2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00060

ELSA: Hardware-Software Co-design for Efficient, Lightweight Self-Attention Mechanism in Neural Networks

Tae Jun Ham*, Yejin Lee*, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, Jae W. Lee

Seoul National University

{taejunham, yejinlee, andyseo247, soosungkim, hyunjichoi, miguel92, jaewlee}@snu.ac.kr

***Abstract*—The self-attention mechanism is rapidly emerging as one of the most important key primitives in neural networks (NNs) for its ability to identify the relations within input entities. The self-attention-oriented NN models such as Google Transformer and its variants have established the state-of-the-art on a very wide range of natural language processing tasks, and many other self-attention-oriented models are achieving competitive results in computer vision and recommender systems as well. Unfortunately, despite its great benefits, the self-attention mechanism is an expensive operation whose cost increases quadratically with the number of input entities that it processes, and thus accounts for a significant portion of the inference runtime. Thus, this paper presents ELSA (Efficient, Lightweight Self-Attention), a hardware-software co-designed solution to substantially reduce the runtime as well as energy spent on the self-attention mechanism. Specifically, based on the intuition that not all relations are equal, we devise a novel approximation scheme that significantly reduces the amount of computation by efficiently filtering out relations that are unlikely to affect the final output. With the specialized hardware for this approximate self-attention mechanism, ELSA achieves a geomean speedup of 58.1× as well as over three orders of magnitude improvements in energy efficiency compared to GPU on self-attention computation in modern NN models while maintaining less than 1% loss in the accuracy metric.**

***Index Terms*—attention, hardware accelerator, neural network**

I. Introduction

The attention mechanism is a relatively recently introduced neural network primitive emerging as one of the most influential ideas in the deep learning community. This mechanism allows neural networks (NNs) to identify the information relevant to the specific input and decide where to *attend*. For example, this mechanism can be used to identify the portion of the information that is relevant to the query from an extensive collection of data (e.g., knowledgebase, image). One specific case of the attention mechanism is the self-attention mechanism, where the attention mechanism is used to identify the relations among input data. Since its first introduction in the seminal paper *Attention Is All You Need* [82] that presents the Transformer NN architecture, the self-attention mechanism has been widely used to lead the breakthroughs in the field of natural language processing (NLP). Self-attention-oriented NLP models from major AI companies such as Google BERT [18], Facebook RoBERTa [52], OpenAI GPT2/3 [5], [64], NVIDIA MegatronLM [71], and Microsoft Turing-NLG [70] established the state-of-the-art results for various NLP tasks. In addition to natural language processing, the self-attention is widely used for computer vision [3], [15], [61], [91] and recommendation systems [20], [43], [73], [78], [94], [95] as well.

Despite its strong potential, the self-attention is a costly operation. This operation identifies the relations among input data, and thus it requires the amount of computation that quadratically increases with the number of entities involved in this operation. Due to this high cost, the self-attention accounts for a substantial amount of time and energy consumption in many self-attention-oriented NN models, which becomes a limiting factor for deployment. For example, many existing NLP models such as Google BERT limit the self-attention to be applied for up to 512 tokens (e.g., words) to avoid the excessive performance and energy overhead. When the input text has more than 512 tokens, the input text needs to be divided into multiple segments (each with up to 512 tokens), and the self-attention is separately applied for each segment. Unfortunately, such a scheme makes NLP models unable to capture the relation between two tokens that do not belong to the same segment.

Thus, we present a hardware-software co-designed solution for **e**fficient, **l**ightweight **s**elf-**a**ttention, called ELSA. Like other hardware accelerators, ELSA exploits hardware specialization to improve the performance and energy efficiency over the conventional hardware like GPU. However, rather than merely porting a provided algorithm to the hardware, our work takes a step further and proposes a novel approximate self-attention scheme as well as a specialized hardware architecture for it. Based on the intuition that irrelevant relations can be effectively filtered out by computing approximate similarity, ELSA substantially reduces computational waste in a self-attention operation. Unlike conventional hardware such as GPUs, which fails to benefit from the proposed approximation, our specialized hardware directly translates this reduction to further improve performance and energy efficiency. This reduced cost of self-attention enables us to apply the self-attention to larger data, which can uncover distant relations within the data that today's models cannot handle effectively. In summary, our work makes the following contributions:

- We present a novel approximate self-attention scheme which exploits approximate, hardware-friendly similarity computation to substantially reduce the amount of computation in the self-attention operation during inference.
- We design ELSA, a specialized hardware accelerator that

* These authors contributed equally to this work.

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00060

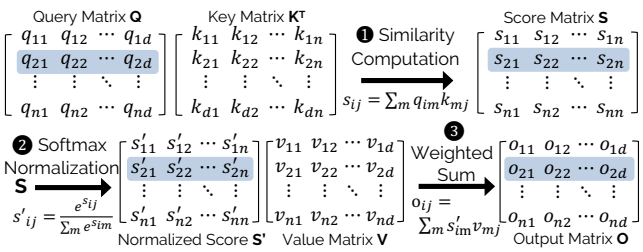

Fig. 1. Self-attention mechanism.

exploits opportunities for approximation and parallelism in the self-attention operation to significantly improve its performance and energy efficiency.

- We evaluate ELSA with multiple representative self-attention-oriented neural network models to demonstrate that our accelerator can achieve substantial performance and energy efficiency gains over the conventional hardware.

II. Background and Motivation

A. Self-Attention Mechanism

Computation. Self-attention is essentially an operation that identifies the relations within the input entities, and Fig. 1 presents the required computations for it. For each input entity, three different d-dimensional dense vector representations need to be provided: query, key, and value. Assuming the input has n entities, n vectors of d dimension are grouped to form the query matrix ($\mathbf{Q}$), the key matrix ($\mathbf{K}$), and the value matrix ($\mathbf{V}$) each having $n \times d$ dimensions. Throughout the paper, we call row vectors of these matrices as queries, keys, and values, respectively. ❶ The very first step of the self-attention is similarity computation, which computes the dot product similarity between each query vector and each key vector. For this purpose, the query matrix is multiplied with the transposed key matrix ($\mathbf{QK}^T$). This results in $n \times n$ matrix (i.e., attention score matrix $\mathbf{S}$), where s_{ij} represents the similarity (i.e., dot product) between the ith query and the jth key vector. Note that some implementations often called *scaled* self-attention divide the resulting matrix by a scalar constant. ❷ The second step is the softmax normalization for each row of the attention score matrix ($s'_{ij} = e^{s_{ij}} / \sum_m e^{s_{im}}$). ❸ The final step computes the output of this operation for each query vector by computing the weighted sum of value matrix ($\mathbf{V}$) rows utilizing the corresponding normalized attention scores as weights. This is equivalent to multiplying the matrix $\mathbf{S}'$ to $\mathbf{V}$ (because $\text{row}_i(O) = \sum_{m=1}^{n} s'_{im} \cdot \text{row}_m(V) \Leftrightarrow o_{ij} = \sum_{m=1}^{n} s'_{im} \cdot v_{mj}$). The result of this is an output matrix ($\mathbf{O}$) where ith row represents the d-dimensional vector that represents the outcome of the self-attention operation for the ith input entity.

Application-Level Description. Each input entity (e.g., a word in a text) gets three different vector representations (query, key, and value). Then, each entity uses its query representation to find the set of other entities that are the most relevant to the current entity. For this purpose, the dot product similarity between the query representation (of the current entity) and the key representation of other entities are computed, then softmax-normalized. Since the softmax function is a differentiable approximation of the `argmax` function, this step is effectively selecting a few most similar entities to the current entity. Finally, the value representations of the selected entries are summed up utilizing the softmax-normalized attention score as the weights. This process is repeated for each input entity, and the output is passed to the next layer in a NN model. In NLP models, this operation is used to identify the specific semantic relation between tokens (e.g., words). For example, a self-attention head (i.e., sub-layer) in a layer lets the *direct objects* to attend their *verbs*, or *noun modifiers* to attend their *nouns* [14].

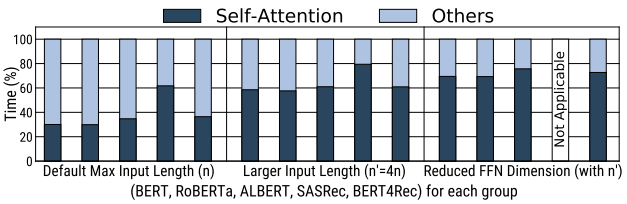

Fig. 2. Portion of the runtime spent for the self-attention mechanism.

B. Cost of Self-Attention Mechanism

As explained before, the self-attention mechanism consists of three steps. The first matrix multiplication requires n^2d multiply-and-accumulate (MAC) operations (since it multiplies $n \times d$ matrix with $d \times n$ matrix). The second softmax operation requires n^2 exponent operations, and the final matrix multiplication also requires n^2d MAC operations ($n \times n$ matrix is multiplied with $n \times d$).

Fig. 2 shows the portion of the runtime spent on self-attention in popular NN models. We run SQuADv1.1 dataset [68] for NLP models (BERT, RoBERTa, ALBERT) and MovieLens-1M [33] for recommendation models (SASRec, BERT4Rec) on NVIDIA V100 GPU [56]. The details of each workload are available in Section V-A. The left side of the figure shows that the self-attention accounts for a significant portion (about 38%) of the runtime across many existing self-attention-oriented NN models. Furthermore, the figure also shows that increasing n further than the published model parameter, say, by 4×, makes the self-attention to account for the even larger portion (about 64%) of the model runtime. Finally, note that several recent research works on NLP models suggest that the portion of the self-attention is going to increase even further. For example, a recent research [88] demonstrates that extraneous dimensions in the feedforward layers are unnecessary and removing them hardly affects the model accuracy while significantly reducing the runtime of the feedforward layers in Transformer-style models. The right side of the figure shows that the runtime portion of the self-attention on these models reach about 73% when the feedforward layer dimension is reduced by 4× [88]. In addition, several recent proposals also investigate the idea of replacing the feedforward layer in Transformer-style models with the self-attention [48], [76]) for better model accuracy. Such trends will make the self-attention to take an even larger portion of the total model runtime in the future.

C. Opportunities for Approximation

All three input matrices ($\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$) of the self-attention are dense. In other words, they mostly consist of nonzero

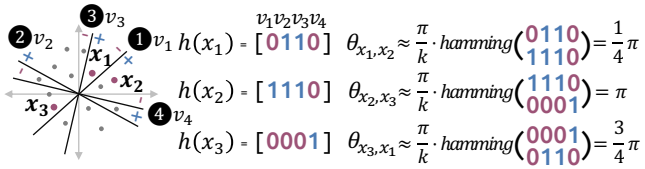

Fig. 3. Visualization of sign random projection (SRP)

elements. However, not all elements of these matrices contribute equally to the output. This is because the softmax operation maps most of the values in the attention score matrix ($\mathbf{S}$) to zeros or near-zero values except for the few largest values of the row. It effectively makes $\mathbf{S}'$ a sparse matrix with many near-zero values, and hence the final matrix $\mathbf{S}'\mathbf{V}$ as well. Simply performing the sparse matrix multiplication for the second matrix multiplication ($\mathbf{S}'\mathbf{V}$) does not completely mitigate the high cost of the self-attention, since the first matrix multiplication $\mathbf{Q}\mathbf{K}^T$ still requires n^2d multiplications. To fully exploit the approximation potential in the self-attention, there should be a way to identify the set of keys (for each query) that will result in large attention scores, without performing expensive n^2d multiplications.

Our intuition is that it is possible to achieve this by performing an approximate and lightweight similarity computation. Instead of performing d multiplications and the softmax operation to identify whether the ith query and the jth key will be relevant or not (i.e., if s'_{ij} will be near-zero or not), an approximate similarity can be computed to quickly filter out a key that is expected to be not very relevant to the query. If this approximate similarity computation indicates that they are potentially relevant, the exact dot product similarity is computed. If not, this similarity computation and all subsequent computations can be skipped. With this scheme, it is possible to eliminate a large amount of computational waste, and our specialized hardware can translate this reduction into performance improvement as well as energy savings.

III. Approximate Self-Attention

A. Overview

Our approximate self-attention scheme consists of three sub-operations. First, we estimate the angle between two vectors (e.g., a key and a query) with minimal computation by utilizing the concise representations (e.g., k-bits hash, also called binary embedding) of the key and the query (Section III-B, Section III-C). Second, an estimated angle is utilized to compute the approximate similarity between a query and a key (Section III-D), based on the intuition that dot product is directly proportional to the cosine of the angle between two vectors. Finally, the approximate similarity is compared with a certain threshold (Section III-E) to identify whether a specific key is relevant to the query or not.

B. Binary Hashing for Angular Distance

Sign Random Projection. Sign random projection (SRP) [7] is a well-known technique that effectively maps each input vector to a binary hash vector in a way that allows the original angular distance between two vectors to be efficiently estimated with the two corresponding binary hash vectors. This mapping is often utilized for the locality-sensitive hashing schemes, but our work focuses on its use as an efficient estimator for the angular distance.

For this process, a random d-dimensional vector v is initialized by setting each of its component to a value sampled from normal distribution $N(0, 1)$. Then, for an input vector x, the hash bit value of 1 is assigned if $v \cdot x \geq 0$ and assigned 0 otherwise. This is repeated for k times with k random vectors $v_1, ...v_k$ to construct k-bits binary hash $h(x)$ for the input vector x. Formally, the hash function is defined as follows.

$$h(x) = (h_{v1}(x), h_{v2}(x), ...h_{vk}(x)) \text{ where } h_v(x) = sign(v \cdot x)$$

Here, $sign(x)$ is a function whose value is 1 if $x \geq 0$ and 0 otherwise. It is proven that the Hamming distance between hashes of the vector x and y (i.e., $hamming(h(x), h(y))$) is an unbiased estimator of their angular distance [7]. Intuitively, if two vectors are on the same side for many of the random hyperplanes each defined by one of k random vectors $v_1, ...v_k$, they are more likely to have a smaller angle. For example, Fig. 3 shows that x_1 and x_2 is on the same side of three random hyperplanes out of four, and thus have a small hamming distance as well as angular distance. The following equation is used to estimate the angle between vector x and y [7].

$$\theta_{x,y} \approx \frac{\pi}{k} \cdot hamming(h(x), h(y))$$

Our work, in fact, employs the slight variant of SRP that utilizes the k orthogonal vectors generated with the modified Gram-Schmidt Process [86]. Utilizing the orthogonal vectors prevents two or more random vectors from pointing to a similar direction, which leads to the unnecessary emphasis on that specific direction. This method is proven to reduce the error of the angular distance approximation [40].

Angle Correction. The estimated angle computed from the hamming distance is not biased, but still has errors. For this reason, if we simply utilize this estimator without any correction, the estimated angles will be larger than the true angle in about half of the cases. Since overestimating the angle (i.e., underestimating the similarity between two vectors) can result in our scheme to miss the keys that have relations with the query, we subtract the bias θ_{bias} to this estimator. Specifically, we set θ_{bias} to be the 80th percentile error of this estimator so that subtracting this bias from the angle makes this estimator underestimate angles in 80% of the cases. The 80th percentile error is obtained by experiments on a synthetic dataset with standard random normal vectors. For a specific case $d = 64$ and $k = 64$, θ_{bias} is 0.127.

C. Efficient Hash Computation

Cost of Hash Computation. To obtain the k-bits hash value for a d-dimensional vector x, a $k \times d$ orthogonal matrix (i.e., a matrix whose row vectors are unit vectors orthogonal to each other) is multiplied to x, and then each element is assigned a hash bit (i.e., 1 if it is positive; 0 if not). With this scheme, computing the hash values for n vectors requires

ndk multiplications (as well as $n(d-1)k$ additions), and since our scheme requires computing hashes for all queries and keys, the total number of multiplications required for hash computation is $2ndk$. This cost is negligible compared to $2n^2d$ (cost of dot product similarity computation and value matrix computation) when $n \gg k$. However, at least for current neural networks with the limited n (e.g., 128 for small models), this is not always the case. To minimize the amount of computation for hash computation, our work exploits Kronecker product, a technique to efficiently compute the matrix multiplication using orthogonal matrices [22], [93].

Kronecker Product. The key intuition of our approach is that we can utilize a structured orthogonal matrix for hash computation. Specifically, we utilize an orthogonal matrix which can be computed by the *Kronecker product* of smaller matrices. A Kronecker product of a $m \times n$ matrix $\mathbf{A}$ and $p \times q$ matrix $\mathbf{B}$ produces the $pm \times qn$ matrix as shown below.

$$\text{Kronecker Product: } \mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \dots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \dots & a_{mn}\mathbf{B} \end{bmatrix}$$

It is well known that Kronecker product of orthogonal matrices results in an orthogonal matrix. Thus, it is possible to obtain the $k \times d$ orthogonal matrix through Kronecker products of smaller orthogonal matrices. This characteristic allows us to utilize the technique [22], [93] to efficiently compute the hash value of the vector x, which is obtained by computing $\mathbf{A}x$.

$$\mathbf{A}x = (\mathbf{A}_1 \otimes \mathbf{A}_2)x = (\mathbf{A}_1 x\texttt{.reshape(8,8)}\mathbf{A}_2^T)\texttt{.reshape(64)}$$

Efficient Computation with Kronecker Product. Fig. 5 visualizes an example case of computing matrix $\mathbf{A}x$ with much fewer computations for a 4×4 matrix $\mathbf{A}$, which is represented as Kronecker product of two 2×2 matrices $\mathbf{A_1}$ and $\mathbf{A_2}$. Similarly, the above equation shows the case for $k = d = 64$ where the 64×64 matrix $\mathbf{A}$ is represented as Kronecker product of two matrices. Here, `x.reshape(8,8)` represents the operation of reshaping 64-dimensional vector x to a 8×8 matrix by dividing the vector by 8 slices and stacking them. With this technique, the amount of multiplications involved in this operation is now reduced to 1024 (i.e., $2d^{3/2}$) from 4096 (i.e., d^2).

$$\begin{aligned}\mathbf{A}x &= (\mathbf{A}_1 \otimes \mathbf{A}_2 \otimes \mathbf{A}_3)x \\ &= (\mathbf{A}_2(x\texttt{.reshape(4,4,4)}\mathbf{A}_3^T)^{T(0,2)}\mathbf{A}_1^T)^{T(0,2)}\texttt{.reshape(64)}\end{aligned}$$

Similarly, the technique can be applied to obtain orthogonal matrix A by computing Kronecker product of three smaller 4×4 matrices $\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3$ using the above equation. Here, $T(0,2)$ means the tensor transpose which maps element with index (i,j,k) to (k,j,i). With this scheme, three batched (with batch size = 4) 4×4 multiplications are required to compute Ax. In other words, this requires a total of twelve 4×4 matrix multiplications which involves 768 (i.e., $3d^{4/3}$) multiplications. Note that the explained efficient computation mechanism also works for cases where $k \neq d$ or A is not a square matrix [93].

D. Approximate Self-attention Algorithm

Fig. 4 illustrates our approximate self-attention algorithm. Below, we explain each sub-operation of the approximate self-attention algorithm in detail.

Preprocessing. ❶ At the beginning, k-bits hash values for keys (Section III-B) are computed with the efficient hash computation scheme (Section III-C). At the same time, the norm of each key is computed and stored as well. This preprocessing requires $3nd^{4/3}$ multiplications for the hash computation and nd multiplications as well as n square root computations for the norm computation. Note that it is possible to compute query hashes during this phase. However, for now, we assume that the query hash is computed when that query is processed so that it matches well with the hardware architecture explained in the next section.

Approximate Similarity Computation. Once the preprocessing ends, the approximate dot product similarity between a query and each key needs to be computed to determine whether they are relevant or not. For a query (Q_x) and each key ($K_y \in \{K_1, ...K_n\}$), the following computations are performed. ❶ First, the query hash value $h(Q_x)$ is obtained using the efficient computation scheme in Section III-C. ❷ Second, the Hamming distances between a query hash and all keys are computed. ❸ Third, these Hamming distances are translated to angles θ_{Q_x,K_y} for all $1 \leq y \leq n$ using the equation in Section III-B, and the θ_{bias} is applied. ❹ Fourth, the cosine function is applied to each of these approximate angles, and then ❺ the corresponding key norm is multiplied to each of them. Note that the resulting value is the estimate of the dot product between the normalized query and the key, which represents the (query-normalized) similarity of those two vectors. The following equations illustrate this relation.

$$\begin{aligned}&Sim(Q_x/\|Q_x\|, K_y) = (Q_x/\|Q_x\|) \cdot K_y = \|K_y\| \cos(\theta_{Q_x,K_y}) \\ &\approx \|K_y\| \ \cos\Big(\max(0, \frac{\pi}{k} \cdot hamming(h(Q_x), h(K_y)) - \theta_{bias})\Big)\end{aligned}$$

❻ Finally, once the above values are computed, we inspect these values and compare them with a constant threshold to determine whether these values are relevant to the query or not. The method to determine this threshold is explained in the next subsection. ❼ At this point, the candidates for the current query have been selected, and the next query is processed (starting from step ❶). Each approximate similarity computation between a key and a query involves i) single Hamming distance computation, ii) a multiplication ($\frac{\pi}{k}$) and a subtraction (θ_{bias}), iii) a cosine function, iv) and another multiplication ($\|K_y\|$). This cost is substantially lower than d multiplications required to compute the exact dot-product similarity. Furthermore, Section IV-C shows we can avoid some of these computations in hardware using a lookup table.

E. Candidate Selection Threshold

Motivation. There can be several different ways to filter out irrelevant keys for a particular query based on the approximate similarity. One possible way is to sort the score and select a certain number of top-scoring elements. However, sorting has

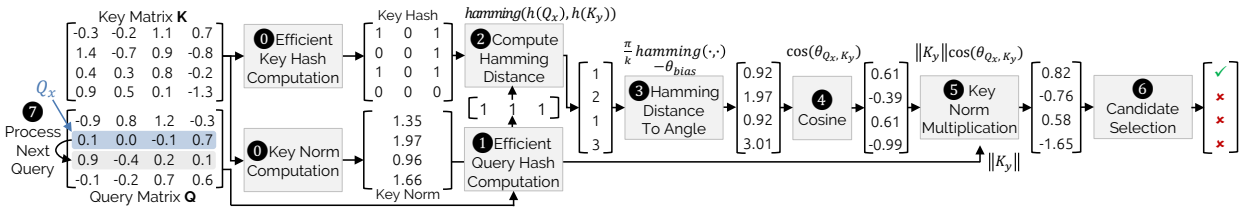

Fig. 4. Approximate Self-attention Algorithm

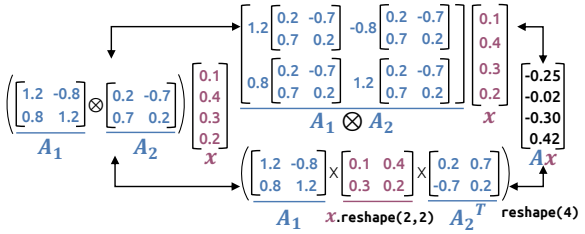

Fig. 5. An example of efficient computation with Kronecker Product.

$nlogn$ time complexity and is difficult to efficiently implement in hardware, especially when n is large. For these reasons, our work focuses on filtering out potentially irrelevant keys by comparing those keys' approximate (query-normalized) similarities with a pre-defined threshold. One major issue is that different layers and sub-layers utilizing the self-attention often require different thresholds since each (sub-)layer often exhibits a different distribution of attention scores. However, it is impractical to leave these layer-specific threshold values as user-defined hyperparameters, especially for models like BERT-large which has 384 sub-layers utilizing the self-attention mechanism. To avoid such an impracticality, we let a user specify a single hyperparameter that represents the degree of approximation, and present a scheme that automatically finds the (sub-)layer-specific thresholds that correspond to the user-specified degree of approximation.

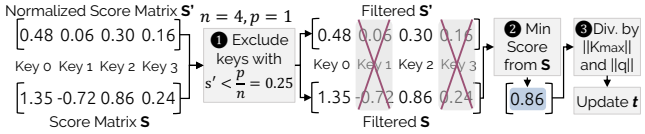

Fig. 6. Process of identifying layer-specific thresholds.

Learning Layer-Specific Thresholds. To find the layer-specific threshold, our scheme runs target neural network model inference on the training set and inspects the characteristics of each layer utilizing the self-attention. Fig. 6 illustrates this process. First, for each invocation of the self-attention operation for a particular (sub-)layer, our scheme inspects the softmax-normalized attention scores for each query. Then, ❶ we identify the set of keys whose softmax-normalized attention score exceeds $p \cdot \frac{1}{n}$ where p is a user-specified hyperparameter, and n is the number of input entities. Here, the hyperparameter p represents the degree of approximation. For example, if $p = 2$ when $n = 200$, this means that the user considers entities whose softmax-normalized score exceeding 0.01 to be relevant. The selection of a larger p implies aggressive approximation, and a smaller p means conservative approximation. ❷ Among those keys, we focus on the key with the minimum softmax-normalized attention score[1]. ❸ Then, we normalize its original attention score by dividing it with the query norm $\|q\|$ and the maximum key norm $\|K_{max}\| = max(\|K_1\|, ..., \|K_n\|)$. We denote the resulting value as the threshold t. This process is repeated for multiple input data in training set to find the average of this value for each (sub-)layer. During an actual inference run, the threshold t multiplied by the maximum key norm ($t \cdot \|K_{max}\|$) is compared with the approximate similarity (Section III-D) to determine whether a key (in the key matrix $\mathbf{K}$) is relevant to the current query. Specifically, the following equation specifies the condition to determine if the computation for the key K_y can be skipped for the query Q_x.

$$t \cdot \|K_{max}\| \geq \|K_y\| \cdot \cos\left(\max(0, \frac{\pi}{k} \cdot hamming(h(Q_x), h(K_y)) - \theta_{bias})\right)$$

IV. ELSA Hardware Architecture

A. Motivation

Hardware specialization is a well-known approach to improving performance and energy efficiency of a specific type of computation. Naturally, this idea can be applied to the self-attention operation, which accounts for a substantial portion of total execution time in many emerging NN models of today. However, we also emphasize more important, often overlooked, benefits of building specialized hardware–exposing unique optimization opportunities for the specific operation that cannot be exploited profitably by the conventional hardware.

We make this point with the proposed approximation algorithm as an example. As explained in Section III-D, the key idea of ELSA approximate attention is to avoid d-dimensional dot product through a hamming distance computation between binary embeddings, multiplication, and a cosine function. Unfortunately, the conventional GPU is not suited for many of these operations, and our internal experiments have found that the approximation scheme results in a 3.14× slowdown because simply performing d-dimensional dot product is faster than performing the approximate similarity computation, even with various manual/automated optimizations for CUDA implementation (e.g., TorchScript Tracing [62]). We find that the true benefits of the proposed approximation scheme can be harnessed only by a specialized hardware that is co-designed with this approximation algorithm. This is where a software-hardware co-optimization uncovers the unique opportunity that pure hardware or software-only optimizations fail to exploit.

[1]Note that there exists a case where all softmax-normalized attention scores are below $p \cdot 1/n$ (this can happen when $p > 1$). In such a case, we simply take the maximum score among all keys.

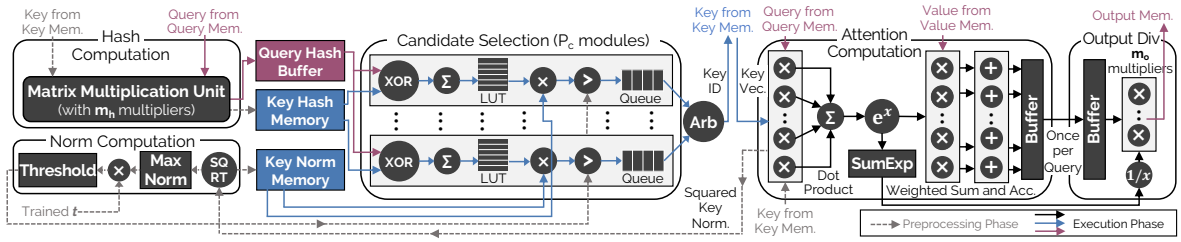

Fig. 7. ELSA Pipeline Block Diagram

B. Hardware Overview

For the efficient processing of the self-attention operation, we design a specialized hardware accelerator that exploits the novel approximation scheme introduced in Section III. One can view the ELSA accelerator as a specialized functional unit for the self-attention mechanism, which can be integrated with various computing devices such as CPUs, GPUs, and other NN accelerators. The host device can issue a simple command to initiate the ELSA accelerator and pass the inputs (i.e., key/query/value matrix and n). When a device with scratchpad memories such as GPUs or NN accelerators is used, matrix inputs (and output buffer) can be passed by reference so that the accelerator can directly read those inputs without making another copy. Once inputs are ready, the accelerator goes through the preprocessing/execution phase and then writes the output matrix to the output memory and notifies the host.

Operation Overview. Fig. 7 shows the block diagram of the ELSA accelerator pipeline, which also presents its high-level dataflow. The ELSA accelerator takes a key matrix, a query matrix, and a value matrix as inputs for self-attention to generate the output matrix. As soon as inputs are ready, the preprocessing phase begins. This phase computes k-bits hash values of each row in the key matrix using a *hash computation module*, and stores them in the key hash memory. Similarly, the norm of each key vector is computed using a *norm computation module* and stored in the key norm memory. Once this phase ends, the execution phase begins where each row of the query matrix is processed in sequence to output a single row of the output matrix at a time. Specifically, for each query, P_c *candidate selection modules* retrieve P_c keys' hashes and norms (along with the query hash) every cycle and outputs up to P_c selected candidate key IDs (i.e., row IDs) to each module's output queue. Then, these selected key IDs are arbitrated and passed to the *attention computation module*, which computes and accumulates the selected key's contribution to the output (for the current query) every cycle. Once all selected keys for this particular query is computed, the *output division module* performs the division on this output. This process is repeated for each row of the query matrix (i.e., each query), and the operation ends when the last query is processed.

C. Design of Hardware Modules

(1) Modules for Approximate Self-attention Computation

Candidate Selection Module. The candidate selection module performs the approximate self-attention mechanism (Section III) to identify the set of potentially relevant rows in the key matrix (i.e., candidates), and then outputs the indices of such elements to the attention computation module. Every cycle, this module takes three inputs: i) k-bits hash value of a key from the key hash memory and ii) the norm of this key from the key norm memory, and iii) k-bits hash value of the current query from the query hash buffer. Then, this module utilizes k-bits XOR unit followed by an adder to compute the Hamming distance between the key hash value and the query hash value. The resulting Hamming distance value is then used as an index to access the pre-populated lookup table, which stores $\cos(\pi/k \cdot d_{Hamming} - \theta_{bias})$. Since the Hamming distance takes an integer value between zero and k, this lookup table has $k+1$ entries. Once this value is retrieved, it is multiplied with the norm of the current key to compute the approximate similarity (Section III-D). This value is compared with the product of threshold t (Section III-E) and the largest vector norm of the key matrix (i.e., $t \cdot \max(\|K_1\|, ..., \|K_n\|)$). If the approximate similarity is greater than this value, the key in question is selected as a potentially relevant key, and the index of this key is then passed to this module's output queue. Multiple (i.e., P_c) candidate selection modules process different keys in parallel, and then their outputs are arbitrated and passed to the attention computation module. The candidate selection module is fully-pipelined and processes one key per cycle.

```
1  def attention_computation (float q[], float key[][],
2     float val[][], vector<int> candidates):
3     for keyid in candidates:
4        /* Dot-Product*/
5        parallel for i = 0 to d-1:
6           temp[i] = key[keyid][i] * q[i]
7        score = ParallelSum(temp)
8        /* Exponent Computation */
9        score = exp(score)
10       sumexp += score
11       /* Weighted Sum */
12       parallel for i = 0 to d-1:
13          output[i] += score * val[keyid][i]
14 def output_division (float output[], float sumexp):
15    reciprocal = 1/sumexp
16    /* Division */
17    for i = 0 to d/mo-1:
18       parallel for j = 0 to mo-1:
19          output[i * mo + j] *= reciprocal
```

Fig. 8. Pseudocode for Attention computation and output division modules.

Attention Computation Module. A single attention computation module is in charge of computing a single row of the final output matrix, along with the output division module. Fig. 8 represents this module's operation in pseudocode. Each cycle, this module takes a key as an input from the arbiter with the longest-queue-first scheduling policy. Then, it first computes

the dot product between a key (K_y) and a query (Q_x) using its d multipliers and an adder tree (Line 5-7 in Fig. 8). After that, for the softmax normalization of the resulting attention score, the exponent of this value is computed using a lookup table (explained in Section IV-E). The resulting exponentiated value is i) accumulated in the sum of exponent register (Line 10), and ii) multiplied with all components of the corresponding value matrix row using the other set of d multipliers and accumulated with d adders (Line 12-13). This module is fully-pipelined and can process a single candidate every cycle. Assuming c candidates are selected for the query Q_x by the candidate selection modules, this module can process them in about c cycles. The resulting output vector and the sum of exponentiated values are then passed to the output division module when it finishes processing all selected keys for the current query.

Output Division Module. Once all (selected) keys are processed, all components of the output vector needs to be divided by the accumulated exponentiated score to complete the softmax normalization. For this purpose, the hardware first utilizes a reciprocal unit (explained in Section IV-E) to compute the reciprocal of the sum of the exponentiated score (Line 15), and then multiply each component of the output vector with m_o multipliers (Line 18-19). Since this module is fully-pipelined, it can handle a single query every d/m_o cycles. Note that this module operates in parallel with the rest of the pipeline (e.g., candidate selection and attention computation modules). However, when other modules are processing the ith query, this module is processing the $(i-1)$th query.

(2) Modules for Key/Query Hash & Norm Computation

Hash Computation Module. This module is in charge of computing hashes for the keys and the queries by performing a series of matrix multiplications as described in Section III-C. Specifically, if we assume the specific case presented in Section III-C (i.e., utilizing three-way Kronecker products of 4×4 matrices for $k = d = 64$), the hash computation for a vector requires a total of twelve (4×4, 4×4) matrix multiplications (the last paragraph in Section III-C). Assuming m_h multipliers for this unit, we carefully design the matrix multiplication unit so that it fully utilizes all m_h multipliers to perform this operation and complete the hash computation in $768/m_h$ (i.e., $3d^{4/3}/m_h$) cycles. For these matrix multiplications, this module contains 48 ($3d^{2/3}$) registers, where each register value is an element of three pre-defined (4×4) matrices for the hash computation (i.e., A_1, A_2, A_3 in Section III-C). Once the matrix multiplications are finished, the sign bits of each component (a total of k-bits) are concatenated and stored in the key hash memory. During the preprocessing phase, this module computes all key hashes ($768n/m_h$ or $3nd^{4/3}/m_h$ cycles) and the first query hash (extra $768/m_h$ or $3d^{4/3}/m_h$ cycles). During the execution phase, this model computes the hash value for the next query while the rest of the pipeline (e.g., candidate selection and attention computation module) is processing the current query.

Norm Computation Module. Norms of the keys are computed during the preprocessing phase in addition to the hashes of the keys. The euclidean (L2) norm of the key vector $\|K_y\|$ is obtained by computing the dot product with itself ($K_y \cdot K_y$) and then taking its square root. For this purpose, instead of having its own set of multipliers, this unit utilizes the d multipliers and the adder tree in the attention computation module (as shown in Fig. 7). Then, this module utilizes its own square root units (see Section IV-E for details) to compute the final result and store it in the key norm memory. In addition, this module also identifies the maximum key norm and multiply the trained t by that value to compute the threshold that is used for the candidate selection modules.

(3) Memory Modules

Key Hash/Norm Memory. These memory modules are implemented as SRAM structures placed within the ELSA accelerator. These structures are initialized during the preprocessing phase and then utilized by the candidate selection module during the execution phase. Key Hash SRAM requires a total of $nk/8$ bytes storage, and Key Norm SRAM requires a total of n bytes assuming an 8-bit representation for the norm. In $n = 512$ and $k = 64$ configuration, the key hash SRAM requires 4KB, and the key norm SRAM requires 512 bytes.

Query/Key/Value/Output Matrix Memory. These matrices are inputs (Query, Key, Value) and output of the self-attention. They can be placed within the ELSA accelerator using the SRAM structures. However, since the ELSA accelerator is expected to be utilized in conjunction with a host device such as GPUs or other neural network accelerators targeting other parts of the neural network models, it is also possible to utilize scratchpad memory structures in those devices (e.g., GPU shared memory) to store these matrices. At $n = 512$ and $d = 64$, each of these matrices requires about 36KB storage space assuming 9-bits representation (including the sign bit).

D. Pipeline Design

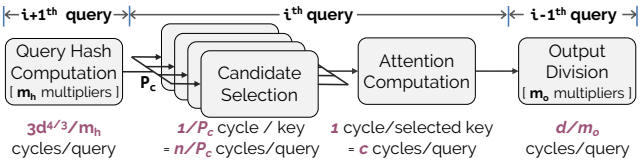

Fig. 9. ELSA accelerator pipeline during the execution phase.

Pipeline Configuration. For a given n and d, this pipeline takes $3d^{4/3}(n+1)/m_h$ cycles for the preprocessing. Figure 9 shows the high-level view of the pipeline during the execution phase and lists each hardware module's latency to process a single query (also explained in Section IV-C). As illustrated in the figure, four hardware modules can potentially bottleneck the pipeline. It takes $\max(3d^{4/3}/m_h,\ n/P_c,\ c,\ d/m_o)$ to process a single query when c is the number of candidates selected by candidate selection modules. To avoid introducing the bottleneck and maximize the throughput, one should carefully select P_c, m_h, m_o to properly balance the pipeline. Specifically, it is ideal to configure parameters in a way that modules other than the attention computation module (takes c cycles) do not become a pipeline bottleneck. For

example, if one aims to design a pipeline that can achieve up to 8× speedup (i.e., it takes $n/8$ or more cycles to process a query) with approximation, each of $3d^{4/3}/m_h$, n/P_c, and d/m_o should be less than or equal to $n/8$. When d is 64, a configuration such as $P_c = 8$, $m_h = 64$, $m_o = 8$ satisfies this requirement as long as $n \geq 96$. With this configuration, the achieved speedup is $min(n/c, 8)$. That is, the speedup is often (i.e., $c \geq n/8$) determined by the effectiveness of the approximation scheme, which reduces the number of keys to process (i.e., c) for the attention computation module.

Parallel Pipeline. We extend the pipeline so that ELSA can utilize multiple attention computation modules in parallel by exploiting the fact that each row of the key/value matrix can be processed independently. To extend the pipeline to utilize P_a attention computation modules in parallel, the key matrix, the value matrix, and the key hash/norm need to be stored in a banked on-chip memory where each bank contains n/P_a keys, values, and key hashes/norms. Then, for each bank, P_c candidate selection modules and a single attention computation module are connected so that they process the set of keys (and values) within a single bank and compute the partial sum of the output as well as the exponentiated score. At the end of each query, such partial sums are passed to the output division module, which sums up these values using an adder tree (requires an extra set of $(P_a - 1) \cdot m_o$ adders) and computes the final output. To avoid a specific stage of the pipeline or the specific phase from forming a bottleneck, pipeline configuration parameters such as m_h (# of multipliers in hash computation module) and m_o (# of multipliers in output division module) may need to be adjusted. This is because the throughput of candidate selection modules and attention computation modules are increased by $P_a\times$ compared to the ones shown in Fig. 9. We find that $m_h = 256$ and $m_o = 16$ work well for $P_a = 4$. For further throughput, the whole ELSA accelerators (including its memory elements) can be replicated to exploit batch-level parallelism as well (e.g., our evaluation utilizes a set of twelve ELSA accelerators to exploit batch-level parallelism).

E. Design Details

Number Representation. The elements of key, query, value matrix are represented in a fixed-point form with a single sign bit, five integer bits, and three fraction bits. The elements of predefined matrices for the hash computation are represented with a fixed-point form with a single sign bit and five fraction bits. The rest of the pipeline utilizes the minimal necessary integer bitwidth to avoid the overflow while maintaining the number of fraction bits. We use custom floating-point representations (e.g., a single sign bit, ten exponent bits, and five fraction bits) to represent the output of the exponent function as well as following computations on it to cover their huge value range. We empirically verified that the use of these number representations has a negligible impact (<0.2%) on model evaluation metric loss across various models when compared to the FP32 baseline.

Choice of n and d. n represents the maximum number of input entities for the self-attention. For a model running very small NLP micro-benchmarks like GLUE [83], a small n (e.g., 128) is sufficient. For longer text such as question-answering benchmarks [47], [68], a larger n (e.g., 512) is often utilized to capture the relation between distant tokens. An even larger n (e.g., 800, 1024) is utilized for tasks like text summarization [51], and text generation [64], [71]. For evaluation, we configure the hardware to fit the largest workload we run, which has $n = 512$. We utilize $d = 64$, which all our evaluated models originally used. ELSA accelerator can be designed for any n or d, and once synthesized, it can efficiently run with any model or input that has smaller n or d.

Choice of Hash Length k. In general, higher k results in the better approximation since the estimate for the angle between two vectors becomes more accurate. However, too large k increases i) the cost of hash computation, ii) key hash storage area, and iii) area/power of the candidate selection modules. For such reasons, we find that $k = d$ is a choice that works well as long as k is not too small (e.g., less than 16). In case where $k > d$, batches of orthogonal vectors are utilized to generate k hash bits [40]. Since all our evaluated workloads use $d = 64$, we set $k = 64$ as well.

Hyperparameter Tuning. Our main hyperparameter p (Section III-E) determines the degree of approximation. We recommend the user to tune this parameter with the validation dataset so that the model maintains a user's desirable accuracy while improving the performance and energy efficiency. Note that this tuning process is simple since p is a hyperparameter that (almost) monotonously increases accuracy as its value decreases. Finally, a user can set p to 0 to easily fall back to the exact version when the highest accuracy is desired.

Special Functional Units. The exponential computation unit computes e^x by utilizing the fact that $e^x = 2^{(\log_2 e)x} = 2^{\text{frac}((\log_2 e)x)} \cdot 2^{\text{floor}((\log_2 e)x)}$. For $2^{\text{frac}((\log_2 e)x)}$, it utilizes 32-entry lookup table where fractional exponents of 2 are stored. For the reciprocal unit, a simple lookup table with 32-entry is used to obtain the reciprocal of a floating point with 5 fraction bits. For the square root unit, a Taylor-expansion-oriented scheme named tabulate and multiply [36], [81] is utilized.

V. Evaluation

A. Workloads

We evaluate several representative self-attention-oriented NN models to demonstrate the effectiveness of the ELSA. For natural language processing models, we select three of the most popular ones: Google BERT (large) [18], Facebook RoBERTa (large) [52], and Google ALBERT (large) [49]. We utilize open-source implementations of those models from HuggingFace [87] (BERT, RoBERTa), FairSeq [57] (RoBERTa), and Google ALBERT repository [24] (ALBERT). For all three NLP models, we run Stanford Question Answering Dataset (SQuAD) [68] 1.1 & 2.0, and RACE dataset [47], which is a large-scale reading comprehension dataset from examinations. For RoBERTa, we additionally run IMDB review sentiment analysis dataset [54]. In addition to these NLP models, we also evaluate ELSA with self-attention-oriented sequential recommendation models such as SASRec (3-layers model) [43]

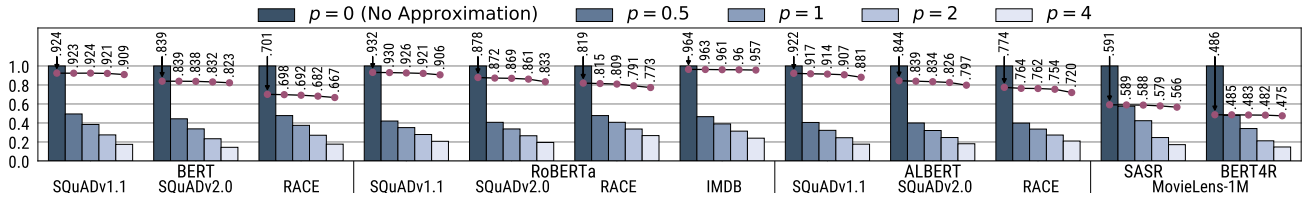

Fig. 10. Model accuracy (**lines**) and portion of selected candidates (**bars**) across varying degree of approximation.

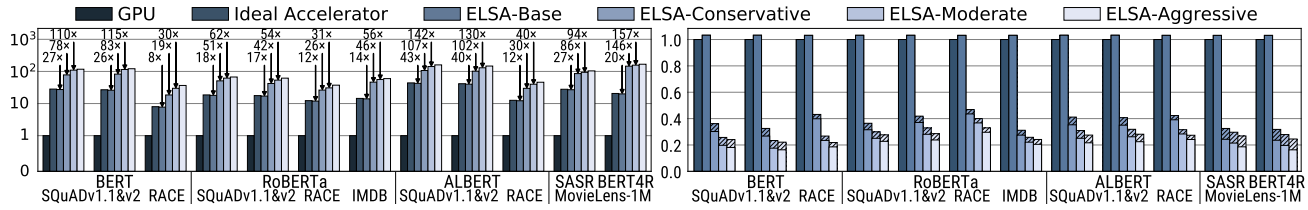

Fig. 11. (a) Normalized self-attention throughput (left) and (b) Normalized self-attention operation latency (right) on various devices. Hatched area on the right figure represents the time spent on preprocessing.

and BERT4Rec (3-layers, 2-head model) [78] with MovieLens 1M dataset [33].

B. Accuracy Evaluation

Methodology. We extend the self-attention layer in each NN model with our approximation scheme and measure the model's end-to-end accuracy metric (i.e., F1 score for SQuAD, raw accuracy for RACE/IMDB, and NDCG@10 [85] for recommendation models) on the test set or the validation set (for the workloads whose test set is not publicly available). For conciseness, we simply refer to these metrics as *accuracy* throughout this section.

Impact of Approximation. Fig. 10 shows the impact of approximation on end-to-end model accuracy (lines) as well as the portion of selected candidates[2] (bars) across a varying degree of approximation hyperparameter p. In general, the small p implies conservative approximation with a relatively small accuracy degradation, while the larger p implies more aggressive approximation. For most of the model-workloads combinations, it is possible to achieve sub-1% accuracy loss by only inspecting less than 40% of the total entities as candidates (i.e., $p = 1$). Furthermore, the figure also shows that it is possible to achieve sub-2% accuracy loss by inspecting about 26% of the total entities on average ($p = 2$). As discussed in Section IV-E, the user can experiment with the train set or the validation set to determine the degree of approximation (p) that provides a reasonable accuracy loss.

C. Performance Evaluation

Methodology. For performance evaluation, we implement a custom simulator for ELSA that is integrated with the PyTorch/TensorFlow implementations of the NN models. For GPU performance evaluation, we utilize a system with the six-core Intel Xeon Gold CPU [35] and the Nvidia V100 GPU [56] with 16GB memory. For each workload, the batch size achieving the best throughput is selected. For ELSA performance evaluation, we use a set of twelve ELSA accelerators, each running at 1GHz and configured as follows: $P_a = 4$, $P_c = 8$, $m_h = 256$, and $m_o = 16$. We specifically evaluate twelve ELSA accelerators so that their peak throughput (= 1.088 TOPS/accelerator $\times$ 12 $\approx$ 13 TOPS) approximately matches with the Nvidia V100 GPU having 14 TFLOPS peak throughput (with FP32[3]).

We select p for each NLP model-workload combination whose worst-case accuracy loss is bounded by 1%, 2.5%, 5% to call them ELSA-conservative, moderate, aggressive, respectively. For recommendation models, 0.5%, 1.0%, 2.0% drop in NDCG@10 metric is used to determine p for those configurations. We also evaluate ELSA-base configuration with no approximation. Finally, we compare ELSA configurations with an *ideal* accelerator, which can sustain 100% peak FP throughput at 1GHz frequency, while having the same number (i.e., 528) of multipliers with the ELSA-base accelerator. This is effectively an upper-bound of performance for the other matrix multiplication accelerators *without* approximation.

Throughput. Fig. 11(a) presents the throughput of the self-attention across different platforms. The figure shows that a set of ELSA-base accelerators achieve substantially better throughput (i.e., 7.99-43.93$\times$) than the GPU, indicating that its specialized architecture can effectively accelerate the self-attention operation. Overall, the speedup of the ELSA-base over GPU varies across workloads and models. Variations across workloads are mostly attributable to the actual number of input entities. There are some inputs where the data has fewer entities than the maximum number of entities the model supports (i.e., n). In such cases, the GPU implementations pad the data and perform the matrix multiplications with n rows. However, ELSA accelerators (and the ideal accelerator) avoid computation for the padded rows and achieve higher

[2]Many software implementations operate with the fixed size n (e.g., 512). If the input text has fewer than n tokens, the software implementation pads the input so that it gets n tokens. We exclude such paddings for the normalization, and the figure shows the portion of selected candidates among the real tokens.

[3]Nvidia GPUs can achieve better raw inference throughput by utilizing the FP16 format accelerated with the tensor core. However, its iso-peak-FLOPS throughput will be lower in this case since the actual throughput increase from FP16 inference is often much lower than 8$\times$ increase in *peak throughput*. Thus, using FP32 throughput gives an advantage to GPU in calculating the normalized throughput.

speedup. Speedup differences across NLP models for the same dataset are mostly due to the GPU performance differences across different models and implementations. The figure also demonstrates that the conservative, moderate, and aggressive approximation scheme enables ELSA to achieve much higher geomean speedups over GPU (57×, 73×, 81×, respectively) than the ELSA-base accelerator. We find that moderate or aggressive approximation performance is sometimes bounded by the pipeline bottleneck caused by the candidate selection modules. Adjusting pipeline configuration parameters such as P_c (Section IV-D) will result in extra speedups in these cases at the expense of extra area/power.

Latency. Fig. 11(b) compares the average latency of performing a single self-attention operation on various models across ELSA accelerators and the ideal accelerator. As shown in the figure, ELSA-base latency is nearly identical (1.03×) to the ideal accelerator. ELSA with the approximation scheme achieves latency reduction over the ideal accelerator by exploiting the approximation opportunities. The average (geomean) normalized latency of ELSA-conservative, ELSA-moderate, and ELSA-aggressive are 0.38×, 0.29×, 0.26× of the ideal accelerator latency. Fig. 11(b) also shows that all workloads spend a small amount of time on preprocessing. If a further reduction in preprocessing time is desired, one can increase the m_h or use multiple hash computation modules.

Impact on End-to-End Performance. Figure 11 compares the throughput and latency for the *self-attention mechanism* (not the end-to-end model throughput or latency). As shown in Figure 2, the portion of the time spent on self-attention varies greatly across models, sequence length (i.e., input length), and the model configuration (e.g., FFN dimension). With ELSA-conservative's 57× average speedup, the use of ELSA accelerators makes the time spent on self-attention to be negligible compared to the time spent on the other operations. The ELSA-conservative accelerators achieve about 1.4-2.5× end-to-end speedup across five models when the default max input length is utilized, and 2.4-5.0× speedup when the 4× larger input length is utilized. Furthermore, if other types of accelerators are utilized to accelerate the rest of the network (e.g., FC layers), the end-to-end speedup from the use of ELSA accelerators becomes even larger, since the portion of the time spent on the self-attention layer becomes larger.

D. Area/Energy Evaluation

Methodology. For area and energy evaluation, we implement the ELSA accelerator with Chisel hardware description language [12], and perform functional verification. Then, we synthesize, place and route the Chisel-generated Verilog code with the 1GHz target frequency using Synopsys Design Compiler [80] and TSMC 40nm standard cell library. For logic synthesis, we assume the following pipeline configuration: $n = 512$, $d = 64$, $P_a = 4$, $P_c = 8$, $m_h = 256$, $m_o = 16$.

Area. Table I reports the ELSA accelerator area characteristics and Fig. 12 shows the layout of the ELSA accelerator. As shown in the table, the single ELSA accelerator utilizes about 1.3mm^2 area (2.1mm^2 with external memory modules), and twelve

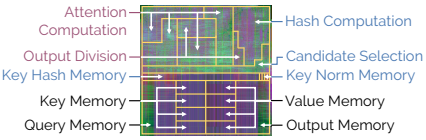

Fig. 12. Post-layout image of the ELSA accelerator.

ELSA accelerators utilize about 15.1mm^2 area (25.8mm^2 with external memory modules). On the other hand, Nvidia V100 GPU has a total die size of 815mm^2 [55]. This implies that integrating the ELSA accelerator to GPU incurs a very little area cost, and such a cost becomes even lower considering that the reported ELSA area is estimated from the 40nm technology node, while the Nvidia V100 GPU die area is from the 12nm technology node. Another important point from the area table is that candidate selection modules (32 copies) utilize a relatively little area. This proves that our approximation mechanism is very hardware-friendly.

Power and Energy Consumption. Table I shows that a single ELSA accelerator consumes about 1.49W (including power consumption from the external memory modules) and twelve ELSA accelerators consume about 17.93W at its peak. This is substantially lower than that of the Nvidia V100 GPU, which has 250W thermal design power (TDP). Furthermore, we measured the actual GPU power consumption with *nvidia-smi* tool and confirmed that the GPU is in fact operating at the power level very close to its peak (e.g., 240W+) while performing the self-attention operation in our workloads. Fig. 13(a) presents the energy efficiency comparison of the ELSA accelerators and the GPU. Combining the power efficiency (over 13×) and the speedup (shown in Fig. 11), the ELSA-base accelerator achieves over two orders of magnitude improvements in energy efficiency (geomean: 442×) over the GPU for the self-attention computation. Moreover, approximation-enabled configurations further increase the energy efficiency improvements: 1265× (conservative), 1726× (moderate), and 2093× (aggressive). Finally, Fig. 13(b) shows the energy consumption breakdown of the ELSA accelerators. The figure shows that our approximation scheme, despite the introduction of new hardware modules, results in the total energy reduction by significantly reducing the energy spent on attention computation and output division modules and external memory modules.

E. Discussion

Comparison with the A^3 accelerator. A^3 [30] is a recent proposal that also applies approximation to the attention. However, A^3 architecture has the following key limitations that make it not well-suited for the self-attention. First, its approximation scheme requires an expensive preprocessing (i.e., sorting all columns of the key matrix). Its preprocessing relies on external hardware (e.g., GPU) that incurs significant performance/energy overheads. Unfortunately, when multiple attention accelerators are used in parallel, the preprocessing time linearly increases while the execution time linearly decreases, to make this preprocessing take the dominating portion of the runtime. Also, storing the outcome of the preprocessing requires a memory that is twice larger than the

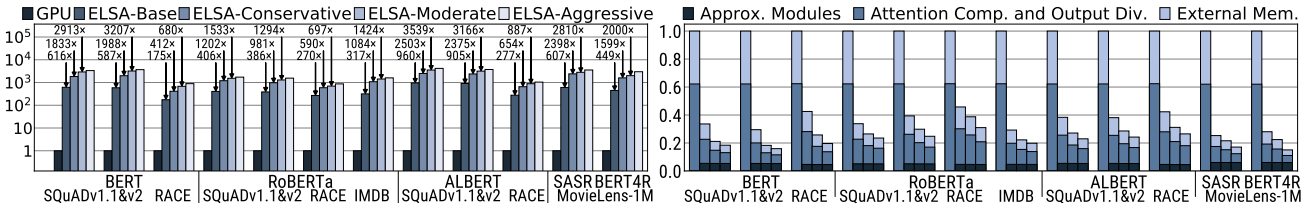

Fig. 13. (a) Normalized energy efficiency (performance/W) (left) and (b) Energy consumption breakdown (right) of the ELSA accelerators: from left-to-right, each bar represents ELSA-base, conservative, moderate, and aggressive.

TABLE I
AREA AND (PEAK) POWER CHARACTERISTICS OF ELSA.

Module Name	Area (mm^2)	Dynamic Pwr(mW)	Static Pwr(mW)
Modules for Approximate Self-attention			
Hash Computation ($m_h = 256$)	0.202	115.08	2.23
Norm Computation	0.006	9.91	0.07
32× Candidate Selection	0.180	78.41	1.95
Modules for Attention Computation			
4× Attention Computation	0.666	566.42	7.53
Output Division ($m_o = 16$)	0.022	11.42	0.19
Internal Memory Modules			
Key Hash Memory (4KB)	0.141	139.91	1.05
Key Norm Memory (512B)	0.038	34.9	0.29
External On-Chip Memory Modules			
Key/Value Mem. (36KB ea.)	0.253	167.39	2.29
Query/Output Mem. (36KB ea.)	0.193	91.03	1.72
ELSA Accelerator			
ELSA Accelerator (1×)	1.255	956.05	13.31
External Memory Modules (1×)	0.892	516.84	8.02
ELSA Accelerators (12×)	15.06	11472.6	159.72
External Memory Modules(12×)	10.704	6202.08	96.24

original key matrix. Second, the A^3's approximation scheme is complex (occupying over 1.7× larger area than ELSA's attention computation module) and has a very low degree of parallelism. A^3's approximation scheme can only select up to two keys (and often fewer) every cycle and is not further parallelizable. This significantly limits its ability to achieve the desired accuracy on time and prevents the use of multiple attention computation modules in parallel. For example, A^3 evaluation results state that it achieves a 1.85× speedup over its baseline accelerator without the approximation on the BERT model running the SQuADv1.1 dataset at the expense of 1.3% accuracy loss. On the other hand, for a similar setting, ELSA-conservative/moderate configurations achieve 2.76×/3.72× speedup over the ELSA-base without approximation with lower than 1%/2.5% accuracy loss. Considering this difference in baseline configurations, ELSA approximate configurations achieve 5.96×/8.04× better raw speedup over the A^3 approximation configuration. Finally, ELSA presents a more scalable, area-efficient attention computation module design that does not require multiple n-element buffers.

Comparison with Google TPU. Google Tensor Processing Unit (TPU) [23] is specialized hardware that targets neural network training as well as inference tasks. To check its effectiveness in self-attention operation, we run ALBERT model [49] that natively supports TPU execution on Google Cloud TPUv2. Our experimental results show that ELSA-base achieves 8.3×, 6.4×, 2.4× better (peak-FLOPS-normalized) throughput[4] on self-attention operations of ALBERT running SQuADv1.1/2, and RACE datasets. For the same workloads, ELSA-moderate achieves 27.8×, 20.9×, 8.0× speedup, respectively. For the references, the measured TPU (peak-FLOPS-normalized) throughput was 5.5×, 6.7×, and 5.4× better than GPU throughput for the same workloads.

NN Models with Lightweight Self-Attention. Several recent works propose changes in the NNs to reduce the computational demand of the self-attention operation. For example, some [4], [11], [13], [16], [27], [45], [63], [65], [75], [79], [84], [88]–[90] augment the architecture of the self-attention layer to efficiently capture the relation between a large number of entities. Our work is compatible with most of them [4], [11], [27], [63], [79], [88], [90] because they decompose a very large self-attention operation (e.g., sequence length $\geq$ 4096) into a sequence of multiple, smaller conventional self-attentions.

Moreover, ELSA is fundamentally different from these software approaches in that it takes a more model-agnostic approach without requiring retraining, which can be very expensive computationally on a large-scale language model. Finally, most software-only approaches [4], [11], [45], [63], [65], [89] in fact fails to achieve the inference speedup for reasonable sequence length (e.g., <2048), despite a theoretical reduction in the number of operations. Specifically, a recent work [84] finds that sparse attention techniques achieve very little speedup (e.g., 20% speedup for 2% accuracy loss), and Reformer [45] fails to achieve any speedup for sequence length less than 2048, due to its huge constant in their time complexity. Even in the case of concurrent works achieving speedup on the commercial hardware for sequence length <2048, their reported speedup from approximation is around 1.3×-1.7× [13], [84], which is far less than what ELSA achieves with approximation.

VI. RELATED WORK

Hardware Support for Attention Mechanisms. A few hardware accelerators related to the attention mechanism are recently proposed. A^3 is the most closely related work, which is discussed in Section V-E. MnnFast [39], Manna [74], and Mann Dataflow accelerator [60] are also relevant in that they contain modules that can potentially be utilized to accelerate the attention mechanism. However, their focus is on the end-to-end hardware implementation of particular

[4]TPUv2 has a peak throughput of 180 TFLOPS with its bfloat16 internal representation. We assume that it has 1/4× peak throughput with FP32 (45 TFLOPS) and then compute its iso-peak-FLOPS throughput by dividing the actual TPU throughput by 45/13 as twelve ELSA accelerators we used for the comparison with GPU has 13 TOPS peak throughput (instead of 180/13).

neural network models without fully exploiting approximation opportunities, such as Google NTM/DNC [25], [26] for Manna and Facebook End-to-End Memory Network [77] for MnnFast.

NN Approximation with Hardware Support. There are prior works presenting various forms of approximation strategies to improve neural network performance and energy efficiency. Specifically, works such as [28], [34], [37], [38], [44], [59] investigate the efficient use of quantization and low-precision operations for neural networks. Furthermore, other works [53], [67] propose the approximate MAC unit to achieve a similar goal. More closely related works are ones focusing on finding values that are less likely to affect the final output of the neural network models. SnaPEA [1], ComPEND [50], ZAP [72], and RnR (Reduce and Rank) [66] are representive examples.

Hardware Accelerators for NN. Various hardware accelerators [6], [8]–[10], [19], [21], [29], [41], [42], [69] have been proposed to accelerate key neural network operations represented as matrix multiplications. Specifically, several proposals [2], [17], [31], [32], [46], [58], [92] focus on the sparsity of the activation and weight matrices to further accelerate such operations. Our work differs from these works in that i) we provide the unique approximation scheme that dynamically sparsifies the key matrix, and ii) specifically targets the self-attention mechanism.

VII. Conclusion

The self-attention mechanism is recently getting a large amount of attention for its ability to capture relations within input entities. Considering that it is emerging as a key primitive of many modern state-of-the-art neural network models in various domains, it is crucial to accelerate this operation for better performance and energy efficiency. Our work focuses on the approximation opportunity within this operation and co-designs a specialized approximation algorithm and hardware for this operation to significantly reduce the amount of computation for this operation. With this reduction, ELSA achieves significant improvements in both performance and energy efficiency over the conventional hardware like GPU.

Acknowledgments

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2021-0-00105, Development of Model Compression Framework for Scalable On-device AI Computing on Edge Applications), National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2020R1A2C3010663), and IC Design Education Center (IDEC), Korea (for EDA tools). Jae W. Lee is the corresponding author.

References

[1] 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 *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA, 2018.

[2] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA, 2016.

[3] I. Bello, B. Zoph, A. Vaswani, J. Shlens, and Q. V. Le, "Attention augmented convolutional networks," in *IEEE/CVF International Conference on Computer Vision*, ser. ICCV, 2019.

[4] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," 2020.

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[6] "Cerebras systems: Achieving industry best ai performance through a systems approach," https://secureservercdn.net/198.12.145.239/a7b.fcb.myftpupload.com/wp-content/uploads/2020/03/Cerebras-Systems-Overview.pdf?time=1584807908, Cerebras Systems, Inc.

[7] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, ser. STOC, 2002.

[8] 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 *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2014.

[9] 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.

[10] 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 *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2014.

[11] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," 2019.

[12] "Chisel," https://chisel.eecs.berkeley.edu.

[13] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, D. Belanger, L. Colwell, and A. Weller, "Rethinking attention with performers," 2020.

[14] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning, "What does BERT look at? an analysis of BERT's attention," in *Proceedings of the ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, 2019.

[15] M. Cornia, M. Stefanini, L. Baraldi, and R. Cucchiara, "Meshed-memory transformer for image captioning," in *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[16] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. Le, and R. Salakhutdinov, "Transformer-XL: Attentive language models beyond a fixed-length context," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.

[17] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-Tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2019.

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics*, ser. NAACL, 2019.

[19] 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 *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA, 2015.

[20] Y. Feng, F. Lv, W. Shen, M. Wang, F. Sun, Y. Zhu, and K. Yang, "Deep session interest network for click-through rate prediction," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, ser. IJCAI, 2019.

[21] 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 *Proceedings of the 45th International Symposium on Computer Architecture*, ser. ISCA, 2018.

[22] Y. Gong, S. Kumar, H. A. Rowley, and S. Lazebnik, "Learning binary codes for high-dimensional data using bilinear projections," in *The IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR, 2013.

[23] "Cloud TPU system architecture," https://cloud.google.com/tpu/docs/system-architecture, Google Cloud.

[24] "ALBERT official implementation," https://github.com/google-research/ALBERT, Google Research.

[25] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *CoRR*, 2014. [Online]. Available: http://arxiv.org/abs/1410.5401

[26] 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*, vol. 538, pp. 471 – 476, 2016.

[27] S. Gray, A. Radford, and D. P. Kingma, "Gpu kernels for block-sparse weights," https://cdn.openai.com/blocksparse/blocksparsepaper.pdf, OpenAI.

[28] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. ICML, 2015.

[29] "Goya hl-10x datasheet," https://habana.ai/wp-content/uploads/2019/06/Goya-Datasheet-HL-10x.pdf, Habana Labs.

[30] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, and D.-K. Jeong, "A^3: Accelerating attention mechanisms in neural networks with approximation," in *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, 2020.

[31] 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 *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA, 2017.

[32] 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 *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA, 2016.

[33] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Transactions on Interactive Intelligent Systems*, vol. 5, no. 4, 2015.

[34] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *Journal of Machine Learning Research*, vol. 18, no. 1, p. 6869–6898, 2017.

[35] "Intel Xeon gold 6128 processor," https://ark.intel.com/products/120482/Intel-Xeon-Gold-6128-Processor-19-25M-Cache-3-40-GHz-, Intel, 2017.

[36] M. Istoan and B. Pasca, "Fixed-point implementations of the reciprocal, square root and reciprocal square root functions," in *HAL Open Archive*, 2015.

[37] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *The IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR, 2018.

[38] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang, "Compensated-dnn: Energy efficient low-precision deep neural networks by compensating quantization errors," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC, 2018.

[39] H. Jang, J. Kim, J.-E. Jo, J. Lee, and J. Kim, "MnnFast: A fast and scalable system architecture for memory-augmented neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA, 2019.

[40] J. Ji, J. Li, S. Yan, B. Zhang, and Q. Tian, "Super-bit locality-sensitive hashing," in *Advances in Neural Information Processing Systems*, ser. NIPS, 2012.

[41] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. 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. Snelham, 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 *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA, 2017.

[42] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2016.

[43] W. Kang and J. J. McAuley, "Self-attentive sequential recommendation," in *Proceedings of the IEEE International Conference on Data Mining*, ser. ICDM, 2018.

[44] S. Kapur, A. K. Mishra, and D. Marr, "Low precision rnns: Quantizing rnns without losing accuracy," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1710.07706

[45] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," in *8th International Conference on Learning Representations (ICLR)*, 2020.

[46] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2019.

[47] G. Lai, Q. Xie, H. Liu, Y. Yang, and E. Hovy, "RACE: Large-scale ReAding comprehension dataset from examinations," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP, 2017.

[48] G. Lample, A. Sablayrolles, M. Ranzato, L. Denoyer, and H. Jégou, "Large memory layers with product keys," *Advances in Neural Information Processing Systems*, 2019.

[49] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A lite bert for self-supervised learning of language representations," in *International Conference on Learning Representations*, ser. ICLR, 2020.

[50] D. Lee, S. Kang, and K. Choi, "ComPEND: Computation pruning through early negative detection for ReLU in a deep neural network accelerator," in *Proceedings of the International Conference on Supercomputing*, ser. ICS, 2018.

[51] Y. Liu and M. Lapata, "Text summarization with pretrained encoders," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP, 2019.

[52] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized bert pretraining approach," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[53] Z. Liu, A. Yazdanbakhsh, T. Park, H. Esmaeilzadeh, and N. S. Kim, "SiMul: An algorithm-driven approximate multiplier design for machine learning," *IEEE Micro*, vol. 38, no. 4, pp. 50–59, 2018.

[54] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, ser. ACL, 2011.

[55] "NVIDIA TITAN V," https://www.nvidia.com/en-us/titan/titan-v/, NVIDIA, 2018.

[56] "NVIDIA V100 Tensor Core GPU," https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf, NVIDIA, 2020.

[57] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *Proceedings of the Conference of the North American Chapter of*

the Association for Computational Linguistics (Demonstrations), ser. NAACL, 2019.

[58] 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 *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA, 2017.

[59] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA, 2018.

[60] S. Park, J. Jang, S. Kim, and S. Yoon, "Energy-efficient inference accelerator for memory-augmented neural networks on an fpga," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, ser. DATE, 2019.

[61] N. Parmar, P. Ramachandran, A. Vaswani, I. Bello, A. Levskaya, and J. Shlens, "Stand-alone self-attention in vision models," in *Advances in Neural Information Processing Systems*, ser. NeurIPS, 2019.

[62] "Torchscript," https://pytorch.org/docs/stable/jit.html, PyTorch, 2020.

[63] J. Qiu, H. Ma, O. Levy, S. W. tau Yih, S. Wang, and J. Tang, "Blockwise self-attention for long document understanding," 2019.

[64] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf, 2019.

[65] J. W. Rae, A. Potapenko, S. M. Jayakumar, C. Hillier, and T. P. Lillicrap, "Compressive transformers for long-range sequence modelling," in *8th International Conference on Learning Representations (ICLR)*, 2020.

[66] 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.

[67] A. Raha and V. Raghunathan, "qLUT: Input-aware quantized table lookup for energy-efficient approximate accelerators," *ACM Transactions on Embedded Computing Systems*, 2017.

[68] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP, 2016.

[69] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA, 2016.

[70] C. Rosset, "Turing-NLG: A 17-billion-parameter language model by Microsoft," https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/.

[71] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using model parallelism," *CoRR*, 2019. [Online]. Available: https://arxiv.org/pdf/1909.08053

[72] G. Shomron, R. Banner, M. Shkolnik, and U. Weiser, "Thanks for nothing: Predicting zero-valued activations with lightweight convolutional neural networks," 2019.

[73] W. Song, C. Shi, Z. Xiao, Z. Duan, Y. Xu, M. Zhang, and J. Tang, "Autoint: Automatic feature interaction learning via self-attentive neural networks," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, ser. CIKM, 2019.

[74] J. R. Stevens, A. Ranjan, D. Das, B. Kaul, and A. Raghunathan, "Manna: An accelerator for memory-augmented neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2019.

[75] S. Sukhbaatar, E. Grave, P. Bojanowski, and A. Joulin, "Adaptive attention span in transformers," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.

[76] S. Sukhbaatar, E. Grave, G. Lample, H. Jégou, and A. Joulin, "Augmenting self-attention with persistent memory," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1907.01470

[77] S. Sukhbaatar, J. Weston, R. Fergus *et al.*, "End-to-end memory networks," in *International Conference on Neural Information Processing Systems*, ser. NIPS, 2015.

[78] F. Sun, L. Wang, Y. Li, D. He, T. Liu, and W. Chen, "A theoretical dummy dumm dumm dumm," in *Proceedings of the 26th Annual Conference on Learning Theory*, ser. COLT, 2013.

[79] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, (AAAI)*, 2020.

[80] "Synopsys Design Compiler," https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.

[81] N. Takagi, "Powering by a table look-up and a multiplication with operand modification," *IEEE Transactions on Computers*, vol. 47, no. 11, p. 1216–1222, 1998.

[82] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS, 2017.

[83] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *International Conference on Learning Representations*, ser. ICLR, 2019.

[84] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," 2020.

[85] Y. Wang, L. Wang, Y. Li, D. He, T. Liu, and W. Chen, "A theoretical analysis of NDCG type ranking measures," in *Proceedings of the 26th Annual Conference on Learning Theory*, ser. COLT, 2013.

[86] "Gram–schmidt process," https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process, Wikipedia.

[87] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Huggingface's transformers: State-of-the-art natural language processing," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1910.03771

[88] Z. Wu*, Z. Liu*, J. Lin, Y. Lin, and S. Han, "Lite transformer with long-short range attention," in *International Conference on Learning Representations*, ser. ICLR, 2020.

[89] Z. Ye, Q. Guo, Q. Gan, X. Qiu, and Z. Zhang, "Bp-transformer: Modelling long-range context via binary partitioning," 2019.

[90] M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big bird: Transformers for longer sequences," 2020.

[91] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, "Self-attention generative adversarial networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. ICML, 2019.

[92] 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 *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2016.

[93] X. Zhang, F. X. Yu, R. Guo, S. Kumar, S. Wang, and S. Chang, "Fast orthogonal projection based on kronecker product," in *IEEE International Conference on Computer Vision*, ser. ICCV, 2015.

[94] C. Zhou, J. Bai, J. Song, X. Liu, Z. Zhao, X. Chen, and J. Gao, "ATRank: An attention-based user behavior modeling framework for recommendation," in *AAAI Conference on Artificial Intelligence*, ser. AAAI, 2018.

[95] G. Zhou, N. Mou, Y. Fan, Q. Pi, W. Bian, C. Zhou, X. Zhu, and K. Gai, "Deep interest evolution network for click-through rate prediction," in *AAAI Conference on Artificial Intelligence*, ser. AAAI, 2019.