

# A<sup>3</sup>: Accelerating Attention Mechanisms in Neural Networks with Approximation

Tae Jun Ham\*   Sung Jun Jung\*   Seonghak Kim\*   Young H. Oh†   Yeonhong Park\*

Yoonho Song\*   Jung-Hun Park\*   Sanghee Lee\*   Kyoung Park‡   Jae W. Lee\*   Deog-Kyoon Jeong\*

\*Seoul National University   †Sungkyunkwan University   ‡SK Hynix

{taejunham, miguel92, ksh1102, ilil96, yhsong, jhpark94, shlee95, jaewlee, dkjeong}@snu.ac.kr,  
younghwan@skku.edu, kyoung.park@sk.com

**Abstract**—With the increasing computational demands of neural networks, many hardware accelerators for the neural networks have been proposed. Such existing neural network accelerators often focus on popular neural network types such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs); however, not much attention has been paid to *attention mechanisms*, an emerging neural network primitive that enables neural networks to retrieve most relevant information from a knowledge-base, external memory, or past states. The attention mechanism is widely adopted by many state-of-the-art neural networks for computer vision, natural language processing, and machine translation, and accounts for a large portion of total execution time. We observe today’s practice of implementing this mechanism using matrix-vector multiplication is suboptimal as the attention mechanism is semantically a content-based search where a large portion of computations ends up not being used. Based on this observation, we design and architect A<sup>3</sup>, which accelerates attention mechanisms in neural networks with algorithmic *approximation* and hardware *specialization*. Our proposed accelerator achieves multiple orders of magnitude improvement in energy efficiency (performance/watt) as well as substantial speedup over the state-of-the-art conventional hardware.

**Keywords**—Attention Mechanism, Accelerators, Approximation, Neural Networks, Machine Learning, ASIC

## I. INTRODUCTION

Neural networks (NNs) are currently one of the most popular techniques to perform complex AI tasks in various domains such as computer vision, natural language processing, robotics, etc. With a large amount of data, NNs can effectively solve a wide range of AI challenges and can often surpass human performance in many domains. However, these advantages of NNs come at a high computational cost involving tens of billions of floating-point operations. In order to minimize the energy cost of such a large number of operations and maximize the throughput of NN processing, many prior works proposed various FPGA or ASIC based accelerators [1], [2], [3]. These prior works are indeed very effective in improving the performance and energy efficiency of the popular NN types such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs). However, such accelerators do not provide full support for an emerging NN primitive such as attention mechanisms.

Attention mechanisms are one of the most important recent advancements in neural networks. Unlike CNNs and RNNs, which has a limited capacity to utilize information from

the past or external knowledge, attention mechanisms (also called memory mechanisms) enable NNs to access and utilize such information by providing extra connections to past state cells, explicit memory cells, and so on. Naturally, not all information from past state cells or explicit memory cells is equally relevant to what NN is currently processing. Hence, the attention mechanism determines what is relevant to the currently processed information through content-based similarity search and decides where to *attend*. This attention mechanism has been recently adopted in many domains of NNs such as computer vision [4], [5], [6] and natural language processing [7], [8], [9], [10]. In addition, this mechanism is also used to enable NNs to solve a class of complex problems that were previously difficult for conventional NNs due to their lack of ability to memorize and retrieve data [11], [12].

In the conventional hardware, an attention mechanism is usually implemented as dense matrix operations and softmax operations. A dense matrix-vector multiplication operation computes the similarity across all search targets and thus the computational complexity of this operation is proportional to the number of search targets. In other words, attention mechanism requires more computation when a NN model wants to retrieve relevant information over the larger external knowledge-base, over a longer period of past information, or from a longer sequence of data. To make it even worse, in some NN models utilizing self-attention mechanisms [7], [13], [14], the computational complexity of attention mechanism is proportional to the square of the search targets (e.g., a length of the reading passage in the reading comprehension task). Naturally, this large computational cost of attention mechanism becomes a limiting factor for the capacity of the NN models and accounts for a significant portion of the performance and energy cost of existing models.

To address this challenge and mitigate the bottlenecks arising from the computational cost of attention mechanisms, we architect A<sup>3</sup>, a hardware accelerator for attention mechanisms in NNs. To design an efficient accelerator, we not only focus on the efficient implementation of the attention mechanism in hardware but also focus on reducing the amount of computation in attention mechanism through algorithmic optimization and approximation. In particular, based on the observation that not all search targets are equally likely to be relevant, our design presents an approximate candidate

```

1 // key:  $n \times d$ , value:  $n \times d$ , query, output:  $d$ 
2 float[] attention_mechanism(float key[],
3 float value[], float query[]) {
4 /* Step 1 : Dot-Product Computation */
5 for i = 0 to n:
6   sum = 0
7   for j = 0 to d:
8     sum += key[i][j] * query[j]
9   dot_product[i] = sum
10 /* Step 2 : Softmax Computation */
11 score = softmax(dot_product)
12 /* Step 3 : Output Computation */
13 for j = 0 to d:
14   sum = 0
15   for i = 0 to n:
16     sum += score[i] * value[i][j]
17   output[j] = sum
18 return output
19 }
20 float[] softmax(float input[]) {
21   sum = 0
22   for i = 0 to n:
23     sum += exp(input[i])
24   for i = 0 to n:
25     output[i] = exp(input[i]) / sum
26   return output
27 }

```

Figure 1. Pseudocode for an attention mechanism.

selection mechanism to reduce the number of search targets, and thus the amount of computation. Furthermore, we propose a specialized hardware pipeline exploiting parallelism to accelerate approximated attention mechanisms while making it even more efficient. With this algorithm-hardware co-design, our proposed accelerator offers significant performance improvements and orders of magnitudes improvements in energy efficiency (performance/W), thereby enabling existing NN models with attention mechanisms to utilize larger external knowledge or a longer sequence of data. Our contributions are summarized as follows.

- We quantify the attention mechanism bottlenecks in NN models and identify that a substantial portion of time in NN models is spent on attention mechanisms.
- We exploit the potential for approximation in attention mechanisms and present an approximation scheme which enables our proposed hardware to find potentially relevant search targets while avoiding an exhaustive search.
- We design a specialized hardware pipeline for an attention mechanism exploiting parallelism and datapath specialization to significantly improve the performance and energy efficiency of the attention mechanism.
- We demonstrate that our proposed accelerator achieves multiple orders of magnitude speedup and energy efficiency over conventional hardware. Furthermore, with our specialized hardware for the approximation scheme,  $A^3$  achieves even higher speedup and energy efficiency while minimizing the degradation of the model accuracy.

## II. BACKGROUND AND MOTIVATION

### A. Attention Mechanism

**Operation.** Attention mechanism is essentially a content-based search. Figure 1 represents the computation of the attention mechanism in pseudocode. Given a query vector

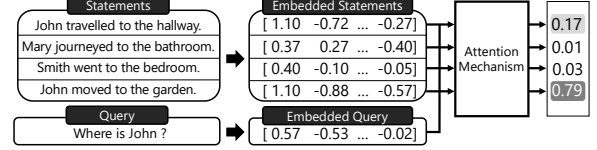


Figure 2. Example application of attention mechanism (Step 1 and 2 in Figure 1) from Facebook bAbI QA [15].

with  $d$  dimensions and a key matrix with  $n$  vectors where each vector has  $d$  dimensions, the attention mechanism first computes a similarity score (i.e., dot-product) for each entry in the key matrix (Step 1 in Figure 1). After this process, an  $n$ -dimensional vector (i.e., `dot_product[]`) is obtained. This array is then processed with softmax function (Step 2 in Figure 1). Finally, the normalized score is used as a weight to retrieve the weighted sum of vectors from the  $n \times d$  value matrix (Step 3 in Figure 1). In short, the set of indices for the set of vectors in the key matrix which are similar to the query vector is first obtained, and these indices (along with weight values) are used to obtain the weighted sum for the set of vectors from the value matrix. This mechanism is often called *soft* attention mechanism since it only consists of differentiable computations, which makes this mechanism well-suited for NNs which are trained with back-propagation.

**Example Application.** Figure 2 introduces a very simple example which shows how the attention mechanism is utilized to enable a NN to find a sentence that is relevant to the question in the Facebook bAbI QA task [15]. In this task, a list of statements and a question are provided in a natural language. The goal is to find the right answer to the question. Note that not all provided statements are necessary to answer a question. In many NN models solving this task, an attention mechanism is utilized to identify the most relevant statements among these provided statements. For example, End-to-End Memory Network [8] first embeds (i.e., converting natural language into vector representation as in Word2Vec [16], Glove [17], FastText [18]) each statement sentence and query sentence. Then, using the attention mechanism, it finds the most relevant sentence to the query from the set of statements. For example, as shown in Figure 2, the attention mechanism can identify that “*John moved to the garden.*” is the most relevant sentence for a query “*Where is John?*” by performing a similarity search using the embeddings. If multiple sentences are required to answer the question, it updates the query with the relevant sentence found in the previous iteration and utilizes the attention mechanism again to retrieve other relevant sentences from the set. After obtaining all relevant sentences, it utilizes a final weight matrix to generate the final answer.

### B. Cost of Attention Mechanism

For a given  $n$  and  $d$ , an attention operation requires i)  $nd$  multiplications and  $n(d - 1)$  additions in Step 1, ii)  $n$  exponent computations and  $n - 1$  additions, and  $n$  divisions

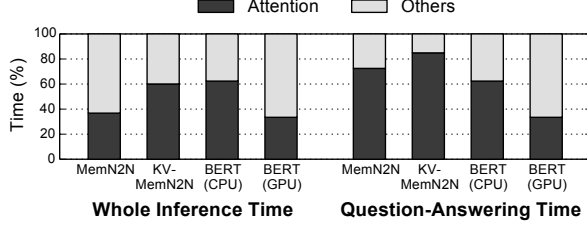


Figure 3. Portion of the time accountable for attention mechanism of NN workloads (for the total inference time and for the query response time).

in Step 2, and iii)  $nd$  multiplications and  $(n-1)d$  additions in Step 3. Naturally, the number of these computations increases almost-linearly with both  $n$  and  $d$ . Here,  $n$  represents the number of data in an external knowledge base or the number of past states that this mechanism allows models to look for. Naturally, the larger  $n$  allows more powerful NN models. On the other hand,  $d$  is the dimension of a single vector entry. A single vector in the key matrix usually represents an embedding of a word, a sentence, a knowledge, or any other portion of a larger entity. The larger  $d$  allows this embedding to have a richer space and thus can provide a higher quality embedding. Between  $n$  and  $d$ ,  $n$  is much more likely to increase further since a larger  $n$  directly makes the NN models more powerful by allowing it to search over a larger number of data to extract useful information.

We analyze the portion of the time spent on attention mechanisms in popular NN models. For this analysis, we focus on three different workloads utilizing attention mechanisms: End-to-End Memory Network (MemN2N) [8] running bAbI QA task [15], Key-Value Memory Network (KV-MemN2N) [19] running WikiMovies QA task [19], and Google BERT [9] running SQuAD task [20] (see Section VI-A for details). We run these workloads on Intel Xeon Gold 6128 CPU (all workloads) and NVIDIA Titan V GPU (BERT only) then report the profiled data below.

Figure 3 shows that the attention mechanism is responsible for the significant portion (i.e., over 35% in all workloads) of the inference runtime. If we take a more in-depth look into the nature of these tasks, the actual cost of the attention model is even higher. Most models handling QA tasks take a substantial amount of time on comprehension (e.g., embedding generation) of the provided knowledge (e.g., list of statements in bAbI QA task or Wikipedia-oriented information about various movies in WikiMovies task). Those processes are query-independent and thus it is possible to preprocess them before a query is provided. Thus, the actual critical path (i.e., query response time) of the question-answering task often does not include such time. The right side of Figure 3 shows query response time (as opposed to the total inference time which includes both comprehension time and query response time) for all workloads. Compared to the result for the total inference time, the attention mechanism takes noticeably a larger portion of time (over 70%) in both MemN2N and KV-MemN2N. The portion of the attention

mechanism in the BERT model remains the same since it performs comprehension and query response in an integrated manner. To accelerate attention mechanism and make it more energy-efficient, we design a specialized hardware accelerator for the attention mechanism. Section III presents how our design efficiently handles this operation.

### C. Opportunity for Approximation

Typically, in many popular NN model implementations, the attention mechanism is implemented as a matrix-vector multiplication (i.e., multiplication of the key matrix and the query vector) followed by a softmax function again followed by another matrix-vector multiplication (i.e., multiplication of the value matrix and the weight vector). Such implementations can utilize the fast matrix-vector multiplication capability provided by popular NN processing frameworks such as TensorFlow [21] and Torch [22]. A dense matrix-vector multiplication-based implementation is functionally correct. However, the nature of this operation is a search, not a dense computation. In reality, most of the computations performed in the first matrix-vector multiplication have very little impact on the final output since most score values become near-zero after the softmax normalization which is essentially a continuous, differentiable approximation of the *argmax* operation which selects the index for the maximum number in an array.

The intuition behind our approximation proposal is to avoid such unnecessary computation. By preprocessing the key matrix, it is possible to obtain a set of candidate rows which are likely to have high score values without much computation. By doing so, our proposed scheme can avoid unnecessary dot product computation, softmax computation, and final result computation. Section IV presents our proposed approximation scheme and Section V presents the hardware accelerator module necessary for this approximated attention mechanism.

## III. $A^3$ BASE DESIGN

We introduce the base design of  $A^3$ , a specialized hardware accelerator for attention mechanisms in neural networks which can be integrated to either CPU, GPU, or an existing hardware accelerator.  $A^3$  accelerates the attention mechanism shown in Figure 1 of Section II. For high throughput and energy efficiency,  $A^3$  employs a pipeline designed with customized datapath exploiting parallelism. This section provides an in-depth overview of the base design of  $A^3$  without approximation, and later sections introduce an approximate attention mechanism and extended hardware designs.

### A. Pipeline Design

Base  $A^3$  takes three inputs — a key matrix ( $n \times d$ ), a value matrix ( $n \times d$ ), and a query vector ( $d$ ) — to compute a  $d$ -dimensional output vector. Figure 5 shows the computation

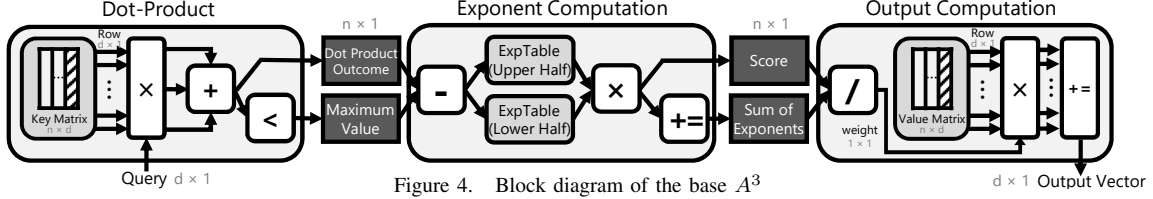


Figure 4. Block diagram of the base  $A^3$

pattern of the Base  $A^3$  in pseudocode. Note that this is slightly different from the pseudocode in Figure 1 as we reorder some computations to make it more suited for hardware implementation. Figure 4 shows the block diagram of the base  $A^3$  pipeline, which implements the pseudocode in Figure 5. As shown in both figures, the base  $A^3$  pipeline consists of three modules: i) dot-product module, ii) exponent computation module, and iii) output computation module. Below, we explain each module in greater details.

```

1 float[] attention_mechanism (float key[][],
2   float value[][], float query[]) {
3   /* Module 1: Dot-Product*/
4   max = -∞
5   for i = 0 to n:
6     parallel for j = 0 to d:
7       temp[i][j] = key[i][j] * query[j]
8     dot_product[i] = ParallelSum(temp[i])
9     if dot_product[i] > max:
10      max = dot_product[i]
11  /* Module 2: Exponent Computation */
12  expsum = 0
13  for i = 0 to n:
14    dot_product[i] -= max
15    score[i] = exp(dot_product[i])
16    expsum += score[i]
17  /* Module 3: Output Computation */
18  for i = 0 to n:
19    weight[i] = score[i]/expsum
20    parallel for j = 0 to d:
21      output[j] += weight[i] * value[i][j]
22  }

```

Figure 5. Base  $A^3$  pipeline represented with pseudocode.

**Module 1: Dot-Product Computation.** The first module of the pipeline (Line 3-10 in Figure 5) computes an inner product between a single row of the key matrix and a query vector. This hardware consists of  $d$  multipliers and a  $d$ -way adder tree for a sum reduction operation. For each cycle, a row of the key matrix is loaded (in sequential order) and each of its vector elements is multiplied by the corresponding element of the query vector using the array of multipliers. Then, once the multiplication finishes, these set of values are passed to the adder tree for a parallel sum reduction. The result of this operation is stored in the corresponding register in the dot product outcome register file. This process is repeated for  $n$  times until all rows in the key matrix are processed. Note that this module also finds the maximum value among all elements of the dot\_product array (Line 9-10 in Figure 5). This value is used by the next module.

**Module 2: Exponent Computation.** Exponent computation (Line 11-16 in Figure 5) computes the exponent of each dot-product value computed by the previous module. Normally, to perform an exponent computation, a hardware exponent computation unit should be used. Instead, our design imple-

ments an exponent function using a lookup table. However, there are two challenges with this approach: handling an overflow and reducing the size of the lookup table. First, the outcome of an exponent function increases very rapidly with the increase in input value, and thus it can easily cause an overflow in a fixed-point representation for large input. To counter this, we leverage the fact that softmax is *invariant* to the addition (or subtraction) of a constant to all elements in the vector. Before performing an exponent computation, this module first starts with subtracting the maximum value of the input vector from the dot-product value being processed. With this subtraction, all elements in the input vector will be less than or equal to zero, and thus the outputs of the exponent function will always be less than or equal to 1.

The second challenge is that the size of the lookup table becomes very large when high precision is required. For example, if we use a 16-bit representation, a lookup table requires 65,536 entries and such an SRAM can incur large power and area overhead. To reduce the size of the lookup table, we exploit the fact that a single exponent operation can be decomposed into a multiplication of two exponent operations. For example, as shown below, computing an exponent for 8-bit input is equivalent to computing an exponent for two 4-bit inputs (i.e., one for the upper 4-bits and another for the lower 4-bits) and multiplying them.

$$e^{0.10101111_2} = e^{0.10100000_2} \times e^{0.00001111_2}$$

With this transformation, instead of building a large lookup table (e.g., one with 65,536 entries), we can utilize two smaller lookup tables (e.g., each with 256 entries) and a multiplier to obtain the same outcome. Once the exponent of the dot-product is computed, this value gets accumulated. This accumulated value is later used as the softmax denominator (Line 19 in Figure 5).

**Module 3: Output Computation.** This module (Line 17-21 in Figure 5) computes the output of the attention operation. Every cycle, an element of the score vector is divided by the sum of all score values for normalization. Then, this value (i.e., weight) serves as a scaling factor for the corresponding row vector in the value matrix. Each element of a row vector is multiplied by this value. The result of this computation is accumulated in the output register. This process is repeated for  $n$  times.

**Throughput and Latency.** Our proposed hardware can handle three queries at a time in a pipelined manner. When a query finishes its computation for a module, it is then passed to the next hardware module, and thus the next query can enter the current module. To balance this pipeline, we

deliberately design all three modules to have the matching throughput (i.e., each module takes  $n$  cycles +  $\alpha$  to process a query). Among three modules, the last module has the longest latency of  $n + 9$  ( $n$  cycles to handle  $n$  rows in a pipelined way, 7 cycles for a division, and 2 cycles for a multiplication and accumulate). Thus, the pipeline latency is  $3n + 27$  cycles and the throughput is  $n + 9$  cycles per query.

### B. Quantization

Most NN tasks are tolerant to certain levels of errors by nature and can thus operate with lower bitwidth representations while keeping the accuracy of higher bitwidth representations. In such a case, utilizing a lower bitwidth is crucial to the accelerator design since many of the registers' or computation unit's energy cost scales linearly or quadratically with the bitwidth of a representation. In addition, since floating-point operations cost much more energy than fixed-point ones, it is often beneficial for specialized accelerators to utilize a fixed-point representation. Our model first quantizes the provided floating-point input to  $i$  integer bits and  $f$  fraction bits (plus a sign bit), and then utilizes different bitwidths for each stage of the pipeline to maintain the precision and avoid an overflow while minimizing the energy cost. Below, we explain our rationale for our choice of fixed-point bitwidths for the pipeline. We evaluate the impact of quantization in Section VI-B.

**Integer Bitwidths.** Integer bitwidths are determined by the dynamic range of values. Suppose  $i$  (e.g.,  $i = 4$  is used for our evaluation) integer bits are required to represent the elements of the input key matrix and the query vector. In other words, the value of such elements is limited to a range between  $-2^i + 1$  and  $2^i - 1$ . In the dot-product module, these input values are first multiplied and stored in `temp[][]`, which requires  $2i$  bits to avoid an overflow and maintain precision. Then, the sum of  $d$  `temp[][]` values are stored in `dot_product[]` which requires  $\log_2(d) + 2i$  bits. Then, in the exponent computation module, the value of the maximum elements in `dot_product[]` is subtracted from all elements in `dot_product[]`, which requires one extra integer bits for `dot_product[]`. After the exponent computation, the value of `score[]` is now limited to a range between 0 and 1 and thus no integer bit is required. For `expsum`, which is a sum of  $n$  values in `score[]`,  $\log_2(n)$  bits are required, and for `weight[]`, no integer bit is required since their value is still bounded to a range between 0 and 1. For the final output[],  $i + \log_2(n)$  bits are required.

**Fraction Bitwidths.** Fraction bitwidth is directly related to the precision of the value. Assume we utilize  $f$  (e.g.,  $f = 4$  is used for our evaluation) bits to represent the elements of both inputs (i.e., key matrix and query matrix). In the dot-product module, inputs with  $f$  fractional bits are multiplied and stored in `temp[][]`, which now requires  $2f$  bits to maintain precision. Since additions (or subtractions) do not change the required number of fractional bits, `dot_product[]` value also

keeps  $2f$  fraction bits. Keeping  $2f$  fraction bits across this exponent computation also does not lose precision<sup>1</sup> and thus `score[]` uses  $2f$  fraction bits as well. `weight[]` also uses  $2f$  fraction bits since division does not require additional precision as long as the divisor (i.e., `expsum` in our case) is larger than 1. Lastly, output requires  $3f = 2f$  (for weights) +  $f$  (for values) fraction bits.

### C. Design Details

**Offloading Mechanism.**  $A^3$  can be integrated with most devices including CPUs, GPUs, or hardware accelerators for deep learning. Since the base  $A^3$  is a relatively straightforward accelerator, it can be integrated to any level of the memory hierarchy. Before invoking  $A^3$ , a key matrix and a value matrix should first be copied to the SRAM buffer of  $A^3$ . Note that the time it takes to copy these matrices is often not a part of the query response time. For example, in question-answering s, a key matrix and value matrix are ready on *comprehension time* (i.e., reads and memorize knowledge) rather than a *query response time* (i.e., reads a question and try to find a relevant knowledge to answer the given question). In other words, a key matrix and a value matrix are copied beforehand and the only communication time included to the query response time is the time it takes for a host device to copy a query vector to  $A^3$ . Once the query arrives,  $A^3$  will buffer it to the query queue. Whenever a dot-product module becomes available, it can start the computation for the query. At the end of the computation, the output vector will be buffered to the output queue for the host.

**Use of Multiple  $A^3$  Units.** Most NN processing tasks have a large amount of parallelism. In such workloads, it is often desirable to handle multiple, independent attention computations in parallel. In such a case, it is possible to use multiple copies of our  $A^3$  units for a different key, value matrices sets. On the other hand, there are often cases where multiple queries are processed to the same set of key and value matrices. In that case, queries can execute in parallel through pipelining in a single  $A^3$  unit. Note that it is also possible to utilize multiple instances of  $A^3$  units for the same set of key, value matrices to increase the throughput.

**Choice of  $n$  and  $d$ .**  $A^3$  can be synthesized for different  $n$  and  $d$  values depending on the needs. In our evaluation, we set  $n$  to 320 and  $d$  to 64 to fit the largest task we evaluated. However,  $A^3$  can still handle a task that requires even larger  $n$  values. When a larger  $n$  is desired, we store first  $n$  vectors to the SRAM while leaving other vectors to the DRAM. Since  $A^3$  accesses both the key matrix and the value matrix in a sequential manner, it is possible to utilize a prefetcher to

<sup>1</sup>When the quantization error is positive (i.e.,  $\epsilon > 0$ ),  $e^{x+\epsilon} - e^x = e^{x+\epsilon}(1 - e^{-\epsilon}) < \epsilon$  holds true because  $e^{x+\epsilon} < 1$  ( $\because x < 0$  indicates  $x+\epsilon < 0$ ) and  $1 - e^{-\epsilon} < \epsilon$ . When  $\epsilon < 0$ , a quantization error  $e^x - e^{x+\epsilon} = e^x \cdot (1 - e^\epsilon) < -\epsilon = |\epsilon|$  because  $e^x < 1$  ( $\because x \leq 0$ ) and  $1 - e^\epsilon < -\epsilon$ . These inequalities prove that error becomes smaller after the exponential function when the exponent part is negative.

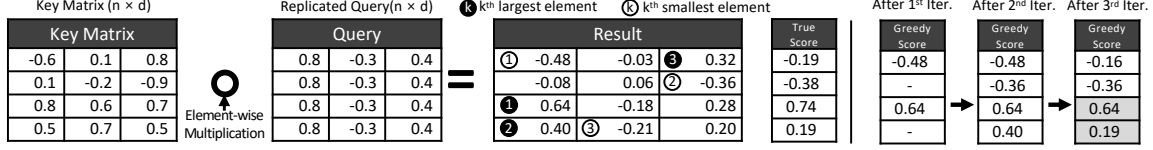


Figure 6. Illustration of the base greedy candidate search algorithm.

read them from a memory without exposing memory latency. Note that the size of the key matrix and the value matrix for one of the largest existing models utilizing attention mechanism (i.e.,  $n = 320$  and  $d = 64$  on BERT model) is still small enough to fit in SRAM so it is likely that the use of DRAM won't be necessary for the near future. Unlike  $n$ ,  $d$  is not likely to vary widely since a choice of too large  $d$  can lead to decrease in model accuracy [23], [24]. For this reason, it often makes sense to simply assume a large enough  $d$  and use zero-padding when smaller  $d$  is desired.

#### IV. APPROXIMATE ATTENTION

##### A. Overview

As explained in Section II, the attention mechanism is essentially a content-based approximate search. In a conventional attention mechanism, the algorithm computes the similarity score between the query vector and all candidates (i.e., each row in the key matrix) and translates scores to weights using softmax function. With those weights, a weighted sum of each row in the value matrix is computed and returned.

Here, one important point is that most of those weight values are often near-zero. Softmax function is a soft (i.e., differentiable) version of the argmax function which amplifies the value differences between a few large entries and other smaller entries. Since score values are transformed to weights with this function, candidates with relatively small score values get near-zero weight. In addition, these near-zero weights often do not contribute to the accuracy of the model. Rather, it is more of a byproduct of utilizing a differentiable version of the max function, which is important for training, but not for inference. So for these near-zero weights, it is actually more beneficial to treat them as zeros and avoid including them for softmax computation and the following weighted sum computation.

An even better way is to avoid computing the score at all for rows of the key value matrix that will end up with the low score, and have near-zero weights after the softmax computation. For this purpose, we present an approximation algorithm which can select the candidates that are likely to have a high score without actually computing the score. The main intuition behind this approximation approach is that it is possible to preprocess the key matrix without affecting the critical path. As explained in Section II, the key matrix (and the value matrix) is obtained at *knowledge comprehension time* rather than *question answering time*. By preprocessing the key matrix, our algorithm tries to reduce the number of operations that can be handled at *question answering time*

where the query becomes available. In addition, on models like BERT where multiple queries (e.g., 320) utilize this same key matrix, the cost of the preprocessing key matrix is amortized and incurs only a limited amount of overhead.

##### B. Base Greedy Candidate Search

Inner product computation between a query vector and a row in a key matrix is a sum of component-wise multiplication result across  $d$ -dimensions. The main idea behind our proposed scheme is that looking at a single component-wise multiplication result provides information about the final outcome. More specifically, our proposed algorithm assumes the following: if a multiplication result of a particular dimension is a large positive number, it is likely that the sum of multiplication results for all dimensions (i.e., dot product result) is large as well. Similarly, if a multiplication result of a particular dimension is a large negative number, it is likely that the dot product result is not a large positive value. In fact, a similar intuition is used for other application domains such as information retrieval [25].

Figure 6 illustrates the basic idea of our algorithm. Given a key matrix and a query vector, this algorithm first replicates query vector across rows to make a replicated query matrix. Then, an element-wise multiplication of these two matrices is performed. Then, an element at the  $i$ th row and the  $j$ th column in the resulting matrix represents a  $j$ th dimension multiplication result between the  $i$ th row and the query vector. Naturally, the sum of all elements in a single row computes the inner product between a row in the key matrix and the query vector (represented as *True Score* in Figure 6).

The main idea of this algorithm is to look at the largest (or the smallest) element in this result matrix in an iterative way. During a  $k$ th iteration, the algorithm checks the  $k$ th largest (and the  $k$ th smallest) element and adds such value to the corresponding row in the greedy score array. This process is repeated for  $M$  (i.e., a user-configurable parameter) times. Once these iterations finish, a greedy score array approximates the true score array. If a row has a positive greedy score, this indicates that the row has one or more relatively large positive components. On the other hand, a row with a negative greedy score indicates that this row has one or more relatively large negative components. Based on this observation, our algorithm selects rows that have positive scores at the end of the iteration as candidates and passes them to the base  $A^3$  dot product module.

This algorithm, in its current form, has a time complexity of  $O(nd \log nd)$ . In order to select the  $k$ th largest element from the result matrix, all elements of the matrix should

be sorted to avoid performing a linear search every time. Naturally, an  $O(nd \log nd)$  time algorithm is not very useful when full dot product computation (i.e., true score computation) takes  $O(nd)$  time. Below, we introduce the efficient implementation of this algorithm which exploits preprocessing steps to make the *query response time* (i.e., the time it takes from query arrival to the output) not dependent on  $n$ .

### C. Efficient Greedy Candidate Search

Figure 7 shows an algorithm that is functionally identical to the one explained in the previous subsection and Figure 8 shows example data structures utilized in this algorithm. While this algorithm is functionally identical to the one presented in the previous subsection, it utilizes a preprocessing step to minimize the latency at the *query response time*. In this algorithm, preprocessing step (Line 1-5) only requires the key matrix, which is often available at *comprehension time*, where an algorithm comprehends knowledge, past states, etc. for future uses. On the other hand, candidate\_selection happens on *query response time* where a query is ready. This is often a critical path since it is desirable to answer the provided query in a short time. Note that there are some models where preprocessing cannot happen off-critical path (e.g., Google BERT). However, models like Google BERT and Transformer utilize self-attention mechanism which utilizes the same key matrix for multiple queries (e.g., 320 times). In such a case, the cost of the preprocessing is amortized so that models' performance can still benefit from approximation.

**Preprocessing.** During a preprocessing step, each column of the key matrix is sorted and the result is stored in sortedKey (Line 1-5 in Figure 7). Then, once the query becomes available, candidate\_selection starts. At the beginning of candidate\_selection, the max\_ptr (and min\_ptr) is initialized for each column. The max\_ptr is set to the row index of the entry with the largest value in the column of the sortedKey matrix if the query component for the corresponding column (i.e., query[i]) is positive. Otherwise, it is set to the entry with the smallest value in the column. The min\_ptr is also initialized in a similar, but in the opposite way. Then, priority queues (i.e., maxQ, minQ) are initialized. Each entry in each column of the sortedKey pointed by max\_ptr (and min\_ptr) is first multiplied with the corresponding query component and then inserted to the maxQ (and minQ, respectively) along with its rowID and colID (Line 12-16).

**Iterative Candidate Selection.** After the preprocessing, the iterative candidate selection (Line 17-25) begins. First, an entry from the maxQ (and minQ) is popped. This entry is the largest (or the smallest in case of minQ) entry among the ones currently pointed by max\_ptr (and min\_ptr). Then, the score value of the entry pointed by max\_ptr (and min\_ptr) is added to the greedy\_score array if it is positive (negative).

```

1 void preprocess (float key[][]) {
2   for i = 0 to d:
3     sortedKey[:,i] = sort(key[:,i])
4     //sortedKey: Sorted List of (val, rowID) pairs
5   }
6 }
7 int[] candidate_selection (float query[]) {
8   maxQ = MaxPriorityQueue()
9   // MaxPriorityQueue: Priority queue of (val, rowID,
10    colID) tuples
11   /* Initialize Pointer */
12   for i = 0 to d:
13     max_ptr[i] = (n-1 if query[i] > 0 else 0)
14   /* Initialize Priority Queue */
15   for i = 0 to d:
16     entry = sortedKey[max_ptr[i]][i]
17     score = entry.val * query[i]
18     maxQ.push(score, entry.rowID, i)
19   /* Iterative Candidate Selection */
20   for iter = 0 to M:
21     maxScore, rowID, colID = maxQ.pop()
22     if maxScore > 0:
23       greedy_score[rowID] += maxScore
24       max_ptr[colID] += (-1 if query[colID] > 0 else 1)
25       nextEntry = sortedKey[max_ptr[colID]][colID]
26       compMultRes = nextEntry.val * query[colID]
27       maxQ.push(compMultRes, nextEntry.rowID, colID)
28   /* Update Candidates */
29   for each (row, score) in greedy_score:
30     if score > 0:
31       candidates.append(row)
32   return candidates
33 }

```

Figure 7. Pseudocode representation of the efficient greedy candidate search algorithm. Note that statements related to minQ are omitted for conciseness since they are mostly symmetric to maxQ operations.

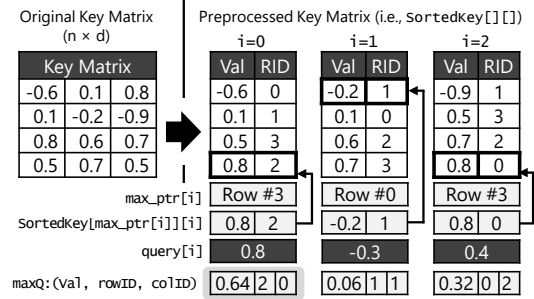


Figure 8. Illustration of the data structures for the efficient greedy search algorithm. minQ operations are omitted for conciseness.

After this, the max\_ptr (and min\_ptr) is updated so that it can point the next largest (or the smallest) entry in the column. Finally, the new entry pointed by the updated max\_ptr (and min\_ptr) is inserted to the respective priority queue. This step is repeated for  $M$  times and then rows with positive greedy scores are selected as candidates. Lastly, we apply a small heuristic — skipping the minQ operation when the cumulative sum of entries selected by max\_ptr and min\_ptr so far is negative — to avoid selecting too few candidates when overall similarity scores are low.

Unlike in the previous version of the algorithm, the complexity of candidate\_selection is  $M \log d$  ( $M$  loops each with  $\log d$  complexity from the iterative candidate selection step). This re-structured algorithm now can select likely candidates with a time complexity that scales with user-defined parameter  $M$ . In practice, to maintain reasonable accuracy,  $M$  needs to increase as  $N$  increases. However,



an important benefit here is that this algorithm provides a user a hyperparameter to adjust the balance between the performance and the accuracy. We evaluate how this algorithm can effectively estimate the set of likely candidates across different  $M$  in Section VI-B.

#### D. Post-scoring Approximation

Once a subset of rows of the key matrix is chosen as candidates, their full scores (i.e., the dot product between the key matrix row and the query) are computed. Then, those scores are used as an input for the softmax function and the outcome of softmax functions are used as weights for the final weighted sum computation. As briefly explained in Section IV-A, most of the relatively small dot-product values become near-zero after the softmax and thus have minimal impact on the final outcome. To minimize the computation required to calculate the softmax and the weighted sum, it is beneficial to avoid including some of the low-scoring candidates for those steps. One way to perform an approximation on this step is to sort candidates based on their dot product results and then only including top scoring rows for the next steps.

One important design choice there is to choose the number of top scoring rows to include for the next step. An easy way is to include a static, predefined number of top scoring rows but a static choice of such number does not necessarily work for all cases. For example, if high-scoring rows form a distribution with very low variance, it is better to include all of those rows for the following softmax computation and the weighted sum computation. On the other hand, if there is only one high-scoring row with many low-scoring rows, it is not beneficial to include more low-scoring rows for the softmax and the weighted sum computation. For this reason, we utilize a dynamic post-scoring approximation scheme which decides whether to include a row for the next steps. Basically, a score for a particular row is compared with the top-scoring row's score and if their difference is larger than threshold  $t$ , such row is excluded for the next steps. If a row's score is smaller than the top row's score by more than  $t$ , this means that this row will have a post-softmax weight that is at least  $e^t \times$  smaller than that of the top-scoring row. This is because softmax functions utilize the current score as an exponent term of the base  $e \approx 2.718$ . Throughout the paper, we use  $T = 100 * (1/e^t)$  instead of directly using  $t$ . With this notation,  $T$  indicates that an entry should have a post-softmax weight that is at least  $T(\%)$  of the maximum weight to be included for the next steps.

### V. $A^3$ WITH APPROXIMATION

To efficiently implement the approximation scheme introduced in Section IV, we design new hardware accelerator modules for candidate selection and post-scoring approximation. Then, we connect them to the base attention mechanism accelerator introduced in Section III to complete  $A^3$  with approximation capability.

#### A. Candidate Selection Module

This hardware is designed to accelerate the algorithm described in Figure 7. By utilizing the customized hardware and exploiting hardware parallelism, our candidate selection module enables us to significantly reduce the execution time and improve the energy efficiency of the algorithm. Figure 9 shows the block diagram for the candidate selection module.

This module has SRAM structures which buffer a preprocessed version of the key matrix as described in Section IV. In these SRAM modules, the sorted version of each column in the original key matrix are stored. And along with the values, each word in the SRAM also includes the row index of the corresponding value in the original key matrix just like in Figure 8. It also has two sets of  $d$  registers (max\_ptr and min\_ptr), two multipliers, two sets of  $d$  circular queues to buffer component multiplication results for each column, two  $d$ -dimensional comparator tree to find the maximum and minimum value from  $d$  component multiplication results, and a set of greedy score registers where greedy scores are updated.

**Pipeline Design.** A naive conversion of the algorithm in Figure 7 to hardware would result in low throughput as the iterative candidate selection (Line 17-25 in Figure 7) has dependency across iterations. Specifically, Line 19 is dependent on Line 25 of the previous iteration and this effectively makes each iteration of the loop to execute in series. We break this backward dependency by pre-executing component multiplication (Line 24) for all  $d$  dimensions for the first few iterations in advance. Assuming the critical path of the loop body takes  $c$  cycles (e.g.,  $c = 4$  in our implementation), our hardware is initialized with  $c \times d$  pre-computed component multiplication results ( $c$  items each for each column) in the component multiplication buffer, which consists of  $d$  circular queues (Figure 9). In steady state, our hardware simply removes one item from this component multiplication buffer (from the column that has the largest component multiplication result) and adds one back to that column  $c$  cycles later. Note that this  $c$ -cycle refill path is fully pipelined to allow our hardware to initiate a new iteration every cycle, thus achieving the throughput of one iteration per cycle. Below, we explain the operation of the pipeline in detail.

**Initialization.** Our hardware first initializes each of max\_ptr and min\_ptr to the appropriate values (i.e., either  $n - 1$  or 0) and reads them from SRAM simultaneously. Then, to fill the component multiplication buffer, our hardware performs i)  $2d$  multiplications where each multiply operation multiplies a component of the sorted key matrix pointed by max\_ptr (and min\_ptr) and the corresponding element of the query, ii) update of max\_ptr (and min\_ptr) register for each column. This process is repeated for a total of 4 times to fill the two sets of  $4 \times d$  buffer. This process requires  $8d$  multiplications, and it will normally take  $4d$  cycles since this module only



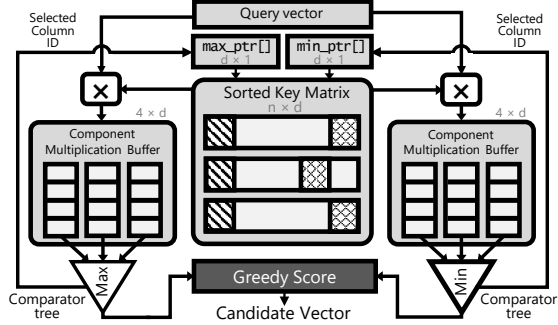


Figure 9. Simplified block diagram of the  $A^3$  candidate selection module

has two multipliers. However, to reduce this cost, we use  $d$  multipliers in the dot product module and  $d$  multipliers in the output computation module of the base  $A^3$  (Figure 4) just for this process. Once this finishes, this module goes to a steady state where only two multiplications happen every cycle.

**Candidate Selection.** In steady state, the hardware performs an iterative candidate selection (similar to Line 19-26 in Figure 7). Every cycle, this hardware performs multiple operations simultaneously: i) load an item from the specific column of the sorted key matrix that is pointed by the corresponding `max_ptr` (and `min_ptr`) register and buffer it near a multiplier, ii) perform a multiplication of the buffered data and the corresponding query elements and update the corresponding circular queue of the component multiplication result buffer, iii) find the maximum value among  $d$  items (i.e., the oldest items of each circular queue) with a  $d$ -way comparator and signal the selected column to update `max_ptr` (and `min_ptr`), and iv) update the greedy score register array with the value outputs from iii). Note that our candidate selection hardware utilizes a  $d$ -dimensional comparator tree to obtain the maximum (and minimum) entry among  $d$  elements in a single cycle instead of  $\log d$  cycles. With this hardware support, the complexity of the algorithm in Figure 7 becomes  $O(M)$  instead of  $O(M \log d)$ . Since this module performs one iteration every cycle in steady state, it takes  $M$  (plus a few extra) cycles to complete the iterative candidate selection part. Once completed, the hardware linearly scans the `greedy_score` registers (up to 16 contiguous entries per cycle) and sends one row ID (with positive score) to the next module.

### B. Post-Scoring Selection Module

The post-scoring selection module is in charge of identifying a certain number of top entries from the dot-product results following the mechanism described in Section IV-D. This hardware is integrated at the beginning of the exponent computation module in base  $A^3$  and its primary function is to select a candidate row with the maximum dot product value among remaining ones. For this purpose, this module simply computes the difference between the maximum dot product value and remaining entries at high throughput (e.g.,

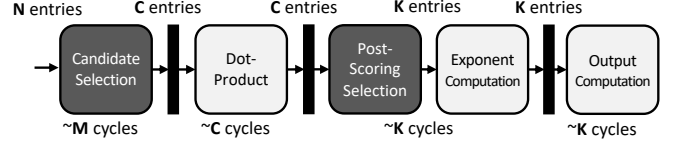


Figure 10. High-level block diagram of the  $A^3$  design with approximation.

16 entries per cycle). If the difference between a compared value and the top-scoring entry's value is larger than the preset threshold  $t$ , this value is passed to the next module (i.e., exponent computation module). This module essentially consists of 16 subtractors and comparators.

### C. $A^3$ HW with Approximation Support

Figure 10 shows the high-level block diagram of the  $A^3$  design. At the very beginning, given a preprocessed key matrix, the candidate selector module extracts the list of rows and passes it to the dot product module. The dot-product module computes the dot product results for the provided list of candidates and passes it to the post-scoring selector, which is integrated with the exponent computation module. The post-scoring selection module selects a few rows to be included for the exponent computation and then the exponent computation module computes the exponent for those rows. Finally, the output computation module generates the output by computing the weighted sum. Assuming  $C$  candidates are selected from the candidate selector module in  $M$  iteration and  $K$  entries are again selected from the post-scoring selection module, the total latency for  $A^3$  is  $M + C + K + K + \alpha$  cycles where  $\alpha$  is a constant. The throughput is limited by the candidate selector module ( $\approx M$  cycles) since our candidate selector module selects less than  $M$  candidates because i) each iteration may update the greedy score for the same row and ii) only the rows ended up with the positive greedy scores are selected.

## VI. EVALUATION

### A. Workloads

We use three representative neural network (NN) models utilizing attention mechanisms. First, we evaluate our  $A^3$  with Facebook's End-to-End Memory Network (MemN2N) [8] running bAbI QA task [15] which consists of twenty types of question-answering tasks. For each task, a set of statements is provided and the model aims to find the right answer through attention mechanisms which find the most relevant statement to the question. Second, we evaluate our design with Key-Value Memory Network (KV-MemN2N) [19] running Wiki-movies [26] question-answering tasks. In this workload, Key-Value Memory Network model first comprehends multiple excerpts about movies from Wikipedia, and then is expected to answer questions about movies. Lastly, we evaluate our proposal with Google BERT (base) [9], which utilizes a self-attention mechanism in Google Transformer [7] to solve many tasks in natural language processing. Among the many tasks

that this model can handle, we evaluate Stanford Question Answering Dataset task (SQuAD v1.1 [20]).

We used the embedding dimension  $d = 64$  for all workloads but each workload has different  $n$ . bAbI QA is a relatively small task whose average  $n$  (i.e., number of statements for a query) across all test inputs is 20 and the maximum is 50. For Wikimovies dataset, average  $n$  (i.e., the number of potentially relevant knowledge) is 186. Lastly, for SQuAD workload,  $n$  (i.e., the maximum length of an input passage and a question in terms of word counts) is 320.

### B. Accuracy Evaluation

**Methodology.** To estimate the impact on the accuracy of our approximation scheme proposed in Section IV, we implement a software model for approximation and integrate this model with our target workload’s official (or endorsed) open-source implementations. Specifically, we use the native Python implementation [27] for End-to-End Memory Network, Torch implementation [28] for Key-Value Memory network, and Tensorflow implementation [29] for BERT. Note that we only apply approximation techniques for an inference, which is our target, and we use test set inputs for accuracy measurements. For accuracy metric, we utilize one of the main metric used in the relevant paper for the task: accuracy for bAbI QA, Mean Average Precision (MAP) for Wikimovies dataset, and F1 score for SQuAD.

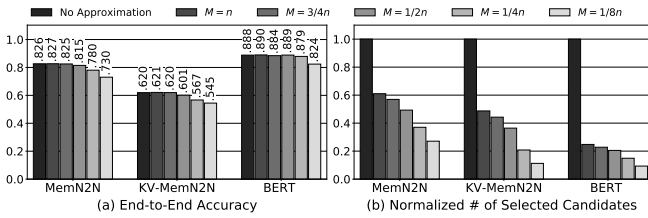


Figure 11. Impact of proposed candidate selection schemes on accuracy across varying iteration counts.

**Impact of Candidate Selection.** Figure 11a shows the percentage differences of accuracy metric for three workloads after applying the proposed candidate selection scheme in Section IV-C. Specifically, we vary  $M$  (i.e., iteration count for candidate selection algorithm in Section IV-C) in Figure 11 to check how varying  $M$  impacts accuracy and the number of candidates selected. As shown in Figure 11a, varying  $M$  from a large number (e.g.,  $n$ ) to a smaller number (e.g.,  $1/8n$ ) results in a change in model accuracy. This is because varying  $M$  results in a different number of selected candidates as shown in Figure 11b. Naturally, the larger the number of selected candidates, the accuracy of the model increases; however, it loses benefits of approximation with a large number of candidates.

**Impact of Post-Scoring Selection.** Figure 12a shows the change in model accuracy with the proposed post-scoring selection scheme (Section IV-D). We vary  $T$  (i.e., the threshold for post-scoring selection algorithm) in Figure 12 to identify how  $T$  affects to model accuracy. Here, note

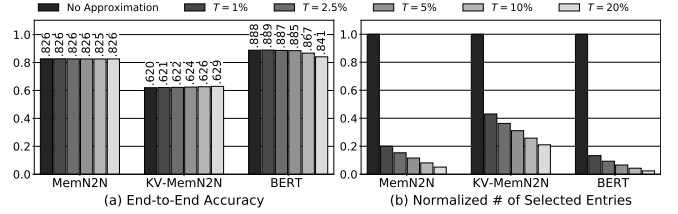


Figure 12. Impact of post-scoring selection schemes on accuracy across varying thresholds.

that an entry is not included for the computation if its post-softmax score would be less than  $T\%$  of the maximal value. Thus, the lower  $T$  indicates more conservative approximation and the higher  $T$  indicates more aggressive approximation. As shown in the figure, relatively high  $T$  (e.g., 10%) can still achieve decent accuracy. This essentially proves our assumption that attention mechanism does not really require all rows to be inspected and any row that would end up with low weight can safely be ignored. Figure 12b shows the normalized number of entries selected in the post-scoring selection scheme. Higher  $T$  results in a lower number of selected entries.

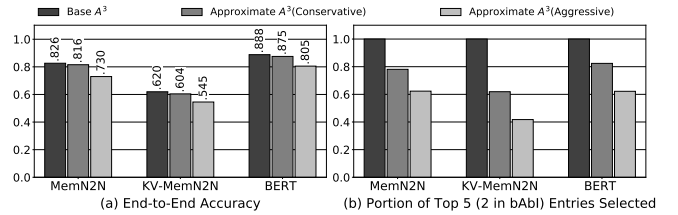


Figure 13. Impact of the approximation scheme on model accuracy across varying workloads.

**Impact of Approximation Scheme.** Figure 13a shows the accuracy change after applying both the proposed candidate selection schemes and the post-scoring selection scheme and Figure 13b shows the portion of true top 2 (bAbI) or top 5 (Wikimovies, SQuAD) entries included after approximation. Here, we evaluate two configurations of our approximation schemes. Approximate (conservative) scheme represents a conservative scheme (with  $M = 1/2n$  and  $T = 5\%$ ), which loses relatively low accuracy (around 1%) but results in a larger selection size during candidate selection and post-scoring selection. On the other hand, approximate (aggressive) scheme represents an aggressive scheme (with  $M = 1/8n$  and  $T = 10\%$ ) loses an extra accuracy (around 8%) but results in a much smaller selection size during candidate selection and post-scoring selection. One of the main strengths of our approach is that  $M$  and  $T$  are configurable. By changing  $M$  and  $T$ , a user always can select the degree of approximation and choose the trade-offs between accuracy and performance/energy efficiency. Figure 13b shows that more aggressive approximation tends to miss some of the true top 2 (bAbI) or top 5 (Wikimovies, SQuAD) entries compared to the conservative scheme. Note that the aggressive approximation configuration may not be practical on its own for its relatively high decrease in accuracy.

In that sense, this configuration is mostly for exploratory purposes. However, one thing to note is that speedups and energy efficiency improvements approximation provide can be translated to improvements in accuracy through the use of larger models. For example, Google BERT has a larger pre-trained model with better accuracy, which naturally spends more time on attention mechanism. The same time/resource-accuracy trade-off is observed in top-performing image classification networks (e.g., Amoebanet [30], NasNet [31]) as well.

**Impact of Quantization Scheme.** To identify the impact of precision in number representation, we quantize the original input to have  $f$  fractional bits (as explained in Section III-B) and then measure its impact on the pipeline. Since our hardware pipeline is carefully designed to avoid precision loss across pipelines, no extra precision loss happens throughout the pipeline. Our experiments show that maintaining a very small number of fractional bits (e.g.,  $f = 4$ -bits) has a negligible impact (i.e., less than 0.1% degradation) on accuracy across all workloads. This proves our assumption that attention mechanism is tolerant to approximation.

### C. Performance Results

**Methodology.** We implement a cycle-level simulator for our proposed accelerator (running at 1GHz) and integrate it into the open-source implementations of our target workloads to evaluate our accelerator’s performance. For comparison, we also profile throughput for attention mechanism processing on Intel Xeon Gold 6128 CPU [32] (used for all workloads) and NVIDIA Titan V (Volta) [33] (only used for BERT since other two workloads did not have a GPU implementation). For CPU and GPU measurements, we tried our best to optimize its throughput following Intel performance optimization guidelines for deep learning workloads [34], [35].

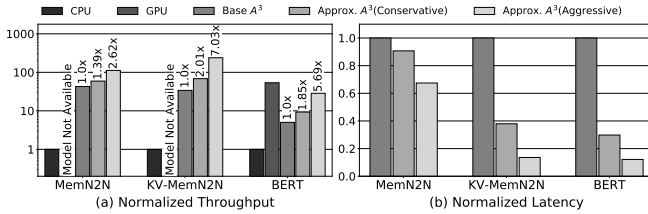


Figure 14. Normalized average throughput/latency of an attention operation for each workload across platforms.<sup>3</sup>

**Throughput.** Figure 14a shows the average throughput of processing an attention operation in the base  $A^3$  and the approximate  $A^3$ , Intel Xeon CPU, and NVIDIA GPU. For each workload, throughput is normalized to that of the CPU. As shown in Figure 14a, both base  $A^3$  and approximate  $A^3$  achieve orders of magnitude higher throughput than Intel Xeon CPU on MemN2N and KV-MemN2N. For BERT,  $A^3$  achieves lower throughput than GPU since BERT’s self-attention mechanism — essentially a batch matrix-matrix multiplication instead of a single matrix-vector multiplication — has easy-to-exploit parallelism. However, since a single  $A^3$

unit consumes multiple orders of magnitudes less energy than the CPU or the GPU (e.g.,  $10^3$ - $10^4\times$ ), it is possible to utilize multiple copies of  $A^3$  for better throughput. Specifically, since BERT’s self-attention mechanism has easy-to-exploit parallelism, using multiple  $A^3$  units can achieve near-perfect scaling behavior. For this reason, it is possible to achieve better throughput than the state-of-the-art GPU by utilizing 6-7 units of approximate (conservative)  $A^3$  together. This is very surprising result considering that  $A^3$  is much smaller than GPUs; this is partly because a large GPU often cannot fully utilize its resources for attention mechanism computation whose matrix size is small and the amount of parallelism is less than what GPU can sustain.

Comparing the base  $A^3$  and two configurations of the approximate  $A^3$  show that approximation enables a larger throughput as well. Note that the throughput increase from approximation is low on MemN2N because of its relatively smaller  $n$ . This can potentially be addressed by utilizing different numbers of modules for each type of module (e.g., use a larger number of candidate selector modules).

**Latency.** Figure 14b shows the average latency of processing an attention operation in the base  $A^3$  and the approximate  $A^3$ . For each workload, latency is normalized to that of the base  $A^3$ . Figure 14 shows both base  $A^3$  and configurations for the approximate  $A^3$  achieve a very low average latency. Furthermore, both approximation-enabled configurations of  $A^3$  achieve significantly better latency than the base  $A^3$ . This is because the approximate  $A^3$  performs a substantially lower number of computations thanks to approximation. Comparing two approximation-enabled configurations shows that aggressive approximation can offer noticeably higher speedup than conservative approximation at a relatively low or moderate accuracy cost shown in Figure 13.

**Preprocessing.** For BERT, the preprocessing happens on the critical path and thus we included the amortized preprocessing overhead (measured on GPU) to  $A^3$  bars (only approximate configurations) in Figure 14. Specifically, BERT utilizes the self-attention mechanism which reuses the same key matrix for  $n$  queries ( $n=320$ ) so the effective overhead per query is only  $1/n$  of the total preprocessing time. This overhead translates to 7% (conservative) or 24% (aggressive) throughput reduction and <11% latency increase. As shown in Figure 14,  $A^3$  achieves significant benefits from approximation even with the (amortized) preprocessing overhead.

### D. Area, Power, Energy and Test Chip

**Methodology.** We implement  $A^3$  with hardware construction language Chisel [36] and compile it to Verilog, and finished all the functional verifications. Then, we synthesize the Verilog code for the 1GHz clock frequency using Synopsys Design Compiler [37] and TSMC 40nm standard cell library.

<sup>3</sup>We show the numbers above each bar, which represent the value normalized to the base  $A^3$  instead of CPU to help readers see the impact of approximation.

Table I  
AREA AND POWER CHARACTERISTICS OF  $A^3$ .

Module	Area (mm <sup>2</sup> )	Dynamic Power(mW)	Static Power(mW)
<b>Modules for Base <math>A^3</math></b>			
Dot Product	0.098	14.338	1.265
Exponent Computation	0.016	0.224	0.053
Output Computation	0.062	50.918	0.070
<b>Modules for Approximation Support</b>			
Candidate Selection	0.277	19.48	5.08
Post-Scoring Selection	0.010	2.055	0.147
<b>SRAM Modules</b>			
Key Matrix (20KB)	0.350	2.901	0.987
Value Matrix (20KB)	0.350	2.901	0.987
Sorted Key Matrix (40KB)	0.919	6.100	2.913
<b>Total</b>			
$A^3$	2.082	98.92	11.502

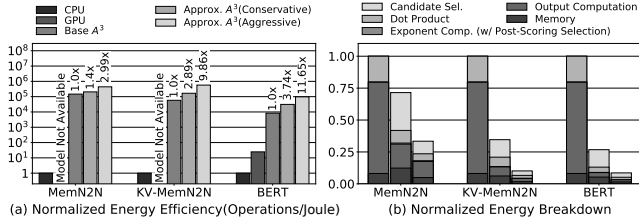


Figure 15. Energy efficiency and energy breakdown of  $A^3$  across workloads <sup>3</sup> For the energy breakdown, three bars represent base  $A^3$ , approximate  $A^3$  (conservative), and approximate  $A^3$  (aggressive) from left to right.

We used  $n = 320$  and  $d = 64$  to fit our largest workload. For number representation (Section III-B), we use  $i = 4$  and  $f = 4$ .

**Area.** As shown in Table I,  $A^3$  utilizes less than 2.082mm<sup>2</sup> area. Note that our CPU baseline Skylake-SP Intel Xeon Core with 14nm process uses a die area of 325mm<sup>2</sup> [38], which is 156 $\times$  larger than a single  $A^3$  unit. Similarly, our baseline GPU Titan V has 815mm<sup>2</sup> die size [39] at a 12nm technology node, which is 391 $\times$  larger than our  $A^3$ . And the effective area difference becomes much larger if we consider that our  $A^3$  is synthesized at 40nm technology.

**Energy and Power.** Table I shows the dynamic and static power usage of  $A^3$  and Figure 15 shows the energy efficiency and energy breakdown of our proposed accelerator running various workloads. For CPU and GPU energy numbers, we assumed their power consumption is equal to their TDPs. Table I shows that  $A^3$  spends less than 100mW when all modules are fully utilized. This is already much lower than that of the Intel Xeon CPU (115W TDP) or NVIDIA Titan V GPU (250W TDP). And when running the real workloads, it consumes even less amount of power than its peak power due to a pipeline imbalance resulting from the approximation. As shown in Figure 15a,  $A^3$ 's low power usage, combined with its high throughput, leads to multiple orders of magnitude energy efficiency improvement compared to the conventional hardware (e.g., over 10<sup>4</sup> $\times$  on CPU 10<sup>3</sup> $\times$  on GPU), despite the fact that our accelerators are synthesized at a 40nm technology node. In addition, comparing the base  $A^3$  and the approximate  $A^3$  demonstrates that approximation leads to a further energy saving by avoiding unnecessary computation.

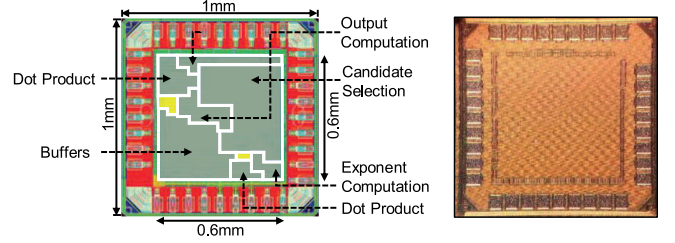


Figure 16. Post-layout image (left) and die photo (right) of a prototype  $A^3$  chip

Lastly, Figure 15b shows that base  $A^3$  spends most of its energy on the output computation module due to its large register structures. On the other hand, approximate  $A^3$  spends most energy on candidate selection module because other modules are not heavily utilized once the candidate selection module substantially reduces the number of rows to process.

**Test Chip.** We have taped out a scaled-down version of the  $A^3$  test chip in TSMC 40nm Low Power (LP) technology with standard cells. The test chip implements the full functionality of  $A^3$  with approximation but is scaled down to fit in a 1mm<sup>2</sup> silicon die, including I/O pads, host interface, and other peripherals. The core area is 0.36mm<sup>2</sup>, which is sized just enough to run the smallest model used for evaluation (MemN2N). Figure 16 shows a post-layout image and a die photo. The test chip communicates with the host ARMv8 CPU via custom JTAG-like serial interface over 3.3V general-purpose I/O (GPIO) pins. To drive the test chip, we have also written a host device driver as well as a Python-based software testing environment that can run MemN2N. We have verified the functionality of the chip to confirm that everything works as intended.

## VII. RELATED WORK

**Attention Mechanism.** Attention mechanism in natural language processing is used for translation [7], [10], question answering [8], [9], [19], [40], [41], [42], language inference [13], [43], summarization [44], [45], document classification [46], [47], etc. In addition to natural language processing, the attention mechanism has also been used in computer vision tasks. Examples include visual (and multi-modal) question-answering tasks [5], [6], [48], image caption generation [4], [49], image classification [50], [51], [52], [53], action recognition [54], saliency detection [55], and so on. Attention mechanism can also be used as a long-term memory mechanism for NNs. Neural Turing Machine [11] and Differentiable Neural Computer [12] from Google Deepmind focus on this to enable a NN to solve a complicated task which requires an explicit, long-term memory. Our work applies to many of these NN models.

**Approximate Similarity Search.** Similarity search is an important technique in other domains such as recommendation systems. There are several prior works relevant to our work to avoid an exhaustive linear search during a similarity

search. For example, some approaches [56], [57], [58] utilize a variant of locality sensitive hashing, a tree-structure, or a clustering algorithm to cluster (or hash) items to different groups before the query arrives. On the other hand, a greedy approach [25] similar to ours performs a greedy iterative search. A few prior works exploit the idea of performing an approximated similarity search for attention mechanisms and implemented them in software [59], [60]. Our work presents a new hardware-friendly algorithm and designs a hardware accelerator to realize the approximation potential in the similarity search.

**Hardware NN Accelerator.** There exist various hardware accelerators for NN computations. CNN accelerators for custom ASIC or FPGA device [1], [2], [3], [61], [62], [63], [64], [65], [66], [67] and RNN accelerators [3], [68], [69], [70] are proposed to accelerate NN processing by exploiting massive parallelism and optimizing data movements. There are hardware accelerators exploiting sparsity [68], [71], [72], [73], [74] for a specific context of CNN/RNNs to improve their efficiency; however, these works mostly focus on sparsity (i.e., a large portion of its data is zero) in weights and activation of CNNs or RNNs while our accelerator focuses on the approximation potential of the attention mechanism (operating on a dense matrix).

**Hardware-supported NN Op. Approximation.** Several works have applied the approximate computing concept to the neural network operations to reduce the amount of computation. Specifically, SnaPEA [75] exploits the unique characteristics of the neural network convolution operation and presents a hardware accelerator that can benefit from an approximation scheme exploiting such characteristics. Similarly, Raha et al [76] presents an RnR (reduce and rank) accelerator that can be used to reduce the energy consumption on certain neural network operations exhibiting the reduce-and-rank pattern through approximation. Finally, there exist several previous works that focus on accelerating MAC (multiply and accumulate) operations prevalent in neural networks through approximate MAC unit [77], [78], [79].

## VIII. CONCLUSION

Neural network (NN) has been a popular target for hardware accelerators for its wide applicability, a large amount of computation, massive parallelism, and static computation pattern. However, the presence of an existing accelerator does not necessarily preclude the need for another accelerator for NN primitives. In fact, when other NN primitives (e.g., CNNs, RNNs) are optimized, it is critical to accelerate relatively less optimized portion according to Amdahl's Law. Our work identifies the importance of the emerging NN primitive — attention mechanism — and accelerates it with software-hardware co-design to achieve orders of magnitude energy efficiency improvement over the conventional hardware.

## ACKNOWLEDGMENTS

This work was supported by a research grant from SK Hynix and National Research Foundation of Korea grant funded by the Ministry of Science and ICT (PE Class Heterogeneous High Performance Computer Development, NRF-2016M3C4A7952587, and Nano-Material Technology Development Program, NRF-2016M3A7B4909668), and by IDEC (CAD tools). The authors thank Kwangho Lee, Han-Gon Ko, Soyeong Shin, Yejun Ko, Dongsuk Jeon, and Jintae Kim for their help with troubleshooting our  $A^3$  test chip design. Jae W. Lee is the corresponding author.

## REFERENCES

- [1] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2014.
- [2] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, 2017.
- [3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-I. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellman, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture, ISCA*, 2017.
- [4] K. Xu, J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *International Conference on Machine Learning, ICML*, 2015.
- [5] M. Tapaswi, Y. Zhu, R. Stiefelhofen, A. Torralba, R. Urtasun, and S. Fidler, "MovieQA: Understanding Stories in Movies through Question-Answering," in *Conference on Computer Vision and Pattern Recognition, CVPR*, 2016.
- [6] S. Na, S. Lee, J. Kim, and G. Kim, "A read-write memory network for movie story understanding," in *International Conference on Computer Vision, ICCV*, 2017.
- [7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, E. Kaiser, and I. Polosukhin, "Attention is all you need," in *International Conference on Neural Information Processing Systems, NIPS*, 2017.
- [8] S. Sukhbaatar, J. Weston, R. Fergus et al., "End-to-end memory networks," in *International Conference on Neural Information Processing Systems, NIPS*, 2015.
- [9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *CoRR*, 2018.
- [10] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, 2014.
- [11] A. Graves, G. Wayne, and I. Danihelka, "Neural Turing machines," *CoRR*, 2014.
- [12] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwinska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu, and D. Hassabis, "Hybrid computing using a neural network with dynamic external memory," *Nature*, 2016.
- [13] A. P. Parikh, O. Täckström, D. Das, and J. Uszkoreit, "A decomposable attention model for natural language inference," in *Empirical Methods in Natural Language Processing, EMNLP*, 2016.
- [14] Z. Lin, M. Feng, C. N. dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," in *International Conference on Learning Representations, ICLR*, 2017.
- [15] J. Weston, A. Bordes, S. Chopra, A. M. Rush, B. van Merriënboer, A. Joulin, and T. Mikolov, "Towards AI-complete question answering: A set of prerequisite toy tasks," in *International Conference on Learning Representations, ICLR*, 2016.
- [16] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *International Conference on Neural Information Processing Systems, NIPS*, 2013.
- [17] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Conference on Empirical Methods in Natural Language Processing, EMNLP*, 2014.
- [18] E. Grave, T. Mikolov, A. Joulin, and P. Bojanowski, "Bag of tricks for efficient text classification," in *Conference of the European Chapter of the Association for Computational Linguistics, EACL*, 2017.

- [19] A. Miller, A. Fisch, J. Dodge, A.-H. Karimi, A. Bordes, and J. Weston, "Key-value memory networks for directly reading documents," in *Empirical Methods in Natural Language Processing*, EMNLP, 2016.
- [20] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Empirical Methods in Natural Language Processing*, EMNLP, 2016.
- [21] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, 2015.
- [22] "Torch 7," <https://github.com/torch/torch7>.
- [23] S. Arora, "Word embeddings: Explaining their properties," <https://www.offconvex.org/2016/02/14/word-embeddings-2/>.
- [24] Z. Yin and Y. Shen, "On the dimensionality of word embedding," in *International Conference on Neural Information Processing Systems*, NIPS, 2018.
- [25] H. Yu, C. Hsieh, Q. Lei, and I. S. Dhillon, "A greedy approach for budgeted maximum inner product search," in *International Conference on Neural Information Processing Systems*, NIPS, 2017.
- [26] Facebook, "The bAbI project," <https://research.fb.com/downloads/babi>, 2015.
- [27] V. Khuc, "MemN2N-babi-python," <https://github.com/vinhkhuc/MemN2N-babi-python>, 2017.
- [28] Facebook, "Key-value memory networks for directly reading documents," <https://github.com/facebook/MemNN/tree/master/KVmemnn>, 2017.
- [29] G. Research, "Tensorflow code and pre-trained models for BERT," <https://github.com/google-research/bert>, 2018.
- [30] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *AAAI Conference on Artificial Intelligence*, AAAI, 2019.
- [31] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *CoRR*, 2017. [Online]. Available: <http://arxiv.org/abs/1707.07012>
- [32] Intel, "Intel Xeon gold 6128 processor," <https://ark.intel.com/products/120482/Intel-Xeon-Gold-6128-Processor-19-25M-Cache-3-40-GHz->, 2017.
- [33] NVIDIA, "NVIDIA TITAN V," <https://www.nvidia.com/en-us/titan/titan-v/>, 2018.
- [34] "Tips to improve performance for popular deep learning frameworks on CPUs," <https://software.intel.com/en-us/articles/tips-to-improve-performance-for-popular-deep-learning-frameworks-on-multi-core-cpus>.
- [35] "Intel software optimization for Torch," <https://github.com/intel/torch>.
- [36] "Chisel," <https://chisel.eecs.berkeley.edu>.
- [37] "Synopsys Design Compiler," <https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html>.
- [38] Wikichip, "Skylake (server) - microarchitectures - Intel," [https://en.wikichip.org/wiki/intel/microarchitectures/skylake\\_\(server\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)), 2019.
- [39] NVIDIA, "Inside Volta: The world's most advanced data center GPU," <https://devblogs.nvidia.com/inside-volta/>, 2018.
- [40] K. M. Hermann, T. Kociský, E. Grefenstette, L. Espeholt, W. Kay, M. Suleyman, and P. Blunsom, "Teaching machines to read and comprehend," in *Advances in Neural Information Processing Systems*, NIPS, 2015.
- [41] M. J. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi, "Bidirectional attention flow for machine comprehension," in *International Conference on Learning Representations*, ICLR, 2017.
- [42] B. Wang, K. Liu, and J. Zhao, "Inner attention based recurrent neural networks for answer selection," in *Annual Meeting of the Association for Computational Linguistics*, ACL, 2016.
- [43] T. Rocktäschel, E. Grefenstette, K. M. Hermann, T. Kociský, and P. Blunsom, "Reasoning about entailment with neural attention," in *International Conference on Learning Representations*, ICLR, 2016.
- [44] A. M. Rush, S. Chopra, and J. Weston, "A neural attention model for abstractive sentence summarization," in *Conference on Empirical Methods in Natural Language Processing*, EMNLP, 2015.
- [45] S. Chopra, M. Auli, and A. M. Rush, "Abstractive sentence summarization with attentive recurrent neural networks," in *The Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL, 2016.
- [46] Z. Yang, D. Yang, C. Dyer, X. He, A. J. Smola, and E. H. Hovy, "Hierarchical attention networks for document classification," in *The Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL, 2016.
- [47] N. Pappas and A. Popescu-Belis, "Multilingual hierarchical attention networks for document classification," in *International Joint Conference on Natural Language Processing*, IJCNLP, 2017.
- [48] K. Kim, M. Heo, S. Choi, and B. Zhang, "Deepstory: Video story QA by deep embedded memory networks," in *International Joint Conference on Artificial Intelligence*, IJCAI, 2017.
- [49] Q. You, H. Jin, Z. Wang, C. Fang, and J. Luo, "Image captioning with semantic attention," in *Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.
- [50] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent models of visual attention," in *International Conference on Neural Information Processing Systems*, NIPS, 2014.
- [51] F. Wang, M. Jiang, C. Qian, S. Yang, C. Li, H. Zhang, X. Wang, and X. Tang, "Residual attention network for image classification," in *Conference on Computer Vision and Pattern Recognition*, CVPR, 2017.
- [52] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial transformer networks," in *International Conference on Neural Information Processing Systems*, NIPS, 2015.
- [53] I. Bello, B. Zoph, A. Vaswani, J. Shlens, and Q. V. Le, "Attention augmented convolutional networks," *CoRR*, 2019. [Online]. Available: <http://arxiv.org/abs/1904.09925>
- [54] S. Sharma, R. Kiros, and R. Salakhutdinov, "Action recognition using visual attention," in *International Conference on Learning Representations*, ICLR, 2016.
- [55] J. Kuen, Z. Wang, and G. Wang, "Recurrent attentional networks for saliency detection," in *Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.
- [56] A. Shrivastava and P. Li, "Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS)," in *International Conference on Neural Information Processing Systems*, NIPS, 2014.
- [57] P. Ram and A. G. Gray, "Maximum inner-product search using cone trees," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD, 2012.
- [58] A. Auvolat and P. Vincent, "Clustering is efficient for approximate maximum inner product search," *CoRR*, 2015.
- [59] J. Rae, J. J. Hunt, I. Danihelka, T. Harley, A. W. Senior, G. Wayne, A. Graves, and T. Lillicrap, "Scaling memory-augmented neural networks with sparse reads and writes," in *International Conference on Neural Information Processing Systems*, NIPS, 2016.
- [60] S. Chandar, S. Ahn, H. Larochelle, P. Vincent, G. Tesauero, and Y. Bengio, "Hierarchical memory networks," *CoRR*, 2016.
- [61] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2014.
- [62] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *International Symposium on Computer Architecture*, ISCA, 2015.
- [63] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2015.
- [64] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2016.
- [65] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale DNN processor for real-time AI," in *International Symposium on Computer Architecture*, ISCA, 2018.
- [66] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded FPGA platform for convolutional neural network," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA, 2016.
- [67] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA, 2015.
- [68] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA, 2017.
- [69] J. Park, J. Kung, W. Yi, and J. Kim, "Maximizing system performance by balancing computation loads in LSTM accelerators," in *Design, Automation Test in Europe*, DATE, 2018.
- [70] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "DeltaRNN: A power-efficient Recurrent Neural Network accelerator," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA, 2018.
- [71] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference Engine on compressed deep neural network," in *International Symposium on Computer Architecture*, ISCA, 2016.
- [72] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *International Symposium on Computer Architecture*, ISCA, 2016.
- [73] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2016.
- [74] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *International Symposium on Computer Architecture*, ISCA, 2017.
- [75] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *International Symposium on Computer Architecture*, ISCA, 2018.
- [76] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan, "Energy-efficient reduce-and-rank using input-adaptive approximations," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 25, no. 2, pp. 462–475, Feb 2017.
- [77] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in *Design, Automation Test in Europe*, DATE, March 2011, pp. 1–6.
- [78] A. Raha and V. Raghunathan, "Qlut: Input-aware quantized table lookup for energy-efficient approximate accelerators," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, 2017.
- [79] M. Masadeh, O. Hasan, and S. Tahar, "Input-conscious approximate multiply-accumulate (mac) unit for energy-efficiency," *IEEE Access*, vol. 7, 2019.