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

Hanrui Wang, Zhekai Zhang, Song Han
Massachusetts Institute of Technology
Cambridge, MA, US
{hanrui, zhangzk, songhan}@mit.edu
https://github.com/mit-han-lab/spatten

Abstract—The attention mechanism is becoming increasingly popular in Natural Language Processing (NLP) applications, showing superior performance than convolutional and recurrent architectures. However, attention becomes the computation bottleneck because of its quadratic computational complexity to input length, 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 heads. Cascade pruning requires sorting token/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 $10.0 \times$ with no accuracy loss, and achieves $1.6 \times$ speedup, and $1.4 \times$, $3.2 \times$, $1193 \times$, $4059 \times$ energy savings over A3 accelerator, MNFFast 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, due to its quadratic computational complexity to input length, 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. 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 inefficiency is even more severe for modern Large Language Models (LLMs) with long context length and large KV cache. 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, 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

As a visual treat, the film is almost perfect.

| Layer | Tokens | Heads | Computation & Memory Access |
|-------|--------|-------|-----------------------------|
| Layer 1 | 11 | 12 | 100% |
| Layer 2 | 6 | 10 | 38% |
| Layer 3 | 2 | 8 | 12% |

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

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 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× 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×), and progressive quantization (5.1×). (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,
Fig. 3. NLP model architecture with attention. BERT only contains the summarization stage, GPT-2 contains summarization and generation stages.

- **Cascade Head Pruning** removes unimportant heads and save DRAM access and computation by another 1.1x.

- **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 51.1x.

- **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], Text103 [9], Pen Tree Bank [10] and Google One-Billion Word [11] with BERT and GPT-2 models. SpAtten reduces DRAM access by 10.0x with no accuracy loss, and achieves 1.6\times 3.0\times 162\times 347\times 1095\times 5071\times speedup, and 1.4\times 3.2\times 1193\times 4059\times 406\times 1910\times energy savings over A\(^3\) accelerator, MNFast 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].

```
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
    for \(r = 0\) to \(L_0\) do
        \(attention\_score \in \mathbb{R}^{L_0 \times L_1};\)
        \(attention\_score = Q[head_id] \cdot K[head_id]^T;\)
        \(attention\_score = attention\_score / \sqrt{D};\)
        \(attention\_prob \in \mathbb{R}^{L_0 \times L_1};\)
        for \(row_id = 0\) to \(L_0\) do
            \(attention\_prob[row_id] = \text{Softmax}(attention\_score[row_id]);\)
        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: \(attention\_out \in \mathbb{R}^{L_0 \times D_in};\)
```

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, 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.

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, \(Q \times K^T / \sqrt{D}\) 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 large. After that, a row-wise softmax computes the attention probabilities (see Figure 3).

**Algorithm 2: Token and Head Pruning (one layer)**

```plaintext```
Input: Number of heads: h;
Token pruning ratio: p_t; Head pruning ratio: p_h;
attention_prob ∈ ℝ^L x D x L x 1, attention_out ∈ ℝ^L x D x m;
Previous cumulative token importance score: st ∈ L x 1;
Previous cumulative head importance score: sh ∈ h;
Reshape attention_out by head, get E ∈ ℝ^h x L x D;
/* accumulate the token importance */
for token_id = 0 to L do
  for l = 0 to L do
    for head_id = 0 to h do
      s_t[token_id] += attention_prob[head_id][l][token_id]
    end
  end
/* accumulate the head importance */
for head_id = 0 to h do
  for l = 0 to L do
    for d = 0 to D do
      s_h[head_id] += abs(E[head_id][l][d])
    end
  end
remained_token_id = top-k(st, L1 × (1 - p_t))
remained_head_id = top-k(sh, h × (1 - p_h))
Output: remained_token_id, remained_head_id, st, sh
```
```

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_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.

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

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 × 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 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 $3.8 \times$ 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_probability $V$ 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.

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 $\Delta s$ to the attention score $s$. We examine the influence of $\Delta s$ on output attention probabilities by computing the softmax derivative:

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

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:

$$\text{error} = |\Delta s_0 \cdot p_0 \cdot (1 - p_0)| + \sum_{i=1}^{L_1-1} |\Delta s_i \cdot p_0 \cdot p_i|$$

$$= \Delta s_0 \cdot (2p_0(1 - p_0)) < \Delta s_0$$

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. Theoreti-
cally, 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 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-bound 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 $32 \times 16$ crossbar for 16 HBM channels. It then gets data back through a reverse $16 \times 32$ 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...
Importance Score
Accumulator
Q, K, V Fetcher
Bitwidth Convertor
FIFO \times 32
Address
Q \times KT
Softmax
... pruned away
0.4 1.0 0.3 1.2 1.7 1.0 0.4 1.8 0.6 1.9 1.4 0.3 0.9 0.4

cumulative importance scores:
Sum each column to get
prob/attention
pivot
R)
+ th
R)
eq
k
pivot
L
eq
RUN
largest
cnt
th
largest, num
pivot
eq
largest
parts: elements smaller or larger than the pivot. It has two a randomly chosen pivot to partition the input array into two and keeps the original order of inputs. The engine leverages find the n
n
log
n
\cdot \log n
n
\cdot \log^2 n
space sorting network, and also completely randomizes data fetching. Instead, we design an innovative logic for the zero eliminator, which also prunes away the largest element as a threshold 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.

B. Top-k Engine
To support cascade head/pruning and local value pruning, we need to find the top k elements of an array. A naïve solution is to use a sort 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.

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\(x\) 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 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_j = \sum_j K_{ij} \times Q_{ij}. 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_{i}s are packed into the same line in the Key SRAM, and the query is broadcast for 512/D times so that all K_{i}s 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_j.

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_{ij} = \sum_i \text{attention prob}_i \times 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 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 SpAtten1/8. SpAtten1/8 is only used for fair comparisons with MNNFast and A^3. The parameters for SpAtten are listed in Table I. The scale of SpAtten1/8 is 1/8 of SpAtten and contains 128 multipliers. We synthesize SpAtten using Cadence Genius 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 exponential function is approximated with Taylor expansion to the 5th order [16] and performed with floating multipication 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 A3 [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 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 $\times$ latency.

### B. Experimental Results

#### Throughput, Power, and Area. SpAtten prunes tokens and value vectors with cascade token pruning and local value pruning by $1.9 \times$ (all models average), and $3.8 \times$ (GPT-2 models average). Cascade head pruning has $1.1 \times$ 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 $2.1 \times$ and DRAM access by $10.0 \times$ 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 $\times$ K and Attention_Prob $\times$ 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 $162 \times$, $347 \times$, $1095 \times$, and $5071 \times$ speedup, and $1193 \times$, $4059 \times$, $406 \times$, and $1910 \times$ 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 FPN layer computations are reduced by token pruning, CPUs and GPUs can also be accelerated. We implement token pruning on CPUs/GPUs.

| Computation Logic | SRAM | DRAM | Overall |
|-------------------|-----|------|--------|
| Power | 1.36W | 1.24W | 5.71W | 8.30W |
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 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 A³ and MNNFast.** A³ [6] and MNNFast [7] are also attention accelerators exploring sparsity. A³ 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 SpAtten₁/₈, A³, and MNNFast in Table III. Specifically:

(i) In A³ 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 (generative GPT-2). (ii) A³ has pre-processing overhead — sorting the keys. (iii) Token pruning in SpAtten is global and cascade, while that in A³ 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 A³ is 64, corresponding to 128 multipliers. We compare A³ with SpAtten₁/₈ which has the same number of multipliers (128), technology (40nm) and bandwidth (64GB/s) in Table III. Under 1GHz, A³ throughput is $2 \times d = 128 \times 64$. Since it has 1.73× geometric speedup, the effective throughput is $128 \times 1.72 = 221 \times 64$. We include the same DRAM power for A³ and SpAtten₁/₈. SpAtten₁/₈ achieves 1.6× better throughput, 1.4× better energy efficiency, and 2.2× better area efficiency over A³. 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, i.e., 1W. Compared to MNNFast, SpAtten has 3× 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 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 $1.9 \times$ for all benchmarks on average, while head pruning can prune $1.1 \times$. Figure 21 shows two trade-off curves between the token/head pruning ratio and 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 $4 \times$ tokens for PTB and $1.2 \times$ 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 $2 \times 1024 \times 64 \times 12$ bits = 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’, ‘admiré’, ‘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’.

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] cuts off the con-
Heavily attends to

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 and automation [25], [26], [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]–[36]. Many of them leverage the sparsity to improve performance [37]–[73]. There also exist general sparse tensor accelerator innovations [74]–[85] 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 topk engines to support on-the-fly cascade token/head pruning. Quantized neural networks are also supported by many accelerators [86]–[95]. 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], [96]–[98]. GOBO [95] proposed to compress BERT model down to 3 bits, thus significantly reducing DRAM access. PoWER-BERT [99] 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], [100] 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 [101] 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 \( 1.6 \times \) and \( 3.0 \times \) 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 ISC, 2019.
[8] P. Rajpurkar et al., “Squad: 100,000+ questions for machine comprehension of text,” EMNLP, 2016.
[77] M. Zhu et al., “Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus,” in MICRO, 2019.
[78] Y. Kwon et al., “Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning,” in MICRO, 2019.
[79] M. Mahmoud et al., “Tensordash: Exploiting sparsity to accelerate deep neural network training,” in MICRO, 2020.
[80] N. Srivastava et al., “Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product,” in MICRO, 2020.
[81] B. Asgari et al., “Alrescha: A lightweight reconfigurable sparse-computation accelerator,” in HPCA, 2020.
[82] N. Srivastava et al., “Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations,” in HPCA, 2020.
[83] J. Weng et al., “A hybrid systolic-dataflow architecture for inductive matrix algorithms,” in HPCA, 2020.
[84] A. D. Lascorz et al., “Shapeshifter: Enabling fine-grain data width adaptation in deep learning,” in MICRO, 2019.
[85] F. Sadi et al., “Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization,” in MICRO, 2019.
[86] J. Lee et al., “Upu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision,” in ISSCC, 2018.
[87] P. Judd et al., “Stripes: Bit-serial deep neural network computing,” in MICRO, 2016.
[88] S. Sharify et al., “Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks,” in DAC, 2018.
[89] Y. Umuroglu et al., “Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing,” in FPL, 2018.
[90] T. Rzayev et al., “Deeprecon: Dynamically reconfigurable architecture for accelerating deep neural networks,” in ICNN, 2017.
[91] H. Sharma et al., “Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network,” in ISCA, 2018.
[92] Nvidia, “Nvidia tensor cores,” in Nvidia, 2018.
[93] N. P. Jouppi et al., “In-datacenter performance analysis of a tensor processing unit,” in ISCA, 2017.
[94] Z. Song et al., “Drq: dynamic region-based quantization for deep neural network acceleration,” in ISCA, 2020.
[95] A. H. o. Zadeh, “Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference,” in MICRO, 2020.
[96] Z. Yan et al., “Micronet for efficient language modeling,” JMLR, 2020.
[97] J. Gu et al., “Non-autoregressive neural machine translation,” in ICLR, 2018.
[98] H. Wang, “Efficient algorithms and hardware for natural language processing,” Massachusetts Institute of Technology, 2020.
[99] S. o. Goyal, “Power-bert: Accelerating bert inference for classification tasks,” ICML, 2020.
[100] P. Michel et al., “Are sixteen heads really better than one?” in NeurIPS, 2019.
[101] S. Sukhbaatar et al., “End-to-end memory networks,” in NeurIPS, 2015.