t$ (Equation 7). Finally, the result of this convolution is sharpened by a scalar λ_h^t and normalized in order to combat the blurring that may occur as a result of the shifting (Equation 8). This produces the final weight vector $\mathbf{w}_h^t$ corresponding to each read and write head key vector, that are then used in the soft read and write operations.

$$\mathbf{wg}_h^t(i) = g_h^t \mathbf{wc}_h^t(i) + (1 - g_h^t)\mathbf{wc}_h^{t-1}(i) \tag{6}$$

$$\mathbf{ws}_h^t(i) = \sum_j^{M_N - 1} \mathbf{wg}_h^t(j)\mathbf{s}_h^t(i-j) \tag{7}$$

$$\mathbf{w}_h^t(i) = \frac{\mathbf{ws}_h^t(i)^{\lambda_h^t}}{\sum_j^{M_N} \mathbf{ws}_h^t(j)^{\lambda_h^t}} \tag{8}$$

3 NTM COMPUTATIONAL BEHAVIOR

For the following experiments, we use the copy task from the original NTM paper [14] as an illustrative example and execute it on an NVIDIA Turing GPU and an Intel Skylake Xeon CPU.

Memory access characteristics. Table 1 categorizes the various NTM kernels depending on their computational primitives and identifies the number of memory accesses associated with each kernel. There are three distinct groups of kernels: the controller kernel that consists of a classical DNN, the addressing kernels that determine how the external memory is accessed, and the kernels that actually access the external memory. The addressing kernels– content-based weighting, location interpolation, shift weighting, and weight sharpening– involve $O(M_N)$ memory accesses per head, since they are used to create and modify the weight column vectors $\mathbf{w}_h$. These kernels also have a relatively low FLOPs/Byte ratio (2 or 3).

On the other hand, the access kernels– key similarity, soft write, and soft read– are extremely memory-intensive kernels with many more memory accesses ($O(M_N \cdot M_M)$) since each of these kernels involve accessing every single element in the differentiable memory at least once for each of the read (write) heads. Furthermore, these kernels exhibit very little reuse, accessing the external memory in a streaming fashion. Note that the only opportunity for reuse in such kernels is across multiple heads, which is usually less than a factor of five. This combination of large memory footprint and extremely low FLOPs/Byte ratio cannot be mitigated through the standard techniques for DNNs, *viz.*, compression and batching. Since the differentiable memory content is dynamic, the memory footprint cannot be reduced by using techniques such as Deep Compression [18]. Moreover, the external memory is unique to each input sequence to the NTM, and therefore it cannot be shared across input batches similar to RNNs and MLPs. Thus, accelerating NTMs requires designing hardware specifically for low FLOPs/Byte ratios, instead of designing for high FLOPs/Byte ratios and using other techniques to achieve those high ratios. It is also important to note that while both key similarity and soft reads involve vector-matrix multiplications, the external memory access patterns for the constituent dot products are fundamentally different. Specifically, dot products for key similarity are performed across the rows, while dot products for soft read are reduced across the columns. Therefore, it is also important to enable efficient dot products along both rows and columns.

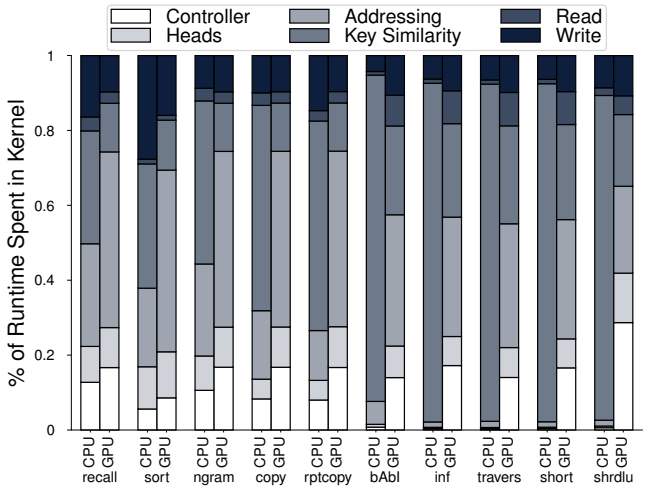

Figure 2: Runtime breakdown of different NTM kernels

Kernel breakdown. Figure 2 presents a breakdown of the runtime during inference spent in different kernels across a suite of ten benchmarks. These benchmarks use differentiable memories that have been scaled to be larger, but still fit in a reasonably-sized on-chip memory (40MiB). For clarity, the read and write head kernels have been grouped together, as have all of the addressing kernels. As shown in the figure, the non-controller kernels, *viz.*, heads, addressing, key similarity, soft reads, and soft writes, dominate the runtime, making up roughly 80% of the total. This is particularly true for benchmarks that require higher differentiable memory capacity (`bAbI`, `inference`, `traversal`, `shortest`, and `shrdlu`). This behavior is expected since MANNs decouple the dynamic state from the network itself, allowing the amount of dynamic state stored to grow dramatically without significantly increasing the size of the controller network. Hence, efficiently realizing these kernels is key to high performance in MANNs.

Interestingly, the kernels that dominate the total runtime within these runtime-intensive kernels vary depending on the platform. For the CPU platforms, the dominant kernels, particularly at larger differentiable memory sizes, are key similarity, soft write, and soft read. This is because these kernels are extremely memory-intensive compared to the remaining kernels as shown in Table 1. However, on GPUs, the vector-only addressing kernels represent an unexpectedly large portion of the runtime, comparable to the memory-intensive access kernels. This is because the memory-heavy access kernels (*e.g.*, key similarity) are large enough to fully utilize the GPU. The addressing kernels, on the other hand, are much smaller, resulting in so-called "narrow tasks" that exhibit poor GPU utilization [40] and thus poor performance due to the GPU kernel

Table 1: Summary of kernels in Neural Turing Machine

Kernel Name	Key Primitive	Mem. Accesses	FLOPs/Byte	Reduction
Key Similarity	Vector-Matrix Mul.	$O(\mathbf{M_N} \cdot \mathbf{M_M} \cdot (\mathbf{H_r} + \mathbf{H_w}))$	$H_w + H_r$	Row-wise
Content-based Weighting	Normalization	$O(M_N \cdot (H_r + H_w))$	3	-
Location Interpolation	El.-wise Mul/Add/Sub	$O(M_N \cdot (H_r + H_w))$	2	-
Shift Weighting	Circular Conv.	$O(M_N \cdot (H_r + H_w))$	S	-
Weight Sharpening	Normalization	$O(M_N \cdot (H_r + H_w))$	3	-
Soft Read	Vector-Matrix Mul.	$O(\mathbf{M_N} \cdot \mathbf{M_M} \cdot \mathbf{H_r})$	H_r	Column-wise
Soft Write	El.-wise Mul/Add/Sub	$O(\mathbf{M_N} \cdot \mathbf{M_M} \cdot \mathbf{H_w})$	H_w	-

call overheads. CPUs, on the other hand, have fewer cores and thus can maximize utilization even for limited parallelism in the addressing kernels. In order to efficiently perform all of the NTM kernels, an accelerator therefore must be able to efficiently execute both the large, extremely parallel matrix operations as well as the comparatively smaller vector operations.

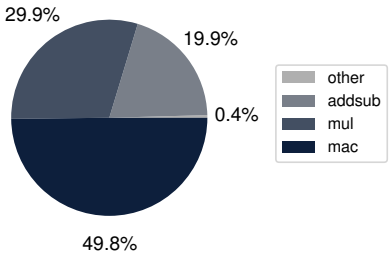

Figure 3: Relative mix of operations in runtime-intensive NTM kernels

Operation breakdown. Finally, we analyze the relative mix of operations that constitute the runtime-intensive kernels by analytically modeling the number of operations of each type that would be executed when running the copy benchmark. As shown in Figure 3, we found that, unlike traditional DNN kernels, the non-controller kernels are equally dominated (49.8% each) by fused multiply-and-accumulate operations and element-wise vector operations, specifically multiplication, addition, and subtraction. Thus, an accelerator for MANNs cannot solely emphasize MAC or dot-product performance but rather a high throughput is required on a wider variety of operations.

In summary, we make several observations for desiderata in a MANN accelerator based on our investigation. First, as demonstrated by Figure 2, the MANN-specific kernels dominate the runtime and therefore a MANN-specific accelerator is warranted. In particular, such an accelerator should: (i) balance the on-chip compute and memory resources for extremely low FLOPs/Byte workloads to maintain high utilization; (ii) provide support for efficient dot products across both rows and columns of the external memory; and (iii) accommodate the higher use of non-MAC compute elements.

4 MANNA ARCHITECTURE

4.1 Overview

Manna is a memory-centric, highly parallel CMOS-based architecture designed explicitly for memory-augmented neural networks. The building blocks of Manna are *DiffMem tiles* that execute the MANN-specific kernels (*i.e.*, read and write head operations, key similarity, addressing, soft read, and soft write) which control the external differentiable memory (hence, the name DiffMem tiles). The entire external memory is partitioned and distributed across the DiffMem tiles. The DiffMem tiles are designed to cater to the low FLOPS/Byte ratio observed in MANN kernels. As a result, the majority of die area of the DiffMem Tiles (and Manna in general) is dedicated to highly banked on-chip memories. The DiffMem tiles are then provisioned with just enough processing elements to match that on-chip memory bandwidth. We also specialize Manna's processing elements to be tailored to the unique mix of operations found in MANN kernels, which exhibit a higher share of non-MAC element-wise operations than the traditional MAC-centric DNN workloads. Finally, we provision Manna with specialized hardware to facilitate efficient transpose operations to accommodate the heterogeneity in memory access patterns seen in MANNs.

Manna also includes *Controller tiles*, which are used to execute the DNN-based controler. We tailor the interconnect topology between tiles to the specific communication patterns that manifest in MANNs. Figure 4 shows the organization of Manna, whose components are described in the following subsections.

4.2 DiffMem Tiles

The DiffMem tiles are used to perform the MANN-specific kernels: read (write) heads, key similarity, addressing, soft reads, and soft writes. Each tile is provisioned with element-wise or multiply-and-accumulate units (eMACs). These eMACs are used to perform element-wise addition, element-wise subtraction, element-wise multiplication, and fused multiply-and-accumulate. Each eMAC also consists of a small register file (RF) that is used to temporarily store inputs and intermediate outputs. The eMACs are connected to a double-buffered scratchpad (`Matrix-Scratchpad`) such that each word of the memory's output is directly connected to the corresponding eMAC unit. The eMACs are also connected to another double-buffered scratchpad (`Vector-Scratchpad`). The `Vector-Scratchpad` either broadcasts a shared value to all eMAC units or unicasts each word of the scratchpad's output to the corresponding eMAC unit. Additionally, there are lateral connections between neighboring eMACs, which enable efficient transpose operations. The `Matrix-Scratchpad` is connected to a larger matrix buffer (`Matrix-Buffer`) through a direct memory access for transpose (`DMAT`) unit that manages data transfers in both regular and transposed form, between the `Matrix-Scratchpad` and the `Matrix-Buffer`. The `Vector-Scratchpad` is similarly connected

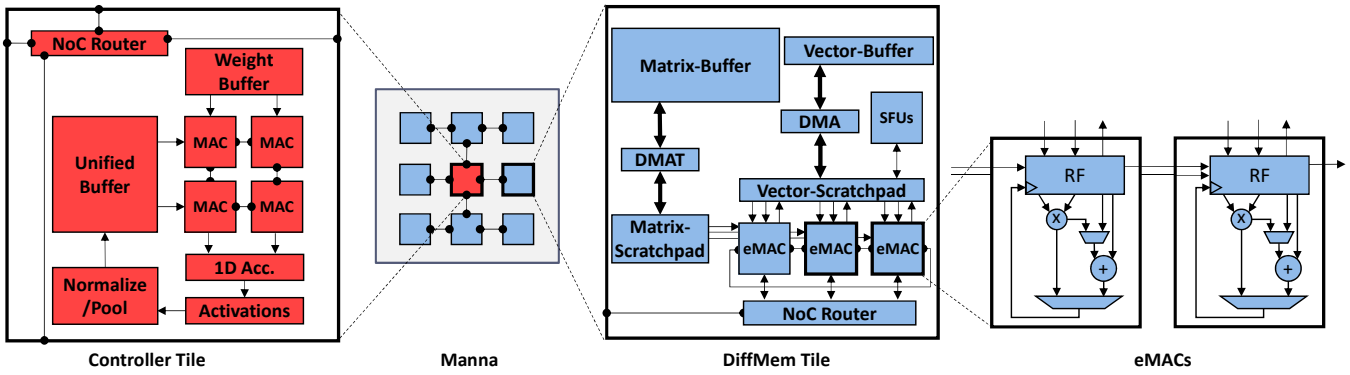

Figure 4: Overview of Manna

to a larger vector buffer (`Vector-Buffer`). However, since these memory elements store the vector data structures which do not require any hardware transpose support, a regular direct memory access (DMA) unit is used. The `Vector-Scratchpad` is also interfaced to a set of Special Function Units (SFUs). The SFUs contain a scalar power function unit, an accumulator, and the logic needed to support various activation functions such as sigmoid, ReLu, *etc.* Each DiffMem tile also has a small instruction memory and control unit used to execute the tile's program. Finally, it also includes an NoC router to perform reduce and broadcast operations between tiles.

4.3 Controller Tiles

The Controller tile is used to execute the DNN controller. Since a wide variety of DNN topologies may be used for the controller, the Controller tile utilizes a more traditional, systolic array-based DNN accelerator architecture. A controller tile consists of a two-dimensional systolic array-based matrix multiplication unit. The matrix multiplication unit is connected to a `Weight Buffer` that supplies the weights, and a `Unified Buffer` that provides the activations. These activations are then accumulated in a one dimensional `Accumulation Unit`, the output of which is connected to the activation units (*e.g.*, ReLu), followed by the normalization and pooling units. The outputs are finally stored in the `Unified Buffer` to be used as inputs to the next layer.

4.4 Implementing MANNs on Manna

Distributing MANN kernels. A key feature of `Manna` is that the entire external memory is partitioned and distributed across the DiffMem tiles. The external memory is divided into independent sections by selecting the number of tiles ($NDistrib$) to distribute the rows across, which results in each tile receiving $n = \frac{M_N}{NDistrib}$ rows of $\mathbf{M}$. The number of tiles to distribute the columns across is then $MDistrib = \frac{NumTiles}{NDistrib}$, resulting in each tile receiving $m = \frac{M_M}{MDistrib}$ columns of $\mathbf{M}$. Therefore, each physical tile is responsible for a logical (n,m) tile of the external memory. The tile is also responsible for the corresponding slices of the various vectors (*e.g.*, key). Figure 5 illustrates this distribution.

The choice of $NDistrib$ and $MDistrib$ affects the performance of the `Manna` accelerator, since it impacts the communication patterns

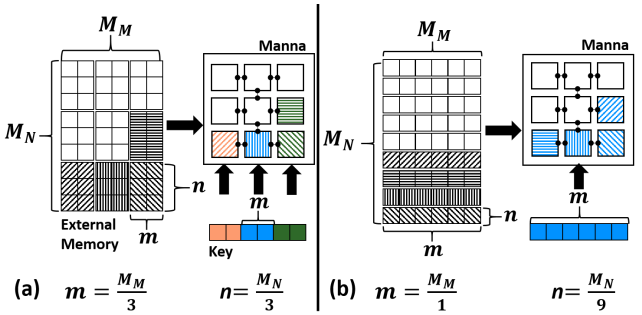

Figure 5: Each DiffMem tile receives a unique, independent section of the external memory. If $MDistrib \neq NumTiles$, as in (a) where $MDistrib = 3$, there will be $MDistrib$ sets of tiles in which each tile in the set receive copies of the same slice of the key vector. If $MDistrib = 1$, as in (b), every tile will receive a copy of the whole key vector

and degree of parallelization that can be achieved in executing the MANN kernels. For example, consider the content weighting kernel which is a softmax operation. In order to obtain the content weights, the content similarities must be normalized. To achieve this, the total sum of the similarities must first be reduced across the $NDistrib$ tiles that partition the weight vector. The final reduced result must then be broadcast back to those tiles so that the final result can be used in a map operation (as described below) to normalize each element in the weight vector. Notice that Equation 5 that describes the content weighting– as with all of the other addressing kernels– is only dependent on M_N. Therefore, if $MDistrib > 1$, there will be a subset of tiles for which no useful computation can be performed. This is due to the fact multiple tiles would have the same slice of the weight vector. To maximize the amount of work that can be done in parallel, thereby maintaining high utilization of the available compute resources, we use the simple heuristic that we will force $MDistrib = 1$ and therefore $NDistrib = NumTiles$.

Realizing Map Operations. Many of the operations that make up the addressing kernels can be envisioned as map operations applied to a given vector. For example, as part of the content weighting kernel, every element in the $\mathbf{w}$ vector is multiplied by the same scalar β. These kernels are executed as follows. First, the value

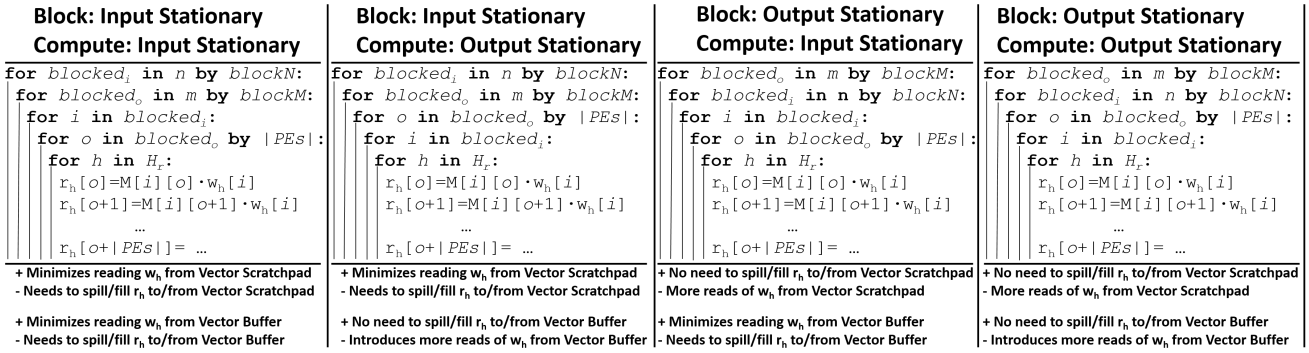

Figure 6: Overview of possible loop orderings for soft read

that is to be mapped to the vector (*e.g.*, the accumulated value for normalizing) is broadcast to the eMACs and stored in their RFs. Subsequently, for each element in the vector that the tile is processing, the values are unicast to the eMAC units such that a different element of the vector is computed in each eMAC unit. In this way, all eMACs in each tile can be fully utilized for executing map operations in parallel, with minimal overhead.

Realizing Vector-Matrix Multiplication. As discussed above, each tile executes a portion of the vector-matrix multiplication in parallel. We store the tile's entire portion of the differentiable memory matrix in a large, highly-banked on-chip memory, the `Matrix-Buffer`. Vectors are stored in a comparatively much smaller on-chip memory, the `Vector-Buffer`. We further break the smaller vector-matrix multiplications into blocks, such that each block consists of a vector-multiplication between *blockN* elements of the vector and (*blockN* x *blockM*) elements of the matrix, resulting in an output partial sum vector of length *blockM*. The portion of the vector and matrix needed for the current block is then brought into the `Vector-Scratchpad` and the `Matrix-Scratchpad`, respectively. To compute the vector-matrix multiplications of a given block, data is fed from the two scratchpads into the eMAC units such that one element from the `Vector-Scratchpad` is broadcast to all of the eMAC units. Unlike the case of map operations, for vector-matrix multiplication, each eMAC unit receives a unique memory word from the `Matrix-Scratchpad`, *i.e.*, vector-matrix multiplication is realized by multiplying one element of the input vector with multiple elements from the matrix to compute a set of partial sums. The eMACs then read the next element from the vector, continuing to hold the partial sums resident in the RF. This strategy is referred to as *output stationary*, since a subset of the output vector of the vector-matrix multiplication is completely finished before moving on to the next subset. This strategy does not require storage for partial sums in the `Vector-Scratchpad`, but may result in rereading the same element from the input vector multiple times if the number of eMAC units is lower than the block size. Alternatively, the element from the input vector can be kept resident and the partial sums spilled to the `Vector-Scratchpad`. This strategy is referred to as *input stationary* and results in minimizing the number of reads of the input vector, but introduces spills (fills) to (from) the `Vector-Scratchpad` since the partial sums are not kept stationary. Note that, while the choice between input stationary and output stationary impacts the number of accesses between the `Vector-Scratchpad` and the eMACs, the total number of transfers between the `Matrix-Scratchpad` and the eMACs remains the same in both cases. In order to maximize reuse (*e.g.*, in the case of multiple heads), the RF is provisioned with enough capacity to allow the eMACs to store the temporary value (input or output) from multiple heads, allowing the eMAC to reuse the element from the `Matrix-Scratchpad` for each head without needing to refetch the input (output) element when moving to the next element from the `Matrix-Scratchpad`.

This loop, which computes the vector-matrix multiplication for a given block, is referred to as the compute loop. Once the multiplication is finished for a given block, we must determine the ordering of the block loop– that is, which block of the vector-multiplication will be executed next. Similar to the compute loop, this can be done in an output-stationary or input-stationary manner. The choice between output-stationary and input stationary determines the number of data transfers between the `Vector-Scratchpad` and the `Vector-Buffer`. The number of transfers between the `Matrix-Scratchpad` and the `Matrix-Buffer` is constant regardless, since each element in the `Matrix-Buffer` is brought into the `Matrix-Scratchpad` exactly once. Figure 6 summarizes the algorithms that result from these different loop orderings and their impact on data transfers.

After each tile has completed computing its portion of the larger vector-matrix multiplication, these partial sums must be reduced across the *NDistrib* tiles in order to obtain the final result (*e.g.*, the final read vector).

Realizing Vector-Transposed Matrix Multiplication. As discussed, the differentiable memory is accessed in conflicting directions. Unfortunately, due to the size of the external memory, as well as the frequency of updates to the memory, it is infeasible to keep an updated copy of both the external memory and its transpose. This means that it is impossible to map the external memory to the physical `Matrix-Buffer` in such as manner as to result in efficient memory accesses for both soft read and key similarity, as illustrated in Figure 7. Therefore, to maintain high utilization of the on-chip bandwidth, we propose an efficient, hardware-based transpose mechanism, outlined in Figure 8.

The proposed transpose mechanism draws inspiration from well-known software techniques in the GPGPU community [31]. This

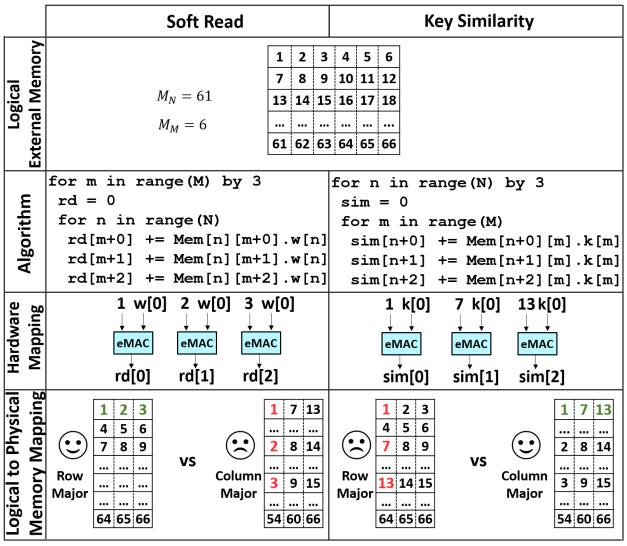

Figure 7: Logical-to-physical mapping of the differentiable memory always results in good memory accesses for one kernel, but poor memory accesses for the other

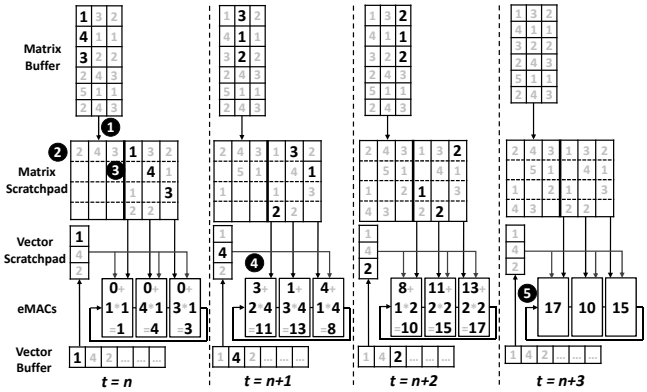

Figure 8: Padding the transfer from buffer to scratchpad and moving the partial sums laterally between eMACs allows for conflict-free direct access to banks

technique relies on splitting the vector-matrix multiplication into blocks, as outlined above. We first fill the `Matrix-Scratchpad` with the entire working set for the current block using efficient reads that utilize the full bandwidth between the `Matrix-Scratchpad` and the `Matrix-Buffer` (1). Once the `Matrix-Scratchpad` is efficiently filled with a block, we begin computing on that block. Notice that since the `Matrix-Scratchpad` is double buffered, this sequence can be pipelined, filling the other half of the scratchpad while the first half is being consumed (2).

This approach fully utilizes the bandwidth between the `Matrix-Buffer` and the `Matrix-Scratchpad`, but there still exists a key inefficiency– there would be many banks conflicts when the eMACs try to read from the `Matrix-Scratchpad` since we have not done anything to fundamentally change the layout of the data yet. To avoid these conflicts, the DMAT unit augments the memory transfers between the `Matrix-Buffer` and the `Matrix-Scratchpad` such that a space is padded between every memory transfer (3). Doing so offsets the elements in the `Matrix-Scratchpad` such that all elements to be read from the scratchpad at a given time are located in different banks, eliminating bank conflicts.

However, now the eMACs would require a full crossbar structure to access the `Matrix-Scratchpad`, instead of direct connections to each bank as before. This is undesirable, as the crossbar will be expensive in terms of chip area and we want to conserve that area for enabling more on-chip bandwidth. In order to avoid requiring this full crossbar, we utilize lateral connections between the eMACs that forward the partial product from one eMAC to its neighbor (4). This way each eMAC can still be directly connected, with the partial sums shifting to keep the proper alignment. When the product is complete, an additional shift is required to realign the partial sums to their original location (5). However, this additional shift can be pipelined with the beginning of the next set of elements.

NoC Design. Given that we have set $MDistrib = 1$ and $NDistrib = NumberofTiles$, only two inter-tile communication patterns are required: reducing across all of the tiles (*e.g.*, to produce the final read vector) or broadcasting to all of the tile (*e.g.*, the output from the controller). Since these are the only patterns required, we implement a simple, H-tree based NoC with a fixed routing strategy, in turn simplifying the design and requiring only $lg(NumTiles)$ communication steps to complete a given reduction (broadcast). Since the output of the controller network must be broadcast to all of the tiles and the controller network must receive the final soft read vectors r_h, the controller tile is chosen as the root node for the H-tree network.

5 ISA & SOFTWARE

Given the abundance of literature on ISAs for DNN accelerators and on mapping arbitrary DNNs to target accelerators, here we focus solely on the Diff Mem Tiles for brevity.

5.1 Execution Model and ISA

To provide the flexibility to perform a wide variety of MANNs, `Manna` provides an instruction set architecture (ISA) that can be used to implement various proposed MANNs. These instructions are grouped into three categories:

Control. The MANN kernels consist entirely of data flows that are easy to generate procedurally. Therefore, we use a set of `control` instructions in order to provide `Manna` with the necessary information to generate the memory accesses needed for a given kernel. `loop` and `end-loop` are used to define the block loop, while `addr-gen` defines the compute loop. The end of the compute loop is inferred from the end of `compute` instructions.

Compute. To enable the execution of a wide variety of MANNs, the key computational primitives of the MANN kernels such as element-wise multiply, vector-matrix multiplication, softmax, *etc.* are expressed as a set of course-grained `compute` instructions. These instructions perform their operations based on the previous `control` instructions.

Communication. Given the execution model outlined in Section 4, the communication patterns used in `Manna` are extremely well defined and consist solely of reduce and broadcast operations that involve all tiles. Therefore, we provide only two communication

instructions– `reduce` and `broadcast`. Since the communication patterns are fixed, these instructions double as synchronization primitives–*i.e.*, when a tile executes a `reduce` operation, the tile will wait until it receives the appropriate message from its neighbor, thereby acting as a fence instruction.

To minimize programmer burden, we also provide a compiler that maps a MANN to `Manna`, which we detail below.

5.2 Compiler

The MANN kernel compiler takes as inputs a description of the target MANN (*e.g.*, number of read/write heads, size of external memory) and a microarchitectural description of the `Manna` accelerator (*e.g.*, size of scratchpads, number of tiles). Using this information, the compiler then generates code for `Manna` in two phases– mapping and code generation.

5.2.1 *Mapping.*

First, a given MANN network is mapped to `Manna` by setting the tile size for loop tiling and choosing the ordering of the loops based on the set tile size.

Loop Blocking. As discussed above, the key similarity, soft read, and soft write kernels are executed in a blocked fashion. A key parameter for optimization, then, is the size of the tiles for loop blocking. Since the main goal of `Manna` is to maximize the utilization of on-chip memory bandwidth, we use the following algorithm for setting the loop tile size for each kernel individually. First, we set the M dimension of the block (*blockM*) to match the Matrix Buffer's memory width in order to maximize bandwidth utilization. This is also required by the proposed transpose mechanism used to ensure bank-conflict free execution. We then maximize the N dimension (*blockN*), while still ensuring that the blocked tile can fit in the `Matrix-Scratchpad`. We also account for the additional storage overhead introduced as a result of padding when computing this dimension.

Loop Ordering. As discussed in Section 4, for matrix-vector operations, there are two sets of pairs of loops that must be ordered. The order of the outer, block loop determines the number of accesses to the scratchpad, while the order of the inner, compute loop determines the number of accesses to the buffers. Figure 6 summarizes these loop orderings along with the advantages and disadvantages of each ordering. In order to select between the four options, the compiler firsts sets the ordering of the block loop, as the accesses to the scratchpad are more expensive and are therefore of a higher priority to optimize. The compiler uses a set of equations to analytically model the cost of the output stationary and input stationary orderings of the block loop and selects the ordering with the lowest cost for each kernel.

Upon determining the outer loop ordering, the compiler also chooses the best ordering of the compute loop for each kernel, choosing the ordering that results in the smallest number of accesses to the buffers.

5.2.2 *Code Generation.*

The code generation phase creates a program for each individual tile; the synchronization between tiles is handled via the communication instructions described above. To generate these programs, the compiler utilizes a library of hand-coded assembly routines for each of the MANN kernels. These routines are parameterized based on both the results of the mapping phase and the microarchitectural parameters.

6 EXPERIMENTAL SETUP

6.1 Methodology

Simulation infrastructure. We developed a detailed, cycle-level architectural simulator in order to evaluate the execution of the benchmarks on `Manna`. The simulator models all necessary events that occur in an execution cycle, including compute, memory, and NoC transactions. To simulate the Controller tile, we used the performance simulator from [32], which has been verified against an RTL implementation. To estimate power, we implemented the logic components of `Manna` in RTL, synthesized them using the 15 `nm` Nangate Open Cell library, and evaluated their power using Synopsys Design Compiler. For the on-chip SRAM memories, we obtained power estimates using CACTI-P [26]. The power consumed by each component was then incorporated into the cycle-level simulators in order to estimate energy consumption.

Benchmarks. Table 2 shows the list of 10 benchmarks modeled on the tasks presented in the NTM and DNC papers [14, 15]. These NTM benchmarks use a diverse range of sizes and aspect ratios for the differential memory, with varying number of read and write heads. `copy`, `rptcopy`, `recall`, `ngram`, and `sort` are small, algorithmic examples. `bAbI` is a question answering task testing logical reasoning. `travers`, `short`, and `inf` are graph tasks that test the MANN's ability to generate directions, find shortest paths, and infer relationships from structured information, respectively. Finally, `shrdlu` tests a network's ability to understand natural language through the form of dialogues about the state of a synthetic block world. These benchmarks have been scaled up from the original works in order to reflect the size of the external memory needed for real-world applications, as projected in the DNC paper [15].

`Manna` Configuration. We evaluated a `Manna` implementation with sixteen DiffMem tiles and one Controller tile. Each DiffMem tile is provisioned with: 32 eMAC units, a 2 MiB Matrix-Buffer; a 16KiB Matrix-Scratchpad; a 32KiB Vector Buffer, and a 4KiB Vector Scratchpad. The Controller tile is provisioned with an 8x8 matrix-multiply unit, and 5 MiB of on-chip storage for the unified and weight buffers. All of the compute units utilize full FP32 precision, in keeping with current MANN software implementations. The entire chip is clocked at 500 `MHz` and provides a total of 1.2TB/s of effective bandwidth for accessing the differential memory.

Comparison with GPUs We compare against two GPU platforms: a Pascal GTX 1080-Ti (`1080-Ti`) and a Turing RTX 2080-Ti (`2080-Ti`). The `1080-Ti` has a total of 11.9 MiB of on-chip memory across the register file, L1, shared memory, and L2, with 484 GB/s memory bandwidth, whereas the `2080-Ti` has a total of 29.5 MiB with a 616 GB/s bandwidth. Both platforms executed a Pytorch 1.0 implementation that utilized the highly optimized cuDNN library. These platforms are summarized in Table 3.

7 EXPERIMENTAL RESULTS

In this section, we present the results of our experiments evaluating the benefits of `Manna`.

Table 2: Summary of benchmarks.

Benchmark	Differentiable Memory Dimension	Controller Dimension	No. of Read Heads	No. of Write Heads
copy (copy)	1024x256	1x100	1	1
repeat-copy (rptcopy)	512x512	1x100	1	1
recall (recall)	1024x64	1x100	1	1
dynamic n-grams (ngrams)	1024x128	1x100	1	1
priority sort (sort)	512x128	2x100	1	4
bAbI (bAbI)	4096x1024	1x256	4	1
shortest path (short)	3648x1400	2x256	5	1
graph traversal (travers)	5056x1000	3x256	5	1
graph inference (inf)	3584x1400	3x256	5	1
mini-shrdlu (shrdlu)	1280x4000	2x256	3	1

Table 3: Summary of platforms.

Platform	Area (mm^2)	Technology Node (nm)	Frequency (MHz)	TDP (W)	On-Chip Memory (MiB)	Bandwidth (GB/s)
Pascal GTX 1080-Ti	470	16	1480	250	11.9	484
Turing RTX 2080-Ti	750	12	1500	250	29.5	616
Manna	40	15	500	16	38	N/A

7.1 Inference Performance

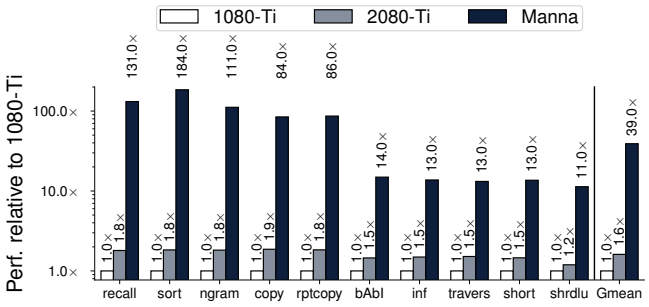

Figure 9: Inference performance

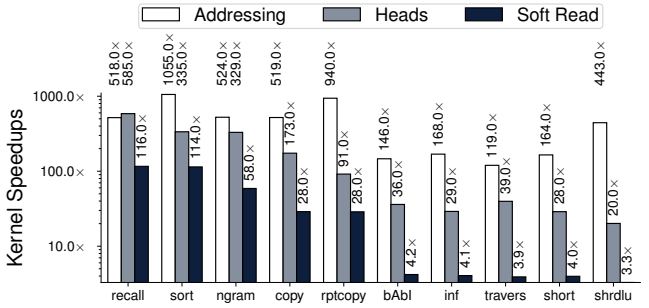

Figure 10: Kernel-specific inference performance

Figure 9 compares the performance of Manna with the GPU implementations, when executing the inference phase with no batching for 10 NTM benchmarks (ordered by size of the external memory). Overall, Manna achieves 11x - 184x speedup (average 39x) over the 1080-Ti. The average speedup achieved by Manna over the 2080-Ti is 24x. There is a substantial difference in speed up for the small benchmarks (recall-rptcopy) as compared to the larger benchmarks (bAbI-shrdlu). To illustrate where the substantial benefits come from, as well as to show why there is such a substantial difference between the small and large benchmarks, we investigate the kernel-specific performance improvements for the addressing kernels, the read and write heads, and soft reads in Figure 10.

Manna demonstrates significantly higher improvement in performance on the addressing kernels because Manna is able to fully parallelize these kernels across the available processing elements, while the GPU is severely underutilized on these kernels. Unlike the addressing kernels, the soft read kernel involves a vector-matrix multiplication against the external differentiable memory. Thus, at smaller memory sizes, Manna exhibits a significant speedup over the GPU platforms, since there is still not enough compute to fully utilize the GPU and mitigate the kernel call overheads. However, as the external memory size increases, the amount of parallelism that can be exposed to the GPU grows rapidly. For the largest benchmarks, Manna's speedup over the GPUs saturates at around 3x, at which point the GPUs are fully utilizing all of their cores (Streaming Multiprocessors) and compute units. We observe a similar trend for the key similarity and soft write kernels. Finally, the head kernels represent a middle ground between these two extremes; the read and write head kernels also involve a vector-matrix multiply, but with a much smaller matrix than the differentiable memory. Correspondingly, Manna's performance improvement is between the two extremes.

7.2 Energy Efficiency

Since MANNs are sequentially evaluated across time steps, to quantify the energy efficiency of the system we consider how many time steps each platform could compute for fixed amount of energy, *i.e.*, steps/J. Based on this measure of energy efficiency, Manna shows substantial improvements of 58x-301x (122x on average) over the baseline GPU platform, as shown in Figure 11. A majority of this energy efficiency is derived from the increase in performance demonstrated in Figure 9, reducing the amount of time the

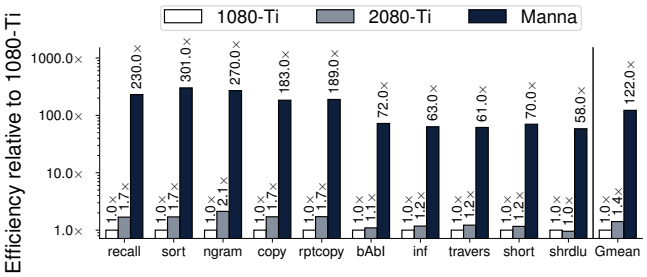

Figure 11: Energy efficiency compared to GPU baselines

system needs to be active to compute a time step. Moreover, Manna consumes an order of magnitude lower power than GPUs, which further maximizes its energy efficiency.

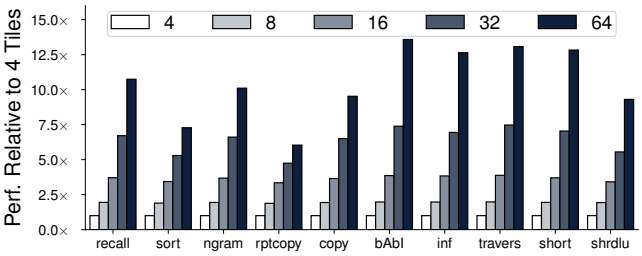

Figure 12: Manna performance trends with strong scaling

7.3 Manna Scaling

Strong scaling. First, we perform a strong scaling analysis in which we compare the performance of a small, four tile baseline configuration of Manna to designs having more tiles. The results of this analysis are shown in Figure 12. Although Manna is able to scale quite well for many of the larger benchmarks, there are some limitations to the strong scaling of Manna. First, by choosing to distribute the external memory across the tiles such that $MDistrib = 1$, we see diminishing returns for smaller N_N, as in the case of smaller benchmarks, or for instances were N_M is similar in size (or greater) than N_N. But even for the large benchmarks where $N_N >> N_M$, we begin to see diminishing returns as the scaling factor increases because Manna becomes limited by the serial accesses to the SFUs in each tile. Therefore, the speedups achievable through strong scaling are limited without (i) also allowing parallelization across M_M, and (ii) increasing the number of SFUs accordingly.

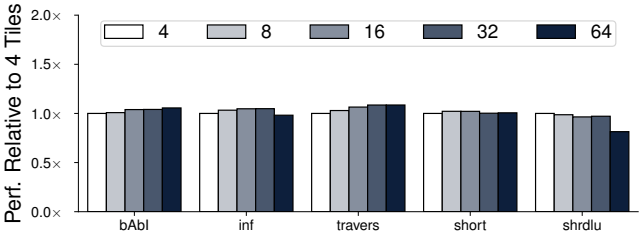

Figure 13: Manna performance trends with weak scaling

Weak scaling. Secondly, we perform a weak scaling analysis. In this analysis, unlike the strong scaling analysis, as we grow the number of DiffMem tiles in Manna, we correspondingly grow the external memory size of the benchmarks used, while maintaining the aspect ratio. So, if we analyze a system with sixteen tiles, then both dimensions of the external memories used in the benchmarks are doubled as compared to those used for the four tile system, resulting in a problem size that is four times larger. As seen in Figure 13, Manna exhibits near-ideal weak scaling, exhibiting very little variability as the number of tiles and problem size are both changed. This is due to Manna's ability to exploit the embarrassingly parallel nature of many of the MANN kernels, meaning the amount of inter-tile communication required is trivial compared to the amount of memory accesses and compute that each tile performs between communications.

Scaling the Differentiable Memory. Finally, we discuss scaling the differentiable memory to be larger than the capacity of a single Manna chip. In this case, multiple Manna chips can be used in a cluster, with the state distributed across them. This scaling method increases the parallelism and compute available proportionally with the capacity of the differential memory. However, in scenarios where scaling out is prohibitive due to cost or space constraints, a High Bandwidth Memory (HBM) can be introduced in the memory hierarchy of a single Manna chip without significant degradation in performance. To understand this further, consider the worst-case scenario in which there is no re-use of the differentiable memory. Each tile has 32 eMACs, each of which requires 4 bytes of differentiable memory elements, for a total of 128 bytes each cycle. An HBM2 module can provide 256GB/s of bandwidth. Since Manna has a 500MHz clock, this means that Manna can receive 512 bytes per clock cycle from HBM2, enough to feed four tiles. Thus, with 4 such memory modules, the 16-tile Manna baseline can have enough bandwidth to cater to all its processing elements. However, adding DRAM is not without its drawbacks. Each HBM2 controller will add $\sim 35mm^2$ [1] area, leading to an increase in total chip area from $40mm^2$ to $180mm^2$. At a 25W power envelope for each HBM2 module, the TDP of Manna will increase to 116W. This design would result in an average 17x improvement in energy efficiency for Manna over the baseline, down from the 122x improvement for the SRAM-only design.

7.4 Ablation

We next compare Manna to three accelerator variants in order to illustrate how the various architectural features of Manna contribute to its performance. The first variant, MemHeavy, is a design optimized for low FLOPs/byte that dedicates the vast majority of the die area to large, highly-banked on-chip memories. However, it does not provide hardware support for transpose or element-wise operations. The second variant builds on the first variant by adding support for transpose (MemHeavy-Transpose), while the third variant replaces the traditional MAC units with the proposed eMAC units (MemHeavy-eMAC). As shown in Figure 14, Manna achieves 2x-4x (3.3x average) performance improvement over the MemHeavy design. Similarly, it achieves 2.3x, and 1.8x average speedup over the designs with only hardware-assisted transpose and eMACs, respectively.

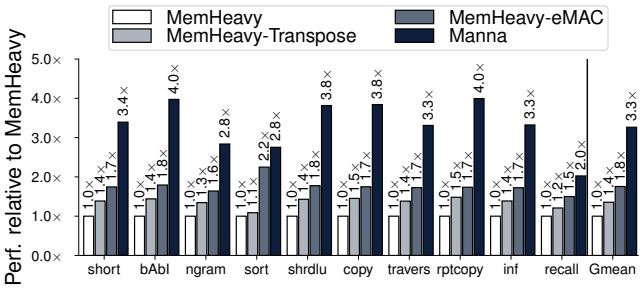

Figure 14: Impact of Manna's architectural features.

8 DISCUSSION AND RELATED WORK

DNN Accelerators. As discussed in Section 1, existing proposed neural network accelerators are insufficient for efficiently executing MANNs for three reasons. First, many DNN accelerators have been designed for high FLOPs/Byte workloads (*i.e.*, CNNs) and attempt to mitigate performance loss in low FLOPs/Byte workloads by utilizing batching [8, 9, 23, 34]. Even with batching, many of these accelerators observe large drops in utilization for low FLOPs/Byte workloads. This is exacerbated even further in MANNs, since MANNs are extremely low FLOPs/Byte workloads and batching cannot be used to increase this ratio. While some RNN-specific accelerators, which are designed specifically for a lower FLOPs/Byte workload, achieve better utilization, these accelerators also rely on techniques such as as compression [17, 35]. However, the memory bottleneck in MANNs– the differentiable external memory– is dynamic and cannot be statically compressed. Second, many accelerators [3, 4, 11, 24, 27, 28] are optimized for a fixed, single data access pattern. This is sufficient when there is only one access pattern for each data structure, as in current DNN inference. However, without sufficient on-chip storage to enable efficient transpose operations, performance suffers when there are both column-wise and row-wise accesses to the same data structure, as in MANNs. Finally, DNN accelerators [5–7, 12] are designed for dominant kernels that are composed of MAC operations. Since other operations, such as non-reductive element-wise addition, subtraction, and multiplication, are relatively infrequent, the throughput of these operations is much lower than that of MACs. However, since MANNs have a nearly equal share of non-MAC operations (*i.e.* element wise operations) and MAC operations, the reduced throughput for non-MAC operations leads to poor performance.

MANN Accelerators.Two recent efforts have addressed acceleration of MANNs [22, 29]. The key difference between Manna and these proposals is that they are fixed-function architectures that are specialized for a specific class of MANNs, end-to-end memory networks (MemNets), whereas Manna has sufficient programmability to realize a broad class of MANNs (*e.g.*, NTMs and DNCs from Google Deepmind), which is important given the evolving nature of MANNs. Moreover, since MemNets do not require soft writes, these accelerators are not designed to support non-MAC operations that constitute a key kernel in soft writes. Consequently, these architectures witness reduced throughput for such element-wise operations, resulting in inefficient soft writes. Besides, since MemNets do not update the differentiable memory, these efforts store a copy of the memory in its transposed form and do not provide any support for on-chip transpose. For the variants of MANNs that do require soft writes, hardware support is crucial; our ablation study indicates that support for element-wise operations and on-chip transpose lead to speedups of `2.8x` and `1.4x`, respectively.

Another related effort, X-MANN [30] realizes the differentiable memory using resistive crossbars. Apart from relying on emerging device technologies that are not widely available, it is also not an end-to-end solution since it does not accelerate the addressing kernels or the DNN controller.

9 CONCLUSION

MANNs are a promising direction in machine learning that enable DNNs to achieve cognitive capabilities well beyond those of classical DNNs. We provide a detailed investigation of the memory and computational characteristics of MANNs, as well as their impact on the design of an efficient accelerator for MANNs. We present Manna, a specialized hardware accelerator for MANNs. Manna is better suited to the low FLOPs/Byte ratio inherent to MANNs, allocating most of the on-chip die area to memory. Manna also utilizes a hardware-assisted transpose mechanism to accomodate the heterogeneity of memory access patterns found in MANNs. We evaluate a 16-tile Manna configuration and demonstrate significant performance and energy benefits over Pascal and Turing GPUs, as well as favorable weak and strong scaling.

ACKNOWLEDGMENTS

This work was supported in part by C-BRIC, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] AMD. [n. d.]. High-Bandwidth Memory: Reinventing Memory Technology.
[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[3] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A Dynamically Configurable Coprocessor for Convolutional Neural Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 247–257. https://doi.org/10.1145/1815961.1815993
[4] Andre Xian Ming Chang and Eugenio Culurciello. 2017. Hardware accelerators for recurrent neural networks on FPGA. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*. IEEE.
[5] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. 2015. Recurrent neural networks hardware implementation on FPGA. *arXiv preprint arXiv:1511.05552* (2015).
[6] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967
[7] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the International Symposium on Microarchitecture (MICRO '14)*. IEEE Computer Society, 609–622.
[8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2018. Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks. *arXiv preprint arXiv:1807.07928* (2018).
[9] 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 (2017), 127–138.

[10] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259* (2014).

[11] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Proceedings of the Conference on Computer Vision and Pattern Recognition Workshops (CVPR Workshops 2011)*. IEEE, 109–116.

[12] Vinayak Gokhale, Aliasger Zaidy, Andre Xian Ming Chang, and Eugenio Culurciello. 2017. Snowflake: A model agnostic accelerator for deep convolutional neural networks. *arXiv preprint arXiv:1708.02579* (2017).

[13] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

[14] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401* (2014).

[15] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (2016), 471.

[16] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to transduce with unbounded memory. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*. 1828–1836.

[17] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2017. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[18] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[19] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR '16)*. 770–778.

[21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[22] 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 *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 250–263. https://doi.org/10.1145/3307650.3322214

[23] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[24] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[25] Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. 2016. Ask me anything: Dynamic memory networks for natural language processing. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[26] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE Press, 694–701.

[27] Sicheng Li, Chunpeng Wu, Hai Li, Boxun Li, Yu Wang, and Qinru Qiu. 2015. Fpga acceleration of recurrent neural network based language model. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[28] Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit Mishra, Srivatsan Krishnan, and Debbie Marr. 2016. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

[29] Seongsik Park, Jaehee Jang, Seijoon Kim, and Sungroh Yoon. 2019. Energy-Efficient Inference Accelerator for Memory-Augmented Neural Networks on an FPGA. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1587–1590.

[30] Ashish Ranjan, Shubham Jain, Jacob R. Stevens, Dipankar Das, Bharat Kaul, and Anand Raghunathan. 2019. X-MANN: A Crossbar Based Architecture for Memory Augmented Neural Networks. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. ACM, New York, NY, USA, Article 130, 6 pages. https://doi.org/10.1145/3316781.3317935

[31] Greg Ruetsch and Paulius Micikevicius. 2009. Optimizing matrix transpose in CUDA. *Nvidia CUDA SDK Application Note* 18 (2009).

[32] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[33] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. 2015. End-to-end memory networks. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[34] 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 International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 13–26. https://doi.org/10.1145/3079856.3080244

[35] Zhisheng Wang, Jun Lin, and Zhongfeng Wang. 2017. Accelerating recurrent neural networks: A memory-efficient approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2763–2775.

[36] Greg Wayne, Chia-Chun Hung, David Amos, Mehdi Mirza, Arun Ahuja, Agnieszka Grabska-Barwinska, Jack Rae, Piotr Mirowski, Joel Z Leibo, Adam Santoro, et al. 2018. Unsupervised predictive memory in a goal-directed agent. *arXiv preprint arXiv:1803.10760* (2018).

[37] Jason Weston, Sumit Chopra, and Antoine Bordes. 2014. Memory networks. *arXiv preprint arXiv:1410.3916* (2014).

[38] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[39] Caiming Xiong, Stephen Merity, and Richard Socher. 2016. Dynamic memory networks for visual and textual question answering. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[40] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. 2017. Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3018743.3018754

[41] Xiang Zhang and Yann LeCun. 2015. Text understanding from scratch. *arXiv preprint arXiv:1502.01710* (2015).