Efficient Algorithms and Hardware for Natural Language Processing

by

Hanrui Wang

B.Eng., Fudan University (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 15, 2020

Certified by .....
Song Han
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Efficient Algorithms and Hardware for Natural Language Processing

by

Hanrui Wang

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2020, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Natural Language Processing (NLP) is essential for many real-world applications, such as machine translation and chatbots. Recently, NLP is witnessing rapid progresses driven by *Transformer* models with the *attention* mechanism. Though enjoying the high performance, Transformers are challenging to deploy due to the intensive computation. In this thesis, we present an algorithm-hardware co-design approach to enable efficient Transformer inference. On the algorithm side, we propose Hardware-Aware Transformer (HAT) framework to leverage Neural Architecture Search (NAS) to search for a specialized low-latency Transformer model for each hardware. We construct a large design space with the novel *arbitrary encoder-decoder attention* and *heterogeneous layers*. Then a *SuperTransformer* that covers all candidates in the design space is trained and efficiently produces many *SubTransformers* with weight sharing. We perform an evolutionary search with a hardware latency constraint to find a *SubTransformer* model for target hardware. On the hardware side, since general-purpose platforms are inefficient when performing the attention layers, we further design an accelerator named *SpAtten* for efficient attention inference. SpAtten introduces a novel *token pruning* technique to reduce the total memory access and computation. The pruned tokens are selected on-the-fly based on their importance to the sentence, making it fundamentally different from the weight pruning. Therefore, we design a *high-parallelism top-k engine* to perform the token selection efficiently. SpAtten also supports *dynamic low-precision* to allow different bitwidths across layers according to the attention probability distribution.

Measured on Raspberry Pi, HAT can achieve **3**× speedup, **3.7**× smaller model size with **12,041**× less search cost over baselines. For attention layer inference, SpAtten reduces DRAM access by **10.4**× and achieves **193**×, **6218**× speedup, and **702**×, **1244**× energy savings over TITAN Xp GPU and Raspberry Pi ARM CPU.

Thesis Supervisor: Song Han
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Song Han. As a great advisor, researcher, and friend, Professor Han always inspires and guides me with his deep insights in both algorithms and hardware, both academia and industry. I am especially grateful for his warm encouragements and supports in my difficulties. His spirit for always targeting impactful research topics profoundly motivates me.

I sincerely appreciate the support from Professor Hae-Seung Lee, who provided invaluable feedback to my first project here at MIT. I am highly thankful to Professor William J. Dally, Professor Joel S. Emer, and Professor Vivienne Sze for offering me guidance and insights on domain-specific accelerator design. I am also especially grateful to Professor David Z. Pan for broadening my horizons about the academia. It was such a great memory to travel to the MLCAD conference in Canada with him. I am very thankful to my academic advisor Professor Charles E. Leiserson, for providing me much guidance on learning at MIT. I also thank Dr. Chuang Gan for always being eager to help and for giving many vital suggestions to this project.

I want to deliver my tremendous appreciation to my undergraduate research advisors Professor Jason Cong, Professor C.-J. Richard Shi, Professor Xiaoyang Zeng, and Professor Yibo Fan for introducing me to the palace of computer architecture and machine learning research.

I am incredibly fortunate to have so many brilliant colleagues and great friends in the MIT HAN Lab. In particular, I thank Zhekai Zhang and Yujun Lin for their selfless help on architecting the accelerators. I also thank Zhanghao Wu and Zhijian Liu for our brainstorms on the NLP algorithms. I am thankful to Ji Lin, Han Cai, Ligeng Zhu, Jiacheng Yang, Driss Hafdi, and Sibo Zhu for the numerous inspiring discussions. Thanks also go to Kuan Wang, Tianzhe Wang, and Muyang Li for our fabulous time on basketball, badminton, and tennis courts.

I am grateful to all my dear friends who support and encourage me in my research and daily life, especially to Ruoyu Han, Yichen Yang, Jiaqi Gu, Zhengqi Gao, Xiao

Gu, Wencan Liu, and Anand Chandrasekhar.

Part of this work was published in the 2020 Annual Conference of the Association for Computational Linguistics. I thank NSF Career Award #1943349, MIT-IBM Watson AI Lab, Semi-conductor Research Corporation (SRC), Intel, AMD, Samsung, and Facebook for supporting this research.

Lastly and most importantly, I would like to thank my parents Guihua Tian and Qinglin Wang for their unconditional and everlasting love and support. They always encourage me to be stronger when facing difficulties and always have great confidence in me. This work would not be possible without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Natural Language Processing (NLP) is the key technique for numerous real-world applications, including machine translation, document summarization, and chatbots. With the rise of Deep Learning (DL) [48], Neural Network (NN) models were widely adopted in the NLP tasks and achieved impressive performance.

Traditionally, the NN models for NLP are Recurrently Neural Network (RNN) models and Convolutional Neural Network (CNN) models [42, 106, 33, 58, 52, 83]. RNN models process the input tokens one by one and store the information with internal states. Typical RNN cells include Long Short-Term Memory (LSTM) [31] and Gated Recurrent Unit (GRU) [13]. CNN models leverage convolution layers to process multiple input tokens at once to extract local features. High-level features can then be obtained by stacking multiple convolution layers.

Lately, the NLP area is witnessing much faster advancements by virtue of the invention of the *attention mechanism* [4, 88]. Attention-based NN models such as Transformer [88], BERT [16] and GPT-2 [69] provide significant performance improvements over RNN and CNN models. BERT [16] even outperforms human performance on the challenging question answering [71] and sentence classification [90] tasks. The better performance derives from attention's capability to capture long-range token dependencies and extract contextualized representations. Specifically, the attention

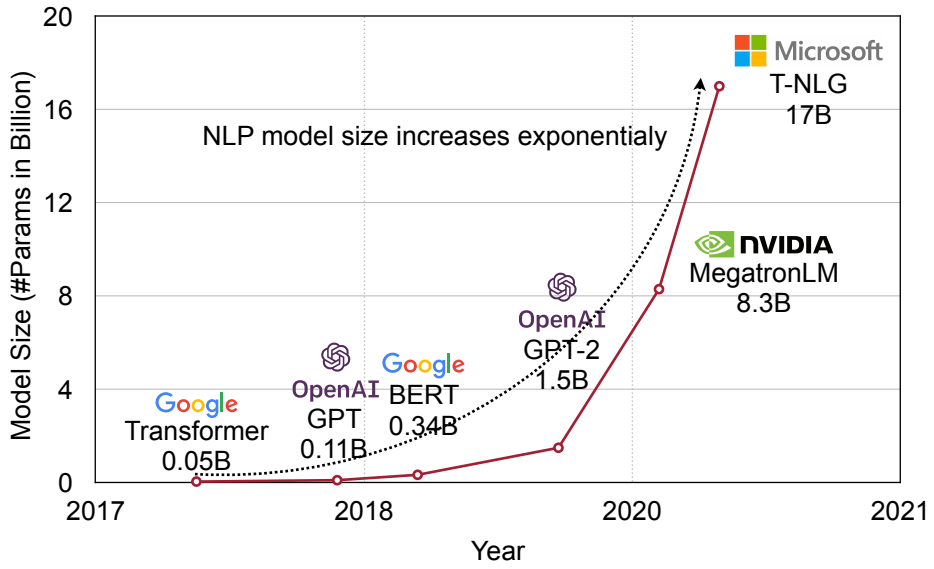

Figure 1-1: Attention-based NLP model size increases exponentially in recent years.

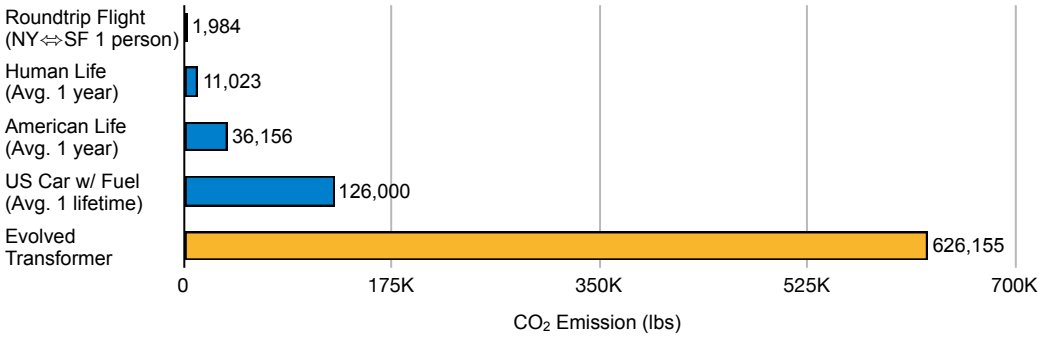

Figure 1-2: Existing method [79] to search for efficient NLP models is expensive [81].

mechanism can assess the importance of a long-range of tokens and select the ones to *attend* in the latter layers, while CNN only has local receptive fields. Besides, attention is easier to parallelize, because the data dependencies are much fewer than the sequential RNN, making large model training feasible.

To pursue higher performance, the sizes of recent attention-based models are increasing exponentially, as in Figure 1-1. The exploding model size and computation complexity bring severe efficiency issues, making it extremely challenging to deploy NLP models on resource-limited edge devices. For instance, in order to translate a sentence with only 30 tokens, a Transformer-Big model needs to execute 13G FLOPs and takes 20 seconds on a Raspberry Pi. Such long latency will hurt the user experience and make real-time NLP applications impossible on mobile devices. There-

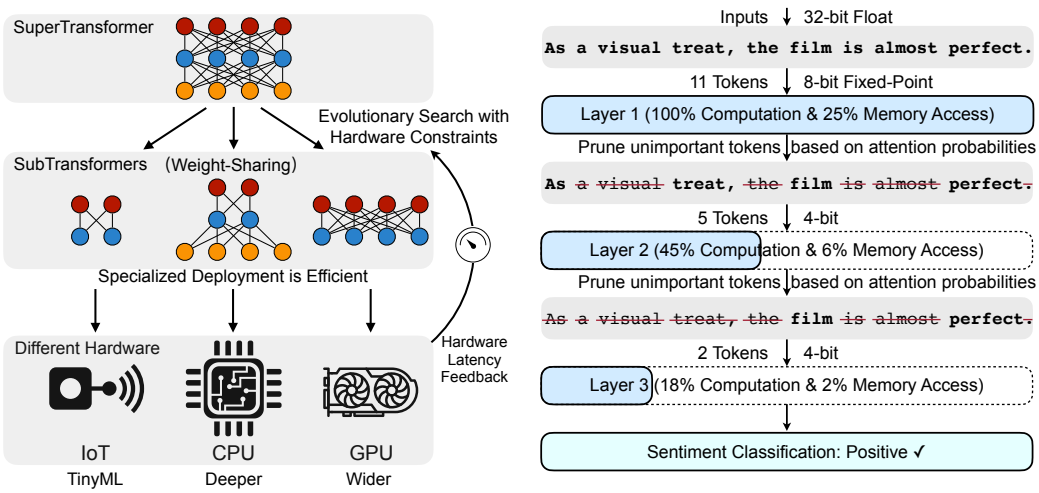

Figure 1-3: Left: Hardware-Aware Transformer (HAT) searches for an efficient and specialized NLP model for each hardware platform (the algorithm contributions of this thesis). Right: SpAtten, a hardware accelerator for the attention mechanism, removes redundant tokens across layers with token pruning, and reduces the bitwidth with dynamic low-precision (the hardware contributions of this thesis).

fore, we need to design hardware-efficient NLP models. There exist research efforts searching for efficient NLP models, such as the Evolved Transformer [79]. However, the search process itself is highly expensive. Figure 1-2 shows the CO_2 emission of searching an Evolved Transformer. The cost of searching one model is five times more than a car in its lifetime, which is clearly unaffordable and unscalable given numerous hardware platforms, from the cloud to the edge. To this end, we propose to leverage a weight-shared supernet to *e ciently* search for *e cient* models.

Furthermore, we find that NLP models' attention layers form the performance bottleneck on general-purpose hardware platforms such as CPUs and GPUs. Attention layers only account for 1% of overall FLOPs but consume over 50% of the processing time due to its low arithmetic intensity and complicated data movements. To this end, it is necessary to design a customized accelerator to process the attention layers.

In this thesis, we propose an algorithm-hardware co-design approach to enable efficient NLP model inference. On the algorithm side, we present the Hardware-Aware Transformer (HAT) framework to leverage Neural Architecture Search (NAS) to search for an efficient model dedicated to run fast on target hardware as shown

in Figure 1-3 left. On the hardware side, we design a specialized accelerator, named *SpAtten* functioning as a co-processor to compute the attention layers in the NLP models to further speedup the inference. SpAtten supports novel *token pruning* and *dynamic low-precision* to reduce the memory access and computation of attention layers, as illustrated in Figure 1-3 right.

1.2 Background

1.2.1 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, sentence-level classification, and regression etc. For instance, Named Entity Recognition (NER) is a token-level classification task classifying tokens into one of several semantic classes such as a person, location, organization, and misc in CoNLL-NER dataset [85]. Sentiment classification such as Stanford Sentiment Treebank V2 (SST-2) [80] is a typical sentence-level classification task which classifies movie reviews into positive or negative ones. Semantic Textual Similarity Benchmark (STS-B) [9] is a sentence-level regression task to regress a score between 1 and 5 as the similarity between two input sentences.

Meanwhile, models for generative tasks need to summarize the input information firstly and then generate new tokens. Examples of generation tasks include Language Modeling (LM) [69], machine translation [88] and text summarization [99]. Language modeling is used to predict a probability distribution of the next word given an input sentence. Machine translation translates sentences from one language to another one. Text summarization summarizes a long sentence or paragraph to a concise one while preserving the meanings.

BERT for discriminative, GPT-2 and Transformer for generative tasks are among the most widely-used attention-based NLP models. Figure 1-4 illustrates BERT,

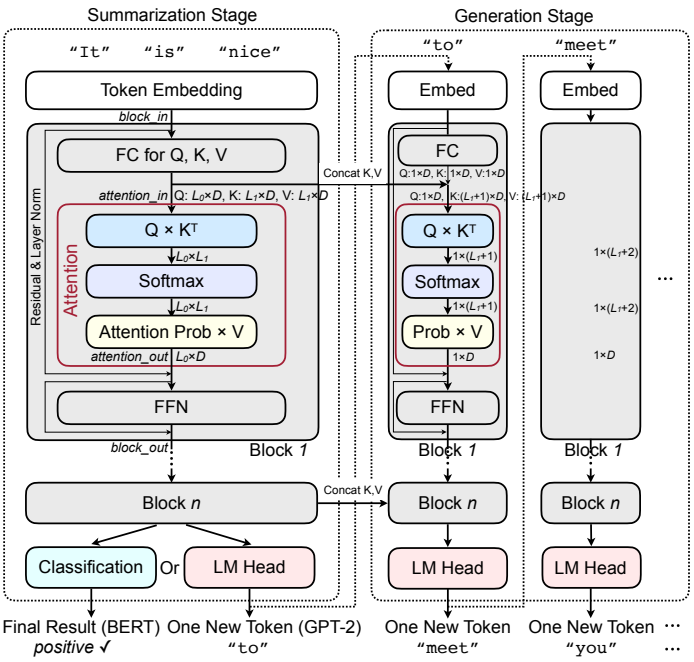

Figure 1-4: Attention-based NLP model architectures. BERT only contains the summarization stage. GPT-2 and Transformer contain both summarization and generation stages.

GPT-2 and Transformer model architectures. BERT only contains the summarization stage, while GPT-2 and Transformer models firstly performs summarization and then generation stage. The summarization and generation stages can also be called ***encoder*** and ***decoder*** respectively. In the summarization stage (Figure 1-4 left), the input tokens are firstly embedded into vectors and processed by blocks. Inside each block, *block_in* is firstly multiplied with three matrices to get Query (Q), Key (K), and Value (V). Then Q, K, and V are processed by attention to get the intermediate features *attention_out*. A residual layer adds the *attention_out* with *block_in* and performs layer normalization. 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 then enter the generation stage.

The generation stage (Figure 1-4 right) has two main differences from the summa-

rization stage: (i) Each iteration only processes *one single token* instead of the whole sentence. (ii) 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. The generation stage ends when the 'end of sentence' token is generated, or the sentence length reaches a predefined limit. The attention in the summarization stage is *self-attention*, and that in the generation stage is *cross-attention*.

The Transformer model is similar to the GPT-2 model with three major differences. (i) There is no LM head after the last block in the summarization stage in Transformer. Therefore, the first token fed to the generation stage is a 'start of sentence' special token. (ii) In each block, there exists one additional self-attention layer before the cross-attention. In the additional self-attention, the Q is a single vector, while K and V are in batch by concatenating Ks and Vs from previous generation iterations. (iii) The cross-attentions in all blocks in the generation stage concatenate the Ks and Vs from only the last block of the summarization stage.

1.2.2 Attention Mechanism

The attention mechanism is shown in Algorithm 1. In the summarization stage, K, Q, 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 K, Q, and V. Different heads capture various dependencies between tokens, some for long-term, and some for short-term. Inside each head, $\mathrm{Q} \times \mathrm{K}^T/\mathrm{sqrt}(D)$ gives attention scores, where D is the dimension of K, Q and V in one head. The attention scores indicate whether too tokens are related. For instance, in Figure 2-6, the score for 'more' attending to 'than' is large, indicating their strong relationship. After that, a row-wise softmax computes the attention probabilities. The exponential of softmax further enlarges the attention scores for highly-related token pairs. The feature of the head is then 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. If

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**
  $attention\_prob[row_{id}] = \text{Softmax}(attention\_score[row_{id}])$
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: $attention\_out \in \mathbb{R}^{L_0 \times D_{in}}$;

there is more than one head, an additional FC layer is applied to the *attention_out*. If only one head, the FC layer is not required.

In BERT-Base and GPT-2-Small, there are 12 attention layers, and each layer has 12 heads. There are 24 attention layers and 16 heads for each in BERT-Large and GPT-2-Medium. Transformer-Base has six layers, and each layer has eight heads, while Transformer-Big has six layers and 16 heads for each.

1.3 Motivation

1.3.1 Hardware-Aware Neural Architecture Search

Through experiments, we find two pitfalls when evaluating the efficiency of an NLP model. (i) *FLOPs does not reflect the measured latency*. Although FLOPs is used as a metric for efficiency in prior arts [32, 100], it is not a good latency proxy. As in Figure 1-5 right, models with the *same* FLOPs can result in very *di erent* measured latencies; (ii) *Di erent hardware prefers di erent model architectures.* As in Table 1.1, the model optimized on one hardware is *sub-optimal* for another because latency is

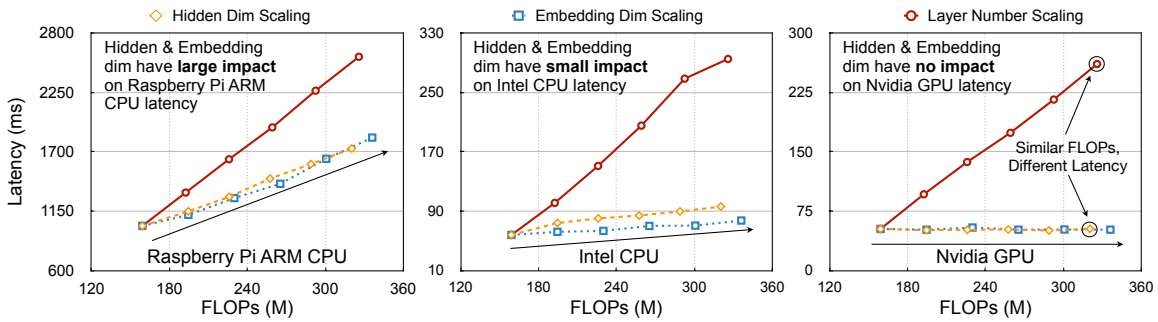

Figure 1-5: Latencies of different Transformer models on different hardware.

Specialized For ↓	Measured On → BLEU	GPU Latency	ARM CPU Latency
HAT (GPU)	28.10	**147 ms**	6491 ms
HAT (ARM CPU)	28.15	184 ms	**6042 ms**

Table 1.1: BLEU score and measured inference latency of HAT models on the WMT'14 En-De task. The efficient model for GPU is not efficient for ARM CPU and vice versa.

influenced by different factors on different hardware platforms. For example, the embedding size has a significant impact on the Raspberry Pi latency but hardly influences the GPU latency (Figure 1-5).

Motivated by the success of Neural Architecture Search (NAS) [6, 23, 66, 7], we propose HAT to search for efficient models involving the latency feedback into the design loop. In this way, we do not need FLOPs as the latency proxy and can explore specialized models for various hardware.

We first construct a large search space with *arbitrary encoder-decoder attention* and *heterogeneous Transformer layers*. Traditional Transformer has an information bottleneck between the encoder and decoder. Arbitrary encoder-decoder attention breaks the bottleneck, allowing all decoder layers to attend to multiple and different encoder layers instead of only the last one. Thus low-level information from the encoder can also be used by the decoder. Motivated by Figure 1-5, we introduce heterogeneous Transformer layers to allow different layers to have different architecture adapting various hardware.

To perform a low-cost search in such a large design space, we first train a Trans-

former supernet [66] – SuperTransformer, which contains many SubTransformers sharing the weights. We train all SubTransformers simultaneously by optimizing the uniformly sampled SubTransformers from the SuperTransformer. The performance of a SubTransformer with inherited weights from the SuperTransformer can provide a good relative performance approximation for different architectures trained from scratch. Unlike conventional NAS, we only need to pay the SuperTransformer training cost for *once* and can evaluate *all* the models in the design space with it. Finally, we conduct an evolutionary search to find the best SubTransformer under the hardware latency constraint. Experiments show that HAT can be naturally incorporated with model compression techniques such as quantization and knowledge distillation.

1.3.2 Attention Accelerator

The attention layers form the performance bottleneck of NLP models. We profiled the end-to-end latency of a GPT-2 model on multiple hardware platforms in Figure 1-6. Attention typically accounts for over 50% latency, even though it only has 1% of overall FLOPs. In Figure 1-6 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 making attention very slow. Since fast execution is critical to interactive applications, it is necessary to build an accelerator to solve the attention bottleneck as a co-processor. The FC layers of NLP models can be processed by GPUs, CPUs, or tensor algebra accelerators as they are highly-optimized for FC, and the co-processor handles all attention layers.

Since the arithmetic intensity of attention can be very low, such as two operations per data (0.5 ops/Byte) for the vector-matrix multiplication Q $\times$ K in the GPT-2 model, the performance of the accelerator is easily bounded by the memory bandwidth. To this end, we propose *token pruning* as shown in Figure 1-3 right, to reduce the DRAM access and computation. Motivated by human languages being highly redundant due to many structural and meaningless tokens such as prepositions, articles, and adverbs, we can safely remove those inconsequential tokens with

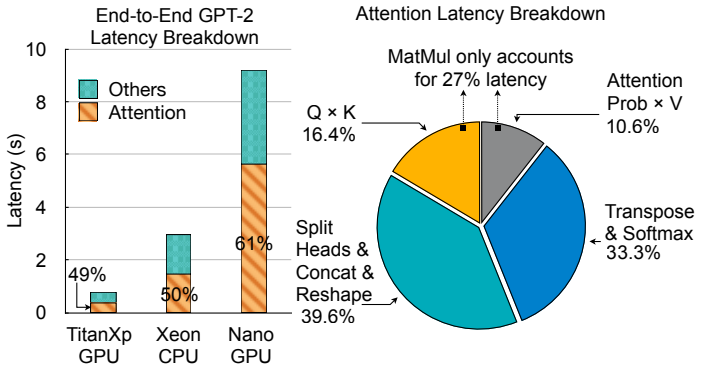

Figure 1-6: End-to-End GPT-2 latency breakdown on various platforms, and attention latency breakdown on TITAN Xp GPU.

little impact on the results. Token pruning is fundamentally different from the classic weight pruning because: (i) There is no trainable weight in attention. (ii) Pruned weights are determined at compile time and consistent for all inputs. By contrast, pruned tokens are selected on-the-fly and vary between different inputs. Specifically, we prune the tokens according to the cumulative importance scores obtained by accumulating across layers the attention probabilities, which are indicators for token influence. Moreover, since long sentences are naturally more redundant, we also adaptively adjust the pruning ratios based on sentence length: the longer, the more tokens are pruned away. To support token pruning, we design and implement a specialized high parallelism top-k engine with $O(n)$ time complexity to get the k most essential tokens and prune others. On average, token pruning can reduce DRAM access and computation by 5.2× on eight GPT-2 benchmarks.

To further reduce the DRAM access, we also propose *dynamic low-precision* for attention inputs. We find an interesting phenomenon that quantization errors are related to attention probability distributions: if the distribution is dominated by a few tokens, the error will be small; while for the flat distribution, the error is large. We also provide a theoretical proof for this phenomenon in Section 2.2.2. Based on this observation, we quantize more aggressively for attention with dominated attention probabilities and more conservatively for others. Concretely, we firstly fetch MSBs of attention inputs to compute attention probabilities. If the max probability is smaller than a threshold, indicating the distribution is flat, we will fetch LSBs on-chip and

redo the computation. In such a way, we trade computation to less memory access, which is beneficial to memory bounded models. With dynamic low-precision, we can save another 5× memory access.

SpAtten accelerator has a broad range of supported applications by virtue of the unique generalization ability of attention-based NLP models. For instance, the BERT model can be used to arbitrary discriminative tasks, such as sentence sentiment classification (SST-2 [80]) and sentence similarity regression (STS-B [9]), 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 [69], document summarization [70] etc. SpAtten also supports Transformers for machine translation [87].

1.4 Thesis Overview and Contributions

This thesis proposes an algorithm-hardware co-design methodology to improve the NLP model inference efficiency.

Chapter 2 introduces the algorithmic optimizations, including the hardware-aware NAS, token pruning, and dynamic low-precision. Chapter 3 describes the hardware architecture for the attention accelerator in which a specialized top-k engine is designed to improve the parallelism. Chapter 4 presents the evaluations of the proposed algorithmic optimizations and hardware accelerator. Chapter 5 lists related work and makes comparisons. Chapter 6 summarizes and concludes the thesis.

This thesis makes the following contributions:

On the algorithm side: (i) **Low-cost Neural Architecture Search with a Large Design Space.** We leverage a weight-shared SuperTransformer to search for efficient models at a low cost. We enlarge the design space with *arbitrary encoder-decoder attention* to break the information bottleneck and *heterogeneous layer* to let different layers alter its capacity. (ii) **Hardware-Aware and Specialization.** We directly involve the hardware feedback in the model design loop to reduce NLP model latency for target hardware, instead of relying on proxy signals such as FLOPs.

For different hardware platforms, specialized models for low-latency inference are explored. (iii) **Design Insights.** Based on the search results, we reveal some design insights: Attending to multiple encoder layers is beneficial for the decoder; GPU prefers shallow and wide models, while ARM CPU prefers deep and thin ones.

On the hardware side, we design an attention accelerator that supports (i) **token pruning** technique to remove unimportant tokens according to the cumulative importance scores, reducing DRAM access and computation by up to 5.2×. (ii) **Dynamic low-precision** is also supported to trade a little more computation for less memory access. We change the bitwidth for different attention heads and layers according to the attention probability distribution, reducing DRAM access by 5×. (iii) A specialized high parallelism **top-k engine** with $O(n)$ time complexity is designed to support token pruning. (iv) The accelerator has **high-parallelism fully-pipelined datapath** for attention to reduce the overhead of its complex data movements and efficiently perform the proposed algorithmic optimizations.

Chapter 2

Methodology

2.1 Hardware-Aware Neural Architecture Search

An overview of our hardware-aware neural architecture search framework (HAT) is shown in Figure 2-1. We firstly construct a large design space with Arbitrary Encoder-Decoder Attention and Heterogeneous Layers. Then we train a weight-shared supernet – SuperTransformer by iteratively optimizing randomly sampled SubTransformers. It can provide a performance proxy for SubTransformers. For a given hardware platform, we collect a dataset of *(SubTransformer architecture, measured latency)* pairs, and train a latency predictor to provide fast and accurate latency feedback. Finally, we perform an evolutionary search with a latency constraint to find an efficient model specialized for the target hardware. The searched models are trained *from scratch* to get the final performance.

2.1.1 Design Space

We construct a large design space by breaking two conventions in the Transformer design: (1) All decoder layers only attend to the last encoder layer; (2) All the layers are identical.

Arbitrary Encoder-Decoder Attention. Different encoder layers extract features on different abstraction levels. Conventionally, all the decoder layers only at-

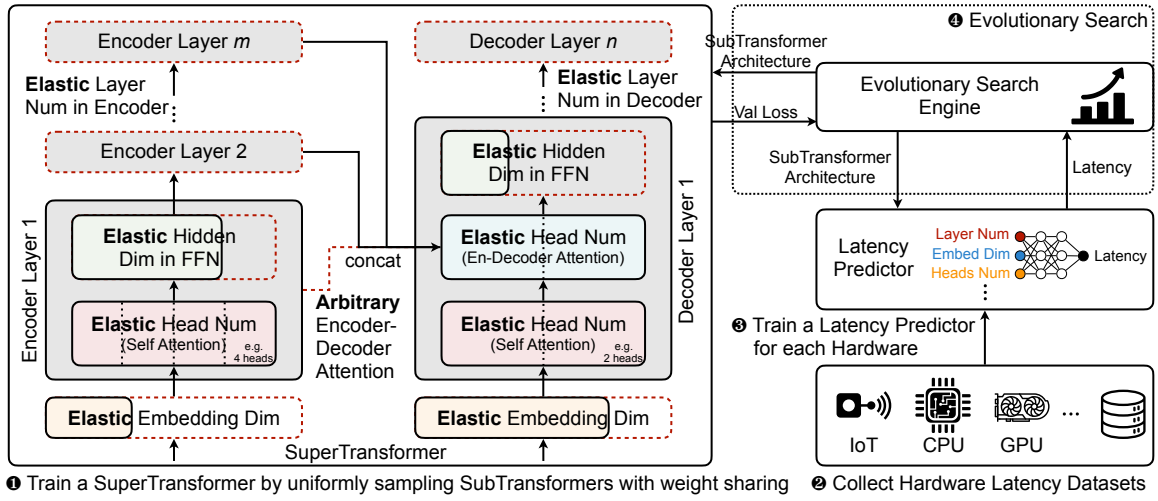

Figure 2-1: Hardware-Aware Transformer Overview.

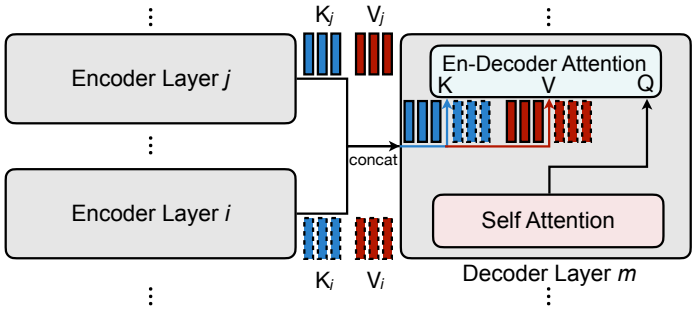

Figure 2-2: Arbitrary Encoder-Decoder Attention.

tend to the last encoder layer. It forms an *information bottleneck* that forces all the decoder layers to learn solely from the high abstraction level and ignore the low-level information. To break the bottleneck, we propose Arbitrary Encoder-Decoder Attention to learn the most suitable connections between the encoder and the decoder. Each decoder layer can choose *multiple* encoder layers to attend. The *key* and *value* vectors from encoder layers are concatenated in the *sentence length dimension* (Figure 2-2) and fed to the encoder-decoder cross attention module. The mechanism is efficient because it introduces no additional parameters. The latency overhead is also negligible. For example, with each decoder layer attending to two encoder layers, the latency of Transformer-Base on Nvidia TITAN Xp GPU barely increases by 0.4%. It improves the model capacity by allowing attention to different abstraction levels.

Heterogeneous Transformer Layers. Previous Transformers repeat one architecture for all layers. In HAT, instead, different layers are *heterogeneous*, with different numbers of heads, hidden dim, and embedding dim. In the attention layers, different heads are used to capture various dependencies. However, [89] shows that many heads are redundant. We thereby make attention head number *elastic* so that each attention module can decide its necessary number of heads.

In the FFN layer, the input features are cast to a higher dimension (hidden dim), followed by an activation layer. Traditionally, the hidden dim is set as $2\times$ or $4\times$ of the embedding dim, but this is sub-optimal since different layers need different capacities depending on the feature extraction difficulty. We hence make the hidden dim *elastic*.

Moreover, we also support *elastic* embedding dim of encoder and decoder, but it is consistent inside encoder/decoder. The number of encoder and decoder layers are also *elastic* to learn the proper level of feature encoding and decoding. Other design choices such as the length of Q, K, V vectors in attention modules can be naturally incorporated in our framework, which we leave for future work.

2.1.2 SuperTransformer

It is critical to have a large design space in order to find high-performance models. However, training all the models and comparing their BLEU scores is infeasible. Thus, we propose SuperTransformer, a supernet for *performance approximation*, which can judge the performance of a model without fully training it. The SuperTransformer is the largest model in the search space with *weight sharing* [66, 51, 7]. Every model in the search space (a SubTransformer) is a part of the SuperTransformer. All SubTransformers share the weights of their common parts. For elastic embedding dim, all SubTransformers share the front portion of the longest word embedding and corresponding FC layer weights. As in Figure 2-3, for elastic FFN hidden dim, the front part of the FC weights is shared. For the elastic head number in attention modules, the whole Q, K, V vectors (the lengths are fixed in our design space) are shared by dividing into *head_number* parts. Elastic layer numbers let all SubTransformers

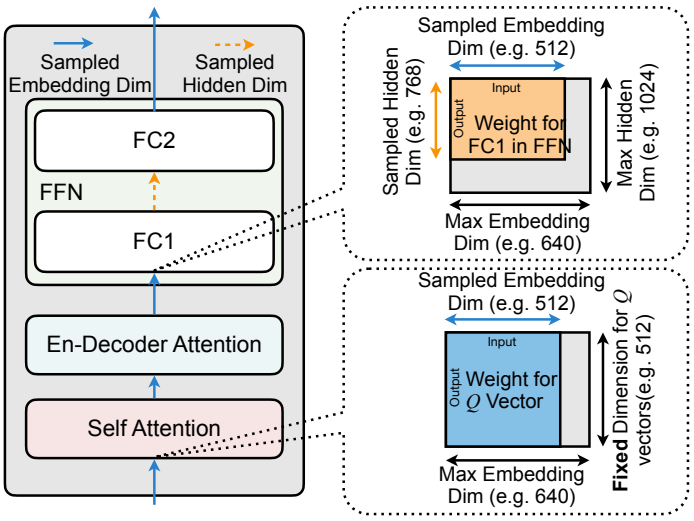

Figure 2-3: Weight Sharing of the SuperTransformer.

share the first several layers.

During the SuperTransformer training, all possible SubTransformers are *uniformly sampled*, and the corresponding weights are updated. In practice, the SuperTransformer only needs to be trained for the same steps as a baseline Transformer model, which is fast and low-cost. After training, we can get the performance proxy of sampled models in the design space by evaluating the corresponding SubTransformers on the validation set without training.

2.1.3 Evolutionary Search for SubTransformers

Given a latency requirement, we perform an evolutionary search to find a satisfactory SubTransformer. There are two ways to evaluate the hardware latency of a SubTransformer: (i) Online measurement in which we measure the model during the search process. (ii) Offline method, where we train a *latency predictor* to provide the latency. We apply the offline one here because it is *fast and accurate*. For the online method, a single sampled SubTransformer requires hundreds of inferences to get an accurate latency, which lasts for minutes and slows down the search process. For the offline method, we encode the architecture of a SubTransformer into a feature vector and predict its latency instantly with a multi-layer perceptron (MLP). Trained with thousands of real latency data points, the predictor yields high accuracy (Figure 2-4).

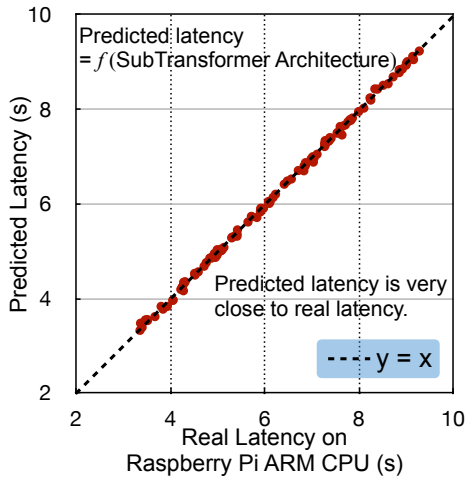

Figure 2-4: The latency predictor is very accurate, with an average prediction error (RMSE) of 0.1s.

Note that the predicted latency is only used in the search process, and we report *real measured latency* in the experiment section. Compared with deducing a closed-form latency model for each hardware, the latency predictor method is more general and faster.

We apply an evolutionary algorithm to conduct the search process. As in Figure 2-1, the search engine queries the latency predictor for SubTransformer latency and validates the loss on the validation set. The engine only adds SubTransformers with latency *smaller than* the hardware constraint to the population. We then train the searched models *from scratch* to obtain the final performance.

2.2 Algorithmic Optimizations in SpAtten

In order to reduce the bandwidth and computation requirements of the attention layers, the SpAtten accelerator supports several novel algorithmic optimizations. The overview of the proposed algorithmic optimizations is shown in Figure 2-5 with the example of one iteration in the generation stage.

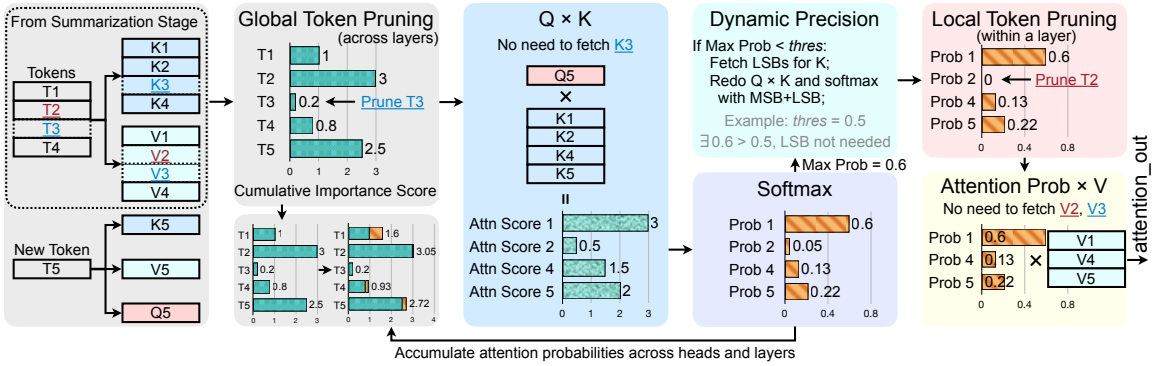

Figure 2-5: Algorithmic optimizations for the attention accelerator.

2.2.1 Token Pruning

Plenty of unessential tokens exist in human languages, which can be pruned away to boost efficiency as long as we can identify them. To achieve this, we propose token pruning to assess the token importance based on the attention probabilities, and remove trivial ones. The corresponding memory fetch and attention computation of the pruned tokens are also skipped. Token pruning has two levels: ***global*** and ***local***.

In ***global*** token pruning, tokens to be pruned are determined by an array of ***cumulative importance score***, one for each token. The scores are obtained by accumulating ***attention probabilities*** in multiple rounds of attention computation. 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 1-4). The *block_in* of the first block are directly from input tokens. For latter blocks, V vectors are still largely determined by the input tokens because of the 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 the example in Figure 2-6, many tokens attend to the word 'fun', implying its high impact. In the summarization stage, the scores are accumulated by L_0 (number of query vectors, see Algorithm 1) times in one attention head. In the generation stage, the scores are accumulated for only once in one attention head because the query is a single vector. In BERT model, we accumulate

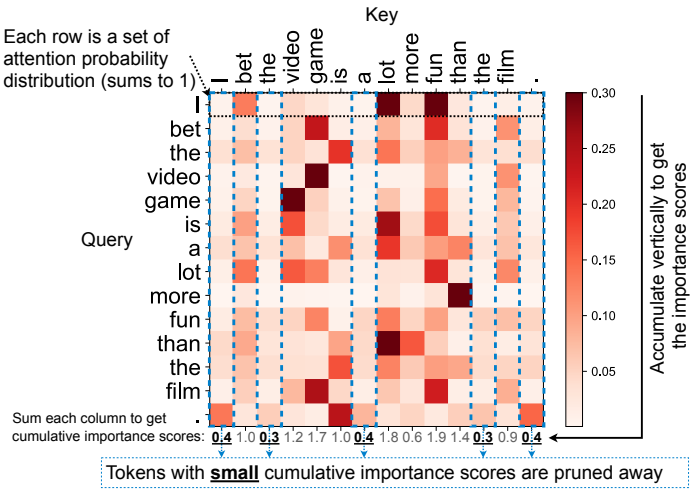

Figure 2-6: The attention probabilities for BERT are summed along the columns, obtaining the importance scores. Tokens with small importance scores are pruned.

importance scores in former heads and layers, and apply token pruning to the latter ones. In GPT-2, we further accumulate importance scores across generation iterations because intuitively, the unimportant tokens for one token generation should be unimportant for another as well.

With the cumulative importance scores, we can remove a pre-defined pruning ratio of tokens. Once a token is pruned in the global token pruning, the Q, K, and V of it will *never* be used in all the following attention layers, thus being *global*. For the example in Figure 2-5, we already have five cumulative importance scores for five tokens. Assuming the pruning ratio as 20%, T3 has the smallest score and thus will be removed. In this way, neither K3 in Q $\times$ K needs to be fetched, nor V3 in attention_prob $\times$ V. Note that after getting attention_out of this layer, we will accumulate the attention probabilities from Softmax to current scores in preparation for global token pruning in the next attention head or layer.

Local token pruning happens after Softmax and the tokens to be pruned are decided *only with attention probabilities of the current layer.* For the pruned tokens, the corresponding V vectors will not be fetched for attention_prob $\times$ V. Naturally, even for tokens that are important for the NLP model as a whole, they may not be critical for some of the attention heads or layers. In Figure 2-5, the attention probability for T2 is the smallest. Thus we reset Prob2 and will not fetch V2 for computation. For

Figure 4-15 example, although 'has' is cumulatively important, it is not important for the 9^{th} layer.

In summary, *global* token pruning removes Q, K, and V of the pruned tokens for the current and all the following attention heads and layers. *Local* token pruning removes V of the pruned tokens for current attention only. In terms of pruning ratios, we apply an adaptive process. For long sentences, we use a higher pruning ratio since the long ones contain more redundancy than the short ones. Note that token pruning can reduce the computation and memory access of both attention in SpAtten, and FC layers outside attention. On eight GPT-2 benchmarks, token pruning can achieve 5.2× reduction of DRAM access with less than 3% relative performance loss. For instance, on the Wikitext-2 LM task with GPT-2-Medium, we can prune away 75% of tokens with only 0.7 perplexity degradation.

2.2.2 Dynamic Low-Precision

To further reduce the memory access of attention layers, we propose *dynamic low-precision*. Instead of using floating-point Q, K, and V, we fetch fixed-point inputs from DRAM and perform fixed-point computations on-chip. We can apply relatively aggressive low-precision since Softmax computation is more quantization friendly.

The Softmax for attention probabilities are computed as: $p_i = e^{s_i} / \sum_{j=0}^{L_1-1} e^{s_j}$, where L_1 is the number of K vectors, p is attention probability and s is attention score. If we apply low-precision on Q and K, then we can consider that as adding a small error Δs to the attention score s. Here we examine the influence of error Δs to the output attention probabilities by computing the softmax derivative:

$$\begin{aligned}
\text{If } i = j: \frac{\partial p_i}{\partial s_j} &= \frac{\partial \frac{e^{s_i}}{\sum_{i=0}^{L_1-1} e^{s_i}}}{\partial s_i} = \frac{e^{s_i} \sum_{i=0}^{L_1-1} e^{s_i} - e^{s_i} e^{s_i}}{(\sum_{j=0}^{L_1-1} e^{s_j})^2} \\
&= \frac{e^{s_i}(\sum_{j=0}^{L_1-1} e^{s_j} - e^{s_i})}{(\sum_{j=0}^{L_1-1} e^{s_j})^2} = p_i \cdot (1 - p_i) \qquad (2.1) \\
\text{If } i \neq j: \frac{\partial p_i}{\partial s_j} &= \frac{\partial \frac{e^{s_i}}{\sum_{i=0}^{L_1-1} e^{s_i}}}{\partial s_j} = \frac{0 - e^{s_j} e^{s_i}}{(\sum_{j=0}^{L_1-1} e^{s_j})^2} = -p_i \cdot p_j
\end{aligned}$$

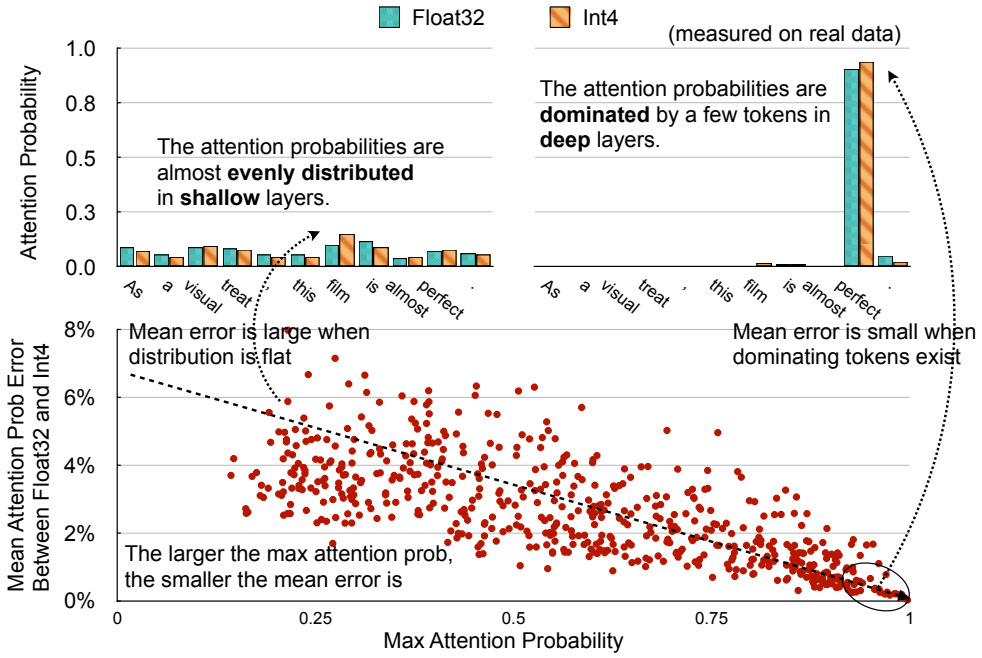

Figure 2-7: When the attention probability is evenly distributed (left), the quantization error is large. When there exists a dominant probability (right), the quantization error is small.

Without loss of generality, we assume s_0 changes by $\Delta s_0 > 0$. Then we compute the sum of absolute errors of all output p with Equation 2.1.

$$\begin{aligned} error &= |\Delta s_0 \cdot p_0 \cdot (1 - p_0)| + \sum_{i=1}^{L_1 - 1} |-\Delta s_0 \cdot p_0 \cdot p_i| \\ &= \Delta s_0 \cdot p_0 \cdot (1 - p_0) + \sum_{i=0}^{L_1 - 1} \Delta s_0 \cdot p_0 \cdot p_i - \Delta s_0 \cdot p_0^2 \\ &= \Delta s_0 \cdot (2 \cdot p_0 \cdot (1 - p_0)) < \Delta s_0 \end{aligned} \tag{2.2}$$

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

On top of low-precision, we further propose dynamic low-precision to adjust the input bitwidth for different attention heads and layers. We find an interesting phenomenon as in Figure 2-7 that if the attention probability distribution is dominated by a few tokens, then the quantization error is smaller; if the distribution is flat, then the error is larger. Intuitively, the front case indicates there existing highly critical tokens, and thus quantization cannot easily undermine its influence. Theoretically, as

in Equation 2.2, errors are in proportional to $p(1-p)$, which is a quadratic function with zero and one as roots. If probability dominators exist, then p is likely close to zero or one; thus errors are smaller. For flat distributions, p is more likely to be in between, resulting in larger errors.

Therefore, for robust layers with probability dominators, the bitwidth can be small, while other sensitive layers should have more bits. To achieve that, we firstly only fetch the MSBs of inputs, 4 bits for instance, 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. For the example in Figure 2-7, the max probability is 0.15 for left Int4 and 0.94 for right Int4. Assume $threshold = 0.8$; the left attention layer requires LSBs while the right one does not need.

In consideration of the fast interaction between SpAtten and hardware platforms that process FC parts, we conduct linear symmetric quantization, which is much faster than K-Means quantization. Different bitwidth representations share the same MSBs so that we only fetch the LSBs when need more bits, thus reducing DRAM access. Dynamic low-precision trades more computation for less memory access and can effectively alleviate the memory-bound issue in the generation stage. Since the summarization stage is computation bounded, we do not need dynamic precision in that part.

Chapter 3

Hardware Architecture

3.1 Overview

The overview of our proposed attention accelerator, SpAtten, is shown in Figure 3-1. The whole architecture is fully pipelined with a specialized module for each operation in attention and thus reducing the data movement overhead. A High Bandwidth Memory (HBM) is used as the off-chip memory to store attention inputs. We also parallelize the computation to increase the throughput and match it with the HBM bandwidth. The architecture also supports the proposed algorithmic optimizations. Specifically, token pruning reduces the computation and memory fetch, but also incurs random access. Thus, we utilize a crossbar to process the addresses, keeping every channel of the HBM busy and increasing the bandwidth utilization. Furthermore, a high-parallelism top-k engine (Figure 3-3) is designed to support the on-the-fly pruning of tokens with $O(n)$ time complexity. To support dynamic low-precision, we also implement an on-chip bitwidth converter to handle the splits of fetched bits and the concatenations of MSBs and LSBs.

Our architecture processes attention head by head and query by query, i.e., one single query of one head is fed to the pipeline at a time. In the summarization stage, the remaining K and V after the global token pruning for one head are firstly fetched to the on-chip SRAM for *data reuse* between multiple queries in this head. In the generation stage, the Q is a single vector, so there is no reuse of K and V, and no need

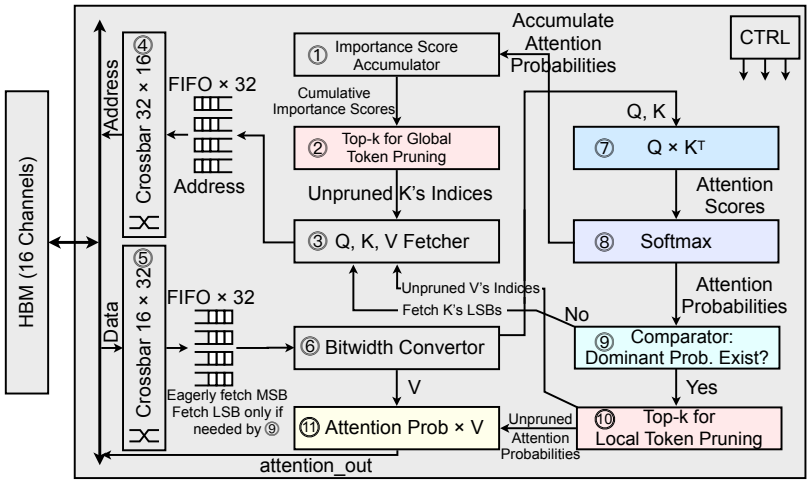

Figure 3-1: SpAtten Architecture Overview. All of the modules are fully pipelined to achieve high performance.

to store them. For each query's attention computation, we first use a top-k engine to process the cumulative importance scores and get the indices for k most important K. A data fetcher then computes the K's 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 3-2) to get the attention scores. Since we support different dimensions D of the query vector, the reconfigurable reduction tree can output attention scores at different levels. A Softmax module (Figure 3-5) then processes the attention scores to get attention probabilities and send them to the dynamic precision determination module (Figure 3-5) to decide whether LSBs are required. The probabilities are also sent to the importance score accumulator to perform accumulations. After that, the local token pruning top-k engine gets the probabilities, computes k most locally important tokens and sends indices of corresponding V to the data fetcher. Finally, the remaining probabilities are multiplied with fetched V, resulting in attention outputs.

3.2 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 inputs are interleaved in different channels. We

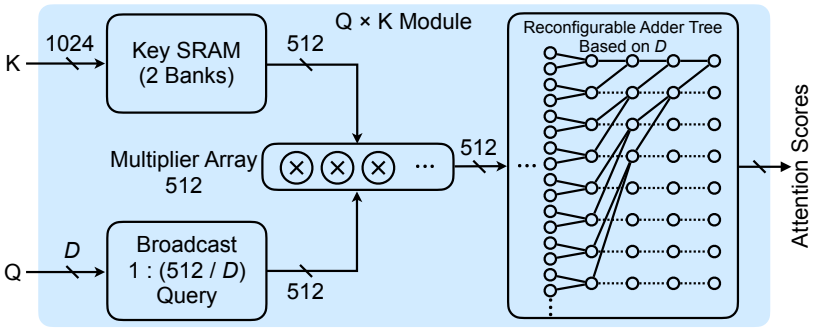

Figure 3-2: Query-Key multiplication module equipped with a reconfigurable adder tree.

use a 32-to-16 crossbar to route these read requests to the correct channels. The master side of the crossbar is larger than the slave side in order to reduce channel conflicts.

In order to support dynamic low-precision and also avoid complex logic overheads, we enforce on-chip SRAMs and multipliers to have 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.

3.3 Query-Key Multiplication Module

The query-key multiplication module (Figure 3-2) 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 Key SRAM, multiplied by Q using the multiplier array and then fed to an adder tree. The adder tree then computes the importance scores by reducing all multiplied results $s_i = \sum_j \mathrm{K}_{ij} * \mathrm{Q}_j$.

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, which

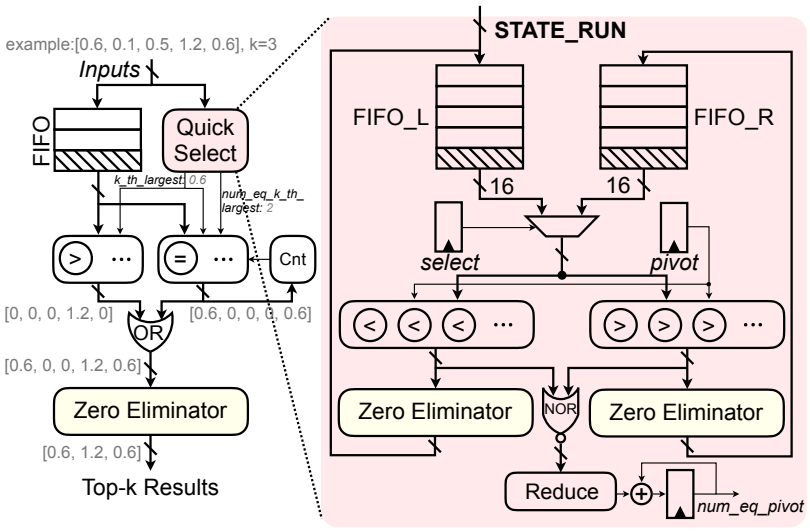

Figure 3-3: High-parallelism top-k engine. The comparators and zero eliminators can process 16 elements per cycle, increasing the throughput of token selection.

are capable of producing multiple results s_i.

3.4 Top-k Engine

To support both global and local token pruning, we need to find the top k elements of the cumulative importance scores and attention probabilities, respectively. A naïve solution is to use a sorter to sort the original array and directly output the first k elements from the sorted array. However, the sorter based solution is costly in time ($O(N*logN)$ time complexity) and space ($O(N*log^2N)$ space sorting network). It is also inefficient in our case because the top-k module's outputs will be used to generate the addresses to fetch keys and values from DRAM. The sorted array loses the original order of keys and values, thus making the data fetching completely random.

Instead of sorting the array, we design a quick-select engine to find the k^{th} largest element and use it as a threshold to filter the input array. It can achieve lower time complexity ($O(n)$ on average) and keep the original order of inputs.

The quick-select algorithm leverages a randomly chosen pivot to partition the input array into two parts: one contains all elements smaller than the pivot, and another contains all elements greater than it. The classical implementation of the

Algorithm 2: 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 ≤ 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              */
    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 3-3)
/* items smaller than pivot → FIFO_L                                    */
/* items larger than pivot → FIFO_R                                     */
num_eq_pivot = 0;
for i = 0 to size(FIFO[select]) 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;
```

partition process on CPUs – Hoare partition scheme uses two pointers to scan the input array and swaps the elements when necessary. This method involves sequential operations that are difficult to parallelize and cannot provide enough throughput. For a typical case of finding the 256^{th} largest element from an array of 1024 elements, our simulation shows that the algorithm needs to scan each element for three times on average. Due to the low parallelism, this method requires around 3000 cycles, resulting in a 0.33 elements/cycle throughput, far below the eight elements/cycle throughput of the multiplication module.

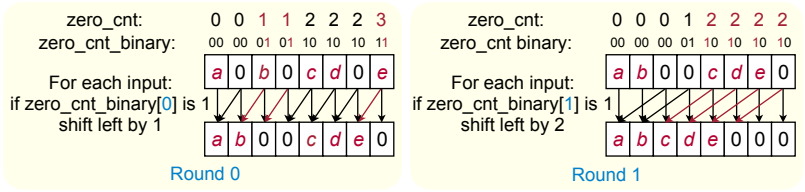

Figure 3-4: Zero Eliminator.

To obtain higher parallelism, we turn to a specialized top-k engine (Figure 3-3). The quick-select module of the top-k engine contains two FIFOs to store the partitioned arrays, FIFO_L and FIFO_R. For an array to be partitioned, it will be fed into two comparator arrays and compared with the pivot. The left comparator array only preserves elements smaller than the pivot, while the right one only preserves larger elements. The other elements will be set to zeros and then be eliminated by a zero eliminator module. The quick-select module runs iteratively until the k^{th} largest element is found. Algorithm 2 describes the control logic of our quick-select module. Once getting the k^{th} largest element, the top-k engine will use it to filter the input array, which is buffered in another FIFO (Figure 3-3 left) before the quick-select process. The filtered out elements will also be eliminated by another zero eliminator to finally get the top-k elements.

The comparator array and the zero eliminator based top-k engine can be easily scaled up to provide high throughput because of their high-parallelism nature. In SpAtten, we apply 16 comparators in each array to make sure it is not the bottleneck of the whole pipeline.

3.5 Zero Eliminator

The zero eliminator 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 or not is determined by the n^{th} bit of its corresponding *zero_cnt*. Figure 3-4 illustrates an example of the zero eliminator.

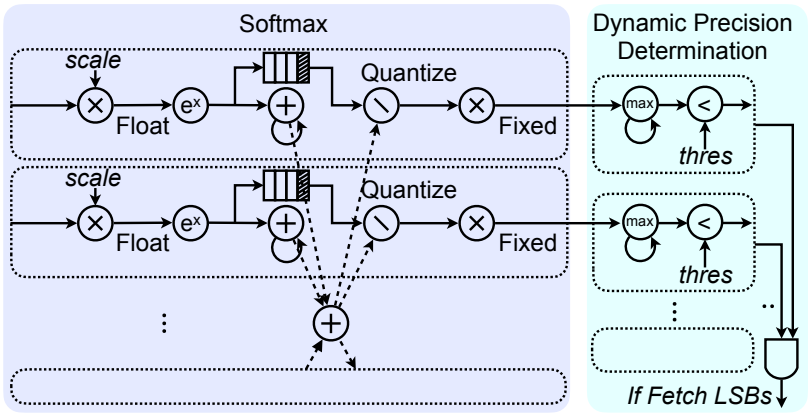

Figure 3-5: Softmax and Dynamic Precision Determination Modules.

3.6 Softmax and Dynamic Low-Precision

We use floating-point units to calculate the Softmax results. The fixed-point attention scores 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 $e^{x_i} / \sum_j e^{x_j}$. 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 dynamic precision 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.

3.7 Attention Prob - Value Multiplication Module

The attention prob-value multiplication unit takes the attention probabilities as inputs, multiply them with the Vs and then accumulate to get attention outputs $A_j = \sum_i \text{attention\_prob}_i * V_{ij}$. 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.

Chapter 4

Evaluation

4.1 Hardware-Aware Neural Architecture Search

4.1.1 Evaluation Setups

Datasets. To evaluate HAT, we conduct experiments on four machine translation tasks: WMT'14 En-De, WMT'14 En-Fr, WMT'19 En-De, and IWSLT'14 De-En, consisting of 4.5M, 36.3M, 43.0M, and 160K pairs of training sentences, respectively. For WMT'14 En-De, we apply 32K source-target BPE vocabulary, validate on newstest2013 and test on newstest2014, replicating [98]; For WMT'14 En-Fr, we use 40K source-target BPE vocabulary, validate on newstest2012&2013, and test on newstest2014, the same as [18]. For WMT'19 En-De, we use 49.6K source-target BPE vocabulary, validate on newstest2017, and test on newstest2018, replicating [39]. We adopt 10K joint BPE vocabulary in lower case for IWSLT'14 De-En [20].

Baselines. Our baseline models are Transformer [87] and Levenshtein Transformer [22]. We use the implementation in [63] for both of them. The Evolved Transformer [79] and Lite Transformer [100] are also baselines.

Evaluation Metrics. For evaluation, we use beam four and length penalty 0.6 for WMT, and beam five for IWSLT [87]. All BLEUs are calculated with case-sensitive

tokenization[1], and we also apply the compound splitting BLEU[2] for WMT, the same as [87]. We test the model with the lowest validation set loss for WMT, and the last ten checkpoints averaged for IWSLT.

We test the latency of the models by measuring translation from a source sentence to a target sentence with the same length. The length is the average output length on the test set – 30 for WMT and 23 for IWSLT. For each model, we measure the latency for 300 times, remove the fastest and slowest 10%, and then take the average of the rest 80%. We conduct experiments on three representative hardware platforms: Raspberry Pi-4 with an ARM Cortex-A72 CPU, Intel Xeon E5-2640 CPU, and Nvidia TITAN Xp GPU.

SuperTransformer Setups. The SuperTransformer for WMT has the following design space: [512, 640] for embedding dim, [1024, 2048, 3072] for hidden dim, [4, 8] for the head number in all attention modules, [1, 2, 3, 4, 5, 6] for decoder layer number. Due to the decoder auto-regression mechanism, it takes the lion's share of the latency, and encoder only accounts for less than 5% of the latency. Therefore, we set the encoder layer number fixed as 6. For arbitrary encoder-decoder attention, each decoder can choose to attend to the last one, two, or three encoder layers. The SuperTransformer design space for IWSLT is the same as WMT except for [2048, 1024, 512] for hidden dim and [4, 2] for head number. We set the Q, K, V vector dim fixed as 512. The design space contains around 10^{15} possible SubTransformers and covers a wide range of model size and latency (largest = 6 $\times$ smallest). We train the SuperTransformers of WMT for 40K steps and 50K steps for IWSLT.

Hardware-Aware Evolutionary Search Setups. The input of the latency predictor is a feature vector of SubTransformer architecture with ten elements: layer number, embedding dim, average hidden dim, average self-attention heads, of both encoder and decoder; plus average encoder-decoder attention heads, and the average number of encoder layers each decoder layer attends to. A dataset of 2000 (SubTrans-

[1]https://github.com/moses-smt/mosesdecoder
[2]https://github.com/tensorflow/tensor2tensor

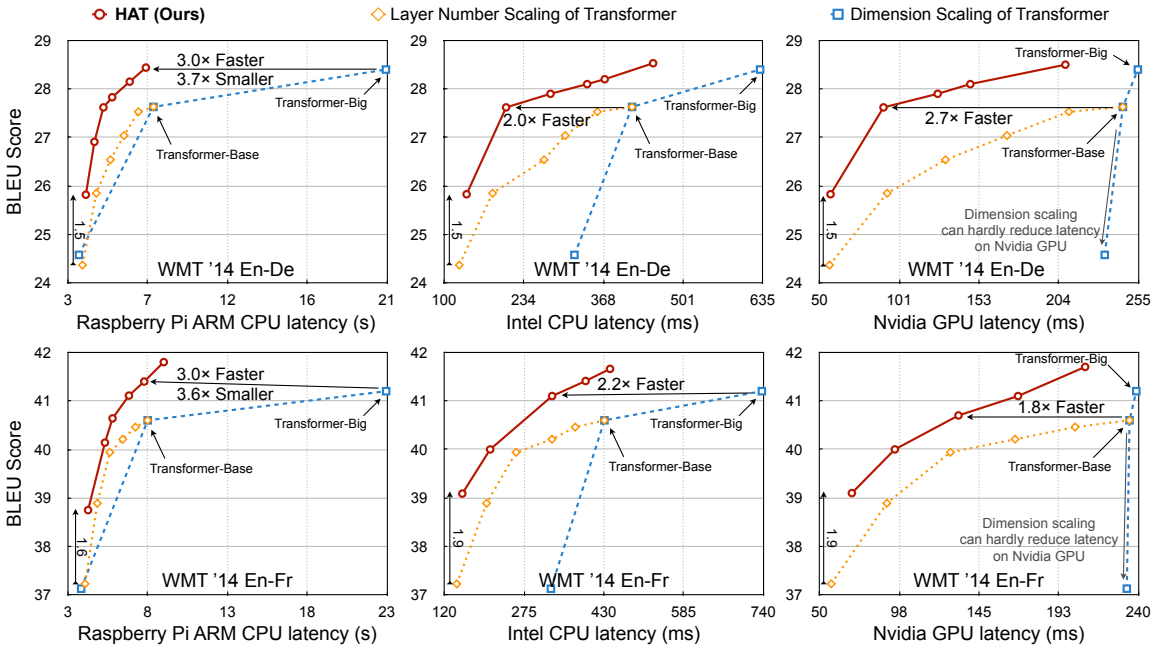

Figure 4-1: Inference latency and BLEU trade-offs of WMT'14 En-De and WMT'14 En-Fr on three hardware platforms.

former architecture, measured latency) samples for each hardware is collected, and split into train:valid:test = 8:1:1. We normalize the features and latencies, and train a three-layer MLP with 400 hidden dim and ReLU activation. We choose three-layer because it is more accurate than the one-layer model, and over three layers do not improve accuracy anymore. With the predictor, we conduct an evolutionary search for 30 iterations in the SuperTransformer, with the population as 125, parents population as 25, mutation population as 50 with 0.3 mutation probability, and crossover population as 50.

Training Settings. Our training settings are in line with [98] and [100]. For WMT, we train for 40K steps with Adam optimizer and a cosine learning rate (LR) scheduler [46, 54], where the LR is linearly warmed up from 10^{-7} to 10^{-3}, and then cosine annealed. For IWSLT, we train for 50K steps with an inverse square root LR scheduler. The baseline Transformers are trained with the *same settings* as the searched SubTransformers for fair comparisons.

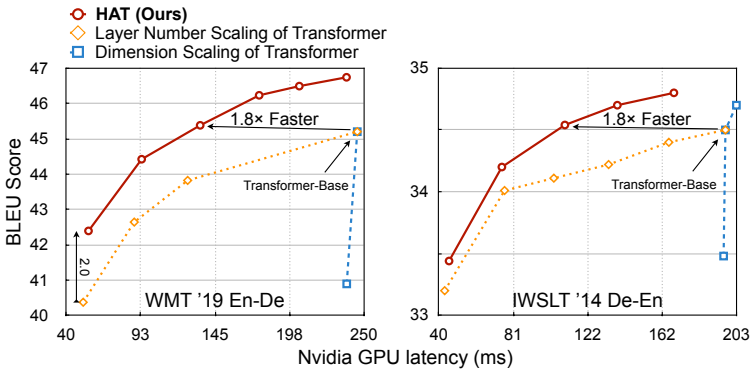

Figure 4-2: Inference latency and BLEU trade-offs of WMT'19 En-De and IWSLT'14 De-En on Nvidia TITAN Xp GPU.

4.1.2 Performance Comparisons

In Figure 4-1 and 4-2, we compare HAT with Transformer baselines on four tasks. The embedding dims are 512 and 1024 for the Transformer-Base and Big, respectively. The hidden dims are 4× and 2× of the embedding dim for WMT and IWSLT. The IWSLT models are smaller to prevent overfitting [98]. We obtain a series of baseline models with layer number scaling (yellow) and dimension scaling (blue). We set different latency constraints on three hardware to get a series of HAT models. HAT consistently outperforms baselines with a large gap under different latency constraints. On ARM CPU, HAT is 3× faster and 3.7× smaller than Transformer-Big with the same BLEU score. On Intel CPU, HAT achieves over 2× speedup. On Nvidia GPU, the blue dash line is nearly *vertical*, indicating that dimension scaling can hardly reduce the latency. In this case, HAT can still find models with lower latency and higher performance than Transformer baselines. We show the specific latency, BLEU and SacreBLEU [68] in Table 4.1.

We further compare various aspects of HAT with Transformer [87] and the Evolved Transformer [79] in Table 4.2. The latency is measured on the Raspberry Pi ARM CPU. On three translation tasks, HAT achieves up to 1.6×, 3×, and 3.4× speedup with up to 1.4×, 3.7×, and 4× smaller size than baselines. We report FLOPs for translating a 23-token sentence for IWSLT and 30 for WMT. We also show the search cost comparisons. The cost of HAT is the overall cost for training both the Super-

Task	WMT'14 En-De			WMT'14 En-Fr		
Hardware	Latency	BLEU	SacreBLEU	Latency	BLEU	SacreBLEU
Raspberry Pi ARM Cortex-A72 CPU	3.5s	25.8	25.6	4.3s	38.8	36.0
	4.0s	26.9	26.6	5.3s	40.1	37.3
	4.5s	27.6	27.1	5.8s	40.6	37.8
	5.0s	27.8	27.2	6.9s	41.1	38.3
	6.0s	28.2	27.6	7.8s	41.4	38.5
	6.9s	28.4	27.8	9.1s	41.8	38.9
Intel Xeon E5-2640 CPU	137.9ms	25.8	25.6	154.7ms	39.1	36.3
	204.2ms	27.6	27.1	208.8ms	40.0	37.2
	278.7ms	27.9	27.3	329.4ms	41.1	38.2
	340.2ms	28.1	27.5	394.5ms	41.4	38.5
	369.6ms	28.2	27.6	442.0ms	41.7	38.8
	450.9ms	28.5	27.9	–	–	–
Nvidia TITAN Xp GPU	57.1ms	25.8	25.6	69.3ms	39.1	36.3
	91.2ms	27.6	27.1	94.9ms	40.0	37.2
	126.0ms	27.9	27.3	132.9ms	40.7	37.8
	146.7ms	28.1	27.5	168.3ms	41.1	38.3
	208.1ms	28.5	27.8	208.3ms	41.7	38.8
Task	WMT'19 En-De			IWSLT'14 De-En		
Hardware	Latency	BLEU	SacreBLEU	Latency	BLEU	SacreBLEU
Nvidia TITAN Xp GPU	55.7ms	42.4	41.9	45.6ms	33.4	32.5
	93.2ms	44.4	43.9	74.5ms	34.2	33.3
	134.5ms	45.4	44.7	109.0ms	34.5	33.6
	176.1ms	46.2	45.6	137.8ms	34.7	33.8
	204.5ms	46.5	45.7	168.8ms	34.8	33.9
	237.8ms	46.7	46.0	–	–	–

Table 4.1: Specific latency numbers, BLEU and SacreBLEU scores for the searched HAT models in Figure 4-1 and 4-2.

Transformer and the searched SubTransformer. The training time is for one Nvidia V100 GPU, and the cloud computing cost is based on AWS in different modes: 'pre-emptable' mode ($0.74/h) is cheaper than 'on-demand' mode ($2.48/h) [81]. HAT is highly affordable since the total GPU-hour is over 12000× smaller than the Evolved Transformer, and is even smaller than Transformer-Big by virtue of the small size and FLOPs of the searched HAT models.

In Table 4.3, we compare HAT with other latest models. We scale down all models to have similar BLEU scores with Levenshtein for fair comparisons. For Levenshtein,

		Latency	#Params	GFLOPs	BLEU	GPU Hours	CO_2 (lbs)	Cloud Computing Cost
IWSLT'14 De-En	Transformer	3.3s	32M	1.5	34.5	2	5	\$12 - \$40
	HAT (Ours)	**2.1s**	**23M**	**1.1**	**34.5**	4	9	\$24 - \$80
WMT'14 En-Fr	Transformer	23.2s	176M	10.6	41.2	240	68	\$178 - \$595
	Evolved Trans.	20.9s	175M	10.8	41.3	2,192,000	626,000	\$1.6M - \$5.5M
	HAT (Ours)	**7.8s**	**48M**	**3.4**	**41.4**	216	61	\$159 - \$534
	HAT (Ours)	9.1s	57M	3.9	**41.8**	224	64	\$166 - \$555
WMT'14 En-De	Transformer	20.5s	176M	10.6	28.4	184	52	\$136 - \$456
	Evolved Trans.	7.6s	47M	2.9	28.2	2,192,000	626,000	\$1.6M - \$5.5M
	HAT (Ours)	**6.0s**	**44M**	**2.7**	28.2	184	52	\$136 - \$456
	HAT (Ours)	6.9s	48M	3.0	**28.4**	200	57	\$147 - \$495

Table 4.2: Comparisons of latency, model size, FLOPs, BLEU score and training cost in terms of CO_2 emissions (lbs) and cloud computing cost (USD) for Transformer [87], Evolved Transformer [79] and HAT.

	Latency	BLEU
Transformer [87]	4.3s	25.85
Levenshtein [22]	6.5s	25.20
Evolved Transformer [79]	3.7s	25.40
Lite Transformer [100]	3.4s	25.79
HAT (Ours)	**3.4s**	**25.92**

Table 4.3: Raspberry Pi ARM CPU latency and BLEU comparisons with different models on WMT'14 En-De. HAT has the lowest latency with the highest BLEU.

we adopt the average iteration time of 2.88 for decoding [22], without limiting the length of the output sentence (12 tokens after decoding). HAT runs 1.3× faster than Transformer with higher BLEU; 1.9× faster than Levenshtein with 0.7 higher BLEU. Under similar latency, HAT also outperforms Lite Transformer. These results demonstrate HAT's effectiveness in low latency scenarios. Our framework can also be adopted to speedup those models.

4.1.3 Analysis

Design Insights. For all HAT WMT models in Figure 4-1, 10% of all decoder layers attend to three encoder layers, 40% attend to two encoder layers. That demonstrates

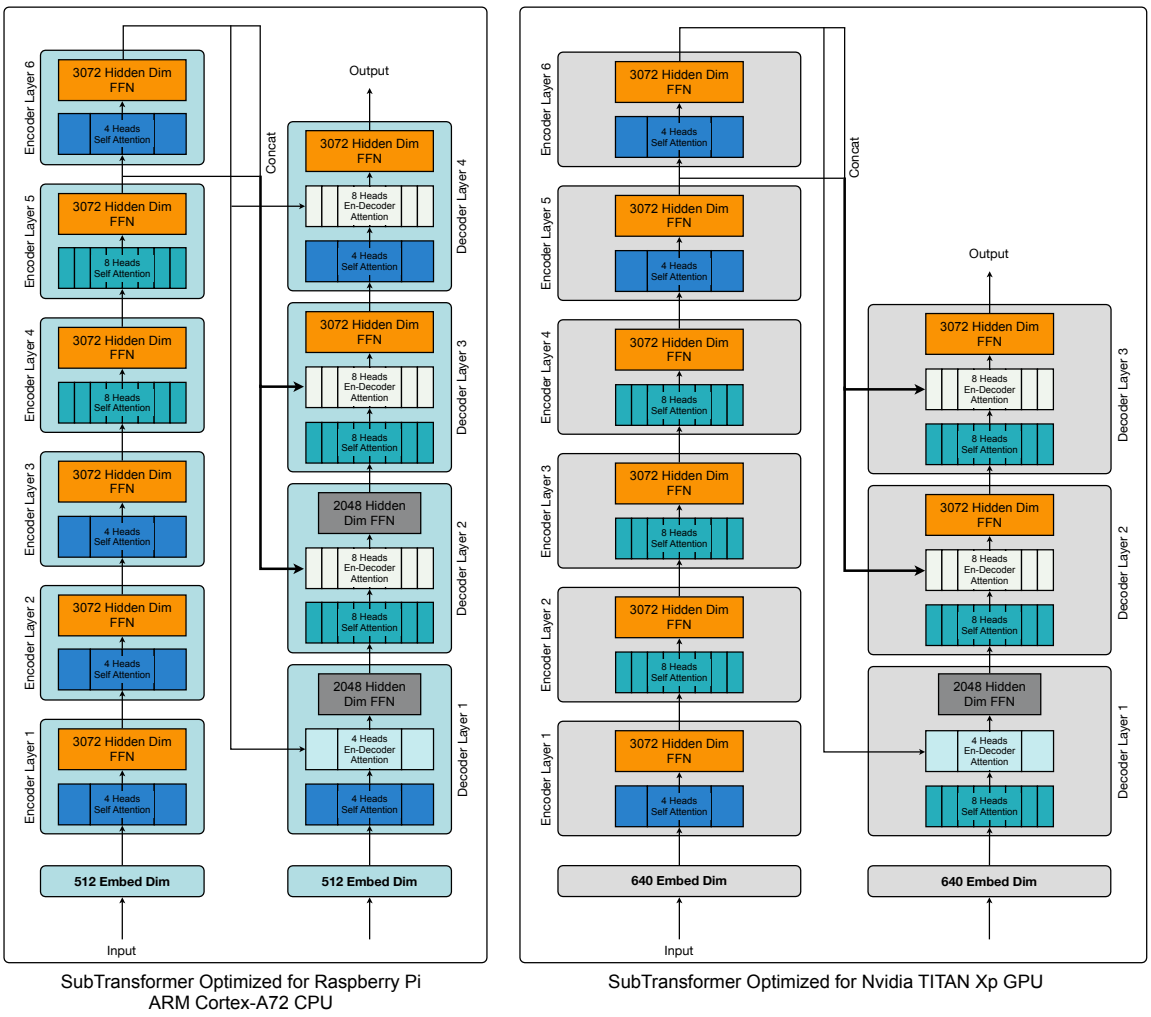

Figure 4-3: SubTransformers optimized for Raspberry Pi ARM CPU and Nvidia GPU on the WMT'14 En-De task are different. The CPU model has BLEU 28.10, and the GPU model has BLEU 28.15.

the necessity of arbitrary encoder-decoder attentions.

We show the HAT models searched for Raspberry Pi ARM Cortex-A72 CPU and Nvidia TITAN Xp GPU in Figure 4-3. The searched model for Raspberry Pi is deep and thin, while that for GPU is shallow and wide. The BLEU scores of the two models are similar: 28.10 for Raspberry Pi CPU, and 28.15 for Nvidia GPU. The phenomenon echos with our latency profiling in Section 1 (Figure 1-5) as the GPU latency is insensitive to embedding and hidden dim, but Raspberry Pi is highly sensitive. It guides manual designs: on GPU, we can reduce the layer number and increase the dimension to reduce latency and keep high performance.

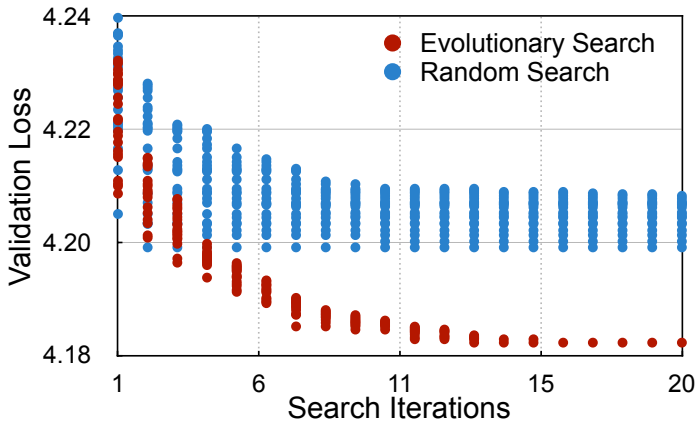

Figure 4-4: Evolutionary search can find better SubTransformers than the random search.

	SubTransformer	Latency	#Params	BLEU
WMT'14 En-De	Largest	10.1s	71M	28.1
	Searched HAT	**6.9s**	**48M**	**28.4**
WMT'14 En-Fr	Largest	10.1s	71M	41.4
	Searched HAT	**9.1s**	**57M**	**41.8**

Table 4.4: Comparisons between searched HAT models with the largest SubTransformer in the design space. Larger models do not necessarily have better performance. HAT can find SubTransformers with lower latency, smaller size, and higher BLEU.

Ablation Study. We compare the HAT searched models with the largest SubTransformers in the design space. HAT achieves higher BLEU with 1.5× lower latency and 1.5× smaller size compared with the largest SubTransformer (Table 4.4). This interesting phenomenon suggests that larger models do not always provide better performance. We also compare the evolutionary search with the random search in Figure 4-4. The evolutionary search can find models with lower losses than the random search.

SubTransformer Performance Proxy. All SubTransformers inside the SuperTransformer are *uniformly sampled* and thus *equally trained*, so the performance order is well-preserved during training. We conduct experiments to show the effectiveness of the SubTransformer performance proxy as in Figure 4-5 and Table 4.5. The BLEUs of SubTransformers with inherited weights and weights trained from

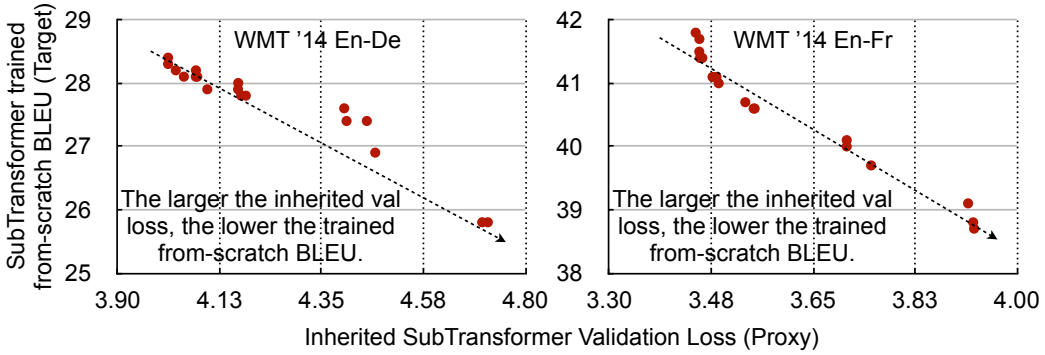

Figure 4-5: The validation loss of SubTransformers is a good performance proxy for BLEU of SubTransformers trained from scratch. The lower the validation loss, the higher the BLEU score. Using this proxy, we directly get feedback without paying for extra training costs.

WMT'14 En-De			WMT'14 En-Fr		
Inherited Val Loss	Inherited BLEU	From-Scratch BLEU	Inherited Val Loss	Inherited BLEU	From-Scratch BLEU
4.71	24.9	25.8	3.92	37.4	38.8
4.40	25.8	27.6	3.71	38.0	40.0
4.07	26.3	28.1	3.48	39.5	41.1
4.02	26.7	28.2	3.46	39.6	41.4
4.01	26.9	28.4	3.45	39.7	41.7

Table 4.5: The performance of SubTransformers with inherited weights is close to those trained from scratch, and have the same relative performance order.

scratch are very close. More importantly, the validation losses of SubTransformers with inherited weights are negatively related to the BLEU scores of SubTransformers trained from scratch. Therefore, we can rely on the validation loss proxy to search for high-performance model architecture, significantly reducing the search cost.

Low Search Cost. As shown in Table 4.2 and Figure 4-6, the search cost of HAT is 12,041× lower than the Evolved Transformer. Although both are using Evolutionary Search, the key difference is that the Evolved Transformer needs to train all *individual models separately* and sort their *final performance* to pick the top ones; on the contrary, HAT trains *all models together* inside SuperTransformer and sorts their *performance proxy* to pick the top ones. The superior performance of HAT proves

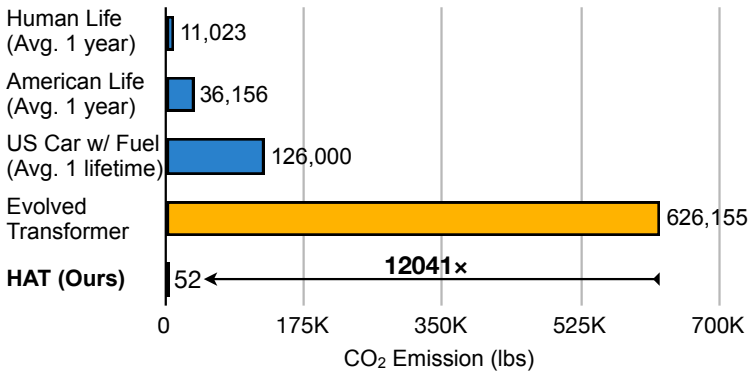

Figure 4-6: The search cost in pounds of CO_2 emission. HAT reduces the cost by four orders of magnitude than the Evolved Transformer [79].

Task	From-Scratch 40K	Inherit-Finetune 10K
WMT'14 En-Fr	41.5	41.7
	40.0	40.2
WMT'14 En-De	28.0	28.0
	27.5	27.4

Table 4.6: SubTransformer with inherited and finetuned weights can achieve similar or better performance than the same SubTransformer trained from scratch. Training steps are saved by 4×.

that the performance proxy is accurate enough to find good models.

Finetuning Inherited SubTransformers. In section 4.1.2, we trained each searched SubTransformer from scratch in order to conduct fair comparisons with baselines. In practice, we can also directly finetune the SubTransformers with the inherited weights from the SuperTransformer to further reduce the training cost. With 10K finetuning steps (1/4 of from-scratch training), the inherited SubTransformers can achieve similar or better performance than trained from-scratch ones (Table 4.6). In this way, the training cost for a model under a new hardware constraint can be further reduced by 4×, as the SuperTransformer training cost is amortizable among all the searched SubTransformer models.

Quantization Friendly. HAT is orthogonal to other model compression techniques such as quantization. We apply K-means quantization to HAT and further reduce the

	BLEU	Model Size	Reduction
Transformer Float32	41.2	705MB	–
HAT Float32	41.8	227MB	3×
HAT 8 bits	**41.9**	**57MB**	**12×**
HAT 4 bits	41.1	**28MB**	**25×**

Table 4.7: K-means quantization of HAT models on WMT'14 En-Fr. 4-bit quantization reduces the model size by 25× with only 0.1 BLEU loss compared with the Transformer baseline. 8-bit quantization even has 0.1 higher BLEU than its full precision version.

model size. We initialize centroids uniformly in the range of [min, max] of each weight matrix and run at most 300 iterations for each of them. Even without any finetuning, 4-bit quantization can reduce the model size by 25× with negligible BLEU loss compared to the Transformer-Big baseline (Table 4.7). Interestingly, the 8-bit model even has 0.1 higher BLEU than the full precision model, indicating the robustness of the searched model. Compared with the Transformer-Base 4-bit quantization baseline, which has 24MB model size and 38.9 BLEU score, HAT has 2.2 higher BLEU with similar model size.

Knowledge Distillation Friendly. HAT is also orthogonal to knowledge distillation (KD) because HAT focuses on searching for an efficient architecture while KD focuses on better training a given architecture. We combine KD with HAT by distilling token-level knowledge (top-5 soft labels) from a high-performance SubTransformer to a low-performance SubTransformer on WMT'14 En-De task. The teacher model has a BLEU of 28.5 and 49M parameters; the student model has 30M parameters. KD can improve the BLEU of the student model from 25.8 to 26.1.

4.2 SpAtten Accelerator

4.2.1 Evaluation Setups

To evaluate the performance of SpAtten, we implement the RTL code with SpinalHDL and simulate each application using Verilator to get the cycle numbers. For HBM

modeling, we use Ramulator [44] with HBM2 settings. The architectural setups are listed in Table 4.8. SpAtten runs at 1GHz. 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. To model the floating-point arithmetic units in the Softmax module, we get the number of multiplications and additions from the simulator, and use energy and area from [17] for addition and multiplication, and those for exponential and division from [74]. We also obtain the width, size, and the number of read/write of each SRAM and FIFO from the simulator and use CACTI [60] to estimate the energy and area of SRAMs and FIFOs. For HBM, we apply the power consumption number from [78, 2].

We extensively select multiple hardware platforms as the evaluation baselines, including server GPU (Nvidia TITAN Xp), mobile GPU (Nvidia Jetson Nano); server CPU (Intel E5-2640 Xeon v4 @ 2.40GHz), mobile CPU (4-core ARM A53 CPU on a Raspberry Pi); and the state-of-the-art ASIC accelerator A^3 [24]. For GPUs and CPUs, we run the attention inference with PyTorch and measure the latency. For TITAN Xp GPU, we measure the power with `nvidia-smi`; and for Xeon CPU, we use the `pcm-power` tool. For Nano GPU and Raspberry Pi ARM CPU, a power meter is used to get their power consumption. For all latency measurements, we repeat the process for 1000 times, then remove the largest 15% and smallest 15%, and average the remaining ones. For all power measurements, we first measure the idle power of the system and then repeatedly run the workloads and measure the total power. The dynamic power is total power minus idle power.

4.2.2 Performance Comparisons

We evaluate the performance of SpAtten on two discriminative models: BERT-Base, BERT-Large, and two generative models: GPT-2-Small and GPT-2-Medium. The tasks for discriminative BERT are nine tasks from the GLUE [90] benchmark set containing sentiment analysis, entailment classification, similarity regression etc. For generative GPT-2, we use language modeling task on four datasets: Wikitext-2 [57], Wikitext-103 [57], Pen Tree Bank [56] and Google One-Billion Word [10]. Therefore,

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.
Attn_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.

Table 4.8: Architectural Setups of SpAtten.

we have in total $2 \times 9 + 2 \times 4 = 26$ benchmarks. For BERT models, we perform low-precision but not dynamic low-precision because they are computation bounded.

For all the tasks, finetuning is performed after token pruning to recover accuracy. We allow at most 3% of relative accuracy loss after performing token pruning and dynamic low-precision optimizations. For GPT-2 tasks, at most 3% increase of cross-entropy loss is tolerated. For each model, we try multiple sets of token pruning ratio and bitwidths for MSBs and LSBs in dynamic low-precision bitwidths to satisfy the accuracy requirements. Therefore, the settings for different models are not the same. Typically, the pruning ratio for BERT is around 2×, while that for GPT-2 is 4×. The typical MSB bitwidth is six, and LSB bitwidth is four. For the latency measurement of BERT models, we set the input sentence length as the average length of the 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 power × latency, which is the total energy consumed to perform the task.

Figure 4-7 shows the speedup of SpAtten. On average, SpAtten achieves 193×, 419×, 1297× and 6218× speedup over TITAN Xp GPU, Xeon CPU, Nano GPU, and Raspberry Pi ARM CPU. SpAtten outperforms the baselines on each of the benchmarks because of its high-parallelism datapath specialized for attention. Meanwhile, token pruning and dynamic low-precision reduce DRAM access by 10.4×, further reducing the computation and memory fetch overhead.

Figure 4-8 compares the energy consumption of SpAtten with others. SpAtten is

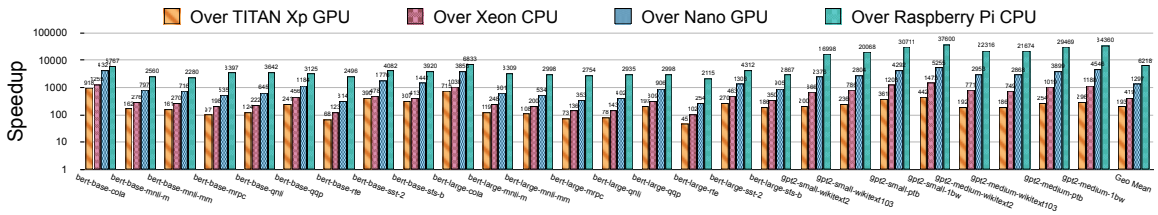

Figure 4-7: Speedup of SpAtten over TITAN Xp GPU, Xeon CPU, Nano GPU, and Raspberry Pi ARM CPU.

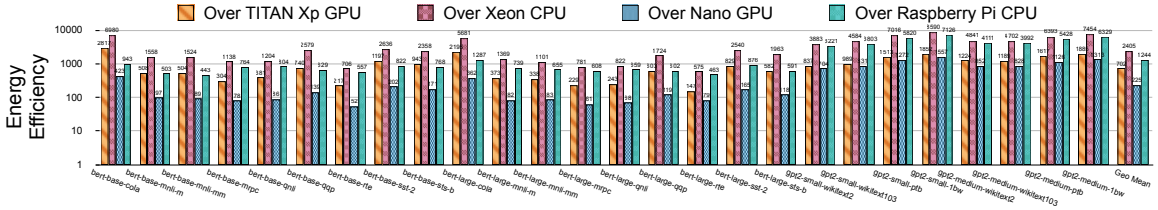

Figure 4-8: Energy efficiency of SpAtten over TITAN Xp GPU, Xeon CPU, Nano GPU, and Raspberry Pi ARM CPU.

702×, 2405×, 225× and 1244× more energy efficient than TITAN Xp GPU, Xeon CPU, Nano GPU, and Raspberry Pi ARM CPU. The energy savings mainly come from SpAtten removing all unessential components such as cache, branch predictor in general-purpose architectures, which are highly energy inefficient. Instead, we design a fully pipelined datapath to reduce the intermediate SRAM fetch, and specialized operators to support a unique set of operations in attention.

We also compare SpAtten with another attention accelerator A^3 [24] in Figure 4-9. The numbers of multipliers are different in two accelerators. Therefore, for fair comparisons, both are compared with their base architecture. Specifically, the base architecture for SpAtten has no token pruning and dynamic low-precision but remains the same dedicated datapath and number of multipliers, which is similar to the settings of A^3 base architecture. Since the techniques in A^3 also incur small accuracy loss, we compare performance in the condition of the same level of accuracy degradation, i.e., less than 3%. Comparison results show that SpAtten is 2.8× faster than A^3 and 1.9× more energy efficient. The main reason is that A^3 only conducts fine-grained pruning of keys while SpAtten prunes the entire token and removes all involved computations.

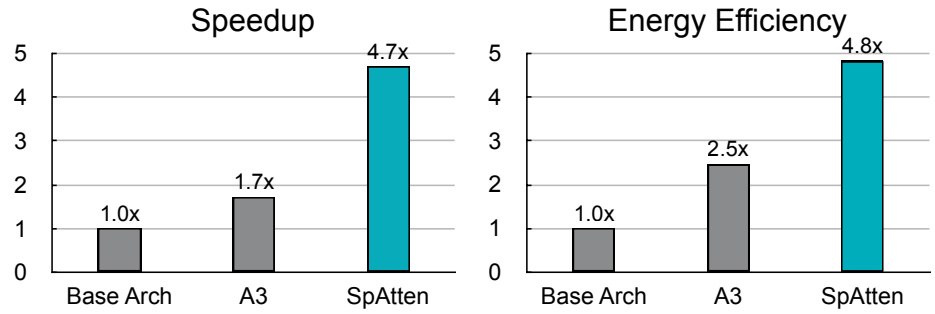

Figure 4-9: Speedup and energy efficiency comparisons with A^3. SpAtten achieves 2.8x speedup over A^3 with similar accuracy loss.

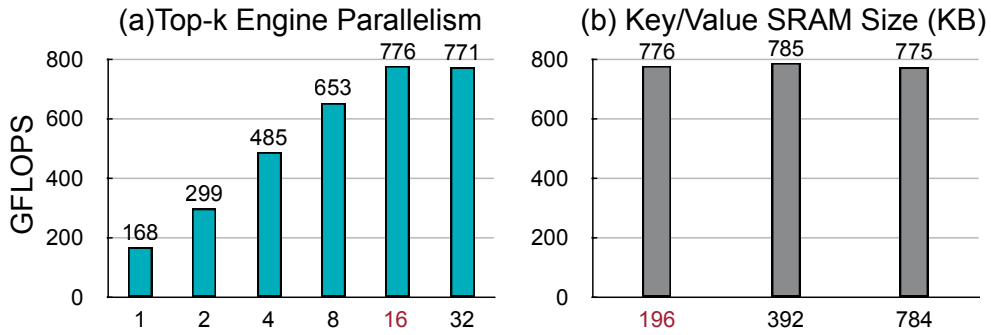

Figure 4-10: Design choice exploration of different top-k engine parallelism and SRAM sizes for SpAtten. Top-k engine with parallelism of 16 and Key/Value SRAM of 196KB are enough for our architecture.

4.2.3 Analysis

Design Choice Exploration. We conduct design choice exploration to find the most suitable architecture hyperparameters. Here we show two examples in Figure 4-10. We select one application on the GPT-2 model to conduct the exploration. On the left side, we change the parallelism of the top-k engine by changing the number of comparators ranging from 1 to 32. In this way, the time for top-k to perform a STATE_RUN stage will be different. We can see that after parallelism 16, the performance does not increase a lot. The reason is that parallelism 16 matches with the speed of data being fed to the top-k engine from the Q × K module. Thus, parallelism larger than 16 makes top-k no longer the bottleneck, and overall performance does not change much. Therefore, we select 16 in our design. On the right side, we change the size of SRAMs storing K and V. Since we want to support up to 1024 context length, the minimum SRAM size is set to $1024 \times 512 \times 12\text{bits} = 196\text{KB}$. We find that

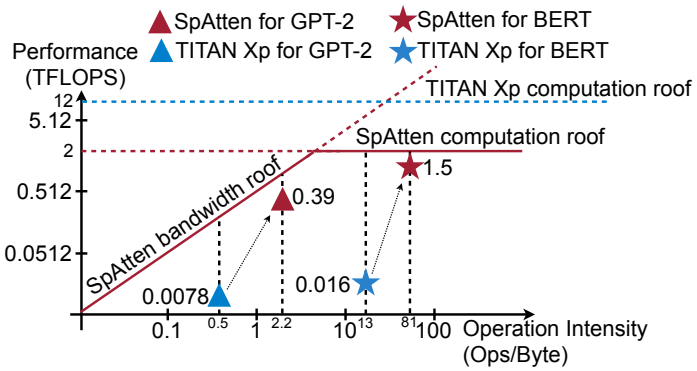

Figure 4-11: Roofline models for TITAN Xp GPU and SpAtten.

increasing the SRAM size does not increase the performance a lot. That is because the whole architecture is fully pipelined. Thus, more intermediate buffers will not much influence the throughput. Therefore, in consideration of reducing the SRAM static power, we select the smallest 196KB in our design.

Roofline Model. To better understand the distance of SpAtten to the theoretical optimal performance, we analyze it with the roofline model in Figure 4-11. We also show the performance of TITAN Xp GPU together for comparison. 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 1.5 TFLOPS on BERT tasks and is close to the computation roof. GPT-2 models, on the contrary, have a low arithmetic intensity and thus appear in the memory-bounded region. SpAtten achieves 0.39 TFLOPS, which is also close to the bandwidth roof. In terms of TITAN Xp GPU, the performance is far away from the roofs in both models because the attention data movement overhead reduces the utilization of on-chip arithmetic units. Token pruning and dynamic low-precision improve the computation intensity, thus the SpAtten points are to the right of TITAN Xp GPU.

Interpreting the Speedup. To look into the performance gain, we also show the speedup breakdown of SpAtten over TITAN Xp GPU on eight GPT-2 benchmarks

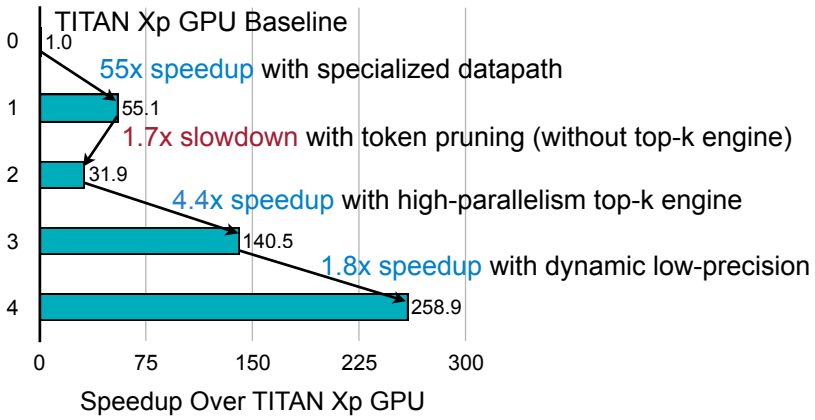

Figure 4-12: Speedup breakdown of SpAtten over TITAN Xp GPU. Token pruning degrades the performance in step 2 since the top-k computation becomes the bottleneck of the system. The bottleneck is resolved by a specialized high-parallelism top-k engine in step 3.

in Figure 4-12. With the dedicated datapath, we can achieve over 55× speedup comparing to the GPU baseline which needs to execute a large number of memory instructions to process the attention. Token pruning is then applied to remove unimportant tokens in the second step. Nevertheless, we find that the performance drops by 1.7× even though the computation is reduced. The reason is that token pruning needs to frequently execute top-k to find unimportant tokens for each set of attention probability. However, this step only has a sequential top-k module which limits the overall performance. After the third step, the dedicated high-parallelism top-k engine resolves the bottleneck and thus brings 4.4× speedup. Finally, the dynamic low-precision reduces the average bitwidth of all inputs, achieving another 1.8× speedup with less DRAM access.

Area and Power Breakdown. We also analyze the on-chip area and power of each module. Figure 4-13 shows the area and power breakdown of SpAtten. The Q × K and Attention_Prob × V modules take the largest portions of both area and power since they are the two most computational intensive modules. Despite the similar structure of Q × K and Attention_Prob × V modules, the latter consumes less energy thanks to the presence of local token pruning. The two top-k engines are relatively efficient as they only take 2.7% of overall area and 3.4% of the power.

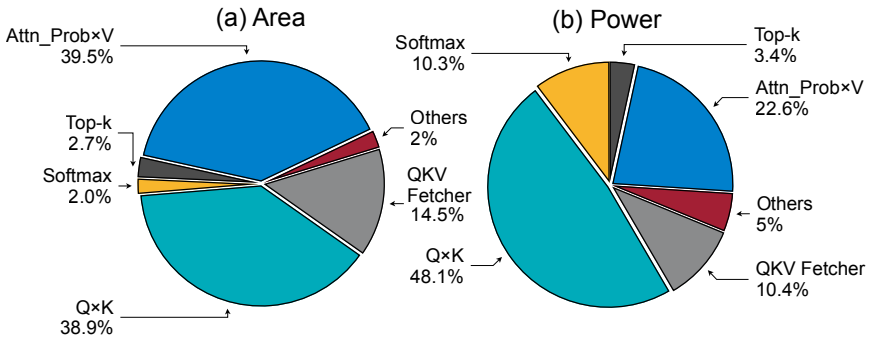

Figure 4-13: Breakdown of on-chip (a) Area and (b) Power of SpAtten.

Bert for Sentence Classification

A wonderful movie, I am sure that you will remember it, you admire its conception and are able to resolve some of the confusions you had while watching it.

~~A~~ **wonderful movie,** ~~I am~~ **sure** ~~that~~ **you will remember** ~~it,~~ **you admire** ~~its~~ **conception and** ~~are~~ **able** ~~to~~ **resolve** ~~some of the~~ **confusion**~~s~~ **you had while watching** ~~it.~~

~~A wonderful movie, I am~~ **sure** ~~that you will~~ **remember** ~~it, you~~ **admire** ~~its conception and are able to~~ **resolve** ~~some of the~~ **confusion**~~s~~ ~~you had while watching it.~~

Bert for Sentence Regression — Sentence similarity regression result: *3.8* (range [1,5], the higher, the more similar)

It does sound like your cat is upset about something, and trying to communicate it to you. [separate] Something is bothering your cat and he wants to tell you.

~~It does~~ **sound like** ~~your~~ **cat** ~~is~~ **upset about something, and trying** ~~to~~ **communicate** ~~it to~~ **you. [separate]** ~~Something is~~ **bothering** ~~your~~ **cat and** ~~he~~ **wants** ~~to~~ **tell** ~~you~~**.**

~~It does~~ **sound** ~~like your~~ **cat** ~~is~~ **upset** ~~about something,~~ **and trying** ~~to~~ **communicate** ~~it to you~~**. [separate]** ~~Something is~~ **bothering** ~~your cat and he wants to tell you~~**.**

GPT-2 for Language Modeling

Du Fu was a great poet of the Tang dynasty. Recently a variety of styles have been used in efforts to translate the work of Du Fu into *English*

Du ~~Fu was a great~~ **poet** ~~of the~~ **Tang dynasty. Recently** ~~a variety of~~ **styles** ~~have been used~~ **in efforts to translate the work of Du Fu into *English***

'***English***' is the generated token.

Du ~~Fu was a great poet of the Tang dynasty. Recently a variety of styles have been used in efforts to~~ **translate** ~~the work of Du Fu~~ **into *English***

Figure 4-14: Examples of token pruning in different models and tasks.

Token Pruning Visualization. Figure 4-14 visualizes the token pruning process for different models and tasks. The tokens pruned are clearly inconsequential ones, showing the reliability of cumulative importance scores. The top example shows the BERT model for sentiment classification. Solely from the remaining tokens such as 'admire' and 'resolve confusion', we can easily tell that the comment is a positive one. The exemplary task in the middle is sentence similarity score regression. The regressed scores range between 1 and 5, and larger scores indicate higher similarity between two sentences. Token pruning 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 bottom example is a generative language modeling task

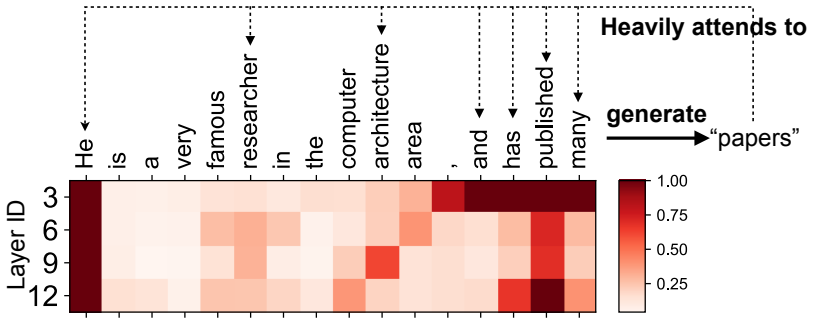

Figure 4-15: Cumulative importance scores across layers in the GPT-2 model.

with GPT-2. The generated token is 'English'. SpAtten aggressively prunes away most tokens as they are irrelevant to what will be generated, and only keeps the 'Du', 'translate' and 'into' tokens. The model may find the name 'Du' not typical in English, so the translation language should be 'English'.

Figure 4-15 shows the cumulative importance score of every single layer in a GPT-2 LM model. The important tokens are consistent across layers, such as the word 'published'. The 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.

Chapter 5

Related Work

5.1 Efficient Transformer Models

Transformer [87] has prevailed in sequence modeling [61, 38]. By stacking identical blocks, the model obtains a large capacity but incurs high latency. Recently, a research trend is to modify the Transformer to improve the performance [11, 98, 82, 94]. Among them, [98] introduces a convolution-based module to replace the attention; [103] and [45] also propose AAN and SSRU to replace the attention operation; [94] proposes to train deep Transformers by propagating multiple layers together in the encoder. HAT [92] is orthogonal to them and can be combined to search for efficient architecture with those new modules. Another trend is to apply non- or partially-autoregressive models to cut down the iteration number for decoding [22, 1, 96, 21]. Although reducing latency, they sometimes suffer from low performance.

Moreover, [5] explores to use learned linear combinations of encoder outputs as the decoder inputs, while HAT concatenates the outputs without linear combinations, thus better preserving the low-level information. [100] investigates mobile settings for NLP tasks and proposes a multi-branch Lite Transformer. However, it relies on FLOPs for efficient model design, which is an inaccurate proxy for hardware latency (Figure 1-5). There are also works [43, 40, 45, 101] using Knowledge Distillation (KD) to obtain small student models. Our method is orthogonal to KD and can be combined with it to improve the efficiency further. Besides, there are also hardware

accelerators [24, 107] for attention and fully-connected layers in the Transformer to achieve efficient processing.

5.2 Neural Architecture Search

In the computer vision community, there has been an increasing interest in automating the efficient model design with Neural Architecture Search (NAS) [111, 112, 66, 28]. Some of them apply black-box optimization methods such as evolutionary search [95] and reinforcement learning [8, 28, 93, 91, 55]; Some leverage backpropagation with differentiable architecture search [51]. Some of them also involve hardware constraints into optimizations such as MNasNet [84], ProxylessNAS [8], FBNet [97] and APQ [95].

To reduce the high design cost of NAS, supernet-based methods [66, 6, 23] apply a proxy for sub-network performance and adopt search algorithms to find good sub-networks. For NLP tasks, the benefits of the architecture search have not been fully investigated. Recently, [79] proposed the Evolved Transformer to search for architectures under model size constraints and surpassed the original Transformer baselines. However, it suffers from very high search costs (*250 GPU years*), making it unaffordable to search specialized models for various hardware and tasks. Besides, hardware latency feedback was not taken into account for better case-by-case specializations. Since different hardware has distinct architecture and features [14], feedback from hardware is critical for efficient NLP architecture search.

5.3 NN Pruning and Quantization

Neural networks tend to be over-parameterized, and many algorithms are proposed to prune away the redundant parameters. Fine-grained pruning [27, 26] cuts off the connections within the weight matrix and can achieve a high pruning ratio. However, it is not friendly to CPUs and GPUs, and requires dedicated hardware [64, 107] to support the sparse matrix multiplication. To this end, structured pruning such as channel-pruning [29, 53, 50, 59, 3, 34, 67] is further proposed to enable acceleration

on general-purpose hardware.

Quantization reduces data precision to shrink the model size and bandwidth requirement. [26] quantizes the NN weights to reduce the model size by grouping the weights with k-means. [35] quantizes both weights and activations for efficient deployments on mobile devices. Some works even push the limits to only one or two bits [15, 72, 108].

SpAtten is fundamentally different from the existing weight pruning and quantization techniques, because there is no weight in attention, and we prune the input tokens based on their importance to the sentence. The quantization in SpAtten is also applied to input Q, K, and V instead of the weights.

5.4 Accelerators for Sparse and Low-Precision NN

There exist various FPGA and ASIC accelerators leveraging the sparsity in neural networks to improve the efficiency. EIE [25] applies a compressed representation for both inputs and weights, and skips zero inputs in multipliers. Cambricon-X [105] exploits weights sparsity with a index module and only transfers required neurons. SCNN [65] applies an outer-product based method to exploit the CNN sparsity. Cambricon-S [109] conducts an algorithm-hardware co-design to reduce the irregularity of sparse patterns, thus reducing the index storing overhead. Sparse ReRam [102] proposes a ReRam-based architecture to conduct sparse FC in the practical small granularity. SparTen [19] leverages a greedy balancing scheme to achieve a better load balance. Laconic [75] decomposes multiplications to the bit-level and reduces bit-level computation by up to 40×. Eager pruning [104] moves the pruning to the early training stage to improve both training and inference efficiency. There are also some general sparse tensor algebra accelerators [12, 30, 41, 110, 47] proposed in recent years that can be used to process sparse FC layers. Most of the mentioned works focus on leveraging weight sparsity. SpAtten, in contrast, leverages token sparsity and is equipped with a specialized top-k engine to support the on-the-fly token pruning.

Low-precision NN models are also supported by many accelerators. Stripes [37]

and UNPU [49] propose the bit-serial computation to support mixed-precision operands. Loom [76] and BISMO [86] further adopt temporal fusion to provide full bit flexibility. DeepRecon [73] and BitFusion [77] dynamically compose and decompose 2-bit multipliers in space to support different bitwidth multiplication. Nvidia Turing GPU architecture [62] has tensor cores to support 1/4/8/16-bit arithmetic operations. Google TPU [36] utilizes a systolic array for matrix multiplication and also supports 8-bit and 16-bit precision. In those works, the bitwidth is fixed at compile time, while in SpAtten, we can adjust the bitwidth based on the attention probability distribution.

Chapter 6

Conclusion

In this thesis, we present an algorithm-hardware co-design methodology for efficient NLP computing. On the algorithm side, we propose a hardware-aware neural architecture search approach with an efficient weight-shared SuperTransformer. The design space is augmented with two novel techniques: *arbitrary encoder-decoder attention* breaks the information bottleneck between encoder and decoder, while *heterogeneous layers* allow each layer of the NLP model vary its architecture. We then perform an evolutionary search with hardware latency feedback to search for a specialized fast model for target hardware. Evaluation results show that it achieves up to 3× speedup, 3.7× smaller size over Transformer without loss of accuracy. With over 12,000× less search cost, it outperforms the Evolved Transformer with 2.7× speedup and 3.6× smaller size.

To resolve the performance bottleneck – attention layers, we further design a hardware accelerator, SpAtten, to enable efficient sparse and low-precision attention inference. The accelerator supports *token pruning* to remove the computation and memory access of inessential tokens. To support the on-the-fly selections of the pruned tokens, a high-parallelism top-k engine with $O(n)$ time complexity is designed. Moreover, we also propose *dynamic low-precision* to allow different bitwidths across layers. MSBs are fetched in all layers, but LSBs are only fetched in the sensitive layers. Experiments on various attention benchmarks show that SpAtten achieves on average 2.8×, 193×, 419×, 1297×, 6218× speedup and 1.9×, 702×, 2405×, 225×,

1244× energy savings over the A^3 accelerator, TITAN Xp GPU, Xeon CPU, Nano GPU, Raspberry Pi ARM CPU, respectively.

Natural language processing is changing our daily lives at an increasingly rapid pace. We hope the algorithm-hardware co-design methodology in this thesis can open up an avenue towards efficient NLP model deployments for both cloud and edge applications.

Bibliography

[1] Nader Akoury, Kalpesh Krishna, and Mohit Iyyer. Syntactically supervised transformers for faster neural machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1269–1281, Florence, Italy, July 2019. Association for Computational Linguistics.

[2] Michael Alfano, Bryan Black, Jeff Rearick, Joseph Siegel, Michael Su, and Julius Din. Unleashing fury: A new paradigm for 3-d design and test. *IEEE Design & Test*, 34(1):8–15, 2016.

[3] Sajid Anwar and Wonyong Sung. Compact deep convolutional neural networks with coarse pruning, 2016.

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

[5] Ankur Bapna, Mia Chen, Orhan Firat, Yuan Cao, and Yonghui Wu. Training deeper neural machine translation models with transparent attention. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3028–3033, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[6] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 550–559, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[7] Han Cai, Chuang Gan, and Song Han. Once for all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[8] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019.

[9] Daniel Cer, Mona Diab, Eneko Agirre, Iñigo Lopez-Gazpio, and Lucia Specia. SemEval-2017 task 1: Semantic textual similarity multilingual and crosslingual

focused evaluation. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 1–14, Vancouver, Canada, August 2017. Association for Computational Linguistics.

[10] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

[11] Mia Xu Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Mike Schuster, Noam Shazeer, Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Zhifeng Chen, Yonghui Wu, and Macduff Hughes. The best of both worlds: Combining recent advances in neural machine translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 76–86, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[12] Yuedan Chen, Guoqing Xiao, Fan Wu, Zhuo Tang, and Keqin Li. tpspmv: A two-phase large-scale sparse matrix-vector multiplication kernel for manycore architectures. *Information Sciences*, 2020.

[13] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[14] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of fpgas and gpus. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96. IEEE, 2018.

[15] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

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

[17] Sameh Galal and Mark Horowitz. Energy-efficient floating-point unit design. *IEEE Transactions on computers*, 60(7):913–922, 2010.

[18] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1243–1252, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[19] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. Sparten: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–165, 2019.

[20] Édouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. Efficient softmax approximation for GPUs. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1302–1310, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[21] Jiatao Gu, James Bradbury, Caiming Xiong, Victor O.K. Li, and Richard Socher. Non-autoregressive neural machine translation. In *International Conference on Learning Representations*, 2018.

[22] Jiatao Gu, Changhan Wang, and Jake Zhao. Levenshtein Transformer. *arXiv*, 2019.

[23] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling, 2019.

[24] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. Aˆ 3: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341. IEEE, 2020.

[25] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 243–254. IEEE Press, 2016.

[26] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2016.

[27] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NeurIPS*, 2015.

[28] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[29] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. *ICCV*, 2017.

[30] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–333, 2019.

[31] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[32] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[33] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. Convolutional neural network architectures for matching natural language sentences. In *Advances in neural information processing systems*, pages 2042–2050, 2014.

[34] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures, 2016.

[35] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *CVPR*, 2018.

[36] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[37] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[38] Marcin Junczys-Dowmunt. Microsoft's submission to the wmt2018 news translation task: How i learned to stop worrying and love the data. *arXiv preprint arXiv:1809.00196*, 2018.

[39] Marcin Junczys-Dowmunt. Microsoft translator at wmt 2019: Towards large-scale document-level neural machine translation. *arXiv preprint arXiv:1907.06170*, 2019.

[40] Marcin Junczys-Dowmunt, Kenneth Heafield, Hieu Hoang, Roman Grundkiewicz, and Anthony Aue. Marian: Cost-effective high-quality neural machine translation in C++. In *Proceedings of the 2nd Workshop on Neural Machine*

Translation and Generation, pages 129–135, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[41] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 600–614, 2019.

[42] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics.

[43] Yoon Kim and Alexander M Rush. Sequence-level knowledge distillation. *arXiv preprint arXiv:1606.07947*, 2016.

[44] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2015.

[45] Young Jin Kim, Marcin Junczys-Dowmunt, Hany Hassan, Alham Fikri Aji, Kenneth Heafield, Roman Grundkiewicz, and Nikolay Bogoychev. From research to production and back: Ludicrously fast neural machine translation. In *Proceedings of the 3rd Workshop on Neural Generation and Translation*, pages 280–288, Hong Kong, November 2019. Association for Computational Linguistics.

[46] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*, 2015.

[47] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 740–753, 2019.

[48] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[49] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 218–220. IEEE, 2018.

[50] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime Neural Pruning. In *NIPS*, 2017.

[51] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019.

[52] Shujie Liu, Nan Yang, Mu Li, and Ming Zhou. A recursive recurrent neural network for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1491–1500, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[53] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *ICCV*, 2017.

[54] Ilya Loshchilov and Frank Hutter. SGDR: Stochastic Gradient Descent with Warm Restarts. In *International Conference on Learning Representations*, 2017.

[55] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, et al. Park: An open platform for learning-augmented computer systems. In *Advances in Neural Information Processing Systems*, pages 2490–2502, 2019.

[56] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[57] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[58] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[59] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference, 2016.

[60] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman Jouppi. Cacti 6.0: A tool to model large caches. *HP Labs*, 2015.

[61] Nathan Ng, Kyra Yee, Alexei Baevski, Myle Ott, Michael Auli, and Sergey Edunov. Facebook FAIR's WMT19 news translation task submission. In *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*, pages 314–319, Florence, Italy, August 2019. Association for Computational Linguistics.

[62] Nvidia. Nvidia tensor cores. In *Nvidia*, 2018.

[63] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[64] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[65] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[66] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4095–4104, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[67] A. Polyak and L. Wolf. Channel-level acceleration of deep face representations. *IEEE Access*, 2015.

[68] Matt Post. A call for clarity in reporting bleu scores. *arXiv preprint arXiv:1804.08771*, 2018.

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

[70] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

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

[72] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net - ImageNet Classification Using Binary Convolutional Neural Networks. In *ECCV*, 2016.

[73] Tayyar Rzayev, Saber Moradi, David H Albonesi, and Rajit Manchar. Deeprecon: Dynamically reconfigurable architecture for accelerating deep neural networks. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 116–124. IEEE, 2017.

[74] Soheil Salehi and Ronald F DeMara. Energy and area analysis of a floating-point unit in 15nm cmos process technology. In *SoutheastCon 2015*, pages 1–5. IEEE, 2015.

[75] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. Laconic deep learning inference acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 304–317, 2019.

[76] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, and Andreas Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[77] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775. IEEE, 2018.

[78] Anton Shilov. Jedec publishes hbm2 specification as samsung begins mass production of chips. In *JEDEC*, 2016.

[79] David So, Quoc Le, and Chen Liang. The evolved transformer. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5877–5886, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[80] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.

[81] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, Florence, Italy, July 2019. Association for Computational Linguistics.

[82] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. In *Proceedings of the 57th Annual*

Meeting of the Association for Computational Linguistics, pages 331–335, Florence, Italy, July 2019. Association for Computational Linguistics.

[83] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014.

[84] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[85] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147, 2003.

[86] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander. Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 307–3077. IEEE, 2018.

[87] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Conference on Neural Information Processing Systems*, 2017.

[88] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[89] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5797–5808, Florence, Italy, July 2019. Association for Computational Linguistics.

[90] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics.

[91] Hanrui Wang, Kuan Wang, Jiacheng Yang, Linxiao Shen, Nan Sun, Hae-Seung Lee, and Song Han. Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In *ACM/IEEE 57th Design Automation Conference (DAC)*, 2020.

[92] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. Hat: Hardware-aware transformers for efficient natural language processing. In *Annual Conference of the Association for Computational Linguistics*, 2020.

[93] Hanrui Wang, Jiacheng Yang, Hae-Seung Lee, and Song Han. Learning to design circuits. In *NeurIPS 2018 Machine Learning for Systems Workshop*, 2018.

[94] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1810–1822, Florence, Italy, July 2019. Association for Computational Linguistics.

[95] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, Hanrui Wang, Yujun Lin, and Song Han. Apq: Joint search for network architecture, pruning and quantization policy. In *Conference on Computer Vision and Pattern Recognition*, 2020.

[96] Bingzhen Wei, Mingxuan Wang, Hao Zhou, Junyang Lin, and Xu Sun. Imitation learning for non-autoregressive neural machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1304–1312, Florence, Italy, July 2019. Association for Computational Linguistics.

[97] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[98] Felix Wu, Angela Fan, Alexei Baevski, Yann Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. In *International Conference on Learning Representations*, 2019.

[99] Felix Wu, Angela Fan, Alexei Baevski, Yann N Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. *arXiv preprint arXiv:1901.10430*, 2019.

[100] Zhanghao Wu, Zhijian Liu, Ji Lin, Yujun Lin, and Song Han. Lite transformer with long-short range attention. In *International Conference on Learning Representations*, 2020.

[101] Zhongxia Yan, Hanrui Wang, Demi Guo, and Song Han. Micronet for efficient language modeling. *Journal of Machine Learning Research*, 2020.

[102] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. Sparse reram engine: joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 236–249, 2019.

[103] Biao Zhang, Deyi Xiong, and Jinsong Su. Accelerating neural transformer via an average attention network. *arXiv preprint arXiv:1805.00631*, 2018.

[104] Jiaqi Zhang, Xiangru Chen, Mingcong Song, and Tao Li. Eager pruning: algorithm and architecture support for fast training of deep neural networks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 292–303. IEEE, 2019.

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

[106] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 649–657. Curran Associates, Inc., 2015.

[107] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020.

[108] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.

[109] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 15–28. IEEE, 2018.

[110] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–371, 2019.

[111] Barret Zoph and Quoc V Le. Neural Architecture Search with Reinforcement Learning. In *International Conference on Learning Representations*, 2017.

[112] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning Transferable Architectures for Scalable Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.