2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10070997

CTA: Hardware-Software Co-design for Compressed Token Attention Mechanism

Haoran Wang[1,2], Haobo Xu[1], Ying Wang[1,2,3], Yinhe Han[1,2,3]
[1]CICS, Institute of Computing Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences, [3]Zhejiang Laboratory
{wanghaoran20g, xuhaobo, wangying2009, yinhes}@ict.ac.cn

***Abstract*—The attention mechanism is becoming an integral part of modern neural networks, bringing breakthroughs to Natural Language Processing (NLP) applications and even Computer Vision (CV) applications. Unfortunately, the superiority of attention mechanism comes from its ability to model relations between any two positions in long sequence, which incurs high inference overhead. For state-of-the-art AI workloads such as Bert or GPT-2, attention mechanism is reported to account up to 50% of the inference overhead. Previous works seek to alleviate this performance bottleneck by removing useless relations for each position and accelerate position-specific operations. However their attempts require selecting from a sequence of relations once for each position, which is essentially frequent on-the-fly pruning and breaks the inherent parallelism in attention mechanism. In this paper, we propose CTA, an algorithm-architecture co-designed solution that can substantially reduce theoretic complexity of attention mechanism, enabling significant speedup and energy saving. Inspired by the fact that the feature sequence encoded by attention mechanism contain a large number of semantic feature repetition, we propose a novel approximation scheme that can efficiently remove that repetition, only calculating attention among necessary features thus reducing computation complexity quadratically. To utilize this algorithmic bonus and empower high performance attention mechanism inference, we devise specialized architecture to efficiently support the proposed approximation scheme. Extensive experiments show that, on average, CTA achieves 27.7× speedup, 634.0× energy savings with no accuracy loss, and 44.2× speedup, 950.0× energy savings with around 1% accuracy loss over Nvidia V100-SXM2 GPU. Also, CTA achieves 22.8× speedup, 479.6× energy savings over ELSA accelerator+GPU system.**

I. Introduction

The attention mechanism has become a critical technique in deep learning for a variety of tasks, such as machine translation [1]–[3], computer vision [4]–[8] and recommendation systems [9]–[13]. The attention mechanism finds the relations between queries and key-value pairs to calculate an output vector for each query, where queries, keys and values are all vectors. Attention also incorporates significant parallelism, which is a primary advantage over traditional sequence modeling methodology such as Recurrent Neural Networks (RNN). On the other hand, Convolutional Neural Networks (CNN) lacks the ability to model relations between distant positions in a sequence. The success of Transformer [14] shows that the attention mechanism combined with even the simplest neural network (NN) structures, such as Feed-Forward Network (FFN), would be sufficient to build a state-of-the-art model, which inspired a dozen of works that followed this idea and make the attention mechanism as the backbone of their models [15]–[27].

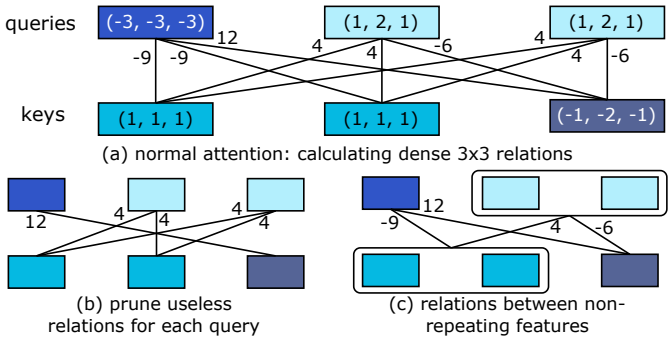

Fig. 1. Visualization of (a) normal attention mechanism, (b) query-specific relation pruning where small relation values are regarded as useless and (c) our innovation. The relations are calculated as dot-product similarities.

Despite the significant ability expansion it has brought to NN, the attention mechanism also incurs a high overhead. This poses severe challenges to general purpose hardware platforms like CPUs and GPUs. All traditional NNs, like CNN and RNN, rely on processing features with weights: CNN organizes its processed features spatially while RNN organizes sequentially. However, attention mechanism lists all the features and find relations among them, which is based on dense matrix multiplications with no pre-trained weights. This fundamental difference makes the techniques used for accelerating traditional NNs [28]–[41] not easily applied to attention.

Based on the characteristics of the attention mechanism, several works [42]–[44] have tried to accelerate attention and explore sparsity contained in attention computation. They all follow the same idea that not all keys are relevant to the query and proposed different techniques to efficiently prune irrelevant keys for each query (Fig. 1 (b)). However this pruning methodology brings about a fundamental problem: Finding relevant keys is specific to each query, which breaks inter-query parallelism of attention mechanism. This methodology defect inevitably confines the previous architectures to process query by query with inefficient computation architecture and memory access patterns.

Different from previous works, we propose a novel methodology to accelerate attention mechanism. Our key observation is that the feature sequence encoded by the attention mechanism contains a large number of semantic feature rep-

etitions. In normal attention, these repetitions incur large amount of repeating computation and unnecessary overhead (Fig. 1 (a)). Thus we propose compressed token attention (CTA), an algorithm-architecture co-design aimed at efficiently extracting non-repeating features to enable high-performance attention inference that is free from repeating computations. Specifically, we extract non-repeating tokens from a token sequence, based on which we calculate non-repeating queries and key-value pairs. Since the number of queries and the key-value pairs are reduced, the number of relations are reduced quadratically, as shown in Fig. 1 (c), whereby 3×3 relations are reduced to 2×2. With all non-repeating relations calculated, model accuracy preserved. Meanwhile, all kinds of inherent parallelism in the attention mechanism are preserved, providing us with abundant architecture design space.

In support of this novel approximation scheme, CTA introduce special algorithmic modifications between major stages of attention computation, while keeping the matrix multiplication formulization of each stage. Our effective token compression scheme reduces the computation needed for all stages of attention and preserves high accuracy. Since CTA contains some special algorithmic operations which are not efficient on CPUs or GPUs, we devised a specialized architecture to accelerate the CTA scheme. The CTA architecture can fully utilize the computation reduction and algorithmic parallelism of CTA approximation scheme to achieve high-performance, energy-efficient and high-accuracy attention inference.

The major contributions of CTA are:

- We propose token compression to efficiently remove feature repetitions, eliminating redundant computations in attention mechanism. Our experiments show that up to 83% of the computations can be avoided with $< 2\%$ accuracy degradation.
- We introduce CTA specialized architecture to fully translate algorithmic computation reduction into high inference performance. Benefiting from the formulation of CTA scheme, CTA architecture is designed with more extensibility compared to previous attention accelerators.
- We validate that CTA can achieve significant speed-up and energy saving compared to general purpose hardware platforms and previous accelerators. Experiments show that CTA achieves up to 67× speedup and 1356× energy efficiency over high-end GPU.

II. Background and Motivation

A. Background

Attention mechanism identifies the relations between input queries and all the key-value pairs. Assuming there are m queries and n key-value pairs, they are embedded to generate two token matrices X^Q ($m \times d_w$) and X^{KV} ($n \times d_w$), where d_w is the dimension of embedded tokens. Attention mechanism first projects tokens to Q, K, V matrices through three linear transformations:

$$Q = X^Q \cdot W^Q,\ K = X^{KV} \cdot W^K,\ V = X^{KV} \cdot W^V$$

Weight matrices are of size $d_w \times d$, so that d dimensional

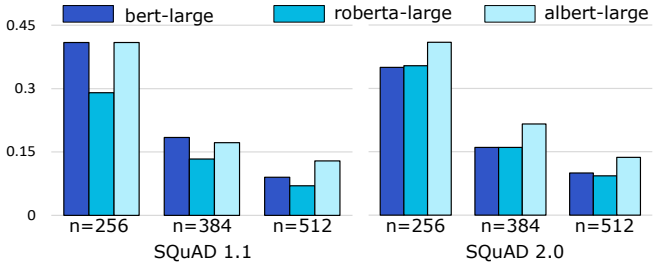

Fig. 2. Proportion of effective relations in attention.

queries, keys and values (row vectors of Q, K, V) are generated. Secondly, attention mechanism calculates scaled-dot product similarities between each pair of query and key resulting in a $m \times n$ attention score matrix S, where S_{ij} represents the similarity between i-th query and j-th key:

$$S = QK^T/\sqrt{d}$$

Thirdly, perform row-wise softmax on attention score matrix to get attention probability matrix:

$$P = Softmax(S), P_{i,j} = e^{S_{i,j}} / \sum_k e^{S_{i,k}}$$

The i-th row of P represents the probabilities that the i-th query attends on all the keys. Finally, attention mechanism produces output vector for i-th query as a weighted sum of the value vectors with attention probabilities, which can be summarized as:

$$O = PV$$

In particular, for self-attention mechanism, queries attend on themselves so there is only one token matrix $X^Q = X^{KV}$, and the procedure described above holds for self-attention.

In NLP applications, attention mechanism is essentially exploring relations between input words and all the words in a context, where the words are embedded to tokens.

B. Motivation

Human languages contain lots of synonyms and similar expressions. Especially in long paragraphs, people tend to use varying expressions to convey the same message. Also, deep NLP models that utilize attention mechanism always stacks multiple layers. It has been studied [45] that each layer of attention only extracts a small span of linguistic structures, such as part-of-speech, constituents, dependencies and so on. For each layer of attention, multiple heads further divide representation space. These result in the clustering phenomenon among both query tokens (X^Q) and key-value tokens (X^{KV}) for each attention head.

The clustering property of tokens is well maintained by linear transformations, and thus queries, keys and values can be clustered accordingly. We exploit novel sparsity in attention mechanism, whereby effective attention relations exist between clustering centroids of queries and keys. Fig. 2 shows the proportion of effective relations in attention from three models. We fixed the sequence length to 256, 384, 512 and evaluated these on datasets SQuAD 1.1&2.0 [46] with a clustering strategy that induces less than 1% accuracy loss. As can be seen, over half of the relations are redundant and the proportion of effective relations decreases significantly as sequence length grows.

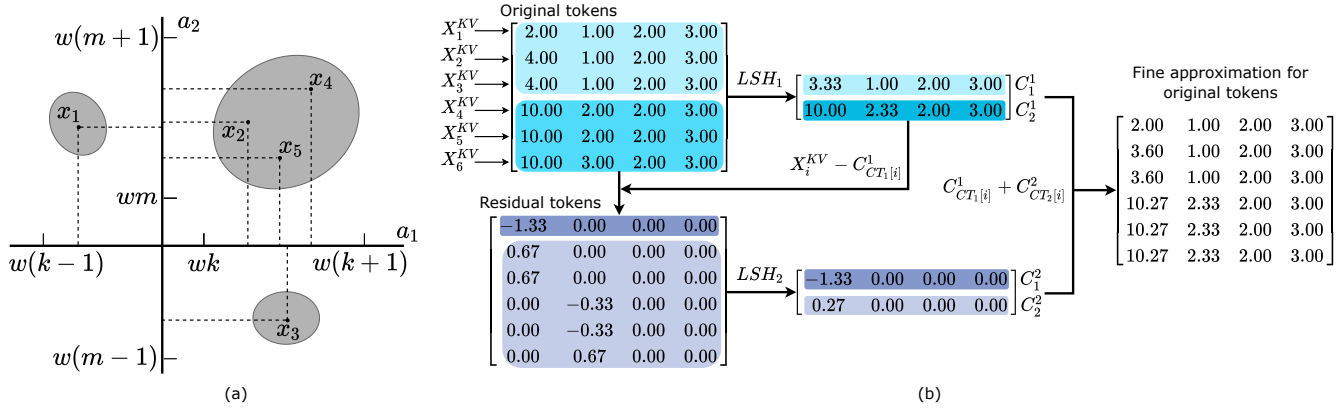

Fig. 3. LSH-based token compression: (a) Hashing of $x_1, ...x_5$ generating 2-dimensional hash codes; (b) Demonstration of two-level clustering token compression.

Inspired by the techniques applied to efficiently process nearest neighbor problems in high-dimensional settings, we argue that token compression can be done with much smaller computational cost than backbone attention computations. Also, we notice that it's possible to transfer attention results on compressed tokens to normal attention results with light-weight algorithmic modifications. These insights suggest that we can develop an efficient approximation scheme to remove redundant computation at the earliest stage of the attention mechanism by compressing tokens, and then together with proper architecture design, we can implement a high performance attention inference system.

III. Compressed Token Attention

Attention mechanism suffers similar high overhead induced by high-dimensional vector operations like traditionally seen in information retrieval and CV tasks. To ease this burden, many vector compression techniques [47]–[52] were proposed. Inspired by them, we develop our token compression technique for accelerating attention operations. We first efficiently cluster tokens and adopt centroids as compressed tokens (Section III-A, B). Then, we project compressed tokens to compressed Q, K, V matrices by corresponding linear transformations (Section III-C). Then scaled dot-product similarities are taken between compressed queries and keys (Section III-C). Finally, we introduce attention probability aggregation, enabling calculating outputs as the weighted sum of compressed value vectors (Section III-C). We analyse the complexity and show that CTA achieves significant computation reduction with low approximation overhead (Section III-D).

A. Locality Sensitive Hashing-based Clustering

Locality sensitive hashing (LSH) [53]–[59] is an efficient technique for grouping high-dimensional vectors. Formally, sample d data from normal distribution $\mathcal{N}(0,1)$ to form a d-dimensional directional vector a, and sample data b from uniform distribution $\mathcal{U}(0,w)$, where w is a predefined width of hash buckets. The sampling process is repeated l times where l is the length of the hash code. For a given d-dimensional vector x, the hash code $h(x)$, which is a l-dimensional integer vector, is calculated as:

$h(x) = (h_{a_1,b_1}(x), ..., h_{a_l,b_l}(x))$, $h_{a,b}(x) = \lfloor (\langle x, a\rangle + b)/w \rfloor$

As shown in Fig. 3 (a), there are two randomly sampled directions a_1, a_2 and five data vectors $x_1, ...x_5$. The projections of data vectors onto direction vectors are shown with dashed lines. The interval where the projection falls in determines the hash value and the hash code combines hash values in all the directions. For example, we can see that x_2, x_4, x_5 share the same hash code (k,m), while the hash codes for x_1, x_3 are ($k-1$,m), (k,$m-1$) respectively.

In our application, we intend to do LSH on all the token vectors, which can be summarized by the following formula:

$$H = \lfloor (A \cdot X^T + B)/w \rfloor \tag{1}$$

X is the token matrix and A, B are LSH hyperparameter matrices with i-th row of A being a randomly sampled direction vector a_i and i-th row of B repeating the bias b_i. The i-th column of the result matrix H is the hash code for the i-th token.

We intend to group tokens with the same hash code into one cluster (x_2, x_4, x_5 in Fig. 3 (a)). To efficiently obtain cluster index for each token, we devise a data structure called cluster tree to dynamically maintain the clustering information. The cluster tree consists of a single root and l layers below the root. Nodes in the i-th layer correspond to i-th values in hash codes and each path from root to leaf corresponds to a unique hash code. In this way, we can establish an one-to-one mapping between leaves and hash codes. This property can be guaranteed if we update cluster tree with the algorithm shown in Fig. 4 (a). The thread receives hash values from hash codes sequentially and updates i-th layer when it receives i-th hash value. Note that up to l threads can work in parallel, as long as they update different layers at the same time.

We denote cluster indices for tokens as cluster table CT, where $CT[i]$ is the cluster index for i-th token.

B. Token Compression

Motivation. If two tokens have small L2 distance, it's safe to conclude that they encode similar features. For a cluster of tokens with small L2 distances, their semantic features can be represented by the cluster centroid. Thus original sequence of

(a) Cluster tree thread

```
/* initialization */
cl_cnt = 0 // shared by multiple threads
curr_depth = 0 // specific to each thread
curr_node = root // specific to each thread
/* a single thread for updating cluster tree */
void thread (int hash_val) {
  if curr_depth == l-1:
    if curr_node->children[hash_val]:
      // existing leaf: report cluster index
      report curr_node->children[hash_val].cl_idx
    else:
      // creating new leaf: new cluster identified
      curr_node->children[hash_val]
        = new leafnode(cl_idx=++cl_cnt)
      report cl_cnt
    curr_node = root
  else:
    if not curr_node->children[hash_val]:
      curr_node->children[hash_val]
        = new internalnode()
    curr_node = curr_node->children[hash_val]
  curr_depth = (curr_depth+1) mod l
}
```

(b) Centroid aggregation

```
/* initialization */
C[:][:] = 0, cntr[:] = 0
/* calculate centroids from tokens */
void centroid_agg (float X[][], int CT[]) {
  for i = 0 to n-1:
    C[CT[i]][:] += X[i][:]
    cntr[CT[i]] += 1
  for j = 0 to k-1:
    C[j][:] /= cntr[j]
}
```

Fig. 4. Pseudocode for cluster tree thread and centroid aggregation.

tokens can be compressed to their clustering centroids with little loss of semantic features.

Token Compression Scheme. The proposed LSH-based clustering groups token vectors with small L2 distances into one cluster. For simplicity, we take the average of all vectors in a cluster to be the centroid. Given cluster table CT, cluster centroids can be easily aggregated according to the algorithm shown in Fig. 4 (b).

Now we formulize our token compression scheme as follows:

1. For query tokens X^Q, we apply a separate LSH clustering LSH_0, compressing X^Q to C^0 with cluster index table CT_0. C^0 consists of k_0 rows of centroids: $C^0_1, ..., C^0_{k_0}$.
2. For key-value tokens X^{KV}, we apply two-level LSH clustering LSH_1 and LSH_2, compressing X^{KV} to C^1, C^2 with cluster index tables CT_1, CT_2. Similar to the notation for LSH_0, C^1 consists of centroids $C^1_1, ..., C^1_{k_1}$ and C^2 consists of centroids $C^2_1, ..., C^2_{k_2}$.

Fig. 3 (b) explains our two-level clustering procedure. Tokens are first clustered by LSH_1, generating LSH_1 centroids C^1_i. The residual tokens are the difference between original tokens and their corresponding LSH_1 centroids. Residual tokens represent the approximation error if we directly use LSH_1 centroids to replace tokens. To lower the approximation error, residual tokens are further clustered by LSH_2, generating LSH_2 centroids C^2_i. In this way tokens can be more accurately approximated as the sum of LSH_1 and LSH_2 centroids. The

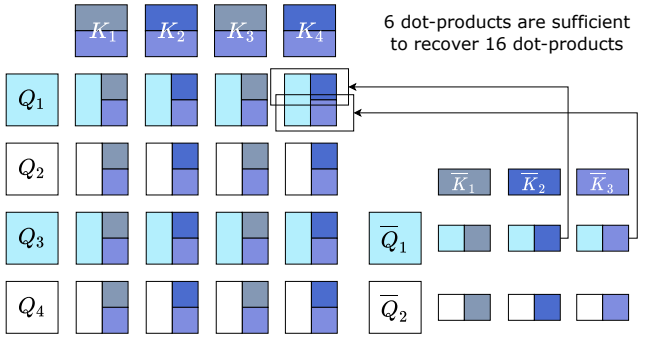

Fig. 5. Visualization of normal attention scores (left part) comparing with CTA compressed scores (right part).

following formula gives fine approximation for tokens:

$$X^Q_i \doteq C^0_{CT_0[i]}, X^{KV}_i \doteq C^1_{CT_1[i]} + C^2_{CT_2[i]} \tag{2}$$

C. Attention among Compressed Tokens

Motivation. After obtaining compressed tokens, it's important to reorganize attention computations and recover normal attention outputs with cheap computational cost. We carefully design CTA to explore computation reduction in all of the computation steps for attention mechanism: linear transformations on tokens, scaled dot-product similarity calculation, score normalization and output calculation.

Linears on Compressed Tokens. In normal attention, queries, keys and values are obtained by applying three linear transformations on tokens. The proposed CTA, however, only needs to apply these linear transformations on compressed tokens. Denote $(k_1+k_2)\times d$ matrix C^{cat} as the concatenation of C^1, C^2 in row dimension, then CTA linear transformations generating compressed queries, keys and values can be formulized as:

$$\overline{Q} = C^0 \cdot W^Q, \overline{K} = C^{cat} \cdot W^K, \overline{V} = C^{cat} \cdot W^V \tag{3}$$

According to (2) and the property of linear transformation, original queries, keys and values can be approximated by the compressed ones.

$$\begin{aligned} Q_i &\doteq \overline{Q}_{CT_0[i]} \\ K_i &\doteq \overline{K}_{CT_1[i]} + \overline{K}_{k_1+CT_2[i]} \\ V_i &\doteq \overline{V}_{CT_1[i]} + \overline{V}_{k_1+CT_2[i]} \end{aligned} \tag{4}$$

Scaled Dot-Product Similarity. Normally, scaled dot-product similarity is calculated between each pair of query and key to generate $m \times n$ ($n \times n$ for self-attention) attention scores. This is really expensive when the number of queries and keys grows large. In CTA scheme, only the similarities between compressed queries and keys need to be calculated, generating $k_0 \times (k_1 + k_2)$ attention scores.

$$\overline{S} = (\overline{Q} \cdot \overline{K}^T)/\sqrt{d} \tag{5}$$

$\overline{S}$ represents the compressed attention score matrix, where $\overline{S}_{i,j}$ is the similarity between i-th compressed query and j-th compressed key. Fig. 5 visualizes how the original attention scores can be recovered from compressed scores. Two compressed queries and three compressed keys are in different colors.

```
Probability aggregation
1  /* initialization */
2  AP[:][:] = 0
3  /* calculate AP from CS */
4  void prob_agg (float CS[][], int CT1[], int CT2[]) {
5    for i = 0 to k0-1
6      for j = 0 to n-1
7        x1 = CT1[j], x2 = k1+CT2[j]
8        p = exp(CS[i][x1]+CS[i][x2])
9        AP[i][x1] += p
10       AP[i][x2] += p
11 }
```

Fig. 6. Pseudocode for attention probability aggregation.

Vertical adjacency of two rectangles means addition, while horizontal adjacency means dot-product. It can be seen that all the similarities in the left part can be recovered by simply adding two of the similarities in the right part. Formally, original attention scores can be approximated as:

$$S_{i,j} \doteq \overline{S}_{CT_0[i],CT_1[j]} + \overline{S}_{CT_0[i],k_1+CT_2[j]} \tag{6}$$

Score Normalization. For CTA scheme, score normalization is also called probability aggregation, which is formulized differently from score normalization in normal attention. The intuition is to approximate normal attention output as weighted sum of compressed value vectors:

$$O_i = \sum\nolimits_j exp(S_{i,j})V_j \doteq \sum\nolimits_j AP_{CT_0[i],j}\overline{V}_j \tag{7}$$

AP in equation (7) is of size $k_0 \times (k_1 + k_2)$, called aggregated attention probabilities in CTA, representing compressed queries' attention probabilities on compressed keys (note that we omit softmax denominator for now). AP can be calculated from $\overline{S}$ using the algorithm shown in Fig. 6 (array CS represents $\overline{S}$). To derive this algorithm, simply substitute $S_{i,j}, V_j$ in equation (7) with the approximation shown in equation (6)(4) and merge the coefficients.

Output Calculation. Finally, output calculation of CTA can be formulized as:

$$\overline{O} = AP \cdot \overline{V} \tag{8}$$

The normal attention output for i-th query O_i can be approximated by CTA output $\overline{O}_{CT_0[i]}$. Notice in normal attention, output vector O_i should be divided by the sum of attention probabilities $\sum_j P_{i,j}$. In CTA, this is equivalent to divide output $\overline{O}_i$ with $\sum_j AP_{i,j}/2$, because approximated attention probabilities are accumulated twice in each row of AP (see line 9, 10 in Fig. 6).

D. Complexity Analysis

For clarity, we analyze complexity for self-attention here and the analysis for cross-attention is similar. The overhead introduced by CTA approximation scheme is divided into three folds:

1) Hashing: LSH_0, LSH_1, LSH_2 induce $3lnd$ multiplications and $3ln(d-1)$ additions (eq (1)).
2) Centroid aggregation: Calculating C^0, C^1, C^2 induces $(k_0 + k_1 + k_2)d$ multiplications and $3dn$ additions (Fig. 4 (b)).
3) Probability aggregation: Calculating AP from $\overline{S}$ induces $3k_0n$ additions (Fig. 6).

The computations of major stages of attention is reduced to:

1) Linears: multiply-and-accumulate (MAC) operations are reduced from $3nd^2$ to $(k_0 + 2k_1 + 2k_2)d^2$ (eq (3)).
2) Dot-product similarity calculation: MACs are reduced from n^2d to $k_0(k_1 + k_2)d$ (eq (5)).
3) Score normalization: exponential operations are reduced from n^2 to k_0n (Fig. 6).
4) Output calculation: MACs are reduced from n^2d to $k_0(k_1 + k_2)d$ (eq (8)) and output divisions are reduced from nd to k_0d.

Note that k_0, k_1, k_2 is substantially smaller than original sequence length n. The computations for linears and score normalization are reduced linearly, while similarity and output calculations are reduced quadratically. Also hash length l is small (6 in our implementation) thus approximation overhead is much smaller than backbone computation.

IV. CTA Hardware Architecture

The token compression in CTA scheme contains sequential logics which can only be implemented into coarse CUDA kernels. We implement a CUDA version of CTA and optimize implemented kernels with Microsoft Antares [60] to minimize the impact from bad implementation. The optimized latency of CTA on GPU is $1.0 - 2.1\times$ (varying with compression ratio) compared to normal attention. Therefore, specialized architecture is needed to release the potential of CTA.

A. Hardware Overview

The proposed CTA specialized hardware takes tokens and weights as input to calculate one head of attention outputs. Fig. 7 shows the high-level block-diagram of CTA hardware. It consists of four parts, including a computation engine and three light-weight modules to support special algorithmic operations introduced in CTA scheme:

(1) Systolic Array Computation Engine (SA): SA consists of processing elements (PEs) and post processing elements (PPEs) that can be mapped to calculate major computation phases in CTA scheme.

(2) Cluster Index Module (CIM): Streams in hash values to maintain the data structure stored in local registers and generates cluster index table accordingly.

(3) Centroid Aggregation Module (CAG): Calculates cluster centroids based on tokens and cluster index table.

(4) Probability Aggregation Module (PAG): Takes compressed attention scores and cluster tables to calculate aggregated attention probabilities.

Our CTA scheme keeps matrix multiplication formulization for all computation intensive stages, thus we design our main computation engine based on systolic array architecture and design PEs and PPEs to support multiple computation stages. The other parts of hardware act as auxiliary modules that provide data support for SA execution.

For one head of attention processing, token/KV memory and weight memory fetches tokens and weights from host

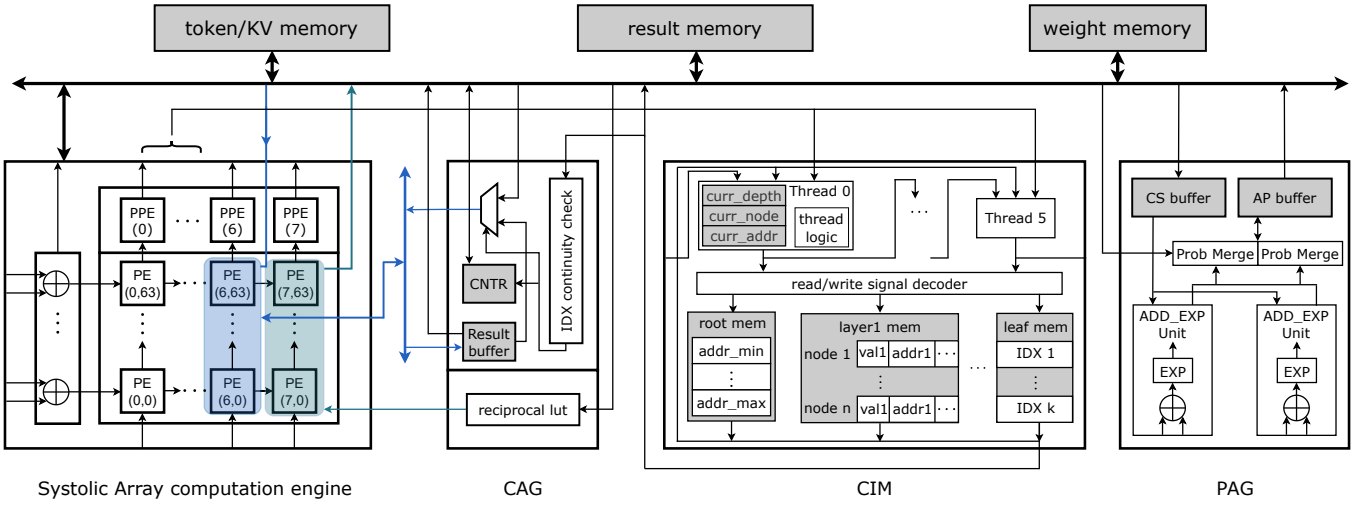

Fig. 7. Block diagram of CTA hardware design. Colored parts of SA are reused by CAG.

device like GPUs or other general purpose devices. SA first reads tokens and LSH parameters to calculate hash codes. The hash codes are streamed directly to CIM, where cluster tables are generated and stored in weight memory. CAG calculates compressed tokens based on cluster tables and stores them in result memory. Then, SA reads linear layer weights and compressed tokens to calculate compressed queries, keys and values in batched manner. Keys and values are stored in token/KV memory, while queries do not need to be stored due to the optimization detailed in Section V-B. After each batch of queries are calculated, SA reads keys to calculate one batch of compressed attention scores, from which PAG calculates aggregated attention probabilities. Finally SA reads values and aggregated probabilities to calculate one batch of outputs and store in corresponding rows of result memory.

B. Modular Design

(1) Systolic Array Computation Engine

We equip our SA with $b \times d$ PEs, where b is SA width, also called batch size in our scenario. Each PE is equipped with a value register, a result register, port registers, an adder, a multiplier, and can be configured to support multiple dataflows. At the top of each column of PEs, there is a PPE. Each PPE consists of an adder, a multiplier, registers and logics to support phase-specific post processing. On the left side of first column of PEs, there is a column of adders to support calculating residual tokens. SA can be configured to run four computation phases under two dataflow configurations, as demonstrated in Fig. 8.

LSH clustering phase (equation 1). SA reads LSH parameter matrix A from memory to prepare the value register of $\text{PE}_{i,j}$ with $A_{i,j}$. At cycle t, $\text{PE}_{i,j}$ adds the product of $A_{i,j}$ (in value register) and $X_{t-(i+j),j}$ (coming from left) to the value coming from bottom. SA is configured with dataflow 1: each PE forward the addition result to the top and the value from left directly to the right (Fig. 8 (a)). PPE_i reads LSH parameters b_i, $1/w$ from memory to add b_i to the value coming from bottom and then multiplies $1/w$. The integer bits of the PPE output is the hash value.

To implement 2-level LSH, first level results C^1, CT_1 are buffered in result and weight memory respectively. When calculating second level, elements of X^{KV}, C^1 are read simultaneously, where reads of C^1 are addressed by CT_1. These elements are forwarded to leftmost column of adders in SA (Fig. 7) to do subtraction, generating residual tokens ready for second level LSH computation.

Linear phase (equation 3). SA is configured as in first phase, only the processed data are different (Fig. 8 (a)). In particular, when calculating queries, the results coming out of the top PE are broadcast through shortcuts to all the PEs in the same column. Each PE is configured to update its value register using the data in shortcut at the next cycle it receives the last weight data from left input stream (see Section V-B). This is possible because each PE completes query calculation after receiving d query weight values and the height of SA is also d. Thus at each cycle, there is a PE in each column who has just finished query calculation and is ready to update its value register with query result generated by top PE.

Score calculation phase (equation 5). SA is still configured as in first phase (Fig. 8 (a)). Note each PPE counts the maximum of the first k_1 scores, and subtract that maximum value from the following k_2 scores when they pass by. By doing so, the approximated attention score (see equation (6)) is kept small, which makes it convenient for our exponent look up table implementation in PAG (see Section IV-(4)). Note that this is equivalent to subtract the same value from all the attention scores in one row, which does not affect the final output after dividing the row-wise softmax denominator.

Output calculation phase (equation 8). Result registers in PEs are cleared to 0 before calculating. At cycle t, $\text{PE}_{i,j}$ accumulates the product of $AP_{i,t-(i+j)}$ and $\overline{V}_{t-(i+j),j}$ into result register, which stores $\overline{O}_{i,j}$ after accumulation completes. The SA is configured with dataflow 2: each PE forward the values from left and bottom directly to right and top (Fig. 8 (b)). PPE_i accumulates the passing probabilities to get $\sum_j AP_{i,j}$, which

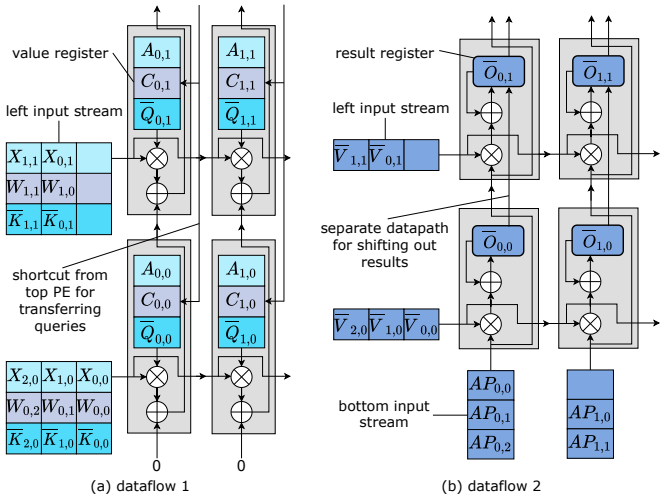

Fig. 8. Two dataflow configurations of SA (only showing 2×2 PEs in the bottom-left corner). In (a), different colors represent data for different computation phases, not physical replications, where data for LSH clustering, linear transformation, score calculation phases are listed from top to bottom.

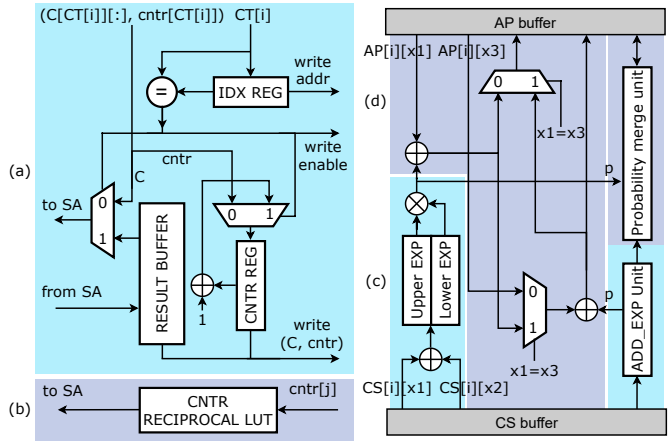

Fig. 9. Architecture of CAG (left part) and PAG (right part). Note that in (c)(d), $x_1 = CT_1[j], x_2 = k_1 + CT_2[j], x_3 = CT_1[j+1], x_4 = k_1 + CT_2[j+1]$, represent indices (Fig. 6 line 7) belonging to two consecutive inner for loop iterations j and j+1.

is twice the softmax denominator. We implement a look up table shared by PPEs to provide softmax denominator values. Once a SA column finishes accumulation, the results in result registers are shifted up along a separate datapath, when each PPE multiplies looked up softmax denominator to the coming value and gives output values.

(2) Cluster Index Module

CIM consists of l thread units, a read/write signal decoder unit and separate memory blocks for different layers of cluster tree. Threads communicate with memory blocks through decoder unit. For root memory, it stores a single root node consists of all the addresses of its children. For internal layer memory, it stores internal nodes, with each node recording pairs of hash value and address of its children. Leaf memory stores all the leaf nodes, each leaf node only records the identified cluster index.

Each thread processes a coming hash value and issues signals to update and read layer memory per cycle. It has three registers curr_node, curr_addr, curr_depth, to maintain the necessary information of currently processing node. The thread control logic is shown in Fig. 4 (a). Note that the pointers in Fig. 4 (a) (also the addresses in CIM) are allocated and managed linearly, making it convenient to implement in accelerators.

Hash values are streamed from SA to threads in parallel, and the pattern of SA generating hash values determines that thread $i+1$ always works at the (cycling) previous layer of thread i. Note that if thread $i+1$ reads the same address as thread i's curr_addr, modifications issued by thread i have not been written, in which scenario modifications from thread i are bypassed to thread $i+1$. Every cycle, there is a thread whose curr_depth register equals l, and the identified cluster index is reported by this thread.

(3) Centroid Aggregation Module

CAG can be divided into two parts, namely CACC for accumulating and CAVG for averaging (Fig. 9 (a)(b)). They implement the logic shown in Fig. 4 (b) line 5-7 and 8-9. In order to reduce area and energy overhead, we reuse arithmetic units in SA to accommodate computations for CAG.

CACC reuse d adders from one SA column to accommodate additions (line 6). At i-th cycle, SA column reads X[i][:] while CACC provides so far accumulated C[CT[i]][:]. Each cycle CACC buffers the addition result from SA and checks whether the coming cluster index is the same as previous one. If so, the accumulation happens to the same centroid as previous cycle, CACC provides buffered result to SA for further accumulation. Otherwise CACC writes buffered result back to result memory and provides another cluster centroid read from memory to SA. CACC also accumulates counters for each cluster with similar control logic.

CAVG reuse d multipliers from one SA column to accommodate divisions (line 9). At j-th cycle, SA column reads accumulated C[j][:] and CAVG provides reciprocal of cntr[j]. CAVG consists of Look-Up-Table indexed by possible counter values, recording their reciprocals.

(4) Probability Aggregation Module

We design PAG with hardware parallelism to implement attention probability aggregation efficiently. The processing logic is shown in Fig. 6. We design a tile-based architecture for PAG, with each tile processing inner for-loop (line 6) sequentially while iterations of outer for-loop (line 5) are unrolled to different tiles to be executed in parallel. PAG reads compressed scores from CS buffer and writes aggregated probabilities to AP buffer, where CS, AP are shared by different tiles. The architecture of one tile is shown in the right part of Fig. 9. Each tile consists of two ADD_EXP units (Fig. 9 (c)) and two Probability merge units (Fig. 9 (d)). Each tile processes two consecutive iterations of inner for-loop per cycle. The ADD_EXP unit implements line 8 to add scores and then take exponent. We implement exponent calculation similarly to the LUT-based method in A^3 [42], with LUT shared among ADD_EXP units. The additions (line 9-10) in two consecutive iterations j and j+1 may happen to same point of AP (for example CT1[j] = CT1[j+1], in Fig. 9 (d) x_1=x_3),

TABLE I
MAPPING PROCEDURE OF CTA ONTO PROPOSED HARDWARE.

	SA column 0-5	SA column 6	SA column 7	CIM	PAG
1	$H_1 = LSH_1(A, X^{KV})$	$C^1, ctr^1 = CACC(CT_1, X^{KV})$	None	$CT_1 = CIM(H_1)$	None
2	$H_0 = LSH_0(A, X^{Q})$	$C^0, ctr^0 = CACC(CT_0, X^{Q})$	$C^1 = CAVG(C^1, ctr^1)$	$CT_0 = CIM(H_0)$	
3	$H_2 = LSH_2(A, rX^{KV})$	$C^2, ctr^2 = CACC(CT_2, rX^{KV})$	$C^0 = CAVG(C^0, ctr^0)$	$CT_2 = CIM(H_2)$	
4	None	None	$C^2 = CAVG(C^2, ctr^2)$	None	
5	for t=0, ..., $\lceil \frac{k_1+k_2}{8} \rceil$	$\overline{K}_{8t+:8} = LIN(C^{cat}_{8t+:8}, W^K)$		None	None
6		$\overline{V}_{8t+:8} = LIN(C^{cat}_{8t+:8}, W^V)$			
7		$\overline{Q}_{0+:8} = LIN(C^0_{0+:8}, W^Q)$		None	None
8		$\overline{S}_{0+:8} = SCORE(\overline{Q}_{0+:8}, \overline{K})$			
9	for t=1, ..., $\lceil \frac{k_0}{8} \rceil$	$\overline{Q}_{8t+:8} = LIN(C^0_{8t+:8}, W^Q)$		None	$AP_{8(t-1)+:8} = PAG($
10		$\overline{S}_{8t+:8} = SCORE(\overline{Q}_{8t+:8}, \overline{K})$			$\overline{S}_{8(t-1)+:8}, CT_1, CT_2)$
11		$\overline{O}_{8(t-1)+:8} = OUT(AP_{8(t-1)+:8}, \overline{V})$			None
12	None			None	PAG last batch
13	$\overline{O}$ last batch				None

in which case the additions should be merged by Probability merge unit.

C. Design Details

Hardware Configuration. We adopt $n = 512, d = 64, b = 8, l = 6$ for CTA hardware configuration. n, d are taken as the maximum sequence length and token dimension in all our evaluated models, which determine size of memory and the height of SA in our hardware. $b = 8$ is the width of SA computation engine and the number of tiles in PAG, one can configure b to larger values to achieve even higher parallelism and further accelerate the computation. We utilize $l = 6$ as the length of hash codes, which determines the number of threads in CIM. In our scenario, long hash codes results in less effective token compression, while short hash codes incurs low accuracy induced by aggressive clustering. Our experiment shows that $l = 6$ achieves good trade-off between compression ratio and model accuracy.

Number Quantization. We adopt fixed-point (FxP) representation throughout the computation. Specifically, we quantize tokens to 13 bit, with 6 integer bits and 7 fractional bits. We quantize the values in weight memory to 12 bits, where different types of weights are quantized with minimal integer bits to cover the value range leaving the rest bits as fractional bits. For example, the hash parameter matrix A, whose values are sampled from standard normal distribution, is quantized with 3 integer bits and 9 fractional bits due to the three sigma guideline. The centroids and compressed queries, keys, values are all quantized to 12 bits with 6 integer bits and 6 fractional bits. Our experiments show that this quantization scheme introduce less than 0.1% accuracy loss.

V. MAPPING AND OPTIMIZATION

A. Mapping Procedure

To run the CTA mechanism on the proposed hardware with maximal efficiency, we optimize the mapping procedure. For clarity we introduce some notations to represent operation mappings. $LSH(), LIN(), SCORE(), OUT()$ represent the mappings of SA calculating under the four phases described in Section IV-B(1). $CACC(), CAVG(), CIM(), PAG()$ represent mappings of auxiliary modules.

Table I lists the mapping procedure of hardware components for a whole CTA attention inference, where rows 1-13 show chronological mapping steps, while columns indicated by the header show different hardware components. The rX^{KV} in the third row represents the residual token matrix (see Section-III B). Note that CACC and CAVG reuse 6-th and 7-th SA column for doing arithmetic, where they are listed correspondingly in Table I. Notations like $\overline{K}_{8t+:8}$ in row 5 means the slice of tensor $\overline{K}$ consisting of 8 consecutive rows starting from $8t$-th row.

B. Optimizations

Minimize overall latency. The overall latency is dominated by latency of SA mappings. Minimizing overall latency is essentially minimizing idle time of SA.

From a coarse granularity perspective, we optimize our mapping order to hide the latency caused by data dependency between operations. Take the mapping order of LSH-based clustering operations as an example, $LSH_1(A, X^{KV})$ is arranged ahead of $LSH_0(A, X^{Q})$ in order to avoid delaying $LSH_2(A, rX^{KV})$.

From a fine granularity perspective, we should also keep every PE in SA busy. The dataflow of SA determines that at the beginning and end of each step, some PEs are doing bubble computations without valid data. It's essential to pack effective inputs belonging to consecutive steps to skip bubble computations since our mapping procedure consists of many steps on SA.

Fig. 10 visualizes our bubble removing schedule. Between steps, value registers may keep old values (a), or be updated with values read from memory (b), or be updated with values from shortcuts (c). Also, we optimize the scenario when one of the step is at output calculation phase (d). For (a), simply configure each SA row to read data for next step right after it reads the last data of current step. For (b), each SA row should first read data to update value registers in PEs before reading data for next step, and each PE is configured to pass data horizontally before being configured to calculate for next step (Fig. 10 (b) cycle 1-2). For (c), each row pauses for 1 cycle to make time for PE updating value register using the value broadcast in shortcut, which prepares value register with query data required for next step calculations (Fig. 10 (c) cycle

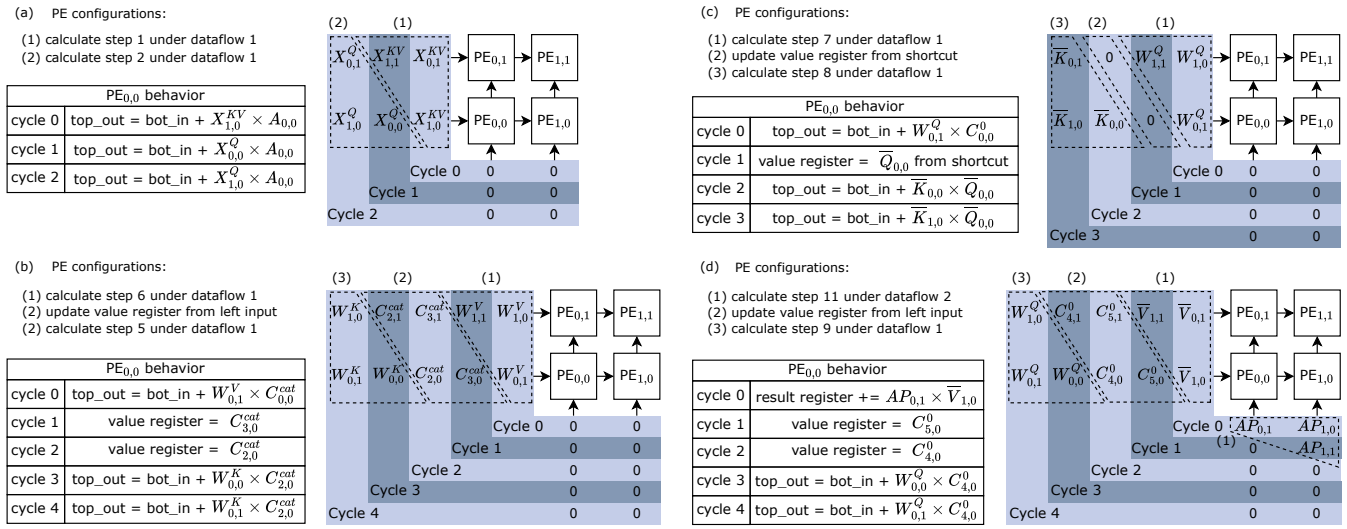

Fig. 10. Demonstration of bubble removing, for simplicity, we assume the SA and processed matrices are of size 2×2 here. All possible situations for bubble removing between consecutive steps (rows in Table I) are shown in (a)-(d). Dashed boxes separate inputs for different PE configurations.

1). For (d), SA row reading is configured similar to (b), SA column reads data for step 11 (output phase) calculation and then reads 0. Note that once all the PEs in a SA column finishes step 11 calculation, output values stored in result registers can be shifted up along separate result register chain (Fig. 8 (b)), not interfering with SA computation dataflow.

Explore hardware parallelism. Different from previous works, we are able to adopt systolic array as major computation architecture. We owe this to the fact that CTA approximation scheme keeps all kinds of inherent parallelism in attention. Besides, auxiliary modules are also designed with proper parallelism. CIM is designed with l threads to process l hash codes coming from SA each cycle. Note the dataflow of SA makes sure that each cycle it produces hash codes belonging to different layers, enabling l threads of CIM work in parallel without hazard. Tile-based PAG architecture utilizes parallelism in line 5-6 of Fig. 6. These parallel design in auxiliary modules make sure that SA computations are not delayed.

Reduce memory overhead. Writing all the intermediate results to dedicated memory and keeping them during the whole process is expensive and unnecessary. Also, some data used in early steps of mapping are useless for later steps, whose occupying memory could be recycled to store later results. The CIM architecture supports processing hash values on-the-fly, saving the memory for storing hash values. Once compressed tokens are obtained, original tokens stored in token/KV memory will never be used, which enable us to reuse that memory to store compressed keys and values. Following the same logic, we design result memory to support both compressed tokens and output storage. Also, mapping the same batch calculation of $\overline{K}$ and $\overline{V}$ next to each other (Table I step 5-6) saves half the reads for updating PE value registers when calculating linear transformations for $\overline{K}, \overline{V}$. Lastly, the shortcut design passing query results directly to PE value registers saves the memory overhead for storing queries, since the the mapping procedure consumes each batch of queries right after they are calculated (Table I step 7-8, 9-10).

VI. Evaluation

A. Workloads

To validate that CTA can effectively accelerate attention mechanism (including linear transformations for QKV), we evaluate both discriminative and generative models on several datasets. For discriminative models, we evaluate BERT (large), RoBERTa (large), ALBERT (large) [15]–[17]. We also evaluate a representative generative model GPT-2 (large) [18]. We evaluate discriminative models on two question answering datasets and a text-classification dataset: Stanford Question Answering Dataset (SQuAD) 1.1 & 2.0 [46], and IMDB movie review dataset [61]. We evaluate GPT-2 (large) on language modeling dataset WikiText-2 [62].

B. Accuracy Evaluation

Methodology. We adopt open-source implementations of models from HuggingFace [63] running with PyTorch 1.9.0 [64]. We implement a PyTorch extension to simulate CTA approximation scheme and integrate it to models' implementations. We make sure that the operations in approximation extension are differentiable, enabling fine-tuning models. We evaluate end-to-end accuracy for each workload: F1 score for SQuAD, raw accuracy for IMDB and perplexity for WikiText-2. Note that perplexity for WikiText-2 measures how well a model predicts a sample with smaller values representing better performance.

Accuracy and Compression ratio. We introduce RL, RA to measure the reduction of computation, which reflects the effect of token compression. RL represents the ratio of the amount of computation for approximated linear transformations compared to that in original attention, and RA represents the ratio for the rest operations in attention mechanism with

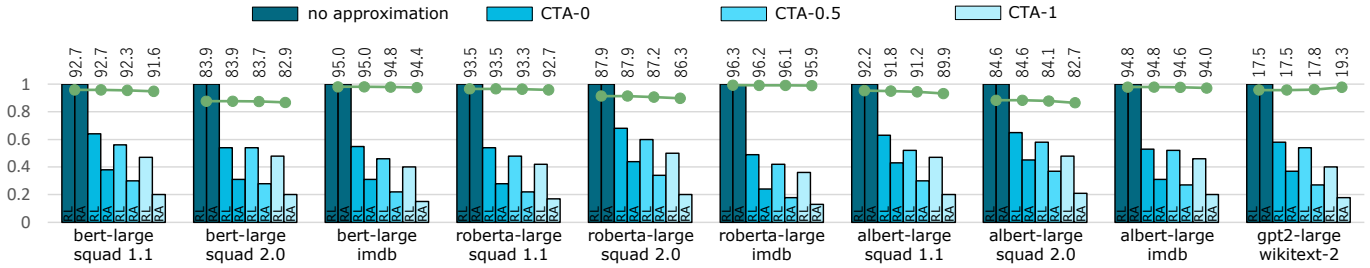

Fig. 11. Model accuracy (lines) and RL, RA (bars) for CTA-0, CTA-0.5, CTA-1 over 10 evaluated model-dataset combinations.

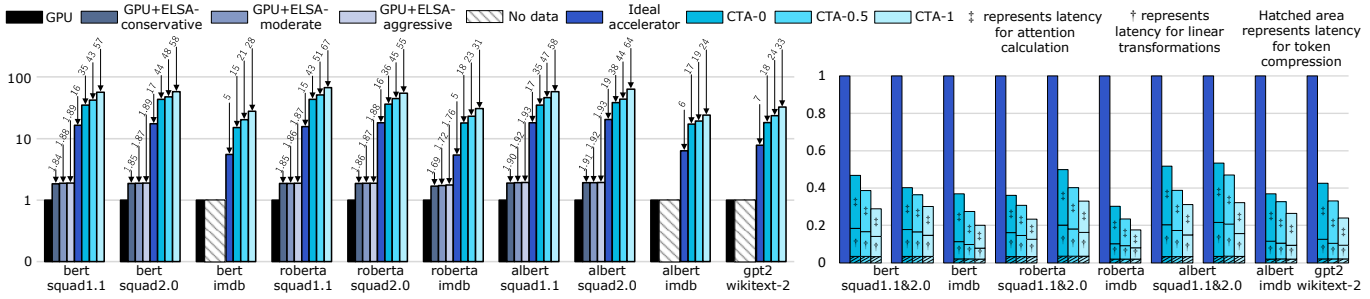

Fig. 12. Left: normalized throughput of attention mechanism. Right: normalized attention mechanism latency.

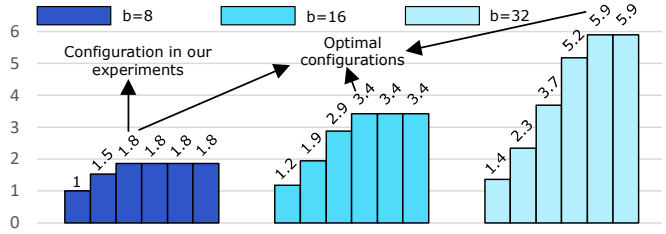

Fig. 13. Normalized throughput of attention mechanism under different hardware configurations. Each cluster of columns represent PAG degree of parallelism = 4, 8, 16, 32, 64, 128 from left to right.

quadratic complexity (including similarity calculation, Softmax normalization and output calculation). In the following, we will simply refer to these two parts of attention mechanism as linear transformations and attention calculations.

For each testcase, we first evaluate model without approximation (RL=1,RA=1) to get baseline accuracy. We then introduce approximation extension to model and fine-tune the model for 1 hour on average to recover accuracy. We validate three more accuracy results with decreasing RL and RA for each testcase. We refer CTA with on average 0%, 0.5%, 1% accuracy loss as CTA-0, CTA-0.5 and CTA-1 respectively. Fig. 11 shows that on average CTA-0, CTA-0.5, CTA-1 consume 58.3%, 52.2%, 44.4% linear computation, and 35.2%, 27.5%, 18.4% attention computation compared to original attention mechanism. This proves CTA can effectively remove repeating relations while keeping representation ability of attention mechanism.

C. Performance Results

Methodology. We implement a cycle-level simulator (summing latency of all mapping steps in Table I) for CTA running at 1GHz and integrate it to model implementations to evaluate throughput performance of 12×CTA. We select server GPU NVIDIA V100-SXM2 with 32GB memory as baseline hardware platform. We measure GPU performance of well-optimized HuggingFace [63] models running dev/test sets using HuggingFace suggested scripts and set maximum sequence length to 512. We choose batch size for each workload to achieve best throughput and measure GPU performance with PyTorch Profiler. We also reproduce the latency of ELSA [43] and compare throughput performance with 12×ELSA. Since ELSA does not support accelerating linear transformations, we compare CTA performance with ELSA+GPU system as suggested in their paper, where linear transformations are mapped on GPU while the rest of attention operations are accelerated by ELSA. Note that 12×CTA satisfies iso-area comparison setting with 12×ELSA. Furthermore, we compare with ideal accelerator equipped with same number of multipliers as CTA, which also runs at 1GHz and always achieves peak throughput, but does not utilize any optimizations introduced in CTA.

Design Space Exploration. We evaluate the average throughput of CTA under different hardware configurations as shown in Fig. 13. The width b of SA computation engine, and the degree of parallelism of PAG (how many iterations in Fig. 6 can PAG process per cycle) are two major factors affecting throughput. We find that setting PAG parallelism degree as twice of SA width provides best throughput with lowest hardware overhead. In our architecture implementation, each tile of PAG's parallelism degree is 2, which means keeping the number of tiles of PAG the same as the width of PE is best design practice. Note in Fig. 13 optimal throughput is not growing linearly with SA width, this is due to more PEs in SA are idle during LSH clustering phase calculation when SA width grows and the increase of time spent on updating value registers. This can be solved by adjusting mapping strategy for a given PE width, and optimizing datapath for updating value registers.

Throughput. Left part of Fig. 12 shows the throughput of different hardware platforms running attention mecha-

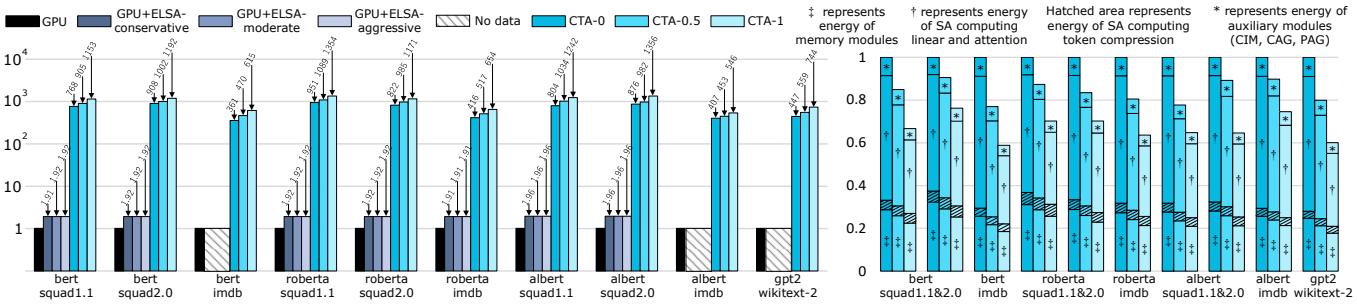

Fig. 14. Left: normalized energy efficiency. Right: energy breakdown of CTA accelerator.

nism. Across different testcases, CTA-0, CTA-0.5 and CTA-1 achieve 27.7×, 33.8× and 44.2× geomean speedups over GPU. The throughput of ELSA+GPU system varies slightly as we change ELSA's approximation strategy from conservative to aggressive. This is largely due to the portion of computation that ELSA accelerates only accounts for about half the computation we measure across these testcases. CTA-0, CTA-0.5 and CTA-1 achieve 18.3×, 22.1× and 28.7× geomean speedups over ELSA-Aggressive+GPU system.

Latency. We analyze latency breakdown of CTA and compare it to ideal accelerator latency in right part of Fig. 12. Across all the testcases, CTA spent on average 59%, 34%, 7% of latency on attention calculations, linear transformations and token compression respectively. The portion of time spent on token compression is small thanks to the efficient LSH-based clustering scheme and dedicated hardware optimization. Also, the latency of CTA-0, CTA-0.5, CTA-1 is on average 41%, 34%, 26% of the ideal accelerator latency, showing that our specialized architecture can fully release the potential benefits from computation reduction in both linear transformations and attention calculations.

End-to-end performance. We evaluate end-to-end speedup by mapping accelerated operations on CTA while other parts of model on GPU. When processing samples with sequence length 512, CTA brings 1.9-2.0× speedup. When extending to process 4× longer sequences, CTA brings 2.9-3.0× speedup. Note that our systolic array-based architecture could be easily extended to accelerate FFN, in which case the end-to-end speedup is further promoted.

D. Area, Energy and Memory Access.

Methodology. We implement CTA with Verilog HDL and compiled to RTL to complete functional verifications. We synthesize the Verilog code with 1GHz clock timing constraint using Synopsys Design Compiler and SMIC 40nm standard cell library to get the area and power of CTA accelerator. For memory modules, we use CACTI [65] to estimate the energy and area, and integrate read/write/static energy into our simulator, which can generate the average memory energy consumption for one attention head in each testcase. We measure GPU power with nvidia-smi to sample the power per 10 seconds when running each testcase. We also evaluate ELSA+GPU system energy consumption using the power provided in ELSA paper.

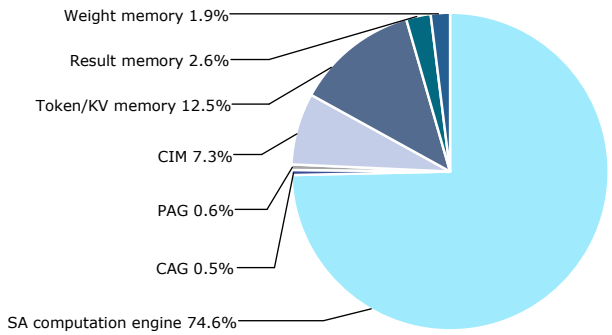

Fig. 15. Area breakdown of CTA accelerator.

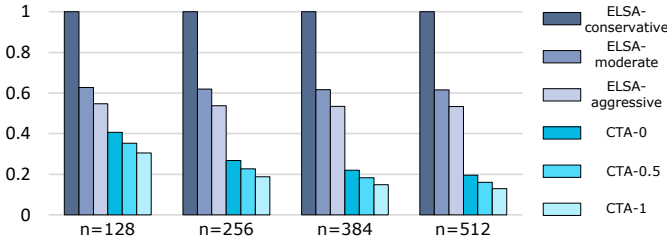

Fig. 16. Normalized memory access under different sequence length.

Area. Fig. 15 shows the area breakdown of CTA. The total area of a single CTA (including memory modules) is 2.150mm^2, where SA computation engine takes up the majority (74.6%) of the area. Note that area overhead for auxiliary modules are low, because the algorithmic operations mapped on auxiliary modules are light-weight with no need to consume too much hardware resources.

Energy. Left part of Fig. 14 visualizes the normalized energy efficiency for running attention mechanism across different hardware platforms. CTA-0, CTA-0.5 and CTA-1 achieves on average 634×, 756× and 950× energy efficiency over GPU, and 399×, 471×, 587× energy efficiency over ELSA+GPU platform. Right part of Fig. 14 shows the energy breakdown of CTA. Memory modules, SA computation engine and auxiliary modules accounts for 29%, 62% and 9% of the total energy consumption respectively. Notice that the portion of energy consumed by memory modules in CTA are smaller than previous architectures based on query-specific pruning, such as ELSA, and LeOPArd [43], [44].

Memory Access. We obtain the number of read/write of CTA running one head of attention from our simulator and compare with the reproduced number of read/write for ELSA. In Fig. 16, we evaluate under four sequence length settings

$n = 128, 256, 384, 512$, the results showed that as the processing sequence length increases, ELSA induces substantially more memory access than CTA. The reasons are two fold: (i) Systolic array provides high memory access efficiency in that data are repeatedly used by communication between PEs. However for ELSA, all relevant keys and values need to be read for each query and the reading is repeated for each query due to query-by-query processing. (ii) CTA benefits from specialized memory access optimization to avoid unnecessary memory read/write (Section V-B).

VII. Conclusion

The attention mechanism has emerged as a new primitive in deep learning, providing great extension to traditional NN. We explore the possibility to accelerate attention mechanism with efficient architecture used to accelerate traditional NN. This is a good practice considering the fact that attention mechanism is usually used in combination with traditional NN in state-of-the-art models. We propose CTA approximation scheme to remove redundant computations while keeping all inherent parallelism in attention mechanism and develop efficient architecture to achieve high performance, energy saving inference. We show the possibility to accelerate attention mechanism based on systolic array, with light-weight auxiliary architecture to support special operations. Future works may follow this route to develop more general architecture and achieve better performance in accelerating attention-based models.

VIII. Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (Grant No. 61834006, 62025404, 62104229, 62104230, 61874124, 62222411), and in part by the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDB44030300, XDB44020300), Zhejiang Lab underGrants 2021PC0AC01. The corresponding authors are Haobo Xu and Ying Wang.

References

[1] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[4] Luowei Zhou, Yingbo Zhou, Jason J Corso, Richard Socher, and Caiming Xiong. End-to-end dense video captioning with masked transformer. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8739–8748, 2018.

[5] Irwan Bello, Barret Zoph, Ashish Vaswani, Jonathon Shlens, and Quoc V Le. Attention augmented convolutional networks. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 3286–3295, 2019.

[6] Marcella Cornia, Matteo Stefanini, Lorenzo Baraldi, and Rita Cucchiara. Meshed-memory transformer for image captioning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10578–10587, 2020.

[7] Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jon Shlens. Stand-alone self-attention in vision models. *Advances in Neural Information Processing Systems*, 32, 2019.

[8] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-attention generative adversarial networks. In *International conference on machine learning*, pages 7354–7363. PMLR, 2019.

[9] Yufei Feng, Fuyu Lv, Weichen Shen, Menghan Wang, Fei Sun, Yu Zhu, and Keping Yang. Deep session interest network for click-through rate prediction. *arXiv preprint arXiv:1905.06482*, 2019.

[10] Wang-Cheng Kang and Julian McAuley. Self-attentive sequential recommendation. In *2018 IEEE international conference on data mining (ICDM)*, pages 197–206. IEEE, 2018.

[11] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1161–1170, 2019.

[12] Chang Zhou, Jinze Bai, Junshuai Song, Xiaofei Liu, Zhengchao Zhao, Xiusi Chen, and Jun Gao. Atrank: An attention-based user behavior modeling framework for recommendation. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[13] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep interest evolution network for click-through rate prediction. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 5941–5948, 2019.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[16] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[17] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[18] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[19] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[20] Mohammad Shoeybi, M Patwary, R Puri, P LeGresley, J Casper, B Megatron-LM Catanzaro, et al. Training multi-billion parameter language models using model parallelism. *arXiv preprint cs.CL/1909.08053*, 2019.

[21] Corby Rosset. Turing-nlg: A 17-billion-parameter language model by microsoft. *Microsoft Blog*, 1(2), 2020.

[22] Zi-Hang Jiang, Weihao Yu, Daquan Zhou, Yunpeng Chen, Jiashi Feng, and Shuicheng Yan. Convbert: Improving bert with span-based dynamic convolution. *Advances in Neural Information Processing Systems*, 33:12837–12848, 2020.

[23] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, pages 10347–10357. PMLR, 2021.

[24] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[25] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[26] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In *International conference on machine learning*, pages 1691–1703. PMLR, 2020.

[27] Liunian Harold Li, Mark Yatskar, Da Yin, Cho-Jui Hsieh, and Kai-Wei Chang. Visualbert: A simple and performant baseline for vision and language. *arXiv preprint arXiv:1908.03557*, 2019.

[28] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014.

[29] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[30] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[31] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[32] Shehzeen Hussain, Mojan Javaheripi, Paarth Neekhara, Ryan Kastner, and Farinaz Koushanfar. Fastwave: Accelerating autoregressive convolutional neural networks on fpga. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[33] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 754–768, 2019.

[34] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.

[35] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278. IEEE, 2016.

[36] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[37] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE international symposium on high performance computer architecture (HPCA)*, pages 541–552. IEEE, 2017.

[38] Amir Yazdanbakhsh, Kiran Seshadri, Berkin Akin, James Laudon, and Ravi Narayanaswami. An evaluation of edge tpu accelerators for convolutional neural networks. *arXiv preprint arXiv:2102.10423*, 2021.

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

[40] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[41] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 382–394, 2017.

[42] 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, et al. Aˆ 3: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341. IEEE, 2020.

[43] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 692–705. IEEE, 2021.

[44] Zheng Li, Soroush Ghodrati, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, and Mingu Kang. Accelerating attention through gradient-based learned runtime pruning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 902–915, 2022.

[45] Ian Tenney, Dipanjan Das, and Ellie Pavlick. Bert rediscovers the classical nlp pipeline. *arXiv preprint arXiv:1905.05950*, 2019.

[46] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[47] Davis W Blalock and John V Guttag. Bolt: Accelerated data mining with fast vector compression. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 727–735, 2017.

[48] Artem Babenko and Victor Lempitsky. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 931–938, 2014.

[49] Julieta Martinez, Holger H Hoos, and James J Little. Stacked quantizers for compositional vector compression. *arXiv preprint arXiv:1411.2173*, 2014.

[50] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[51] Jorge Sánchez and Florent Perronnin. High-dimensional signature compression for large-scale image classification. In *CVPR 2011*, pages 1665–1672. IEEE, 2011.

[52] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2953, 2013.

[53] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[54] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

[55] Edith Cohen, Mayur Datar, Shinji Fujiwara, Aristides Gionis, Piotr Indyk, Rajeev Motwani, Jeffrey D Ullman, and Cheng Yang. Finding interesting associations without support pruning. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):64–78, 2001.

[56] Taher Haveliwala, Aristides Gionis, and Piotr Indyk. Scalable techniques for clustering the web. 2000.

[57] Jeremy Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.

[58] Jeremy Buhler and Martin Tompa. Finding motifs using random projections. In *Proceedings of the fifth annual international conference on Computational biology*, pages 69–76, 2001.

[59] Bogdan Georgescu, Ilan Shimshoni, and Peter Meer. Mean shift based clustering in high dimensions: A texture classification example. In *Computer Vision, IEEE International Conference on*, volume 2, pages 456–456. IEEE Computer Society, 2003.

[60] Microsoft. An engine to auto generate optimized kernels for multi backends. https://github.com/microsoft/antares.

[61] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pages 142–150, 2011.

[62] Stephen Merity. The wikitext long term dependency language modeling dataset. *Salesforce Metamind*, 9, 2016.

[63] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[64] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[65] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.