2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2235-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/HPCA51647.2021.00018

SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning

Hanrui Wang, Zhekai Zhang, Song Han
EECS
Massachusetts Institute of Technology
Cambridge, MA, US
{*hanrui, zhangzk, songhan*}*@mit.edu*

***Abstract*—The attention mechanism is becoming increasingly popular in Natural Language Processing (NLP) applications, showing superior performance than convolutional and recurrent architectures. However, general-purpose platforms such as CPUs and GPUs are inefficient when performing attention inference due to complicated data movement and low arithmetic intensity. Moreover, existing NN accelerators mainly focus on optimizing convolutional or recurrent models, and cannot efficiently support attention. In this paper, we present SpAtten, an efficient *algorithm-architecture co-design* that leverages token sparsity, head sparsity, and quantization opportunities to reduce the attention computation and memory access. Inspired by the high redundancy of human languages, we propose the novel *cascade token pruning* to prune away unimportant tokens in the sentence. We also propose *cascade head pruning* to remove unessential heads. Cascade pruning is fundamentally different from weight pruning since there is no trainable weight in the attention mechanism, and the pruned tokens and heads are selected on the fly. To efficiently support them on hardware, we design a novel *top-k engine* to rank token and head importance scores with high throughput. Furthermore, we propose *progressive quantization* that first fetches MSBs only and performs the computation; if the confidence is low, it fetches LSBs and recomputes the attention outputs, trading computation for memory reduction.**

Extensive experiments on 30 benchmarks show that, on average, SpAtten reduces DRAM access by 10.0× with no accuracy loss, and achieves 1.6×, 3.0×, 162×, 347× speedup, and 1.4×, 3.2×, 1193×, 4059× energy savings over A^3 accelerator, MNNFast accelerator, TITAN Xp GPU, Xeon CPU, respectively.

***Index Terms*—Natural Language Processing; Attention; Domain-Specific Accelerator; Algorithm-Architecture Co-design; Pruning; Quantization**

I. Introduction

Natural Language Processing (NLP) has witnessed rapid progress in recent years driven by the attention mechanism [1]. Attention models such as Transformer [1], BERT [2], and GPT-2 [3] provide significant performance improvements over models based on Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). BERT [2] even outstrips human performance on the challenging sentence classification [4] tasks.

Unfortunately, the high accuracy is at the expense of efficiency. Attention runs extremely slow on general-purpose platforms such as GPUs and CPUs, due to its complex data movement and low arithmetic intensity. For instance, to generate a sentence with only 30 tokens, a GPT-2 model takes a total of 370ms on a TITAN Xp GPU to perform attention inference.

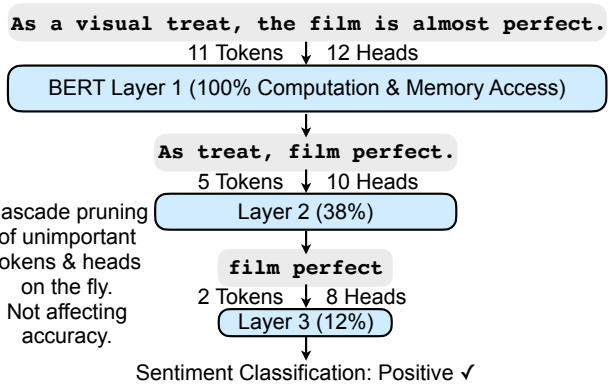

Fig. 1. *Cascade token and head pruning* removes redundant tokens and heads globally across layers. Evaluated with BERT-Base on SST-2 dataset.

That is two orders of magnitude slower than MobileNet-V2, which takes only 6ms to classify an image. On resources-limited Raspberry Pi ARM CPU, attentions cost 43s, making interactive dialog applications impossible. The efficiency barrier prevents attention models from being deployed on mobile devices. Many accelerators have been proposed to accelerate CNN and RNN, but they cannot be easily applied to attention due to the distinct operations.

In this paper, we propose SpAtten[1], an algorithm-architecture co-design to enable efficient attention inference. We propose three algorithmic optimizations: *cascade token pruning*, *cascade head pruning* and *progressive quantization* to reduce computation and memory access. Different from conventional techniques, pruning is applied to the tokens and heads, not weights. Cascade means: once a token/head is pruned, it is removed in all following layers, so one layer only needs to process remaining tokens/heads from previous layers. The deeper the layer, the more tokens/heads are pruned. The three techniques are input-dependent since the pruned computation and bit-width are *adaptive* to input instances. Cascade pruning requires sorting token/head importance scores on the fly. Thus we design hardware architecture with high parallelism top-k engines for token/head selections, specialized memory hierarchy, and fully-pipelined datapath to translate theoretical savings to real speedup and energy reduction.

[1]SpAtten is homophonic with *spartan*, meaning simple and frugal. It is analogous to *token and head pruning*, making sentences shorter and simpler.

978-1-6654-2235-2/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00018

The inputs of attention contain Query (Q), Key (K), and Value (V), each split into multiple heads. Then attention probabilities are computed as the softmax of Q $\times$ K. Multiplying the attention probabilities with V gives the result of one head; concatenating all heads together gives the attention output. The arithmetic intensity of attention in the generation stage is low: only two operations per data (0.5ops/Byte) for the vector-matrix multiplication (Q $\times$ K). Generation takes the largest part of overall latency in GPT-2 models (97% when generating 32 tokens); thus, the overall performance is memory-bounded. For BERT, the overall performance is computation-bounded.

Therefore, we propose *cascade token pruning* as shown in Figure 1 to reduce both DRAM access and computation. Inspired by human languages being highly redundant due to many structural and meaningless tokens such as prepositions, articles, and adverbs, we can safely remove unimportant tokens with little impact on the results. Moreover, attention uses many heads to capture various dependencies, but some of them are also redundant [5]. Therefore, we also propose *cascade head pruning* to determine the importance of heads based on their influence on the outputs, then prune unessential heads. Cascade token and head pruning are fundamentally different from the classic weight pruning and classic head pruning because: (i) There is no trainable weight in attention. (ii) Conventionally, pruned weights and heads are determined at compile-time and consistent for all inputs. In contrast, tokens and heads to be pruned in SpAtten are selected on the fly and vary between different inputs. Token pruning is also different from classic activation pruning because it depends on attention probabilities, not activation magnitude. Specifically, we prune the tokens according to cumulative token importance scores obtained by accumulating attention probabilities (indicators for token influence) across layers. Since long sentences are naturally more redundant, we also adjust the pruning ratios based on sentence length: the longer, the more tokens are pruned away. Moreover, the heads are pruned according to cumulative head importance scores, which are computed by accumulating each head's magnitude across layers. To support cascade token and head pruning, we design and implement a specialized high parallelism top-k engine with $O(n)$ time complexity to get the k most essential tokens or heads. On average, token and head pruning can reduce DRAM access and computation by 3.8$\times$ and 1.1$\times$ on eight GPT-2 models.

To further reduce the DRAM access, we also propose *progressive quantization* for attention inputs. We find an interesting phenomenon that quantization errors are related to attention probability distributions: if a few tokens dominate the distribution, the quantization error is small – only MSB is needed; for a flat distribution, the error is large – both LSB and MSB are needed. We also provide a theoretical proof for this phenomenon in Section III-D. Based on this observation, we quantize more aggressively for attention with dominated attention probabilities and more conservatively for others. Concretely, we first fetch MSBs of attention inputs to compute the attention probabilities. If the max probability is smaller than a threshold, indicating the distribution is flat, we will fetch

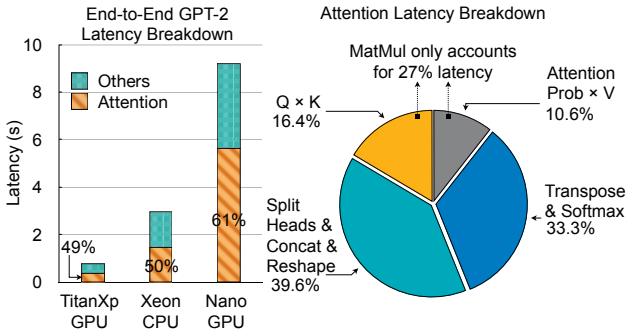

Fig. 2. End-to-End GPT-2 latency breakdown on various platforms, and attention latency breakdown on TITAN Xp GPU. Attention accounts for over 50% of total latency. Data movements account for 73% of attention latency.

LSBs on-chip and recompute attention probabilities. In such a way, we trade computation to less memory access, which is beneficial to memory-bounded models. With progressive quantization, we can save another 5.1$\times$ memory access.

Previous state-of-the-art attention accelerators A^3 [6] and MNNFast [7] also leverage sparsity. However, they have three main limitations. (i) A^3 and MNNFast need to fetch everything from DRAM before calculating what can be pruned. Thus the overhead is already paid, and no DRAM access is reduced. They only optimize computation-bounded discriminate models, and cannot accelerate memory-bounded generative models. SpAtten not only improves computation-bounded discriminative ones (BERT), but also solves the challenge of memory-bounded generative ones (GPT-2). It significantly reduces QKV DRAM access with token pruning (3.8$\times$), head pruning (1.1$\times$), and progressive quantization (5.1$\times$). (ii) Head sparsity is an opportunity to further reduce DRAM access and computation, but A^3 and MNNFast do not support head pruning. (iii) A^3 prunes QKV vectors of a token only locally in one head, and MNNFast only prunes V vector locally. Therefore they only reduce the attention layer's computation but not Feed-Forward Network (FFN) layers. SpAtten prunes the token *globally*: once a token is pruned, the involved computations in *all* following layers are skipped. Therefore, the computations in FFN layers are also reduced in SpAtten.

SpAtten has a wide support range thanks to the generalization ability of attention-based NLP models. For instance, BERT can be used for arbitrary discriminative tasks, such as sentence sentiment classification and sentence similarity regression, for which the backbone of BERT is the same and only the last layer needs to be changed. Likewise, GPT-2 can handle all generative tasks, including language modeling, document summarization, etc.

In summary, SpAtten performs algorithm-architecture co-design for sparse and quantized attention computing while preserving the accuracy. It makes four contributions:

- **Cascade Token Pruning** removes unimportant tokens according to the cumulative token importance scores, reducing DRAM access and computation by up to 3.8$\times$.
- **Cascade Head Pruning** removes unimportant heads and save DRAM access and computation by another 1.1$\times$.

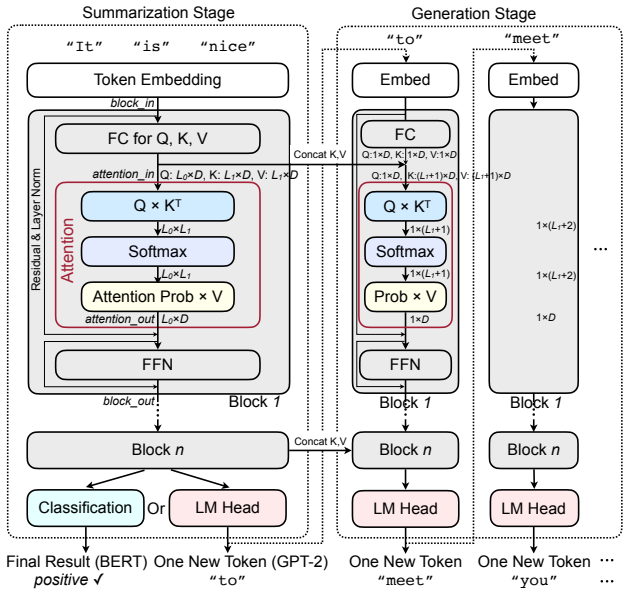

Fig. 3. NLP model architecture with attention. BERT only contains the summarization stage. GPT-2 contains summarization and generation stages.

- **Progressive Quantization** trades a little more computation for less memory access. We change the bitwidths of different attention heads and layers based on attention probability distribution, reducing DRAM access by 5.1×.
- **Specialized High Parallelism top-k Engine** with $O(n)$ time complexity to efficiently support on-the-fly token and head selections.

We extensively evaluate SpAtten on 30 benchmarks including GLUE set [4], SQuAD [8], Wikitext-2 [9], Wikitext-103 [9], Pen Tree Bank [10] and Google One-Billion Word [11] with BERT and GPT-2 models. SpAtten reduces DRAM access by 10.0× with *no accuracy loss*, and achieves 1.6×, 3.0×, 162×, 347×, 1095×, 5071× speedup, and 1.4×, 3.2×, 1193×, 4059×, 406×, 1910× energy savings over A^3 accelerator, MNNFast accelerator, TITAN Xp GPU, Xeon CPU, Nano GPU, Raspberry Pi ARM CPU, respectively.

II. Background and Motivation

A. Background

Attention-Based NLP Models. NLP tasks can be categorized into two types: discriminative and generative. For discriminative ones, the models need to summarize the input information and make predictions. Discriminative tasks include token-level classification [12], sentence-level classification and regression [13] etc. Meanwhile, models for generative tasks need first to summarize the input information and then generate new tokens. Exemplary generation tasks include Language Modeling (LM) [3] and machine translation [1].

BERT for discriminative and GPT-2 for generative tasks are the most widely-used models as illustrated in Figure 3. BERT only contains the summarization stage, while GPT-2 contains summarization and generation stages. In summarization (Figure 3 left), the input tokens are first embedded into vectors and processed by blocks. Inside each block, $block\_in$ are first multiplied with three matrices to get Query (Q), Key (K) and, Value (V). Then QKV are processed by attention to get the intermediate features $attention\_out$. A residual layer adds the $attention\_out$ with $block\_in$ and conducts layer normalization. There will be an additional FC on $attention\_out$ if there is more than one head. Furthermore, a Feed-Forward Network (FFN) layer containing two Fully-Connected (FC) layers is applied. Finally, another residual operation is conducted and outputs $block\_out$. The same block is repeated multiple times such as 12 times for BERT-Base. The last block is followed by one classification layer in BERT to get the *final result*. In contrast, GPT-2 applies an LM head to generate one new token and enters the generation stage.

Algorithm 1: Attention

Input: $Q_{in} \in \mathbb{R}^{L_0 \times D_{in}}$, $K_{in} \in \mathbb{R}^{L_1 \times D_{in}}$, $V_{in} \in \mathbb{R}^{L_1 \times D_{in}}$;
Number of Heads: h ;
Split Q_{in}, K_{in}, V_{in} to h chunks:
$Q \in \mathbb{R}^{h \times L_0 \times D}, K \in \mathbb{R}^{h \times L_1 \times D}, V \in \mathbb{R}^{h \times L_1 \times D}, D = \frac{D_{in}}{h}$;
for $head_{id} = 0$ **to** h **do**
  $attention\_score \in \mathbb{R}^{L_0 \times L_1}$;
  $attention\_score = Q[head_{id}] \cdot K[head_{id}]^T$;
  $attention\_score = attention\_score/\text{sqrt}(D)$;
  $attention\_prob \in \mathbb{R}^{L_0 \times L_1}$;
  for $row_{id} = 0$ **to** L_0 **do**
    $attention\_prob[row_{id}] =$ Softmax$(attention_score[row_{id}])$
  end
  $E[head_{id}] = attention\_prob \cdot V[head_{id}]$;
end
Concatenate heads of $E \in \mathbb{R}^{h \times L_0 \times D}$ as output;
Output: $attention\_out \in \mathbb{R}^{L_0 \times D_{in}}$;

Generation stage (Figure 3 right) has two main differences from summarization: 1) Each iteration only processes *one single token* instead of the whole sentence. 2) Ks and Vs from the summarization stage are concatenated with current K and V, and sent to attention *in batch*, while the query is still *one single vector*. After the last block, another new token will be generated. Generation stage ends when `end_of_sentence` token is generated, or the sentence length reaches a pre-defined limit. One generation iteration's runtime is similar to the whole summarization stage on GPU, because the summarization is processed in batch.

Attention Mechanism. The attention mechanism [1] is shown in Algorithm 1. In the summarization stage, Q, K, and V are matrices with the same dimension, while in the generation stage, Q is one single vector, and K, V are matrices.

Attention has multiple *heads*, each processing a chunk of Q, K, and V. Different heads capture various dependencies between tokens, some for long-term, some for short-term. Inside each head, $\text{Q} \times \text{K}^T/\text{sqrt}(D)$ gives attention scores, which indicate whether two tokens are related. For instance in Figure 5, the score for 'more' attending to 'than' is large. After that, a row-wise softmax computes the attention probabilities. The exponential of softmax further enlarges the score for highly-related token pairs. The head's feature is then

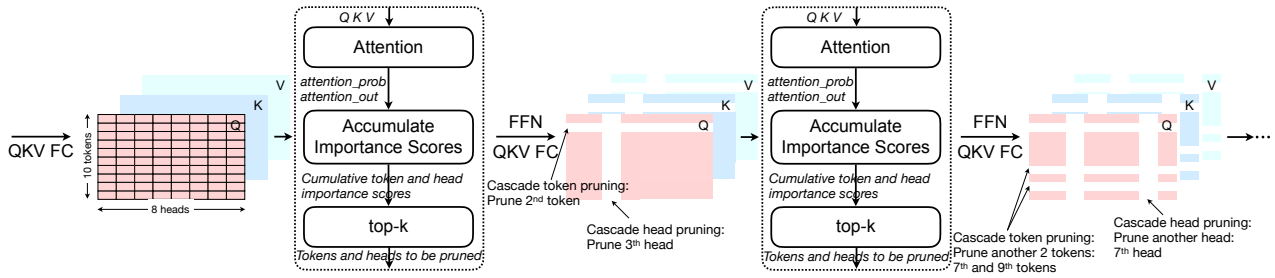

Fig. 4. Cascade token pruning removes redundant tokens and corresponding entire Q K V vectors according to the cumulative token importance scores computed from $attention\_prob$. Cascade head pruning removes unimportant heads and corresponding chunks in all Q K V vectors according to the cumulative head important scores computed from $attention\_out$. Once a token/head is pruned, it will never appear in any following layers, thus named cascade pruning. More tokens and heads are pruned away as the layer goes deeper.

Algorithm 2: Token and Head Pruning (one layer)

Input: Number of heads: h;
Token pruning ratio: p_t; Head pruning ratio: p_h;
$attention\_prob \in \mathbb{R}^{h \times L_0 \times L_1}$, $attention\_out \in \mathbb{R}^{L_0 \times D_{in}}$;
Previous cumulative token importance score: $s_t \in L_1$;
Previous cumulative head importance score: $s_h \in h$;
Reshape $attention\_out$ by head, get $E \in \mathbb{R}^{h \times L_0 \times D}$;

```
    /* accumulate the token importance                    */
for token_id = 0 to L_1 do
    for l_0 = 0 to L_0 do
        for head_id = 0 to h do
            s_t[token_id] +=
              attention_prob[head_id][l_0][token_id]
        end
    end
end
/* accumulate the head importance                         */
for head_id = 0 to h do
    for l_0 = 0 to L_0 do
        for d = 0 to D do
            s_h[head_id] += abs(E[head_id][l_0][d])
        end
    end
end
```

$remained\_token\_id = \text{top-k}(s_t, L_1 \times (1 - p_t))$
$remained\_head\_id = \text{top-k}(s_h, h \times (1 - p_h))$
Output: $remained\_token\_id, remained\_head\_id, s_t, s_h$

computed with $attention_prob \times$V. This step lets each token fetch information from their cared tokens. Finally, multiple heads are concatenated together as the attention output. In BERT-Base and GPT-2-Small, there are 12 blocks, and each attention layer has 12 heads. There are 24 blocks and 16 heads for each attention layer in BERT-Large and GPT-2-Medium.

B. Motivation

We profile the end-to-end latency of a GPT-2 model on multiple hardware platforms in Figure 2. Attention typically accounts for over 50% latency, even though it only has around 10% of overall FLOPs. In Figure 2 right, around 73% of the time is spent on data movements such as splitting heads, K and V concatenations, reshape, and transpose. GPUs and CPUs are well-optimized for matrix multiplications but are poor on the complex memory operations, thus slowing down attention. Therefore, it is necessary to build an accelerator to solve the attention bottleneck as a co-processor. The FC layers of NLP models are processed by GPUs, CPUs, or tensor algebra accelerators as they are highly-optimized for FC, and our SpAtten co-processor handles all attention layers.

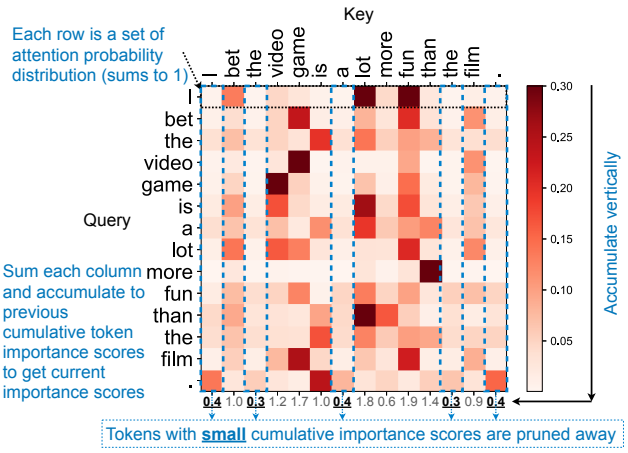

Fig. 5. Attention probabilities for BERT are summed over each column to get importance scores. Tokens with small importance scores are pruned.

III. Algorithmic Optimizations

A. Cascade Token Pruning

Plenty of unessential tokens exist in human languages, which can be pruned away to boost efficiency. Therefore, we propose cascade token pruning to assess token importance based on attention probabilities and remove trivial ones. Cascade means that once a token is pruned, it is removed in all the following layers, so one layer only needs to process *remaining* tokens from previous layers.

In cascade token pruning, tokens to be pruned are determined by an array of *cumulative token importance scores*, one for each token (Figure 4 and Algorithm 2). The scores are obtained by accumulating attention probabilities across multiple rounds of attention. The probability indicates whether a token is important to the sentence, because if the probability is large, then the outputs are more influenced by the corresponding token. Specifically, $attention_out = attention_prob \times$V while V is computed from input features $block\_in$ (see Figure 3). The $block\_in$ of the first block are directly from input tokens. For latter blocks, V vectors are still largely determined by input tokens because of residual connections. Therefore, the probability is an indicator of the token importance, and the accumulations across several heads and layers make the

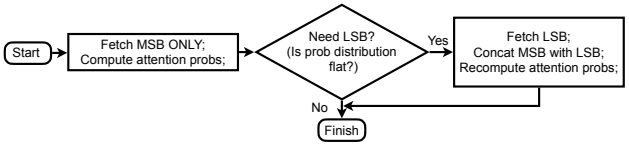

Fig. 6. Progressive Quantization. MSBs are fetched first. Only if the resulting probability distribution is flat, LSBs are fetched, and attention probabilities are recomputed – this trades computation for memory saving.

importance more reliable. For example, in Figure 5, many tokens attend to the word 'fun', implying its high usefulness. In each head, the scores are accumulated by L_0 (number of query vectors) times in the summarization stage and one time in the generation stage. In BERT, we accumulate importance scores in former heads and layers and apply token pruning to latter ones. In GPT-2, we further accumulate importance scores *across generation iterations* because intuitively, the unimportant tokens for one token generation should also be unimportant to others. With the cumulative importance scores, we can remove a pre-defined pruning ratio of tokens. Once a token is pruned, the QKV of it will *never* be used in all the following attention heads and layers; in every layer/head, several new tokens can be selected and pruned away, thus being *global and cascade*. Token pruning can reduce the computation and memory access of both attention, and also FC layers outside attention. On eight GPT-2 benchmarks, token pruning can achieve 3.8× reduction of DRAM access.

B. Cascade Head Pruning

Each QKV vector has multiple chunks corresponding to multiple heads, which are used to capture various token dependency relationships. However, some of the heads are redundant [5] and have little influence on outputs. Token pruning reduces the sentence length. Head pruning reduces the feature length. Hence, redundancies in both dimensions are removed. The head importance is calculated by accumulating the absolute value of $attention\_out$ elements of each head across layers to get *cumulative head importance scores* (Figure 4 and Algorithm 2). The magnitude of the head's outputs indicates its importance because there exists one FC layer processing the concatenation of all heads. If the magnitude of one head is large, the outputs of the FC layer and the whole block $block\_out$ will be more heavily influenced by the head. Similar to token pruning, head pruning is also cascaded: once removed, a head will not appear in the following layers.

C. Local Value Pruning

SpAtten also supports local Value (V) pruning, which is performed after Softmax. the V vectors to be pruned are decided *solely with the current head's attention probabilities.* A pre-defined ratio of V vectors with the smallest attention probabilities are pruned and will not be fetched for the $attention\_prob$×V computation. Compared to cascade token pruning which removes Q, K, and V of pruned tokens for current and all following attention heads and layers, local V pruning only removes V vectors of the current head.

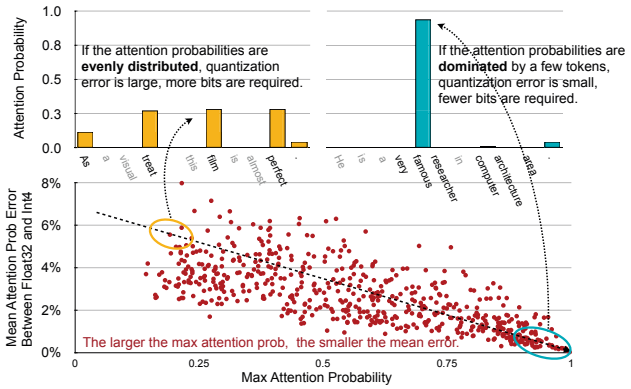

Fig. 7. When the attention probability is evenly distributed (left), the quantization error is large; both MSB and LSB are required. When dominant probability exists (right), the quantization error is small; only MSB is required. Grey tokens are pruned by token pruning, thus no need to compute their attention probabilities.

D. Progressive Quantization

Softmax layers are abundant in attention, which allows us to apply more aggressive quantization than CNN/RNN models because Softmax can reduce quantization error. Softmax for attention probabilities is: $p_i = e^{s_i} / \sum_{j=0}^{L_1-1} e^{s_j}$, where L_1 is the number of K vectors, p is attention probability, and s is attention score. Quantization on Q and K can be considered as adding a small error Δs to the attention score s. We examine the influence of Δs on output attention probabilities by computing the softmax derivative:

$$\begin{aligned} \text{If } i = j: \frac{\partial p_i}{\partial s_j} &= \frac{\partial \frac{e^{s_i}}{\sum_{i=0}^{L_1-1} e^{s_i}}}{\partial s_i} = p_i \cdot (1 - p_i) \\ \text{If } i \neq j: \frac{\partial p_i}{\partial s_j} &= \frac{\partial \frac{e^{s_i}}{\sum_{i=0}^{L_1-1} e^{s_i}}}{\partial s_j} = -p_i \cdot p_j \end{aligned} \tag{1}$$

Without loss of generality, we assume s_0 changes by $\Delta s_0 > 0$, and sum the absolute errors of all output with Equation 1:

$$\begin{aligned} error &= |\Delta s_0 \cdot p_0 \cdot (1 - p_0)| + \sum_{i=1}^{L_1-1} |-\Delta s_0 \cdot p_0 \cdot p_i| \\ &= \Delta s_0 \cdot (2 \cdot p_0 \cdot (1 - p_0)) < \Delta s_0 \end{aligned} \tag{2}$$

Since $0 \leq p_0 \leq 1$, $2p_0(1 - p_0)$ is always smaller than 0.5, so the total quantization error is reduced after Softmax.

On top of static quantization, we propose *progressive quantization* (Figure 6) to progressively increase the input bitwidth if aggressive quantization hurts accuracy. An interesting phenomenon in Figure 7 shows that if the attention probability distribution is dominated by a few tokens, then the 4-bit quantization error is smaller; if flat, the error is larger. Intuitively, dominant tokens are semantically important; thus cannot be easily influenced by quantization. Theoretically, errors are proportional to $p(1 - p)$ (Equation 2). When probability dominators exist, p is closer to zero or one; thus errors are smaller. Otherwise, errors are larger. Therefore, for robust layers with probability dominators, the bitwidth can be

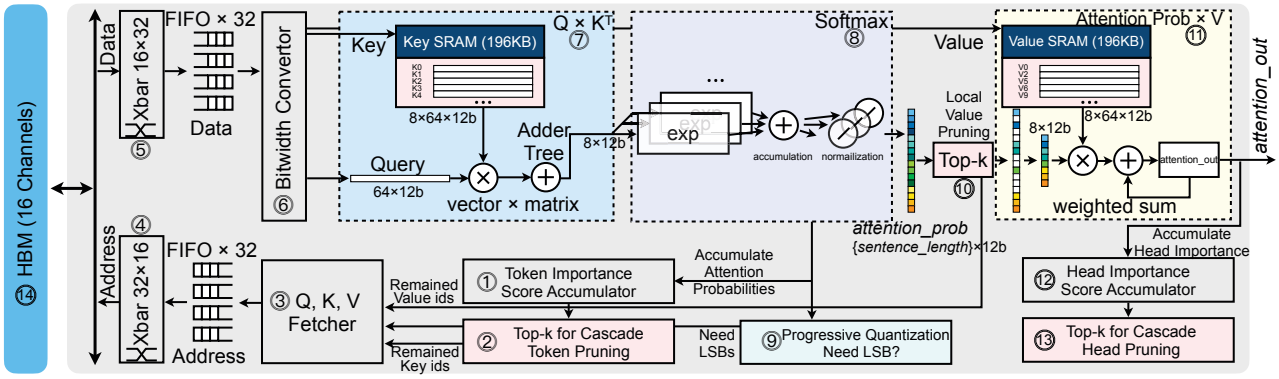

Fig. 8. SpAtten Architecture Overview. Modules on the critical path (6,7,8,10,11) are fully pipelined to maximize the throughput.

small; while other sensitive layers should have more bits. In Figure 6, we firstly apply an aggressive bitwidth (only fetch MSBs) for inputs and compute the attention probabilities. If the max of computed probability is smaller than a threshold, indicating the distribution is flat, we will fetch LSBs for inputs and recompute the attention probabilities for once. Otherwise, LSBs are not needed. Intuitively, progressive quantization finds which input sample is *more difficult* and applies a higher bitwidth instead of using a high bitwidth universally, thus reducing DRAM access under the same accuracy.

For fast interactions between SpAtten and hardware for FC parts, we conduct *linear symmetric quantization*, which is much faster than K-Means quantization. We have five different MSB+LSB settings: 4+4, 6+4, 8+4, 10+4, and 12+4. The settings can be different across tasks but are the same within one task. Different inputs of one task determine *whether to fetch LSB on the fly*. We store MSBs continuously and LSBs continuously in DRAM, so that they can be fetched separately. Progressive quantization trades more computation for less memory access and can effectively accelerate memory-bounded generative models such as GPT-2. It also improves energy efficiency since DRAM access takes around 70% of SpAtten power, much expensive than computation. On average, only 5.9% input samples require LSB. For BERT, we only apply static quantization because BERT models are computation-bounded, and fetching LSB for recomputation will degrade BERT's performance.

IV. Hardware Architecture

A. Overview

An overview of SpAtten is shown in Figure 8. To support token/head pruning, a novel top-k engine (Figure 9) is designed to rank the token/head importance. Pruning reduces computation and memory traffic but incurs random access. Thus, a crossbar is applied to process the addresses, keep each memory channel busy, and increase the bandwidth utilization rate. To support progressive quantization, we implement an on-chip bitwidth converter to handle the splits of fetched bits and concatenations of MSBs and LSBs.

SpAtten processes attention head by head and query by query, thus well balancing the pruning granularity and parallelism. One query of a head is fed to the pipeline at a time, enabling token pruning in both head-wise and layer-wise granularity. Inner-head parallelism can keep all on-chip computation resources busy, so no need for inter-head parallelism. In the summarization stage, K and V that survive cascade token pruning are fetched to the on-chip SRAM and will be *reused* across multiple queries. In the generation stage, the Q is a single vector, so there is no reuse of K and V, and no need to store them in the on-chip SRAM.

For each fetched Q, the top-k engine (Figure 9) first ranks the token importance scores and get k most important Ks. A data fetcher then computes the Ks' addresses and feeds them to a 32×16 crossbar for 16 HBM channels. It then gets data back through a reverse 16×32 crossbar to preserve the correct order. Q and K are processed by a matrix-vector multiplication module (Figure 11) to get the attention scores. A Softmax module (Figure 12 left) then processes the attention scores to get attention probabilities, and sends them to the progressive quantization module (Figure 12 right) to decide whether LSBs are required. The probabilities are also sent to the token importance score accumulator to perform accumulations. After that, the local Value pruning top-k engine gets the probabilities, computes k most locally important Vs, and sends their indices to the data fetcher. Finally, the remaining probabilities are multiplied with fetched V, getting attention outputs. After computing one head, the head importance score will be accumulated. After finishing all heads in a layer, a top-k module prunes unimportant heads, which will not be computed in any following layers.

Our dataflow guarantees to avoid fetching pruned tokens and value vectors, thus bringing DRAM access reductions. The critical path (module 6,7,8,10,11) is fully pipelined. The accumulation of token/head importance scores and token/head top-k are performed *in parallel* with the critical path. For module 3 branches, when multiple sources of Q/K/V requests come simultaneously, the fetcher processes requests one by one and sends addresses to FIFOs. The branches after module 8/11 send $attention\_prob/attention\_out$ to accumulators and the next corresponding computation module simultaneously. For the module 9 branch, if LSB is required, it discards $attention\_prob$ and initiates recomputation; then modules

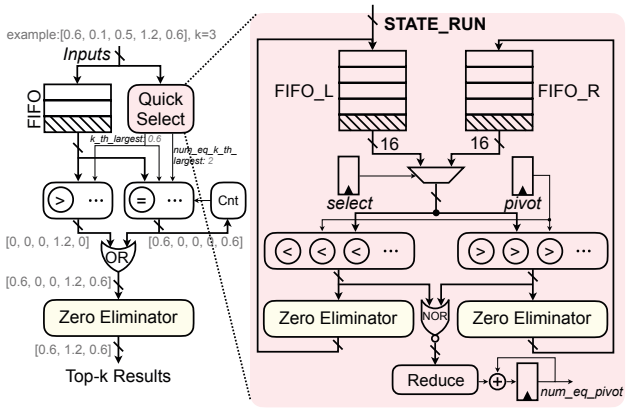

Fig. 9. High Parallelism top-k Engine.

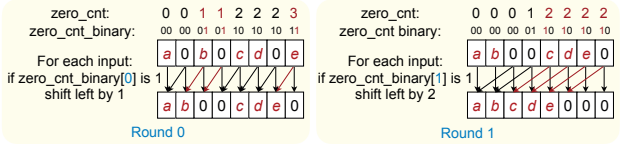

Fig. 10. Zero Eliminator. A zero eliminator of n elements has $\log n$ stages.

10 and 11 will be idle, waiting for recomputed attention probabilities. For module 10 branch, fetching un-pruned V from DRAM is part of the coarse-grained pipeline.

The on-chip memory system has two main SRAMs for Key and Value, 196KB each in module 7 and module 11. They store K and V from QKV Fetcher. Since we process queries one by one, the Q vector is stored in registers. We also have 32 64-depth×8B address FIFOs after QKV fetcher and 32 64-depth×16B data FIFOs before bitwidth converter.

B. Top-k Engine

To support cascade token/head pruning and local value pruning, we need to find the top k elements of an array. A naïve solution is to use a sorter to sort the original array and directly output the first k elements. However, it needs $O(n \cdot \log n)$ time complexity and $O(n \cdot \log^2 n)$ space sorting network, and also completely randomizes data fetching. Instead, we design a novel top-k engine (Figure 9) with a quick-select module to find the k^{th} largest element as a threshold to filter the input array. It has much lower time complexity ($O(n)$ on average) and keeps the original order of inputs. The engine leverages a randomly chosen pivot to partition the input array into two parts: elements smaller or larger than the pivot. It has two FIFOs, FIFO_L and FIFO_R, to store the partitioned arrays. An array to be partitioned will be fed into two comparator arrays and compared with the pivot. The left/right comparator array only preserves elements smaller/larger than the pivot. Others will be set to zeros and eliminated by a zero eliminator. Quick-select runs iteratively until the k^{th} largest element is found. The control logic is in Algorithm 3. Afterwards, the k^{th} largest element is used to filter the input array, which is buffered in another FIFO (Figure 9 left) before the quick-select process. The filtered outputs will be processed by another zero eliminator to finally get the top-k elements. It can be easily scaled up with more comparators. In SpAtten, we apply 16 comparators in each array to make it not the whole pipeline's bottleneck. Since the frequency of head pruning is much lower than token pruning and local Value pruning, we reuse the token pruning top-k engine for head pruning. Therefore, we have two top-k engines in SpAtten architecture.

Algorithm 3: top-k Engine

Input: Top-k k, Data vector $inputs$;
FIFO_L, FIFO_R depth: 64; FIFO = [FIFO_L, FIFO_R];
Initialize FIFO_L with $inputs$, FIFO_R = ϕ;
$target = k$, $num\_eq\_pivot = 0$;
START:
if size(FIFO_R) + $num\_eq\_pivot \leq target$ **then**
  /* the selected pivot is too large */
  $target = target$ − size(FIFO_R) − num_eq_pivot;
  FIFO_R = ϕ; $select = 0$; $pivot$ = FIFO_L[rand];
  Goto **STATE_RUN**;
else if size(FIFO_R) > $target$ **then**
  /* the selected pivot is too small */
  FIFO_L = ϕ; $select = 1$; $pivot$ = FIFO_R[rand];
  Goto **STATE_RUN**;
else
  /* size(FIFO_R) ≤ $target$
    size(FIFO_R) + $num\_eq\_pivot > target$ */
  $k\_th\_largest = pivot$;
  $num\_eq\_k\_th\_largest$ = $target$-size(FIFO_R);
  Output: [$k\_th\_largest, num\_eq\_k\_th\_largest$];
end
STATE_RUN: (Figure 9 Right)
/* items smaller than pivot → FIFO_L */
/* items larger than pivot → FIFO_R */
$num\_eq\_pivot = 0$;
for $i = 0$ **to** size(FIFO[$select$]) − 1 **do**
  Pop FIFO[$select$] as $item$;
  if $item < pivot$ **then**
    Push $item$ to FIFO_L;
  else if $item > pivot$ **then**
    Push $item$ to FIFO_R;
  else
    /* $item == pivot$ */
    $num\_eq\_pivot = num\_eq\_pivot + 1$;
  end
end
Goto **START**;

C. Zero Eliminator

We design an innovative logic for the zero eliminator, which first uses a prefix sum module to calculate the number of zeros before each element $zero\_cnt$. These $zero\_cnt$ will guide a logN layer shifter to shift the input array by 1, 2, 4, ... positions. In the n^{th} layer, whether to shift an element is determined by the n^{th} bit of its corresponding $zero\_cnt$. An example is illustrated in Figure 10.

The top-k engine equipped with zero-eliminators has much higher parallelism than a direct implementation of quick select. Without high parallelism, the performance will be bottlenecked by finding top-k. In Figure 20, we show that SpAtten with a parallelized top-k engine can achieve 3× speedup over a baseline top-k engine with parallelism=1. We also compare a regular full sorting unit (a Batcher's Odd-Even Sorter [14] to perform merge-sort) to the worst case of the top-k engine (selecting the median) with an input length of 1024. Experimental

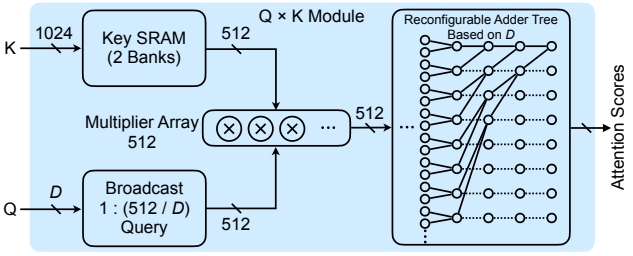

Fig. 11. Query-Key multiplication module with a reconfigurable adder tree. The adder tree can output a various number of partial sums according to the query vector's dimension.

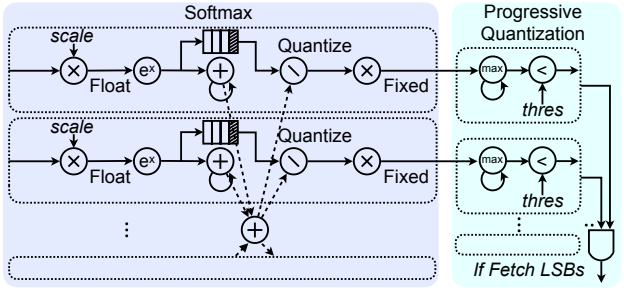

Fig. 12. Softmax and progressive quantization determination modules.

TABLE I
ARCHITECTURAL SETUPS OF SPATTEN.

Q-K-V Fetcher	A 32×16 address crossbar and a 16×32 data crossbar; each port has a 64-depth FIFO.
Q × K	196KB Key SRAM; 512×12-bit multipliers; Adder tree outputs at most 8 items/cycle.
Softmax	FIFO depth for softmax: 128; Parallelism: 8.
Attention_Prob × V	196KB Value SRAM; 512×12-bit multipliers.
HBM	HBM2, 16×128-bit HBM channels @ 2GHz; each channel has 2×64-bit pseudo-channels and provides 32GB/s bandwidth.

results show that we can achieve 1.4× higher throughput with 3.5× smaller power consumption over the full sorting unit.

D. Data Fetcher and Bitwidth Converter

The Q-K-V data fetcher is designed to send multiple random read requests per cycle to all 16 HBM channels, where the QKV are interleaved in different channels. We use a 32-to-16 crossbar to route these read requests to the correct channels. The master side is larger than the slave side. There is no memory access conflict because the crossbar generates at most one memory request for each channel at a time.

In order to support progressive quantization but avoid complex logic overheads, we enforce on-chip SRAMs and multipliers to have a fixed bitwidth. We use a bitwidth converter to convert the data loaded from DRAM (4,8,12 bits) uniformly into on-chip bitwidth (12 bits). The converter consists of MUXes to select correct bits from the input and a shifter to allow reading data from an unaligned address.

E. Query-Key Multiplication Module

The query-key multiplication module (Figure 11) is designed to calculate the matrix-vector multiplication between K and Q. In each cycle, a row of the key matrix K_i is loaded from the K SRAM, multiplied by Q with a multiplier array, and fed to an adder tree. Adder tree then computes attention scores by reducing all multiplied results $s_i = \sum_j \mathrm{K}_{ij} \times \mathrm{Q}_j$. We use 512 multipliers in this module to fully utilize the DRAM bandwidth. To support queries and keys with dimension D lower than 512, we get $512/D$ attention scores in each cycle. The corresponding multiple K_is are packed into the same line in the Key SRAM, and the query is broadcast for $512/D$ times so that all K_is can have access to the same Q. The adder tree is also configurable to output the results of the last several layers, making it function like $512/D$ separate D-way adder trees capable of producing multiple results s_i.

F. Softmax and Progressive Quantization

The fixed-point attention scores s from query-key multiplication are first dequantized using a scaling factor. The attention score normalization factor sqrt(D) is also included in the scaling factor, so that we can perform attention score dequantization and normalization simultaneously. After that, a pipeline of floating-point exponential, accumulation, and division operations are applied to calculate the Softmax results $e^{s_i} / \sum_j e^{s_j}$. The results are finally quantized again so that the operations after Softmax can be performed in fixed-point.

The Softmax results are then fed to the progressive quantization determination module to examine whether LSBs are required. Specifically, we compare the max attention probability with a pre-defined threshold. If smaller than the threshold, the Q-K-V data fetcher will be informed to fetch the LSBs.

G. Attention Prob-Value Multiplication

The attention prob-value multiplication unit takes the attention probabilities as inputs, multiplies them with the Vs and then accumulates to get attention outputs $A_j = \sum_i attention\_prob_i \times \mathrm{V}_{ij}$. It contains another broadcast-multiply-reduce pipeline, similar to the one in the query-key multiplication module, to support the processing of multiple attention probabilities at the same time. The module has 512 multipliers. The attention probabilities are broadcast for D times, and the adder tree is configured to function like D $512/D$-way adder trees. The $attention\_out$ are in 12 bits.

V. EVALUATION

A. Evaluation Methodology

We implement SpAtten with SpinalHDL and compiled to RTL, and simulate each application using Verilator to get the cycle numbers. For HBM modeling, we use Ramulator [15] with HBM2 settings. We synthesize two versions of SpAtten: SpAtten and $\text{SpAtten}_{1/8}$. $\text{SpAtten}_{1/8}$ is only used for fair comparisons with MNNFast and A^3. The parameters for SpAtten are listed in Table I. The scale of $\text{SpAtten}_{1/8}$ is 1/8 of SpAtten and contains 128 multipliers. We synthesize SpAtten using Cadence Genus under TSMC 40nm library to estimate the area and power consumption of the logic, including all fixed-point adders and multipliers. We get the number of floating-point operations in Softmax from the simulator. The

TABLE II
POWER BREAKDOWN OF SPATTEN.

	Computation Logic	SRAM	DRAM	Overall
Power	1.36W	1.24W	5.71W	8.30W

exponential function is approximated with Taylor expansion to the 5^{th} order [16] and performed with floating multiplication accumulation units (FMA). The power and area of FMA are obtained from [17]. We perform the division and estimate power and area with the floating-point unit (FPU) from [17]. The FMAs and FPUs are in 45nm technology, and we use them as an upper bound estimation of 40nm units. We also obtain the width, size, and the number of read/write of each SRAM and FIFO from the simulator and use CACTI [18] to estimate the energy and area of SRAMs and FIFOs. For HBM, we simulate the number of row activation, read/write with Ramulator, and use the energy numbers from [19] to calculate overall energy.

We extensively select various hardware platforms as the evaluation baselines, including server GPU (NVIDIA TITAN Xp GPU), mobile GPU (NVIDIA Jetson Nano), server CPU (Intel Xeon E5-2640 v4 @ 2.40GHz), mobile CPU (4-core ARM A53 CPU on a Raspberry Pi-4), and state-of-the-art accelerators A^3 [6] and MNNFast [7]. For GPUs and CPUs, we run attention with PyTorch and use cuDNN on GPU and MKL on CPU, which are well-optimized libraries. `torch.cuda.Event` on GPU, and `time.time` on CPU are used to measure the latencies. We measure the power with `nvidia-smi` and `pcm-power` for TITAN Xp GPU and Xeon CPU, respectively. For Nano GPU and Raspberry Pi ARM CPU, we use a power meter to get power. For latency measurements, we repeat 1000 times, remove the largest 15% and smallest 15%, and average the remaining. For power measurements, we first measure the system's idle power, and then repeatedly run workloads and get the total power. The dynamic power is total power minus idle power.

We evaluate SpAtten on attention layers of two discriminative models: BERT-Base, BERT-Large, and two generative models: GPT-2-Small and GPT-2-Medium. Tasks for BERT are nine from GLUE [4], V1.1 and V2.0 of SQuAD [8]; For GPT-2, we use language modeling task on four datasets: Wikitext-2 [9], Wikitext-103 [9], Pen Tree Bank [10] and One-Billion Word [11]. In total, we have 30 benchmarks. For all tasks, finetuning is performed for 2 hours on average on GPU after token pruning to recover accuracy. For each task, we try multiple sets of token/head pruning ratios and quantization bitwidths to not lose accuracy, except 2% for BERT-large on SQuAD tasks. Given the same overall pruning ratio, ratios among layers/heads do not have a significant influence. We typically keep the 15% front layers un-pruned, then compute the average ratio of the rest layers r_{avg}. We set a start ratio r_{start} and an end ratio r_{end}, $r_{start} + r_{end} = 2 \times r_{avg}$ and interpolate the ratios of the rest layers. For head pruning, we keep 30% front layers un-pruned and apply a similar method as token pruning. For progressive quantization, the typical

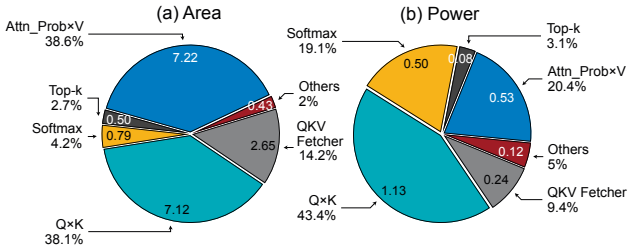

Fig. 13. On-chip (a) Area and (b) Power Breakdowns of SpAtten.

max attention probability threshold is 0.1, and the common MSB+LSB combinations are 6+4 and 8+4.

To measure BERT latency, we set input sentence length as the average length of the each task's dev set. For GPT-2 models, we set the initial length of the input sentence as 992 and measure the latency of generating 32 tokens. The energy efficiency of the models is assessed by energy consumption, which is power×latency.

B. Experimental Results

Throughput, Power, and Area. SpAtten prunes tokens and value vectors with cascade token pruning and local value pruning by 1.9× (all models average), and 3.8× (GPT-2 models average). Cascade head pruning has 1.1× reduction on average. Note that the pruning ratio can be larger when the input sentence of a task is longer because they contain more redundancy. GPT-2 models have longer inputs than BERT, so their pruning ratios can be larger. SpAtten reduces the computation by 2.1× and DRAM access by 10.0× on average. It achieves 1.61TFLOPS on 22 computation-bounded BERT models and 0.43TFLOPS on 8 memory-bounded GPT-2 models. SpAtten consumes 8.30W power as in Table II breakdown and is 18.71mm^2 in area. Figure 13 shows the area and on-chip power breakdown. The Q×K and Attention_Prob ×V modules consume largest portions of energy and area since they are two most computational intensive modules. The latter consumes less energy thanks to local V pruning. top-k engines are relatively efficient, only taking 1.0% of overall power and 2.7% of area so will not cause severe congestion issues.

Comparisons with CPUs and GPUs. Figure 14 shows the speedup and energy efficiency comparisons of SpAtten with baselines on attention layers of the benchmarks. On average, SpAtten achieves 162×, 347×, 1095×, and 5071× speedup, and 1193×, 4059×, 406×, and 1910× energy saving over TITAN Xp GPU, Xeon CPU, Nano GPU, and Raspberry Pi ARM CPU. SpAtten obtains high speedup because it has a highly parallelized and pipelined datapath. Meanwhile, cascade pruning and progressive quantization further reduce computation and DRAM access. Energy savings mainly come from DRAM fetch reduction. The specialized datapath also reduces intermediate SRAM fetch. Since FFN layer computations are reduced by token pruning, CPUs and GPUs can also be accelerated. We implement token pruning on CPUs/GPUs. We use `topk` and `gather` operations to select un-pruned tokens and QKV matrices to reduce matrix sizes, thus reducing computation, latency, and memory footprint. 3× pruning ratio

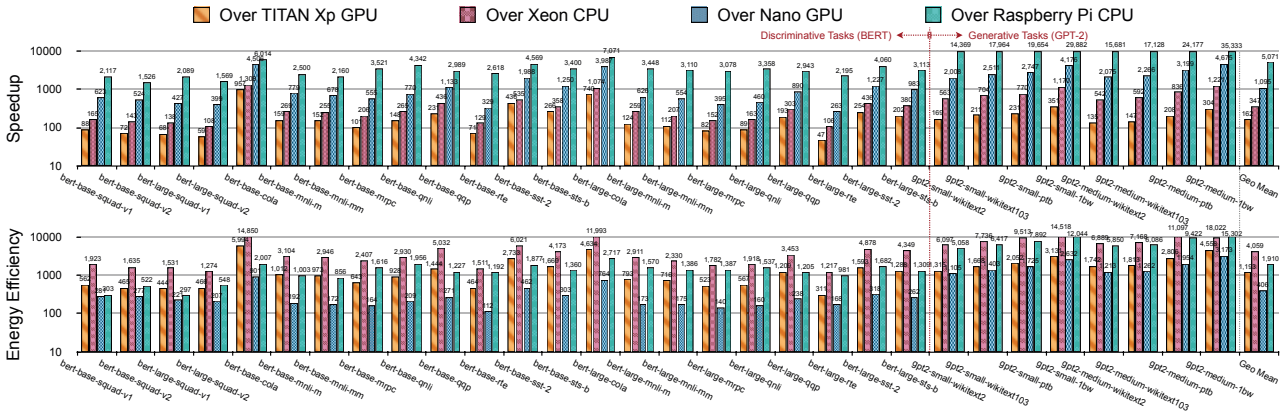

Fig. 14. Speedup and Energy Efficiency of SpAtten over baselines on attention layers. SpAtten can accelerate both discriminative and generative models.

TABLE III
COMPARE SPATTEN$_{1/8}$ WITH PRIOR ART A^3 AND MNNFAST.

	MNNFast	A^3	**SpAtten$_{1/8}$**
Cascade Head Pruning	✗	✗	✓
Cascade Token Pruning	✗	✗	✓
Interpretable Pruning	✗	✗	✓
Local Value Pruning	✓	✓	✓
Progressive Quantization	✗	✗	✓
Preprocessing Overhead	✗	✓	✗
Reduce Computation of	Attention only	Attention only	Attention and FFN
Accelerate	BERT only	BERT only	BERT & GPT-2
Technology	FPGA (28nm)	ASIC (40nm)	ASIC (40nm)
Frequency	1GHz (projected)	1GHz	1GHz
Area (mm^2)	-	2.08 mm^2	1.55 mm^2
Throughput (GOP/s)	120 (1×)	221 (1.8×)	360 (3.0×)
Energy Effi. (GOP/j)	120 (1×)	269 (2.2×)	382 (3.2×)
Area Effi. (GOP/s/mm^2)	-	106 (1×)	238 (2.2×)

brings up to 2.3× speedup for BERT in batch mode (assume performing token pruning twice). GPT-2 results in Figure 14 do not have Beam Search. However, our techniques can also accelerate the Beam Search case because when a token (and its K, V) is pruned, it *will not be used by any beams*.

Comparisons with A^3 and MNNFast. A^3 [6] and MNNFast [7] are also attention accelerators exploring sparsity. A^3 first sorts each dimension of the key vectors among all keys. Then it uses a pre-specified number of largest/smallest elements in the keys to conduct multiplications with a query and get *partial* attention scores. If a score is smaller than a threshold, then the corresponding key will be pruned. MNNFast removes V vectors whose attention probabilities are smaller than a threshold. We compare the differences and performance of SpAtten$_{1/8}$, A^3, and MNNFast in Table III. Specifically: (i) In A^3 and MNNFast, all QKV vectors need to be fetched from DRAM to on-chip buffers before determining what can be pruned, so it cannot reduce DRAM access. Thus, they can only accelerate computation-bounded models (discriminative BERT), but cannot accelerate memory-bounded models

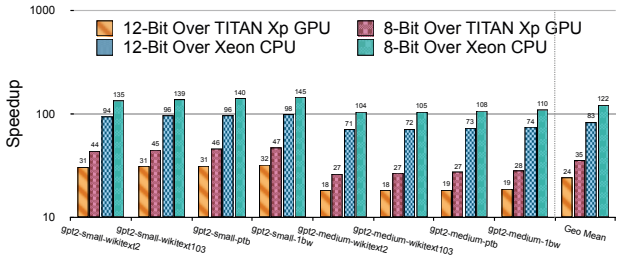

Fig. 15. End-to-End speedup of SpAtten-e2e over baselines. FC layer weights are quantized to 8 bits or 12 bits.

TABLE IV
FC & ATTENTION FLOPS & LATENCY BREAKDOWN ON GPT-2-MEDIUM.

	FC GFLOPs	Attn GFLOPs	FC Latency (ms)	Attn Latency (ms)
GPU	19.3 (85.6%)	3.3 (14.4%)	388.3 (51.4%)	366.7 (48.6%)
SpAtten-e2e	19.3 (95.5%)	0.9 (4.5%)	25.75 (92.4%)	2.13 (7.6%)

(generative GPT-2). (ii) A^3 has pre-processing overhead – sorting the keys. (iii) Token pruning in SpAtten is *global and cascade*, while that in A^3 is local in one head. Therefore, only SpAtten can reduce the computation in both attention and FFN layers. (iv) Cascade token pruning is interpretable and can be intuitively visualized step by step (Figure 22). (v) SpAtten also supports head pruning and progressive quantization.

The parallelism d in A^3 is 64, corresponding to 128 multipliers. We compare A^3 with SpAtten$_{1/8}$ which has the same number of multipliers (128), technology (40nm) and bandwidth (64GB/s) in Table III. Under 1GHz, A^3 throughput is $2\times d$=128GFLOPS. Since it has 1.73× geomean speedup, the effective throughput is 128×1.72=221GFLOPS. We include the same DRAM power for A^3 and SpAtten$_{1/8}$. SpAtten$_{1/8}$ achieves 1.6× better throughput, 1.4× better energy efficiency, and 2.2× better area efficiency over A^3. MNNFast essentially only supports local Value pruning. We get its throughput number with our reproduced simulator under the same bandwidth and number of multipliers. MNNFast was originally a Zynq-7020 FPGA design (10W power). As an optimistic estimation, we assume ASIC implementation consumes 10× less power,

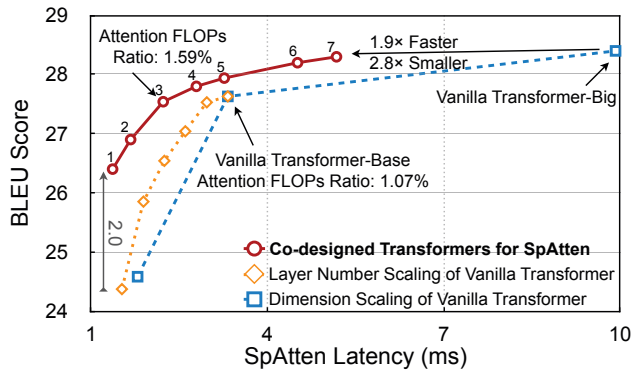

Fig. 16. Co-designed Transformers for SpAtten-e2e can achieve 1.9× speedup and 2.8× model size reduction over vanilla Transformers [1].

i.e., 1W. Compared to MNNFast, SpAtten$_{1/8}$ has 3.0× higher throughput, and 3.2× better energy efficiency.

End-to-End Performance with FFN Support. To compare the end-to-end performance of SpAtten with baselines. We extend our SpAtten to support the FC in the Feed-Forward Network (FFN) layers by reusing the multiplier arrays. The extended architecture is named SpAtten-e2e. FC weights are linear symmetrically quantized to 12 bits and 8 bits and stored on DRAM. Since the FCs in GPT-2 generation stage are matrix-vector multiplications, the performance of SpAtten-e2e is memory-bounded. As shown in Figure 15, on eight GPT-2-Medium benchmarks, 8-bit FC SpAtten-e2e achieves on-average 35× and 122× speedup over TITAN Xp GPU and Xeon CPU; 12-bit FC SpAtten-e2e achieves 24× and 83× respectively. The breakdowns of computation and latency of FC and attention parts of four GPT-2-Medium benchmarks averaged are shown in Table IV. Head pruning is not employed in this comparison. SpAtten-e2e applies token pruning to reduce the attention FLOPs. The FC FLOPs are the same because token pruning can only reduce FC computation in the summarization stage (BERT), not the generation stage (GPT-2). On GPU, the attention only accounts for 14.4% computation but consumes 48.6% latency, echoing with our analysis in Section II-B. By contrast, attention on SpAtten-e2e can be efficiently supported, thus only taking 7.6% latency.

Co-design Model Architecture with SpAtten. Besides the experiments above that leverage existing model architecture, we also explore the potentials of co-designing SpAtten with model architecture by searching a Hardware-Aware Transformer (HAT) [20] for SpAtten-e2e. The search space contains [512, 640, 768] for embedding dim, [512, 1024, 2048, 3072] for FFN layer hidden dim, [1, 2, 3, 4, 5, 6] for decoder layer number, and last three layers for arbitrary encoder-decoder attention. Because the FC layers form the bottleneck of the SpAtten performance, we intentionally configure the lower bound of FFN hidden dimension as low as 512 in expectation of reducing the FC ratio. We set different latency constraints and obtain a series of co-designed Transformers as shown in Figure 16. They are compared with layer number scaling and embedding dimension scaling of vanilla Transformer models [1]. The co-designed Transformer-7 can achieve 1.9× faster

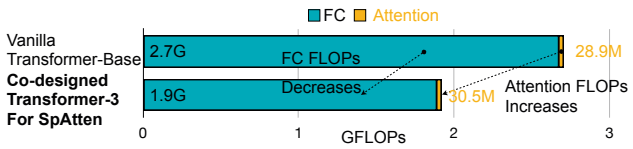

Fig. 17. Co-designed Transformer has slightly more attention but much less FC computations because of SpAtten-e2e's high efficiency on attention.

speed and 2.8× smaller size over the vanilla Transformer-Big model. We also show the computation breakdowns of the vanilla Transformer-Base and the co-designed Transformer-3 in Figure 17. The two models have similar accuracy. Since SpAtten-e2e can support attention with better efficiency, the co-designed model has a larger attention FLOPs. By virtue of the increased attention capacity, the FC computation can be largely shrunk without compromising the accuracy.

C. Performance Analysis

Roofline Analysis. To better understand the distance of SpAtten to the theoretical optimal performance, we analyze its roofline model in Figure 18 and compare it with TITAN Xp GPU. We use theoretical operational intensity: only memory access for input QKV and attention outputs are counted. HBM has 512GB/s bandwidth; thus, the slope of the bandwidth roof is 512G. SpAtten has 1024 multipliers; hence the theoretical computation roof (multiplication and addition) is 2TFLOPS. For BERT, the operation intensity is high, so the performance is computation-bounded. SpAtten achieves 1.61TFLOPS on BERT tasks. That is close to the computation roof and higher than GPU's 0.02TFLOPS. GPT-2 models, on the contrary, have a low arithmetic intensity and appear in the memory-bounded region. SpAtten achieves 0.43TFLOPS, close to the bandwidth roof and higher than GPU's 0.01TFLOPS. GPU performance is far from the roofs in both models because of the low utilization of computation units. Progressive quantization improves the computation intensity; thus, the points of SpAtten are to the right of GPU.

Breakdown of Speedup. Figure 20 shows the speedup breakdown of SpAtten over TITAN Xp GPU on eight GPT-2 benchmarks. With a dedicated datapath, SpAtten is 22.1× faster than GPU baseline, which needs to execute numerous memory instructions for attention. Cascade pruning is then applied to remove unimportant tokens and heads in the second step, reducing computation by 3.8× and 1.1×, respectively. However, the performance only improves by 1.1× for both. The reason is that cascade pruning needs to frequently execute top-k to find unimportant tokens/heads, which becomes a bottleneck without a high throughput top-k engine. Therefore, after adding the high-parallelism top-k engine, the bottleneck is resolved, and the performance jumps by 3×. Finally, the progressive quantization reduces the average bitwidth of inputs, achieving another 2.8× speedup with less DRAM access.

Efficiency-Accuracy Trade-offs. Without accuracy loss, token pruning can prune 1.9× for all benchmarks on average, while head pruning can prune 1.1×. Figure 21 shows two trade-off curves between the token/head pruning ratio and

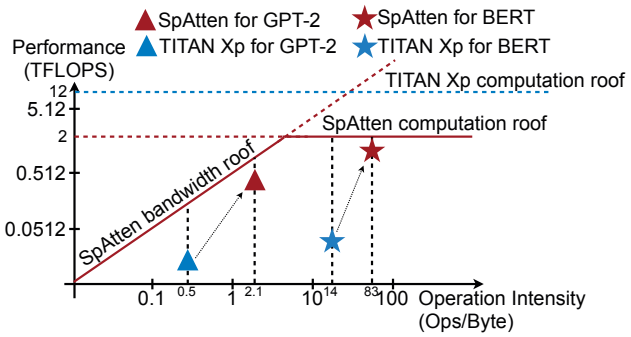

Fig. 18. Roofline Model. The performance of SpAtten is close to bandwidth and computation roofs.

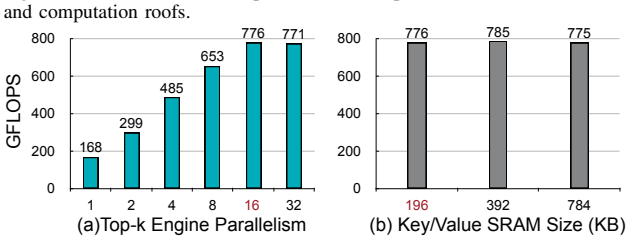

Fig. 19. Design Space Exploration. A top-k engine with a parallelism of 16 and 196KB key/value buffer is sufficient for our settings.

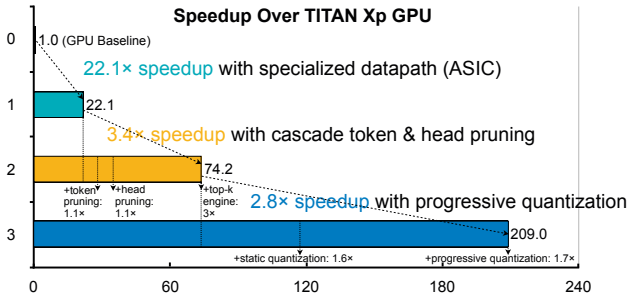

Fig. 20. Speedup Breakdown of SpAtten on GPT-2 models. Cascade pruning and progressive quantization bring 3.4× and 2.8× speedup, respectively.

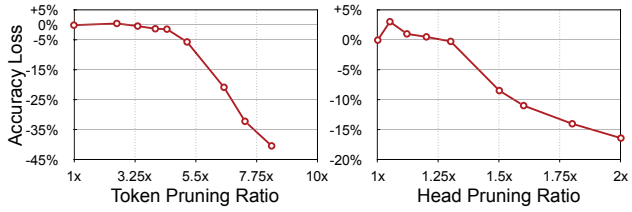

Fig. 21. Trade-off curves between token/head pruning ratio and accuracy loss.

accuracy of GPT-2-Small on PTB (left) and BERT-Base on CoLA (right). For the token pruning curve, head pruning is not applied, and vice versa. We apply 12-bit quantization and disable progressive quantization for both. We can prune around 4× tokens for PTB and 1.2× heads for CoLA without accuracy loss. Small pruning ratios even improve the accuracy. Note that the pruning ratio is related to the input sentence length. Since the sentence length of GPT-2 benchmarks (around 1000) is much longer than BERT ones (less than 100), GPT-2's pruning ratios can be larger while preserving the accuracy.

Design Choice Explorations. We also explore the best architectural settings for SpAtten in Figure 19 on one GPT-2 application. The left side shows the performance with different parallelism (comparator number) of the top-k engine. Comparator number influences the time to perform a STATE_RUN stage (see Algorithm 3). After parallelism 16, the performance does not increase much because 16 matches the top-k engine input data rate from the Q×K module. Thus parallelism larger than 16 makes top-k no longer the bottleneck and cannot much influence the overall performance. We select 16 in our design. On the right side, we change the size of SRAM storing K and V. Since SpAtten supports up to 1024-length context, the minimum SRAM size is set to 2×1024×64×12bits=196KB. '2' is for double buffering. Increasing SRAM size hardly increases the performance because the whole architecture is fully pipelined. More intermediate buffers will not significantly impact the throughput. Therefore, in consideration of reducing SRAM static power, we select the smallest 196KB.

Interpretation and Visualization. Figure 22 visualizes the cascade token pruning process on various tasks. The pruned tokens are redundant ones such as 'it, are, to, is', showing the effectiveness of SpAtten's importance score mechanism. In the first example, the tokens that survive pruning are 'remember', 'admire', 'resolve confusion'; we can easily interpret why the sentence is classified as a positive sentiment. The second example is the similarity score regression. The regressed scores range from 1 to 5, and larger scores indicate higher similarity between two sentences. SpAtten can effectively prune away the meaningless tokens such as 'your' and 'is', and keep the token pairs in two sentences such as 'upset' and 'bothering'. The last example is a generative language modeling with GPT-2. The generated token is 'English'. SpAtten aggressively prunes away most tokens as they are irrelevant to the generated token, and only keeps 'Du', 'translate' and 'into' tokens. The model may find the name 'Du' not typical in English, so the translation language should be 'English'.

Figure 23 shows the cumulative importance scores of every single layer in a GPT-2 LM model. The important tokens are consistent across layers, such as 'published'. Generated 'papers' token heavily attends to several nearby tokens such as 'published' and 'many'. It also attends to some important tokens such as 'researcher' and 'architecture' even though they are far from it. In summary, token pruning reduces the model complexity and shows which tokens are attended most by the model, bringing better interpretability than A^3 and MNNFast.

VI. Related Work

A. Neural Networks Pruning and Quantization

Neural networks tend to be over-parameterized, and many algorithms have been proposed to prune away the redundant parameters. Fine-grained pruning [21], [22] cut offs the connections within the weight matrix and achieves high pruning ratios. However, it is not friendly to CPUs and GPUs and requires dedicated hardware [23], [24] to support sparse matrix multiplication, which may consume extra efforts for design

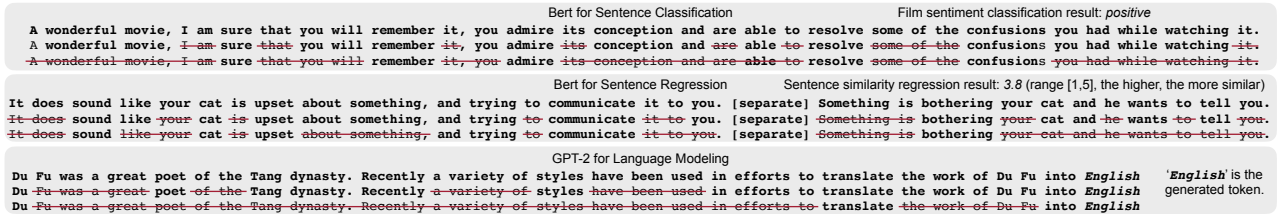

Fig. 22. Examples for cascade token pruning in different models and tasks. SpAtten supports both discriminative models (BERT) and generative models (GPT-2). Unlike A^3 and MNNFast, cascade token pruning in SpAtten is structured and interpretable.

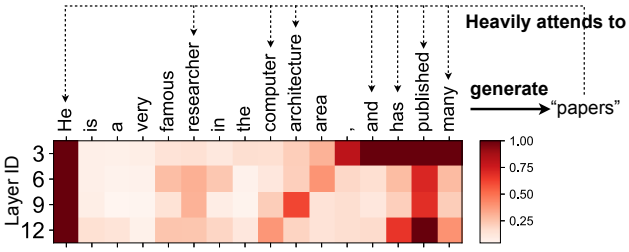

Fig. 23. Cumulative importance scores in GPT-2. Unimportant tokens are pruned on the fly. Important tokens are heavily attended.

and automation [25]–[27]. To this end, structured pruning such as channel-pruning [28], [29] was further proposed to remove the entire channel to enable acceleration on general-purpose hardware. [30] further proposed to enable fine-grained pruning speedup on general-purpose platforms. However, it requires a complicated guided sparsity learning process. Quantization reduces data precision to shrink the model size and bandwidth requirements. SpAtten is fundamentally different from the existing weight pruning and quantization because there is no weight in attention, and we prune the input tokens/heads based on their importance to the sentence. Quantization in SpAtten is applied to input QKV instead of weights.

B. Accelerators for Sparse and Quantized Neural Networks

There have been various domain-specific FPGA and ASIC accelerators proposed for neural networks [31]–[44]. Many of them leverage the sparsity to improve performance [45]–[57]. There also exist general sparse tensor algebra accelerators [58]–[64] proposed in recent years, which can be used to process sparse FC layers. Most of the prior work focuses on leveraging weight sparsity. By contrast, SpAtten leverages activation (token/head) sparsity and employs specialized top-k engines to support on-the-fly cascade token/head pruning. Quantized neural networks are also supported by many accelerators [65]–[74]. In those accelerators, the bitwidth is fixed at the compile time, while in SpAtten, we can adjust the bitwidth according to the attention probability distributions.

C. Efficient Natural Language Processing

The large computation of attention-based NLP models raised much interest in improving their efficiency [20], [75]–[77]. GOBO [74] proposed to compress BERT model down to 3 bits, thus significantly reducing DRAM access. PoWER-BERT [78] prunes tokens based on the *instant* attention probabilities of only one layer, which is different from SpAtten's *cumulative* attention probabilities of multiple layers. It cannot support *per-head granularity* token pruning or local V vector pruning either. Head pruning is proposed in [5], [79] but they only prune head weights instead of activations as in SpAtten. The head pruning in [5] is not cascaded since pruned heads in one layer appear in latter layers. Our token pruning idea can also be generalized to Memory-Augmented Networks [80] to remove unimportant memory vectors and improve efficiency.

VII. Conclusion

We propose SpAtten, a software-architecture co-design to enable efficient sparse and quantized attention inference. We first propose cascade token and head pruning to remove the computation and memory access of inessential tokens and heads. A novel top-k engine is designed to support on-the-fly token and head importance ranking with $O(n)$ time complexity. Moreover, we propose progressive quantization to allow different bitwidths across layers. SpAtten achieves orders of magnitude speedup and energy savings over traditional platforms, and is 1.6× and 3.0× faster than A^3 and MNNFast. We also provide detailed performance analysis, breakdown of each technique, and design space explorations, offering insights to future NLP accelerator designs.

Acknowledgement

Part of this work was supported under NSF CAREER Award #1943349 and DARPA SDH program. We thank MIT Data Science and AI Lab (DSAIL) for supporting this research. We thank Joel Emer, Stephen Keckler, Mike O'Connor, Donghyuk Lee for inspiring discussions.

References

[1] A. Vaswani *et al.*, "Attention is all you need," in *NeurIPS*, 2017.
[2] J. Devlin *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv*, 2018.
[3] A. Radford *et al.*, "Language models are unsupervised multitask learners," *OpenAI Blog*, 2019.
[4] A. Wang *et al.*, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *EMNLP Workshop BlackboxNLP*, 2018.
[5] E. Voita *et al.*, "Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned," *ACL*, 2019.
[6] T. J. Ham *et al.*, "A3: Accelerating attention mechanisms in neural networks with approximation," *HPCA*, 2020.
[7] H. Jang *et al.*, "Mnnfast: A fast and scalable system architecture for memory-augmented neural networks," in *ISCA*, 2019.
[8] P. Rajpurkar *et al.*, "Squad: 100,000+ questions for machine comprehension of text," *EMNLP*, 2016.
[9] S. Merity *et al.*, "Pointer sentinel mixture models," *arXiv*, 2016.
[10] M. P. Marcus *et al.*, "Building a large annotated corpus of English: The Penn Treebank," *Computational Linguistics*, 1993.
[11] C. Chelba *et al.*, "One billion word benchmark for measuring progress in statistical language modeling," *arXiv*, 2013.

[12] E. F. Tjong Kim Sang *et al.*, "Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition," in *HLT-NAACL*, 2003.

[13] R. Socher *et al.*, "Recursive deep models for semantic compositionality over a sentiment treebank," in *EMNLP*, 2013.

[14] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[15] Y. Kim *et al.*, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, 2015.

[16] P. Nilsson *et al.*, "Hardware implementation of the exponential function using taylor series," in *NORCHIP*, 2014.

[17] S. Salehi *et al.*, "Energy and area analysis of a floating-point unit in 15nm cmos process technology," in *SoutheastCon*, 2015.

[18] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to model large caches," *HPL*, 2015.

[19] M. O'Connor *et al.*, "Fine-grained dram: energy-efficient dram for extreme bandwidth systems," in *MICRO*, 2017.

[20] H. Wang *et al.*, "Hat: Hardware-aware transformers for efficient natural language processing," in *ACL*, 2020.

[21] S. Han *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *ICLR*, 2016.

[22] T. Wang *et al.*, "Apq: Joint search for network architecture, pruning and quantization policy," in *CVPR*, 2020.

[23] S. Pal *et al.*, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *HPCA*, 2018.

[24] Z. Zhang, H. Wang *et al.*, "Sparch: Efficient architecture for sparse matrix multiplication," in *HPCA*, 2020.

[25] H. Wang *et al.*, "Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning," *DAC*, 2020.

[26] H. Mao *et al.*, "Park: An open platform for learning-augmented computer systems," *NeurIPS*, 2019.

[27] H. Wang, J. Yang, H.-S. Lee, and S. Han, "Learning to design circuits," *NeurIPS MLSys Workshop*, 2018.

[28] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," *ICCV*, 2017.

[29] Y. He *et al.*, "Amc: Automl for model compression and acceleration on mobile devices," in *ECCV*, 2018.

[30] J. Park *et al.*, "Faster cnns with direct sparse convolutions and guided pruning," *arXiv*, 2016.

[31] M. H. Mubarik *et al.*, "Printed machine learning classifiers," in *MICRO*, 2020.

[32] A. K. Ramanathan *et al.*, "Look-up table based energy efficient processing in cache support for neural network acceleration," in *MICRO*, 2020.

[33] Z. Song *et al.*, "Vr-dann: Real-time video recognition via decoder-assisted neural network acceleration," in *MICRO*, 2020.

[34] H. Kim *et al.*, "Duplo: Lifting redundant memory accesses of deep neural networks for gpu tensor cores," in *MICRO*, 2020.

[35] L. Liu *et al.*, "Duet: Boosting deep neural network efficiency on dual-module architecture," in *MICRO*, 2020.

[36] H. Mo *et al.*, "Tfe: Energy-efficient transferred filter-based engine to compress and accelerate convolutional neural networks," in *MICRO*, 2020.

[37] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *ISCA*, 2020.

[38] M. Vilim *et al.*, "Gorgon: accelerating machine learning from relational data," in *ISCA*, 2020.

[39] L. Ke *et al.*, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *ISCA*, 2020.

[40] R. D. Evans *et al.*, "Jpeg-act: accelerating deep learning via transform-based lossy compression," in *ISCA*, 2020.

[41] U. Gupta *et al.*, "Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference," *ISCA*, 2020.

[42] Y. Zhao *et al.*, "Smartexchange: Trading higher-cost memory storage/access for lower-cost computation," *ISCA*, 2020.

[43] E. Baek *et al.*, "A multi-neural network acceleration architecture," in *ISCA*, 2020.

[44] Y. Ji *et al.*, "Fpsa: A full system stack solution for reconfigurable reram-based nn accelerator architecture," in *ASPLOS*, 2019.

[45] S. Han *et al.*, "Eie: Efficient inference engine on compressed deep neural network," in *ISCA*, 2016.

[46] S. Zhang *et al.*, "Cambricon-x: An accelerator for sparse neural networks," in *MICRO*, 2016.

[47] 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," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[48] X. Zhou *et al.*, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *MICRO*, 2018.

[49] T.-H. Yang *et al.*, "Sparse reram engine: joint exploration of activation and weight sparsity in compressed neural networks," in *ISCA*, 2019.

[50] A. Gondimalla *et al.*, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *MICRO*, 2019.

[51] S. Sharify *et al.*, "Laconic deep learning inference acceleration," in *ISCA*, 2019.

[52] J. Cong *et al.*, "Understanding performance differences of fpgas and gpus," in *FCCM*, 2018.

[53] D. Yang *et al.*, "Procrustes: a dataflow and accelerator for sparse deep neural network training," in *MICRO*, 2020.

[54] Z. Gong *et al.*, "Save: Sparsity-aware vector engine for accelerating dnn training and inference on cpus," in *MICRO*, 2020.

[55] R. Hwang *et al.*, "Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations," *ISCA*, 2020.

[56] E. Qin *et al.*, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *HPCA*, 2020.

[57] A. Delmas Lascorz *et al.*, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *ASPLOS*, 2019.

[58] Y. Chen *et al.*, "tpspmv: A two-phase large-scale sparse matrix-vector multiplication kernel for manycore architectures," *Information Sciences*, 2020.

[59] K. Hegde *et al.*, "Extensor: An accelerator for sparse tensor algebra," in *MICRO*, 2019.

[60] K. Kanellopoulos *et al.*, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *MICRO*, 2019.

[61] M. Zhu *et al.*, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *MICRO*, 2019.

[62] Y. Kwon *et al.*, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *MICRO*, 2019.

[63] M. Mahmoud *et al.*, "Tensordash: Exploiting sparsity to accelerate deep neural network training," in *MICRO*, 2020.

[64] N. Srivastava *et al.*, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *MICRO*, 2020.

[65] J. Lee *et al.*, "Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *ISSCC*, 2018.

[66] P. Judd *et al.*, "Stripes: Bit-serial deep neural network computing," in *MICRO*, 2016.

[67] S. Sharify *et al.*, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *DAC*, 2018.

[68] Y. Umuroglu *et al.*, "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing," in *FPL*, 2018.

[69] T. Rzayev *et al.*, "Deeprecon: Dynamically reconfigurable architecture for accelerating deep neural networks," in *IJCNN*, 2017.

[70] H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ISCA*, 2018.

[71] Nvidia, "Nvidia tensor cores," in *Nvidia*, 2018.

[72] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[73] Z. Song *et al.*, "Drq: dynamic region-based quantization for deep neural network acceleration," in *ISCA*, 2020.

[74] A. H. o. Zadeh, "Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference," in *MICRO*, 2020.

[75] Z. Yan *et al.*, "Micronet for efficient language modeling," *JMLR*, 2020.

[76] J. Gu *et al.*, "Non-autoregressive neural machine translation," in *ICLR*, 2018.

[77] H. Wang, "Efficient algorithms and hardware for natural language processing," *Massachusetts Institute of Technology*, 2020.

[78] S. o. Goyal, "Power-bert: Accelerating bert inference for classification tasks," *ICML*, 2020.

[79] P. Michel *et al.*, "Are sixteen heads really better than one?" in *NeurIPS*, 2019.

[80] S. Sukhbaatar *et al.*, "End-to-end memory networks," in *NeurIPS*, 2015.