ON THE EFFICIENCY OF TRANSFORMERS

by

Zhanpeng Zeng

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: 05/07/2024

The dissertation is approved by the following members of the Oral Committee:
Junjie Hu, Assistant Professor, Biostatistics & Med. Info., Computer Sciences
Yixuan Li, Assistant Professor, Computer Sciences
Yingyu Liang, Associate Professor, Computer Sciences
Frederic Sala, Assistant Professor, Computer Sciences
Karthikeyan Sankaralingam, Professor, Computer Sciences
Vikas Singh (Advisor), Professor, Biostatistics & Med. Info., Computer Sciences

© Copyright by Zhanpeng Zeng 2024
All Rights Reserved

This one is for my parents. Thank you for your love and support, and for encouraging me to pursue a PhD.

ACKNOWLEDGMENTS

I was fortunate enough to collaborate with an incredible group of labmates, collaborators, and supervisors. In no specific order, these individuals have greatly contributed to my research through their collaboration, discussions, and support in numerous ways: Shailesh Acharya, Rudrasis Chakraborty, Liang-Chieh Chen, Michael Davies, Glenn Fung, Cole Hawkins, Ju He, Mingyi Hong, Jeffery Kline, Yin Li, Qihao Liu, Sourav Pal, Nikolaos Pappas, Pranav Pulijala, Sathya N. Ravi, Xiaohui Shen, Vikas Singh, Karu Sankaralingam, Mingxing Tan, Yunyang Xiong, Qihang Yu, Aston Zhang, Li Zhang, and Shuai Zheng. I would also like to thank my thesis committee: Junjie Hu, Yixuan Li, Yingyu Liang, Frederic Sala, Karthikeyan Sankaralingam, and Vikas Singh for the time spent to review my dissertation and provide valuable feedback for my dissertation, as well as valuable advice for my future career.

I would like to express my gratitude to my advisor, Vikas, for his continuous support and guidance throughout my PhD studies. He has helped me become an independent researcher, a better collaborator, and a more responsible person. His vision and extensive knowledge have encouraged me to view my ideas from new perspectives, inspiring deeper and more interesting insights, and enabling the development of more significant ideas.

I would like to extend a special thank you to Yunyang for his guidance and patience during our collaboration when I first joined the lab as a complete novice in research. He helped me navigate the entire research process, from formulating and evaluating an idea to writing a paper. He also provided invaluable help and advice for my career development.

I would like to give a sincere thank you to my parents for their love, support, and encouragement. Without their support and encouragement, I would not have even started pursuing a PhD degree or accomplished what I did.

Finally, I thank American Family Insurance for supporting my Research Assistantship through much of the graduate program via funding through the Data Sciences Institute.

CONTENTS

Contents iii

List of Tables vi

List of Figures ix

Abstract xii

1 Introduction 1
- 1.1 Transformer Architecture . . . 4
- 1.2 On the Efficiency of Transformers . . . 7
- 1.3 Contribution and Scope of Thesis . . . 15

2 Background 21
- 2.1 Notations . . . 21
- 2.2 Transformers . . . 22
- 2.3 Efficient Structured Projections . . . 25
- 2.4 Locality Sensitive Hashing . . . 28
- 2.5 Importance Sampling . . . 32
- 2.6 Multi-Resolution Analysis . . . 35
- 2.7 Gradient Estimates of Selection Operators . . . 38
- 2.8 Hardware for Matrix Multiplication . . . 40

3 Memory Lookup based Approximation for Efficient Self-Attention 42
- 3.1 Locality Sensitive Hashing based Importance Sampling . . . 43
- 3.2 YOSO Attention . . . 45
- 3.3 Backpropagation . . . 52
- 3.4 Experiments . . . 56
- 3.5 Summary . . . 65

4 Memory Lookup based Approximation for Compute-lite Feedforward Network **66**
4.1 Preliminaries 68
4.2 FFN as Lookups 71
4.3 BH4: Efficient and Expressive Projection for Hashing 76
4.4 Experiments 78
4.5 Summary 87

5 Multi-Resolution based Approximation for Efficient Self-Attention **88**
5.1 MRA view of Self-attention 89
5.2 A Practical Approximation scheme 95
5.3 Link to Sparsity and Low Rank 105
5.4 Experiments 107
5.5 Summary 117

6 Multi-Resolution based Approximation for Faster Transformer Block **118**
6.1 Notations 120
6.2 VIP-Token Centric Compression (VCC) 121
6.3 A Specific Instantiation via Multi-Resolution Compression 125
6.4 Efficient Data Structure for De/compression 138
6.5 Experiments 146
6.6 Limitations 159
6.7 Summary 160

7 Low Precision Integer Computation for General Matrix Multiply **162**
7.1 Round to Nearest: What do we lose? 164
7.2 What happens with Low Bit-Width? 175
7.3 IM-Unpack: Integer Matrix Unpacking 177
7.4 Limitations 193
7.5 Summary 193

8 Conclusions **195**

8.1 Future Directions 197

References **202**

LIST OF TABLES

3.1 Time/memory complexity of self-attention and YOSO-attention in forward/backward computation 53
3.2 Dev set results on MLM and SOP pretraining and GLUE tasks 60
3.3 Dev set results on MLM and SOP pretraining and GLUE tasks for baseline comparisons. 61
3.4 Test set accuracy of LRA tasks 62

4.1 Log perplexity of each baseline 81
4.2 Downstream performance of RoBERTa-small models. 82
4.3 Log perplexity when scaling to a RoBERTa-base model. 82
4.4 Ablation study evaluating the effects of different hyper-parameters on model performance 83
4.5 Average latency for LookupFFN compared to baselines. 84
4.6 Analysis of performance characteristics for LookupFFN 85

5.1 Hyperparameters for all experiments. 108
5.2 Summary of 512 length RoBERTa-base models 112
5.3 Summary of 512 length RoBERTa-small models 113
5.4 Summary of 4096 length RoBERTa-base models 114
5.5 Summary of 4096 length RoBERTa-small models. 115
5.6 Test set accuracy of LRA tasks 116
5.7 Summary of ImageNet results trained on 4-layer Transformers 116

6.1 Approximation quality. 147
6.2 Dev set results for encoder-only models. 148
6.3 Dev set results for encoder-decoder models 150
6.4 Dev results for encoder-decoder models on MultiNews. 151
6.5 Dev accuracy and FLOPs for encoder-only models 151
6.6 Dev results and FLOPs for encoder-decoder models. 152

6.7 Dev results of NarrativeQA on base model when scaling sequence length from 16K to 128K. . . . 152
6.8 Test set accuracy of LRA tasks. . . . 154
6.9 Length statistics of each dataset 157
6.10 Hyperparameters for all experiments. . . . 158

7.1 Standard deviation vs percentile when removing largest outliers 166
7.2 Inference: Comparison on LLaMA-7B zero-shot performance and ViT ImageNet classification when only quantize parameters 168
7.3 Inference: Comparison on LLaMA-7B zero-shot performance and ViT ImageNet classification when using quantized computations in all linear layers 169
7.4 Inference: Comparison on LLaMA-7B and ViT when quantize computation in all GEMMs 169
7.5 Inference: Comparison on LLaMA-13B when we quantize computation in all linear layers. . . . 170
7.6 Inference: LLaMA-13B when we quantize computation in all GEMMs. 170
7.7 Inference: Mistral-7B and Phi-2 when we quantize computation in all linear layers. . . . 170
7.8 Training: Validation log perplexity of RoBERTa. . . . 172
7.9 Training: Validation top-1 accuracy of ViT-Small. . . . 172
7.10 Validation metrics of T5-Large finetuning on 1/4 of XSum dataset for 1 epoch. . . . 173
7.11 Maximal ratios between the maximum and 95-percentile of magnitudes of each matrix involved in GEMMs. . . . 176
7.12 Maximal ratios between the maximum and 95-percentile of magnitudes of each matrix involved in GEMMs during the training of RoBERTa-Small.176
7.13 Catastrophic performance degradation when restricting outliers to a representable range of quantized domain or clipping the outliers . . . 176
7.14 Averaged unpack ratios of each type of GEMMs in LLaMA-7B 187
7.15 Averaged unpack ratios of each type of GEMMs in ViT-Large 188

7.16 Averaged unpack ratios of each type of quantized GEMMs in both forward and backward of a RoBERTa-Small 188
7.17 Averaged ratios of quantized GEMMs ($\beta = 15$) in linear layers on ViT-Large 189
7.18 Inference: end to end baseline comparison combining information from Table 7.3 and Table 7.14 for LLaMA-7B. 190
7.19 Runtime overhead of UnpackColumn and UnpackRow compared to GEMM and Clone 191
7.20 Inference: estimated speedup of the entire model for LLaMA-7B. 192

LIST OF FIGURES

1.1 Applications of the Transformer architecture on various domains. . . . 2
1.2 High level overview of the Transformer architecture. 5
1.3 Illustration of how the encoder processes an input sequence. 5
1.4 Structure of an encoder block. 6
1.5 High level illustration of self-attention. 6
1.6 Structure of an decoder block 8
1.7 Overall scope of this thesis. 16

2.1 Illustration of transforming different data to the input of Transformer models 23
2.2 Detailed computation within a multi-head attention module. 25
2.3 Illustration of fast Hadamard transform algorithm 28
2.4 An example of using the Monte Carlo Method for estimating π 33
2.5 Illustration of the nested structure of the wavelet transform. 37

3.1 Comparison between collision probability and exponential function . . 47
3.2 A high-level overview of YOSO-Attention 49
3.3 Overview of YOSO-Attention algorithm 51
3.4 MLM and SOP result of 512 sequence length language model pretraining 58
3.5 MLM and SOP result of pretrained pretraining when altering the number of hashes in inference. 59
3.6 Running time and memory consumption results on various input sequence length 63
3.7 Averaged Radian between outputs of YOSO-E and YOSO-m 64
3.8 Attention matrices generated by Softmax and YOSO 64

4.1 Issues of LSH 69
4.2 High level comparison of each method 72
4.3 Illustration of structure matrix $\mathbf{S}$ for $\tau = 3$ 73
4.4 Illustration of LookupFFN operations. 74

4.5 Approximation capacity vs FLOPs and parameters for each efficient projections . . . 77
4.6 Visualization of efficient projections . . . 78
4.7 Future performance opportunities of LookupFFN . . . 86

5.1 Visualization of MRA approximation for self-attention matrices . . . 89
5.2 Visualization of MRA components . . . 91
5.3 Illustration of our MRA approximation scheme in log scale . . . 100
5.4 Illustration of our MRA approximation scheme in linear scale . . . 100
5.5 Theoretical Workload vs Approximation Error . . . 106
5.6 Comparison between optimal sparsity and block sparsity found by our method . . . 108
5.7 Approximation Error versus Runtime versus Memory . . . 110
5.8 Entropy versus Approximation Error . . . 111

6.1 Model efficiency of processing one sequence . . . 118
6.2 Diagram that illustrates a Transformer layer with VIP-token centric sequence compression. . . . 122
6.3 Illustration of multi-resolution compression . . . 127
6.4 Visualization of $\mathbf{b}_{s,x}$ for a specific scaling s and translation x. . . . 128
6.5 An example of approximating an 1D signal using a truncated wavelet transform . . . 129
6.6 Proposed data structure $\mathcal{T}(\mathbf{C})$ . . . 140
6.7 Model runtime vs WikiHop dev accuracy when using different model specific hyperparameters . . . 149

7.1 Overall illustration of this chapter . . . 164
7.2 Training: Comparison of RoBERTa loss curves. . . . 171
7.3 Training: Comparison of ViT-Small . . . 172
7.4 Loss curves of T5-Large finetuning on 1/4 of XSum dataset for 1 epoch. 173
7.5 Illustration of unpacking row vectors . . . 179
7.6 Failure cases for different unpacking strategies . . . 181

7.7 Illustration of unpacking column vectors 182
7.8 Illustration of unpacking both rows and columns based on the OOB counts 183
7.9 Illustration of the bit representation of a matrix 193

ABSTRACT

Deep learning has been widely applied in various fields, such as understanding image content, analyzing text semantics, and modeling biological and medical data, among others. The introduction of the Transformer architecture in 2017 represented a significant advancement in deep learning due to its adaptability across different domains and its scalability. This marked the beginning of the era of large language models and foundational models across various domains. The widespread success of Transformer models can be attributed to two main factors: their ability to process massive context effectively and their scalability of model size. However, these advantages also introduce new efficiency challenges related to computational burden and environmental impact. This thesis focuses its main efforts on developing effective and efficient methods for improving Transformer's efficiency and making its adoption more environment-friendly, cost-efficient, and accessible for everyone. The computation within a Transformer model can be broken down and abstracted into three levels: Transformer blocks, Multi-Head Attention (MHA) and Feed-Forward Network (FFN) that comprise these blocks, and the GEneral Matrix Multiply (GEMM), which forms the foundation for all compute-intensive operations. We explore various methodologies to improve efficiency at each abstraction level, including the approximation of the Transformer blocks, MHA computation, FFN, and the fundamental GEMM operation. Directly applying these methodologies can often be sub-optimal and introduce specific challenges when designing efficiency algorithms for the specific compute abstractions of Transformer models. In this thesis, we analyze these challenges and develop specific algorithms to address these problems, and show the effectiveness and efficiency of our algorithms. These algorithmic improvement significantly reduce the costs associated with training and deploying Transformer models, enabling Transformer models to comprehend massive context that are previously infeasible and to run faster with smaller memory footprint and to deploy on cheaper devices with limited computational resources.

1 INTRODUCTION

Deep learning methods now power applications across a range of fields, including image content understanding, text semantic analysis, and biological and medical data modeling. Such tools have demonstrated remarkable effectiveness in tasks that were traditionally exclusive to human. In a landmark event in 2012, AlexNet (Krizhevsky et al., 2012) won the ImageNet image recognition competition (Russakovsky et al., 2015). This deep convolutional neural network (CNN) significantly outperformed existing methods and showcased the potential of deep learning for computer vision problems. Following AlexNet, we witnessed a rapid development of deep learning architectures in the mid-2010s. Models like VGGNet (Simonyan and Zisserman, 2015), GoogLeNet (Szegedy et al., 2015), and ResNet (He et al., 2016) brought new ideas enhancing the depth and complexity of neural networks and setting new state of the art accuracy for understanding of visual content. Concurrently, the field of natural language processing (NLP) saw substantial progress with recurrent neural networks (RNNs) (Elman, 1990) in handling language-related tasks (Tang et al., 2015; Cho et al., 2014). In 2017, the introduction of the Transformer architecture (Vaswani et al., 2017) marked a paradigm shift in deep learning in terms of its adaptability to different domains as well as its scalability (illustrated in Figure 1.1), and initiated the era of large language models (LLMs) and large foundational models for various domains.

Prior to Transformer models, the NLP domain was dominated by RNNs. RNNs sequentially read one word of an input sentence at each time step and update their understanding of the sentence. In contrast to the sequential processing nature of RNNs, the unique core module of the Transformer architecture, Multi-Head Attention, made it both effective for modeling linguistic content and highly parallelizable on modern hardware by evaluating the entire sentence at once. Although the Transformer was originally introduced as a novel method for language translation tasks (Vaswani et al., 2017), the core attention mechanism not only enhances the speed and performance of language translation models but also opens up new

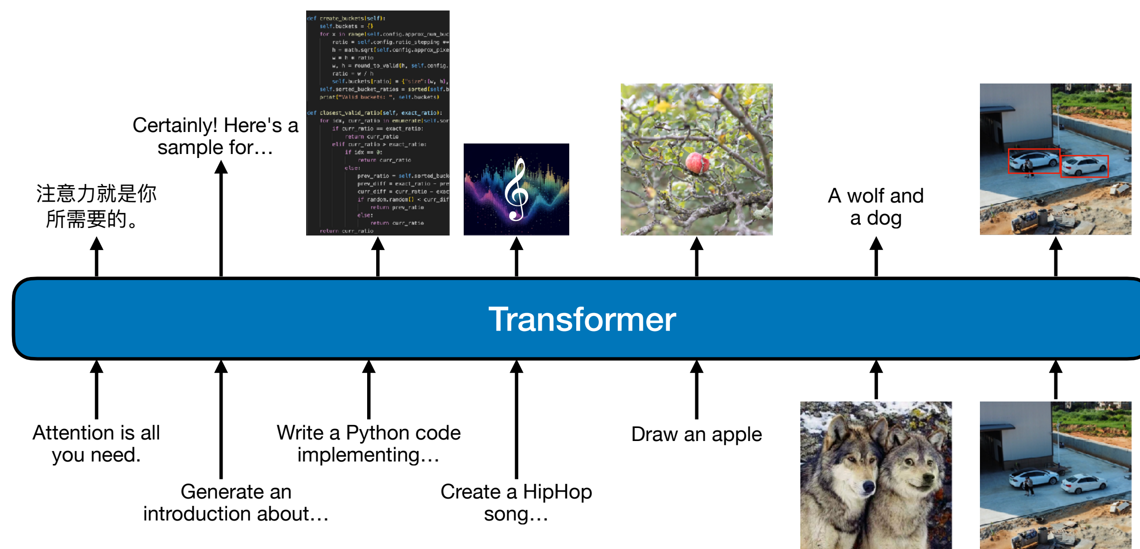

Figure 1.1: Applications of the Transformer architecture on various domains.

opportunities in processing and comprehending complex linguistic data at a much larger scale.

Subsequent to the initial success, the Transformer architecture was adapted for broader applications within the NLP domain. Specifically, the development of the BERT language model (Devlin et al., 2019) led to a significant advancement in the field of NLP. By utilizing a two-stage approach, pretraining via a method called bidirectional masked language modeling on massive text datasets in order to understand the text/language followed by finetuning on downstream NLP applications with limited data, this approach enabled BERT to achieve outstanding effectiveness in a variety of NLP tasks, including language inference, comprehension, and question answering, to name a few (Devlin et al., 2019). This success of BERT subsequently ignited a surge of explorations and innovations within the Transformer architecture for more effective and robust NLP approaches, including improved pretraining and better formulation of diverse NLP tasks. Notable examples of enhancing pretraining include RoBERTa (Liu et al., 2019a) and XLNet (Yang et al., 2019), which build upon and refine BERT's pretraining task. In terms of reformulating various NLP tasks, a notable example is T5 (Raffel et al., 2020), which

adopts a unified approach to handling a diverse range of language-related challenges by formulating all tasks as sequence-to-sequence generation tasks, resulting in a deployment for downstream applications.

Contrasting with bidirectional masked language modeling, OpenAI's GPT series explored an alternative direction: Transformer-based autoregressive language modeling (Radford et al., 2018, 2019; Brown et al., 2020; OpenAI et al., 2023) and led to a significant discovery: the impressive zero-shot performance of large language models (LLMs) (Brown et al., 2020; Touvron et al., 2023; Le Scao et al., 2023; Team et al., 2024) in a wide range of NLP tasks. By using massive compute resources and data in language model pretraining, these models often show human level proficiency in a wide range of NLP tasks, including generating coherent text, solving complex problems, or even programming code. This proficiency in language understanding is transformative and has sparked a global competition in the development of LLMs.

Inspired by the achievements of Transformer models in NLP, a significant development in the field of vision emerged with the introduction of the Vision Transformer (ViT) (Dosovitskiy et al., 2021), marking the first successful application of the Transformer architecture to vision tasks. This advancement led to a rapid and widespread integration of Transformer models within various aspects and problem settings in computer vision. These models began to excel and often outperformed CNNs in a variety of diverse tasks besides image classification including object detection, image segmentation, and generation of visual content. Examples of these tasks includes Detection Transformer (DETR) (Carion et al., 2020) for object detection, Segment Anything Model (SAM) (Kirillov et al., 2023) for image segmentation, and Diffusion Transformer (DiT) (Peebles and Xie, 2023) for image generation. Recently, OpenAI released an impressive video generative model, SORA (Brooks et al., 2024), which is built upon DiT. SORA demonstrates the capacity of Transformers to generate visual content with high fidelity and creativity. The efficacy demonstrated by Transformer models in handling visual information showcased their potential as a universal architecture suitable for various data domains.

Given the success of Transformer models in both NLP and vision, it was a natural

progression for the research community to explore their application across other modalities and domains. For audio processing, Transformer models have shown promise in tasks like speech recognition (Latif et al., 2023) and music generation (Suno-AI, 2023). For biomedical sciences, Transformer models have been applied to tasks like protein and genomic modeling (Choromanski et al., 2021; Choi and Lee, 2023). The extension of Transformer models to these diverse domains is not only a testament to their adaptability but also highlights a broader trend in deep learning towards more unified and generalized approaches across multiple modalities.

1.1 Transformer Architecture

To better understand how Transformers function, we need to dive into some technical details of a Transformer model. Originally, the Transformer architecture consists of two primary components: the encoder and the decoder (Vaswani et al., 2017), as shown in Figure 1.2. Each serves a unique role in handling sequence-to-sequence tasks, such as language translation. However, as the applications of Transformer models evolved, many models began to use either the encoder or the decoder exclusively. For instance, models like BERT (Devlin et al., 2019), ViT (Dosovitskiy et al., 2021), and DiT (Peebles and Xie, 2023) utilize only the Transformer encoder. This encoder is responsible for computing the contextual embeddings of input sequences for subsequent downstream tasks. In contrast, GPT (Brown et al., 2020), LLaMA (Touvron et al., 2023), and most contemporary LLMs employ the Transformer decoder for autoregressive text generation tasks. The decoder is similar to the encoder with a few minor differences, so we will first focus on the encoder's function before highlighting the differences with the decoder.

The encoder processes the input as a sequence of tokens, each represented by a vector. We denote the sequence of tokens as a matrix $\mathbf{X}$ encoding the input's information. The row vectors of $\mathbf{X}$ correspond to tokens in the sequence. These tokens are transformed into a sequence of contextual embeddings by passing through a stack of Transformer encoder blocks with an identical structure, as illustrated

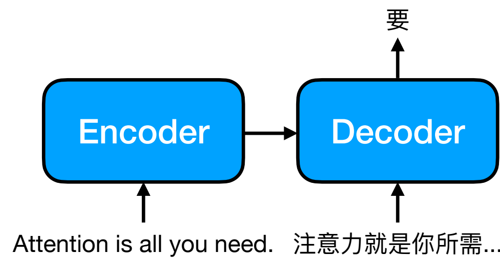

Figure 1.2: High level overview of the Transformer architecture.

in Figure 1.3. Since the Transformer model is invariant to the order of sequence, we need a way to inform the model about the location of each token within the sequence. So, a positional embedding, $\mathbf{E}$, encoding the location of each token, is added to $\mathbf{X}$ to form the input $\mathbf{X}_0$ to the first encoder block. The first encoder block then outputs $\mathbf{X}_1$, which is the input to the next encoder block. The process continues until we get the output $\mathbf{X}_\mathrm{L}$ of the last encoder block. Here, $\mathbf{X}_\mathrm{L}$ is the contextualized output of the encoder, which will be used for subsequent downstream tasks.

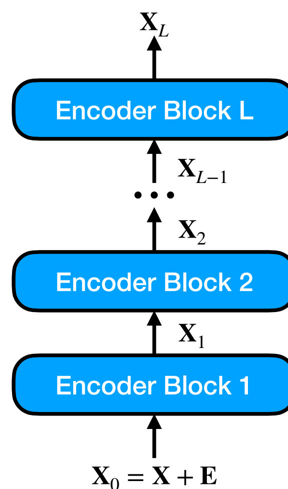

Figure 1.3: Illustration of how the encoder processes an input sequence.

The internal structure of an encoder block consists of two components: a multi-head attention (MHA) mechanism and a pointwise feedforward network (FFN), which will be discussed in more details in §2.2. The overall structure of an encoder block is illustrated in Figure 1.4. Now, we can describe MHA and FFN.

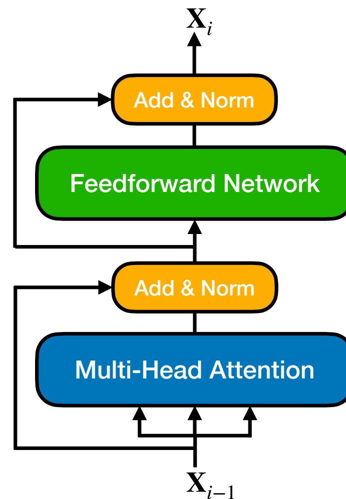

Figure 1.4: Structure of an encoder block.

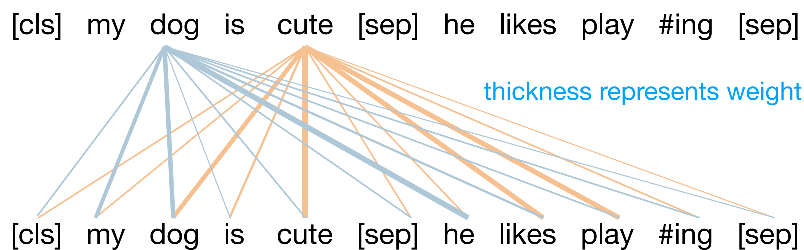

Figure 1.5: High level illustration of self-attention.

At the i-th encoder block, the encoder block first feeds X_{i-1} into a MHA module. MHA consists of multiple self-attention in parallel and is central to Transformer based models and provides a flexible global receptive field to exchange information among input tokens. The output of self-attention is a combination of all tokens where coefficients are determined by the similarities among tokens, as demonstrated in Figure 1.5. Compared to a RNN, self-attention allows all pairwise dependencies among tokens to be computed and aggregated in parallel, which is much more suitable to run on a GPU. Further, the length of signal paths [1] between any two tokens are simply $O(1)$, making it easier for Transformers to learn long-range

[1]The path of the information of a token to be propagated to another token. For example, for a sequence of N tokens, in RNN, the information of the first token needs at minimum a $O(N)$ length path to be propagated to the last token.

dependencies (Vaswani et al., 2017). Then, the output of MHA is fed into two-layer FFN to compute a pointwise non-linear transformation of each token embeddings. An encoder consists of multiple such encoder blocks with an identical structure, which make it easier to scale the width and depth of a Transformer model with very few additional design choices.

The decoder shares a similar structure with the encoder. The decoder consists of multiple decoder blocks. As illustrated in 1.6, at each block, the input is fed into a multi-head causal attention, which is essentially a MHA but with an additional mask applied to the attention matrices $\mathbf{A}_h$ to avoid leakage of information from future tokens, which is critical for autoregressive training. Then, when the decoder is used along with the encoder, the intermediate output is fed into a multi-head cross attention using the intermediate output as queries and output embeddings $\mathbf{X}_L$ from the encoder as keys and values so that the decoder tokens can gather useful information from the encoder output for sequence-to-sequence tasks. Finally, the output is fed into a FFN for a pointwise non-linear transformation of token embeddings. In this thesis, our focus is on the MHA and FFN in the encoder, and many of the findings described in the subsequent chapters can be transferred to the decoder, so we will not dive into the details of the decoder.

1.2 On the Efficiency of Transformers

The success of Transformer models across various domains can be attributed to two key factors: **Ability to Process Sequences Effectively**, and **Scalability of Model Size**. These factors contribute to the success of Transformers, but also pose new challenges in terms of the compute burden and environmental concerns. In this section, we will discuss each of these factors and related works that try to address these challenges.

To better understand the challenges, we need to first know how much compute is needed to run a Transformer model. In §1.1, we described how an input is processed using a Transformer model. The overall compute complexity of a Transformer

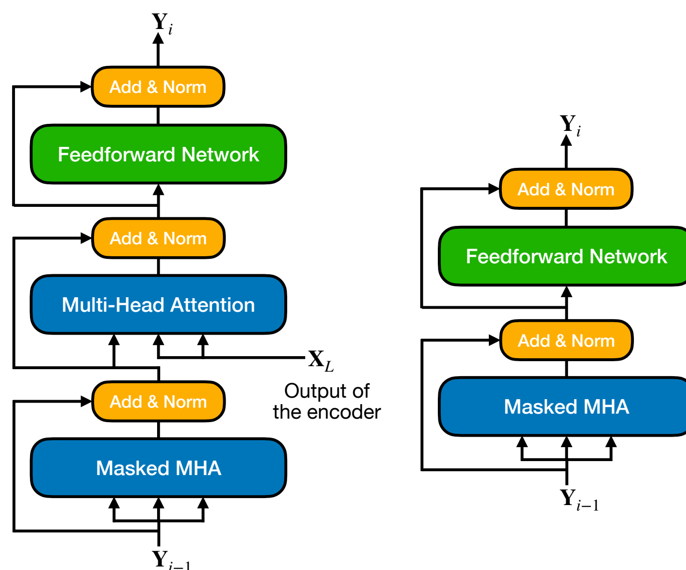

Figure 1.6: Structure of an decoder block. The left block is the decoder block used in encoder-decoder models, and the right block is the decoder block used in decoder-only models.

model is

$$O(LN^2D + LND^2)$$

where L is the number of Transformer blocks, D is the model dimension, and N is the sequence length of input. The first term is the complexity of computing the MHA module, and the second term refers to the cost of computing the FFN module.

1.2.1 Ability of Processing Sequences Effectively

The Transformer architecture is very effective in modeling sequences, evidenced by its superior performance on various domains. The superior ability of processing sequences effectively is due to its MHA module. The self-attention mechanism provides a flexible mechanism to exchange information among different tokens globally within a sequence, allowing the model to focus on different parts of the input data. The multi-head mechanism augments this ability by running multiple self-attention blocks in parallel, allowing each self-attention mechanism to attend

to different semantic embedding spaces for a more expressive receptive field. While MHA provides various benefits making it exceptionally good at understanding the input context, it is also a bottleneck when training and serving models involving long input sequences.

Each MHA incurs a $O(N^2D)$ complexity, which restricts its use in applications where capturing long term context dependencies is important. For example, the sequence length N can range from less than 32 to more than 512,000 for different applications. While we do not know the statistics of sequence lengths for all real world applications, the existing datasets for different tasks should give a reasonable estimate of the expected sequence lengths. For example, most tasks in GLUE (Wang et al., 2018a), a benchmark to analyze model performance in learning a wide range of linguistic phenomena, has most sequences with less than 100 tokens. In question answering settings, for example, WikiHop (Welbl et al., 2018) has more than a half of the sequences longer than 2,000 tokens. Further, in NarrativeQA (Kočiský et al., 2018), a question answering based on the content of a book, more than a half of the instances are longer than 50,000 tokens (with the longest being more than 512,000 tokens). Arxiv (Cohan et al., 2018), a task of generating a summary of a scientific article, has instances whose lengths are more than 10,000 on average.

In the NLP setting, many current models have certain constraints on the sequence length, e.g., BERT, T5, and other transformer-based language models (Yang et al., 2019; Liu et al., 2019a; Raffel et al., 2020) limit the sentence length to be at most 512, for example, via truncating the sequence and only keeping the first 512 tokens. Of course, this would result in severe loss of accuracy due to incomplete or partial information. Recent results have reported longer sequence significantly improve model performance even with weaker attention mechanisms (Beltagy et al., 2020; Zaheer et al., 2020; Guo et al., 2022). This quadratic cost has motivated many ongoing efforts to mitigate the resource needs of such models.

One line of work addressing the efficiency of long sequence modeling is efficient self-attention. For example, Linformer (Wang et al., 2020) shows that using a low-rank assumption, self-attention can be approximated via random projections

along the sequence length dimension. The authors replace random projections with learnable linear projections and achieve a $O(ND)$ complexity via a fixed projection dimension. For example, linear Transformers (Katharopoulos et al., 2020) replace the softmax activation by applying a separable activation on the queries and keys. Based on the connection between the softmax activation and the Gaussian kernel, Performer (Choromanski et al., 2021) and Random Feature Attention (Peng et al., 2021) approximate softmax as the dot product of finite dimensional random feature vectors with a guarantee of convergence (to softmax). Nyströmformer (Xiong et al., 2021) and SOFT Lu et al. (2021), on the other hand, uses a landmark-based Nyström method to approximate the attention matrices. These methods achieve $O(ND)$ complexity by avoiding direct calculation of attention matrices.

Multiple approaches have been developed to exploit sparsity and structured patterns of attention matrices observed empirically. This line of work includes Sparse Transformer (Child et al., 2019), Longformer (Beltagy et al., 2020), and Big Bird (Zaheer et al., 2020), which involve complexity of $O(N\sqrt{N}D)$ or $O(ND)$. The Reformer method (Kitaev et al., 2020) also utilizes the sparsity of self-attention. But instead of predetermining a sparsity pattern, it uses Locality Sensitive Hashing (LSH) as a tool to approximate nearest neighbor search, and dynamically determines the sparsity pattern to achieve $O(N\log(N)D)$ complexity.

Later developments of efficient self-attention move away from low rank or sparsity based construction. Chen et al. (2021a) suggests that approximations relying solely on low rank or sparsity are limited and a hybrid model via robust PCA offers better approximation. Scatterbrain (Chen et al., 2021a) uses a sparse attention + low rank attention strategy to avoid the cost of robust PCA. H-Transformer-1D (Zhu and Soricut, 2021) proposes a hierarchical self-attention where the self-attention matrices have a low rank structure on the off-diagonal entries and attention is precisely calculated for the on-diagonal entries.

When accounting for the method specific hyperparameters controlling the trade-off between quality and efficiency of these efficient mechanisms, most of these self-attention mechanisms have a complexity of $O(NMD)$ where M is a method specific

hyper-parameter. With the development of efficient self-attentions, Transformers have been applied to tasks with longer sequences, which were previously infeasible. Impressive performance has been demonstrated (Beltagy et al., 2020; Zaheer et al., 2020; Guo et al., 2022).

There are also alternative approaches that allow Transformer models to process long sequences without quadratically increasing the compute cost by compressing prior context into a fixed size memory in autoregressive setting. For example, Memorizing Transformers (Wu et al., 2022) and RMT (Bulatov et al., 2022) follow a recurrent design and store the past context in an external memory module. By sequentially processing one segment of input sequences at one time, they avoid blowing up the memory when processing long sequences.

One distinct line of work to improve the compute efficiency of self-attention is a faster implementation of exact self-attention computation but without reducing the quadratic complexity of self-attention. For example, by exploiting the benefit of minimizing global memory IO, FlashAttention (Dao et al., 2022) proposes a fused self-attention computation CUDA kernel that greatly accelerates the self-attention computation and reduces memory usage. RingAttention (Liu et al., 2024) describes a ring network communication pattern that allows distributing self-attention across multiple devices without increasing network bandwidth requirements.

1.2.2 Scalability of Model Size

The Transformer architecture can be easily scaled up by setting up proper number of layers L and model dimension D due to the identical structure in every layer. And the recent advance in Transformer-based LLMs and vision foundation models significantly benefits from this scaling up: larger models typically demonstrate superior performance, and as the size of the model increases, so does its performance on various applications and benchmarks (Radford et al., 2019; Brown et al., 2020; OpenAI et al., 2023; Dosovitskiy et al., 2021; Dehghani et al., 2023). This trend has motivated the increase in model sizes. For instance, in 2018, the large model of BERT (Devlin et al., 2019) included $L = 24$ layers with a dimension of $D = 1024$, by

2020, GPT-3 (Brown et al., 2020) expanded to $L = 96$ layers with a dimension of $D = 12288$.

While the research on Transformer models has consistently shown that increasing the size of the model improves performance across various domains, a more groundbreaking discovery by OpenAI, the scaling law of LLMs (Kaplan et al., 2020), takes this a step further. It reveals that as both model size and dataset size increases, we can expect a predictable model performance improvement on language modeling. This insight motivated an exponential growth in the sizes of LLMs. Additionally, the phenomenon of emergent abilities in LLMs (Wei et al., 2022) reveals that once the model size surpasses a certain unpredictable threshold, new abilities, such as solving few shot prompted problems, starts to emerge in larger models, capabilities that smaller models do not exhibit. This further reinforces the value of and the interest in developing larger language models.

Motivated by the success of the GPT series and the potential ability of LLMs, these discoveries initiated a global competitive race among technology companies and AI startups to develop and refine large language models. Major players in the tech industry, as well as emerging AI startups, are investing heavily in the development of these LLMs. This race is characterized by an exponential increase in the model sizes of newly released language models. For instance, the GPT series has progressively increased in model size, with GPT-1, GPT-2, and GPT-3 having 0.1 billion, 1.5 billion, and 175 billion parameters, respectively (Radford et al., 2018, 2019; Brown et al., 2020). While the exact size of GPT-4 is not confirmed, it is speculated to exceed 1 trillion parameters (LeCun, 2023). The information about open-sourced LLMs is more accessible. The publicly available LLMs like the LLaMA series feature parameter counts of 7 billion, 13 billion, and 70 billion (Touvron et al., 2023), whereas BLOOM has 176 billion parameters (Le Scao et al., 2023).

Similar to the developments in LLMs, the field of vision also witnessed a parallel trend with vision foundational models following a trajectory of increasing model sizes, leading to new state of the art results. For example, while the base model in

the initial proposal of ViT has only 86 million (Dosovitskiy et al., 2021) parameters, later development pushed the model sizes to 2 billion (Zhai et al., 2022) and even 22 billion (Dehghani et al., 2023) parameters.

As the model size grows larger, the amount of compute required to train this model and serve such models increases as well. The training of these large models is expensive. When using the most advanced GPU, NVIDIA's A100s, the training of LLaMA2 series takes 184320, 368640, 1038336, and 1720320 GPU hours, respectively for LLaMA 7B, 13B, 34B, and 70B (Touvron et al., 2023). A total of more than 3 million GPU hours for LLaMA2 series training correspond to running 4096 NVIDIA's A100s for more than a month and a carbon footprint of 539 tons Carbon dioxide equivalent (tCO2eq). OpenAI (Brown et al., 2020) provides an estimate of 3,640 petaflop/s-days for training GPT-3, which corresponds to running thousands of NVIDIA's A100s for more than two weeks for GPT-3 training, while noting that the less compute capable V100s were the mainstream for deep learning at the time when GPT-3 was released. The estimated financial cost is over $4 million (Vanian and Leswing, 2023). For the next generation GPT-4, while the estimate of the required compute is unknown, OpenAI's CEO Sam Altman claims that GPT-4 costs over $100 million (Knight, 2023). The training of these large models is only part of the compute story. When serving these LLM to the public, the inference cost is even more expensive. Some reports (Patel and Ahmad, 2023) estimate that the cost of serving GPT-3 is roughly $0.7 million per day, and OpenAI requires close to 30,000 A100s for serving these models. As the number of users increases, the inference cost increases linearly as well.

With this exponential growth in compute requirements, the hardware infrastructure to provide the required substantial computing capacity is critical. The hardware community has consistently advanced the development of faster computing devices to accommodate the growing computational demands of large models. For instance, consider the evolving compute capabilities of NVIDIA's flagship GPUs across generations: V100, A100, H100, and the upcoming B100. For 16-bit floating-point matrix multiplication, the V100, released in 2017, achieves 125 teraFLOPs (NVIDIA,

d). This capacity significantly increased with the A100 in 2020, which delivers 312 teraFLOPs and 624 teraFLOPs with sparsity (NVIDIA, a), followed by the H100 in 2022 with 990 teraFLOPs and 1979 teraFLOPs with sparsity (NVIDIA, c). The planned B100, set for release in 2024, is expected to reach 3.5 petaFLOPs with sparsity (NVIDIA, b). The FLOPs of B100 for dense matrix multiplication is unknown, but based on the FLOPs for A100 and H100, the estimate is 1.75 petaFLOPs. Alongside the advancements in computing speed, unfortunately there has been an increase in both power consumption and cost for each generation of GPU. For example, the power usage for the V100, A100, and H100 GPUs stands at 350, 400, and 700 watts, respectively (NVIDIA, d,a,c). And the current retail price for A100 and H100 is around $10,000 and more than $30,000 per GPU (Leswing, 2023a,b). While the rapid improvements in computing speed are impressive, a closer look at these numbers reveals that the increasing trend of the teraFLOPs will start to saturate.

Maintaining the cutting-edge computing infrastructure is increasingly becoming both an environmental and financial challenge. While players in the tech industry have the financial and human resources to compete in upgrading their computing infrastructure and developing state-of-the-art models, those in the academic community are significantly falling behind. This disparity is creating a gap where academic institutions struggle to keep pace with the rapid development, affecting their ability to contribute to and participate in research and development.

Many approaches have been developed to mitigate the limits on compute hardware via architecture adjustments or compute adjustments. For example, the mixture of experts (MoE) approach is becoming increasingly popular as this approach allows increasing parameter count without increasing compute requirement via adaptive computations. Switch transformer (Fedus et al., 2022) uses a MoE based feedforward network that routes different tokens to different experts and expands parameters to 1 trillion. Mixtral (Jiang et al., 2024) uses a similar MoE approach to built a relatively light weight LLM in term of compute that performs well on most benchmarks compared to larger and heavier LLM competitors. Another approach

for improving efficiency without changing the hardware is adjusting the algorithms for computing GEMM. There are many ongoing methods seeking to improve low precision computation to approximate the full precision GEMMs. Most of them focus on serving these large models during inference, while a few of them target speeding up the training (Banner et al., 2019; Nagel et al., 2019; Kim et al., 2021; Dettmers et al., 2022; Li and Gu, 2023; Xiao et al., 2023; Li et al., 2023a; Wang et al., 2018b; Wu et al., 2018; Zhu et al., 2020; Wortsman et al., 2023). One of the most impactful adaptations is a transition from 32-bit floating point training to 16-bit floating point (Micikevicius et al., 2018), and 16-bit training has became a norm in the deep learning community. There are also other approaches such as pruning (Chen and Zhao, 2019) for speeding up the computation.

In this thesis, we will explore these areas of research and spot any opportunities that we can exploit to improve Transformers' efficiency.

1.3 Contribution and Scope of Thesis

This thesis focuses its main efforts on developing effective and efficient methods for improving Transformer's efficiency and making its adoption more environment-friendly, cost-efficient, and accessible for everyone.

In this thesis, we improve the efficiency of Transformer models by improving efficiency of different (**a**) components or (**b**) compute abstractions of an encoder block, which can significantly reduce the cost of training and using large Transformer models and allow expanding the use of Transformer models on new applications that were previously infeasible. For example, these ideas can create opportunities for training and using large language models on consumer devices with limited memory, computation, and scalability. Also, the formulations can enable addressing complex problems involving significant amount of information by fully exploiting Transformer models' ability of capturing information from a very long context. The contribution of this thesis is to design efficient algorithms for accelerating the com-

putation of Transformer models for training and inference for diverse applications. Our contributions of Transformer models' efficiency can be categorized into three methodologies: memory lookup based approximations, multi-resolution based approximation, low precision integer computation. The overall computation of a Transformer block can be abstracted into three levels: a Transformer block is built upon MHA and FFN, and MHA and FFN are built upon GEneral Matrix Multiply (GEMM). We explore the possibility of applying the aforementioned methodologies to different abstractions: approximating Transformer encoder block as a whole, approximating MHA computation, approximating FFN, and approximating the basic operator of Transformer: GEMM, as shown in Figure 1.7. From the list of methodologies and abstractions, we now give a brief background to the specific problems that we attack, identify their specific challenges, and present how we will deal with them later in each chapter.

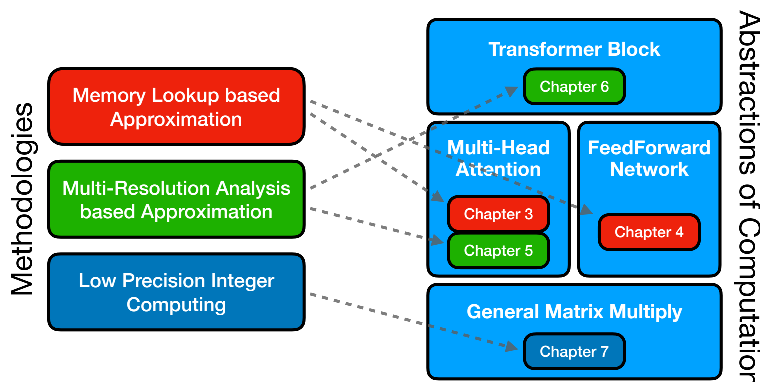

Figure 1.7: Overall scope of this thesis.

1.3.1 Chapter 3. Memory Lookup based Approximation for Efficient Self-Attention

Central to the transformer model is the MHA mechanism, which captures the interactions of token pairs in the input sequences and depends quadratically on the sequence length. Training such models on longer sequences is expensive as

discussed above in §1.2.1, so this quadratic cost limits Transformer models' use in applications that requires long context understanding.

In Chapter 3, we show that a Bernoulli sampling attention mechanism based on Locality Sensitive Hashing (LSH), effectively decreases the quadratic complexity of such models to linear. We bypass the quadratic cost by considering self-attention as a sum of individual tokens associated with Bernoulli random variables that can, in principle, be sampled at once by a single hash (although in practice, this number may be a small constant). This leads to an efficient sampling scheme to estimate self-attention which relies on specific modifications of LSH to enable deployment on GPU architectures and relies mostly on memory lookup operation with minuscule compute.

We evaluate our algorithm on the GLUE benchmark (Wang et al., 2018a) with the standard 512 sequence length where we see favorable performance relative to vanilla Transformer models. On the Long Range Arena (LRA) benchmark (Tay et al., 2021), for evaluating performance on long sequences, we find our method achieves results consistent with vanilla self-attention but with sizable speed-ups and memory savings and often outperforms other efficient self-attention methods, demonstrating the scalability of this approach.

1.3.2 Chapter 4. Memory Lookup based Approximation for Compute-lite Feedforward Network

The GEMM based FFN is one of the two primary components within a Transformer encoder block, and also a workhorse within modern deep neural networks, but it requires heavy computation. While efficiency on the compute front has tapered out, memory efficiency per watt is expected to continue to get better over the coming years creating efficiency opportunities for memory lookup based algorithms.

In Chapter 4, we assess the extent to which FFN can be made compute-(or FLOP-) lite using similar memory lookup algorithms described in Chapter 3 and explore the possibility of deploying this algorithm on CPUs. Specifically, we propose an

alternative formulation (we call it LookupFFN) to GEMM based FFNs inspired by the recent studies of using LSH to approximate FFNs. Our formulation recasts most essential operations as a memory look-up, leveraging the trade-off between the two resources on any platform: compute and memory. By viewing table look-ups as a selection problem and approximating it via a softmax whose logits is a structured transformation of the input, this memory look-up based FFN can be computed efficiently and is fully backpropagatable.

For performance comparison, our formulation achieves similar performance compared to GEMM based FFNs in RoBERTa language model pretraining, while dramatically reducing the required FLOPs. Further, our development is complemented with a detailed hardware profiling of strategies that will maximize efficiency – not just on contemporary hardware but on products that will be offered in the near/medium term future.

1.3.3 Chapter 5. Multi-Resolution based Approximation for Efficient Self-Attention

Developing better and efficient self-attention mechanisms is an active research area. Recent efforts including our own on training and deploying Transformer models more efficiently have identified many strategies to approximate the self-attention. In Chapter 3, we described a memory lookup based approach for efficient self-attention. Other effective ideas include various prespecified sparsity patterns and low-rank basis expansions.

In Chapter 5, motivated by the multi-resolution structure presented when visualizing self-attention matrices, we revisit classical Multi-Resolution Analysis (MRA) concepts such as Wavelets, whose potential value in this setting remains underexplored thus far. We show that simple approximations eventually yield a MRA-based approach for self-attention with an excellent performance profile across most criteria of interest.

We undertake an extensive set of experiments and demonstrate that this multi-

resolution scheme yield accurate approximations to the ground-truth self-attention matrices and outperforms most efficient self-attention proposals and is favorable for both short and long sequences.

1.3.4 Chapter 6. Multi-Resolution based Approximation for Faster Transformer Blocks

Despite recent works devoted to reducing the quadratic cost $O(LN^2D)$ of MHA computation in the last few years, including our results on Chapter 3 and 5, we find this is not sufficient to run Transformer models on ultra long sequences (e.g., $>$16K tokens) efficiently since the linear cost (with respect to N) of the FFN module can also be expensive. Applications such as answering questions based on a book or summarizing a scientific article, which require Transformer models to look at the entire book or article, are inefficient or infeasible. This limitation undermines Transformers' ability to learn long range dependencies.

In Chapter 6, we propose to approximate the computation of the entire Transformer block to significantly improve the efficiency of Transformers for ultra long sequences, by compressing the sequence into a much smaller representation at each layer. Specifically, by exploiting the fact that in many tasks, only a small subset of special tokens, which we call VIP-tokens, are most relevant to the final prediction, we develop a VIP-token centric compression (VCC) scheme which selectively compresses the sequence based on their impact on approximating the representation of the VIP-tokens via the inspiration of Multi-Resolution approximation that we exploited in Chapter 5.

Compared with competitive baselines, our algorithm is not only efficient (achieving more than $3\times$ compute efficiency gain compared to baselines on 4K and 16K lengths), but also offers competitive/better performance on a large number of tasks. Further, we show that our algorithm scales to 128K tokens (or more) while consistently offering accuracy improvement.

1.3.5 Chapter 7. Low Precision Integer Computation for General Matrix Multiply

The majority of operations in a Transformer model relies on the basic operation GEneral Matrix Multiply (GEMM). GEMM is also a central operation in deep learning and corresponds to the largest chunk of the compute footprint. Therefore, improving its efficiency is an active topic of ongoing research. A popular strategy is the use of low bit-width integers to approximate the original entries in a matrix. This allows efficiency gains, but often requires sophisticated techniques to control the rounding error incurred.

In Chapter 7, we first verify that if the low bit-width restriction is removed, for a variety of Transformer models, integers are, in fact, sufficient for all GEMMs needed – for *both* training and inference stages, and achieve parity compared to floating point counterparts. No sophisticated techniques are needed. We find that while a large majority of entries in matrices (encountered in such models) can be easily represented by *low* bit-width integers, the existence of a few heavy hitter entries makes it difficult to achieve efficiency gains via the exclusive use of low bit-width GEMMs alone.

To address this issue, we develop a simple algorithm, Integer Matrix Unpacking (IM-Unpack), to *unpack* a matrix with large integer entries into a larger matrix whose entries all lie within the representable range of arbitrarily low bit-width integers. This allows *equivalence* with the original GEMM, i.e., the exact result can be obtained using purely low bit-width integer GEMMs. This comes at the cost of additional operations – we show that for many popular models, this overhead is quite small.

Finally, in Chapter 8, we will summarize the contributions of the thesis and discuss the future directions of our research.

2 BACKGROUND

In this chapter, we define some common notations that will be used throughout the thesis. We will also cover some basic background knowledge that our development of efficiency algorithms relies on, Specifically, we will review relevant details regarding the Transformer architecture, efficient structured projection locality sensitive hashing, importance sampling, and multi-resolution analysis to facilitate our discussion of efficient computation of Transformer models.

2.1 Notations

In this section, we will define some notation conventions that will be frequently used in this thesis. We use bold uppercase letters to denote matrices ($\mathbf{A}, \mathbf{B}, \mathbf{C}, \cdots$) and bold lower case letters to denote vectors ($\mathbf{a}, \mathbf{b}, \mathbf{c}, \cdots$). For scalars, we use regular letters $a, b, c, \cdots$ and $A, B, C, \cdots$. The upper case scalars ($A, B, C, \cdots$) are used to denote the size of set, sequence, or dimension, while the lower case scatters ($a, b, c, \cdots$) are used to denote iterating indices. Further, we use $\mathbb{A}, \mathbb{B}, \mathbb{C}, \cdots$ to denote sets or spaces and $|\cdot|$ as the cardinality/size of input set. Also, we use $\mathcal{A}, \mathcal{B}, \mathcal{C}, \cdots$ to denote functions and mappings. We preserve $\mathbb{R}$ for the space of real numbers, $\mathbb{C}$ for the space of complex numbers, and $\mathbb{Z}$ for the space of integers.

Given a space $\mathbb{A}$ and scalar A, B, C, $\mathbb{A}^A$, $\mathbb{A}^{A \times B}$, and $\mathbb{A}^{A \times B \times C}$ correspond to the product space. For example, $\mathbf{A}$ in $\mathbb{A}^{A \times B}$ is an element from the product space and can be materialized as a matrix containing $A \times B$ entries.

For a matrix $\mathbf{A}$, we use $[\mathbf{A}]_{i,:}$ or $[\mathbf{A}]_i$ to denote the i-th row vector and $[\mathbf{A}]_{:,j}$ to denote the j-th column vector. Also, we use $[\mathbf{A}]_{i,j}$ to denote the (i, j) entry of matrix $\mathbf{A}$. Similar for a vector $\mathbf{a}$, we use $[\mathbf{a}]_i$ to denote the i-th entry of vector.

Given two D dimensional vectors $\mathbf{a}$ and $\mathbf{b}$, the inner product between $\mathbf{a}$ and $\mathbf{b}$ is

defined as

$$\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=1}^{D} [\mathbf{a}]_i [\mathbf{b}]_i$$

Given a vector, we always assume it is a column vector, and we might also use $\mathbf{a}^\top \mathbf{b}$ vector-vector multiplication to denote the same inner product. We also define the inner product between two matrices $\mathbf{A}$ and $\mathbf{B}$ of size $D \times T$ as

$$\langle \mathbf{A}, \mathbf{B} \rangle = \sum_{i=1}^{D} \sum_{j=1}^{T} [\mathbf{A}]_{i,j} [\mathbf{B}]_{i,j}$$

2.2 Transformers

In Chapter 1, we briefly described how a Transformer model processes the input into a contextualized output for subsequent tasks. In this section, we will describe the detailed computation of the Transformer encoder, which will be used throughout this thesis.

The encoder processes input as a sequence of tokens. We denote the sequence of tokens as a matrix $\mathbf{X}$ of size $N \times D$, encoding the input's information. Here, N is the number of tokens. For a more specific example, consider the sentence "Attention is all you need." as an input to a NLP model. This sentence is initially tokenized, converting each of the 6 words, including the period, into integers using a tokenizer, a dictionary mapping vocabulary words to unique integers. The NLP model then uses these integers as indices to retrieve corresponding vector representations from a D-dimensional embedding table, forming a matrix $\mathbf{X} \in \mathbb{R}^{6 \times D}$. In the context of a vision model, when an image is the input, it undergoes a different process. An image in $\mathbb{R}^{64 \times 64 \times 3}$ format (with 64 representing height and width, and 3 for the RGB channels) is converted into a tensor in $\mathbb{R}^{4 \times 4 \times D}$. This is achieved by extracting non-overlapping patches of $16 \times 16 \times 3$ and linearly projecting each patch to D-dimensional embeddings. The result is a matrix $\mathbf{X} \in \mathbb{R}^{16 \times D}$, formed by flattening the first two dimensions of these embeddings. We provide an illustration of the examples in Figure 2.1.

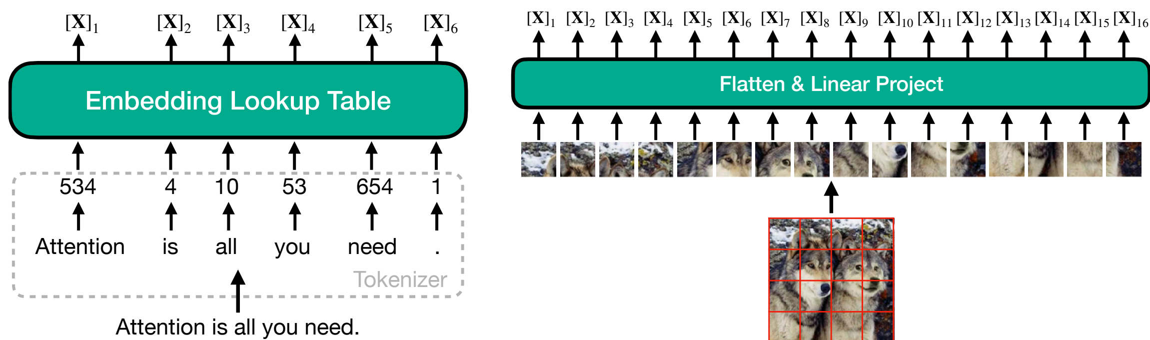

Figure 2.1: Illustration of transforming different data to the input of Transformer models. The left figure is for NLP models, and the right figure is for vision models.

Since the Transformer model is invariant to the order of sequence, which will be discussed later, we need a way to inform the model about location of each token within the sequence. So, a positional embedding, $\mathbf{E} \in \mathbb{R}^{N \times D}$ representing N D dimensional embeddings encoding location $1, 2, \cdots, N$, is added to $\mathbf{X}$ to form the input $\mathbf{X}_0$ to the first encoder block. Let $\mathcal{B}_i$ be the i-th encoder block in the encoder, then the output of each encoder block is computed as (as illustrated in Figure 1.3)

$$\begin{aligned}\mathbf{X}_0 &= \mathbf{X} + \mathbf{E} \\ \mathbf{X}_i &= \mathcal{B}_i(\mathbf{X}_{i-1})\end{aligned}$$

Then, for an encoder of L blocks, $\mathbf{X}_L$ is the contextualized output of the encoder, which will be used for subsequent downstream tasks.

The internal structure of an encoder block $\mathcal{B}_i$ consists of two components: a multi-head attention mechanism (MHA) and a pointwise feedforward network (FFN). Let us take a look at the detailed computation of i-th encoder block $\mathcal{B}_i$.

$$\begin{aligned}\mathbf{Y}_i &= \mathrm{LN}_1(\mathcal{A}(\mathbf{X}_{i-1}, \mathbf{X}_{i-1}, \mathbf{X}_{i-1}) + \mathbf{X}_{i-1}) \\ \mathbf{X}_i &= \mathrm{LN}_2(\mathcal{F}(\mathbf{Y}_i) + \mathbf{Y}_i)\end{aligned}$$

where $\mathcal{A}$ and $\mathcal{F}$ are MHA and FFN within the encoder block $\mathcal{B}_i$. And LN_1 and LN_2 are Layer Normalization (Ba et al., 2016), functionally similar to Batch Nor-

malization (Ioffe and Szegedy, 2015) in CNN. Subsequent studies showed shows that the placement of Layer Normalization was not optimal and proposed to apply these normalization schemes on the input of $\mathcal{A}$ and $\mathcal{F}$ (Vaswani et al., 2017; Xiong et al., 2020). Since this is not essential to the discussion of this thesis, in our later discussion, we omit these operations for a simplified notation.

$$\begin{aligned} \mathbf{Y}_i &= \mathcal{A}(\mathbf{X}_{i-1}, \mathbf{X}_{i-1}, \mathbf{X}_{i-1}) + \mathbf{X}_{i-1} \\ \mathbf{X}_i &= \mathcal{F}(\mathbf{Y}_i) + \mathbf{Y}_i \end{aligned} \tag{2.1}$$

The overall structure of an encoder block is illustrated in Figure 1.4. We can now start discussing the internal structure of MHA and FFN.

The encoder block first feeds X_{i-1} into a MHA module. The overall structure of MHA is shown in Figure 2.2. Let $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ be the queries, keys, and values embeddings corresponding to the three inputs of $\mathcal{A}$ The MHA output is defined as

$$\mathcal{A}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \sum_{h=1}^{H} \text{softmax}\left(\frac{\mathbf{Q}\mathbf{W}_{q,h}\mathbf{W}_{k,h}^{\top}\mathbf{K}^{\top}}{\sqrt{D_H}}\right)\mathbf{V}\mathbf{W}_{v,h}\mathbf{W}_{o,h} \tag{2.2}$$

where H is the number of attention heads, a hyper-parameter for Transformers. Here, $\mathbf{W}_{q,h}, \mathbf{W}_{k,h}, \mathbf{W}_{v,h}$ are common trainable linear layers projecting D dimensional vectors into a D_H dimension space. $\mathbf{W}_{o,h}$ are also trainable linear layers projecting D_H dimensional vectors back to a D dimension space. Biases are omitted for simplicity for our discussion. Further, we also omit the weights $\mathbf{W}_{q,h}, \mathbf{W}_{k,h}, \mathbf{W}_{v,h}, \mathbf{W}_{o,h}$, scaling factor $\sqrt{D_H}$, and normalization in the softmax function and assume that $H = 1$. They are still used in the implementation, but omitted from discussion for notational simplicity. Then, (2.2) becomes

$$\mathcal{A}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \exp(\mathbf{Q}\mathbf{K}^{\top})\mathbf{V}$$

After omitting normalization in the softmax function, the softmax becomes an exponential function, denoted as exp.

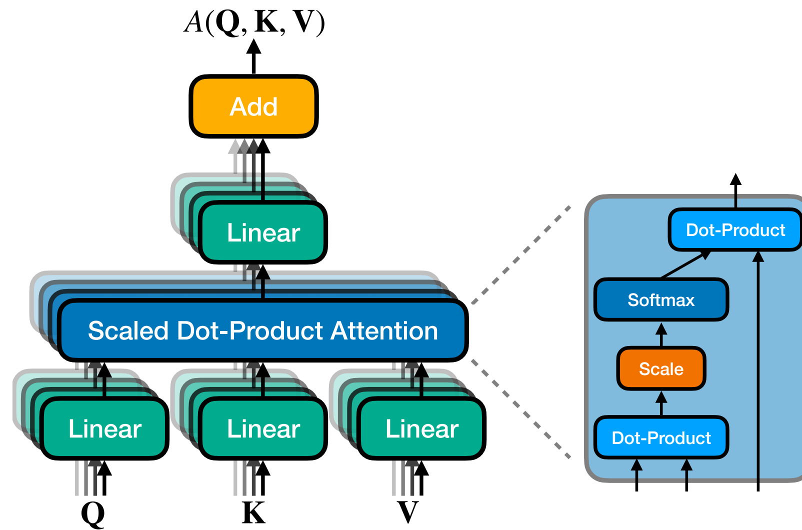

Figure 2.2: Detailed computation within a multi-head attention module.

The output of the MHA module is added to $\mathbf{X}_{i-1}$ for a residual connection to form an intermediate representation $\mathbf{Y}_i$ in (2.1). Then, this $\mathbf{Y}_i$ is fed into a two-layer FFN to compute a pointwise non-linear transformation of each row of $\mathbf{Y}_i$.

$$\mathcal{F}(\mathbf{Y}_i) = \sigma(\mathbf{Y}_i \mathbf{W}_1^\top)\mathbf{W}_2$$

where $\mathbf{W}_1, \mathbf{W}_2$ are trainable linear projections. Here, σ is the activation function of choice, and the default activation function is GELU (Hendrycks and Gimpel, 2016). An encoder consists of multiple such encoder blocks with an identical structure, which make it easier to scale the width and depth of a Transformer model with very few design choices.

2.3 Efficient Structured Projections

In this section, we will cover some background regarding the structured projections and corresponding fast algorithms.

2.3.1 Discrete Fourier Transform

The discrete Fourier Transform (Heideman et al., 1985) is a commonly used structured, orthogonal, symmetric, linear transformation. Consider we are given a vector $\mathbf{x}$ of dimension D, where D is a power of 2. The transformed output $\mathbf{y}$ is defined as

$$[\mathbf{y}]_k = \sum_{n=1}^{D} [\mathbf{x}]_n \, e^{-i2\pi \frac{(k-1)(n-1)}{D}}$$

Here, $[\mathbf{y}]_k$ is the k-th entry of the output and $[\mathbf{x}]_n$ denotes the n-th entry of the vector $\mathbf{x}$. And the inverse Transform is defined as

$$[\mathbf{x}]_n = \frac{1}{N} \sum_{k=1}^{D} [\mathbf{y}]_k \, e^{i2\pi \frac{(k-1)(n-1)}{D}}$$

In matrix form, for example, the DFT of size 4 is

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & e^{-i2\pi\frac{1}{4}} & e^{-i2\pi\frac{2}{4}} & e^{-i2\pi\frac{3}{4}} \\ 1 & e^{-i2\pi\frac{2}{4}} & e^{-i2\pi\frac{4}{4}} & e^{-i2\pi\frac{6}{4}} \\ 1 & e^{-i2\pi\frac{3}{4}} & e^{-i2\pi\frac{6}{4}} & e^{-i2\pi\frac{9}{4}} \end{bmatrix}$$

The discrete Fourier Transform has a well-known fast algorithm called the Fast Fourier Transform (Heideman et al., 1985), which reduces the $\mathcal{O}(D^2)$ complexity for the naive algorithm to $\mathcal{O}(D \log(D))$ by recursively decomposing a discrete Fourier Transform of size M to two discrete Fourier Transforms of size $\frac{M}{2}$.

2.3.2 Hadamard Transform

The Hadamard Transform (Shanks, 1969) is also a structured, orthogonal, symmetric, linear transformation. In matrix form, the Hadamard Transform $\mathbf{H}_M$ is a $2^M \times 2^M$ matrix whose entries are $-\frac{1}{\sqrt{2^M}}$ and $\frac{1}{\sqrt{2^M}}$ where M is an integer. $\mathbf{H}_M$ can

be defined recursively from $\mathbf{H}_{M-1}$ using

$$\mathbf{H}_M = \frac{1}{\sqrt{2}} \begin{bmatrix} \mathbf{H}_{M-1} & \mathbf{H}_{M-1} \\ \mathbf{H}_{M-1} & -\mathbf{H}_{M-1} \end{bmatrix}$$

with the base case $\mathbf{H}_0$ being a 1×1 matrix whose entry is 1. For example, $\mathbf{H}_0, \mathbf{H}_1, \mathbf{H}_2$ are defined as

$$\mathbf{H}_0 = \begin{bmatrix} 1 \end{bmatrix}$$
$$\mathbf{H}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$
$$\mathbf{H}_2 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

The inverse of the Hadamard Transform is equivalent to the Hadamard Transform itself. Naively, a Hadamard Transform of a vector of size D (a power of 2) incurs $\mathcal{O}(D^2)$ cost, but similar to the discrete Fourier Transform, it also has a fast algorithm (Shanks, 1969). Via the Fast Hadamard Transform algorithm, this transform only incurs $\mathcal{O}(D \log(D))$. An example of transforming a 8 dimensional vector is shown in Figure 2.3.

The Fast Hadamard Transform is often used in efficient random projections (Le et al., 2013; Andoni et al., 2015). These efficient random projections are not only fast due to the $\mathcal{O}(D \log(D))$ algorithm, but also produce lower variance estimations due to the orthogonality of Hadamard Transform. Another common application of the Fast Hadamard Transform is a learnable structured projection, as a replacement to regular learnable linear projection within deep learning models to reduce the compute footprint and parameter count (Cheng et al., 2015; Le et al., 2013; Yang et al., 2015; Moczulski et al., 2016).

In next section, we will show how the Fast Hadamard Transform can be used for

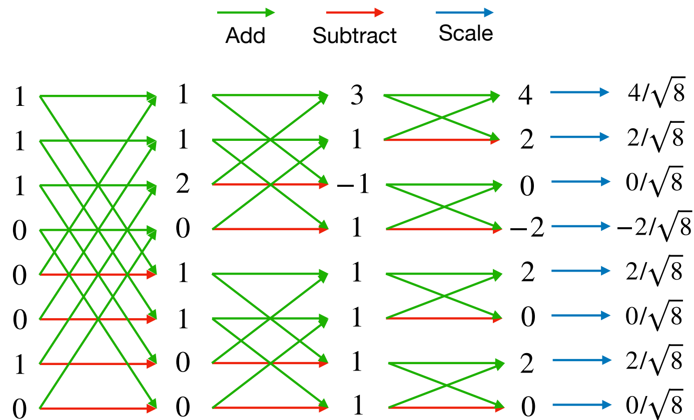

Figure 2.3: Illustration of fast Hadamard transform algorithm

speeding up the hash computation for locality sensitive hashing.

2.4 Locality Sensitive Hashing

In this section, we will cover some basics of Locality Sensitive Hashing (LSH), which will be useful in Chapter 3.

2.4.1 Maximum Inner Product Search

Maximum inner product search (MIPS) (Shrivastava and Li, 2014) is a problem of finding the data vector that would have the maximal inner product with the query vectors. Formally, for a set $\mathbb{S}$ of size N containing vectors $\mathbf{x}_i$ in a D dimensional space, given a query vector $\mathbf{q} \in \mathbb{R}^D$, MIPS attempts to find $\mathbf{x}_i$ satisfying

$$\arg\max_{i \in \mathbb{S}} \langle \mathbf{x}_i, \mathbf{q} \rangle$$

where $\langle \cdot, \cdot \rangle$ is the inner product. The naive algorithm for solving this MIPS problem is brute force: computing $\langle \mathbf{x}_i, \mathbf{q} \rangle$ for every $\mathbf{x}_i$ and pick the $\mathbf{x}_i$ corresponding to the maximal $\langle \mathbf{x}_i, \mathbf{q} \rangle$. This procedure takes $\mathcal{O}(ND)$ computational complexity, which is linear in the size of $\mathbb{S}$ and is generally slow for use in practice. When the vector

norms of vectors are the same (by the structures of these vectors or via specialized construction discussed later), the MIPS problem is the same as nearest neighbor search (NNS). Depending on the structure of $\mathbf{x}_i$ in the set $\mathbb{S}$, there exists different types of efficient algorithm for solving this MIPS problem. For a high dimensional space (D is large), a common efficient algorithm is LSH (Shrivastava and Li, 2014).

A special construction for constant norm vectors. In many applications, the norm of data vectors might not be the same, resulting in misalignment between MIPS and NNS. Neyshabur and Srebro (2015) describes a simple construction to bridge the gap.

Assume that the norms of data vectors $\mathbf{x}_i$ are bounded by some $\sqrt{\tau}$ for all $\mathbf{x}_i \in \mathbb{S}$. Then we can construct new data vectors as follows

$$\hat{\mathbf{x}}_i = [[\mathbf{x}_i]_1, \cdots, [\mathbf{x}_i]_D, \sqrt{\tau - \|\mathbf{x}\|^2}]^\top$$

Here, we assume the norm is Euclidean norm. This construction can be adjusted for other norms. Then, for a query $\mathbf{q}$, the new query vector can be constructed as

$$\hat{\mathbf{q}}_i = [[\mathbf{q}]_1, \cdots, [\mathbf{q}]_D, 0]^\top$$

Note that $\langle \hat{\mathbf{x}}_i, \hat{\mathbf{q}} \rangle = \langle \mathbf{x}_i, \mathbf{q} \rangle$ while $\|\hat{\mathbf{x}}_i\| = \sqrt{\tau}$ for all $\mathbf{x}_i \in \mathbb{S}$. Via this construction, MIPS becomes a NNS problem.

2.4.2 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) (Gionis et al., 1999) is a probabilistic hashing algorithm that hashes similar (in terms of distance) vectors to the same hash bucket with high probability. As a result, given a query vector, the vectors sharing the same LSH hashcode are approximately close to the query vector, so we can obtain approximated NNS result efficiently using LSH. Formally, given a metric space $\mathbb{M}$ equipped with a distance function $\mathcal{D}$ (this distance $\mathcal{D}$ can be L1 distance, Euclidean distance, angular distance, and others). Define a set $\mathbb{F}$ of hash functions $\mathcal{H} : \mathbb{M} \rightarrow \mathbb{H}$

where $\mathbb{H}$ is a set of hash buckets. $\mathbb{F}$ is an LSH family if it satisfies the condition that for any two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{M}$ and a hash function $\mathcal{H}$ uniformly sampled from $\mathbb{F}$:

1. $\mathcal{D}(\mathbf{a}, \mathbf{b}) \leqslant r$, then $\mathcal{H}(\mathbf{a}) = \mathcal{H}(\mathbf{b})$ with probability at least p_1 (a.k.a. **a** and **b** collide to the same hash bucket).

2. $\mathcal{D}(\mathbf{a}, \mathbf{b}) \geqslant cr$, then $\mathcal{H}(\mathbf{a}) = \mathcal{H}(\mathbf{b})$ with probability at most p_2.

for some constants $r > 0$, $c > 1$, and $p_1 > p_2$. Then, this $\mathbb{F}$ is (r, cr, p_1, p_2)-sensitive.

Amplification. Given a base (r, cr, p_1, p_2)-sensitive family $\mathbb{F}$, we can use it can construct (r, cr, q_1, q_2)-sensitive family $\mathbb{G}$ for different q_1, q_2 as the following manner:

1. AND-Amplification: A (r, cr, p_1^K, p_2^K)-sensitive family $\mathbb{G}$ can be constructed by defining a hash function $\mathcal{G} \in \mathbb{G}$ such that $\mathcal{G}(\mathbf{a}) = \mathcal{G}(\mathbf{b})$ if and only if $\mathcal{H}_i(\mathbf{a}) = \mathcal{H}_i(\mathbf{b})$ for all $i \in \{1, 2, \cdots, K\}$ where $\mathcal{H}_i$ are sampled uniformly from $\mathbb{F}$.

2. OR-Amplification: A $(r, cr, 1-(1-p_1)^K, 1-(1-p_2)^K)$-sensitive family $\mathbb{G}$ can be constructed by defining a hash function $\mathcal{G} \in \mathbb{G}$ such that $\mathcal{G}(\mathbf{a}) = \mathcal{G}(\mathbf{b})$ if and only if $\mathcal{H}_i(\mathbf{a}) = \mathcal{H}_i(\mathbf{b})$ for at least one $i \in \{1, 2, \cdots, K\}$ where $\mathcal{H}_i$ are sampled uniformly from $\mathbb{F}$.

Via amplification, we can construct LSH family more freely with fewer base LSH families, which is useful for our algorithm development.

2.4.3 Random Hyperplane Hashing

When the distance $\mathcal{D}$ is the angular distance, a common LSH family is the random hyperplane hashing. Formally, each $\mathcal{H} \in \mathbb{F}$ is associated with a hyperplane passing through the origin (or a vector) $\mathbf{p}$ where $\mathbf{p} \sim \text{Normal}(0, \mathbf{I}_D)$, and $\mathcal{H}$ is defined as

$$\mathcal{H}(\mathbf{a}) = \text{sign}(\mathbf{p}^\top \mathbf{a})$$

where sign is a sign function returning 1 for a positive input and 0 for a negative input. It is not difficult to prove that given two vectors $\mathbf{a}$ and $\mathbf{b}$, the probability of $\mathcal{H}(\mathbf{a}) = \mathcal{H}(\mathbf{b})$ (the collision probability) is

$$1 - \frac{\mathcal{A}(\mathbf{a}, \mathbf{b})}{\pi}$$

where $\mathcal{A}(\mathbf{a}, \mathbf{b})$ is the angle between $\mathbf{a}$ and $\mathbf{b}$.

$$\mathcal{A}(\mathbf{a}, \mathbf{b}) = \arccos\left(\frac{\mathbf{a}^\top \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|}\right)$$

AND-Amplification. We could concatenate multiple hash functions in $\mathbb{F}$ to boost the difference in collision probability between two vectors. Define $\rho : \mathcal{M} \to \{0, 1\}^K$ where each output dimension is determined by a $\mathcal{H}_i$ sampled from $\mathbb{F}$:

$$\mathcal{G}(\mathbf{a}) = \text{sign}(\mathbf{P}\mathbf{a})$$

where $\mathbf{P} \in \mathbb{R}^{K \times D}$ whose entries are i.i.d. sampled from a Normal$(0, 1)$. Then, the collision probability of $\mathcal{G}(\mathbf{a}) = \mathcal{G}(\mathbf{b})$ becomes

$$\left(1 - \frac{\mathcal{A}(\mathbf{a}, \mathbf{b})}{\pi}\right)^K$$

Efficient Random Projections. Computing the hashcode of an input $\mathbf{a}$, $\mathcal{G}$ takes $\mathcal{O}(DK)$ computational complexity due to the matrix-vector multiplication $\mathbf{P}\mathbf{a}$. There are faster variants (Andoni et al., 2015) for random projections using Fast Hadamard Transform discussed in §2.3. Specifically, assuming D is a power of 2 and $K = D$, then

$$\mathbf{H}\mathbf{D}_3\mathbf{H}\mathbf{D}_2\mathbf{H}\mathbf{D}_1\mathbf{a}$$

can function similar to $\mathbf{P}\mathbf{a}$. Here $\mathbf{H}$ is the $D \times D$ Hadamard matrix and $\mathbf{D}_3, \mathbf{D}_2, \mathbf{D}_1$ are diagonal matrices. Via the fast Hadamard transform algorithm, the above computation only takes $\mathcal{O}(D \log(D))$ complexity.

In Chapter 3, we will discuss how we use LSH to estimate self-attention computation efficiently that leads to our memory lookup based methodology.

2.5 Importance Sampling

In this section, we will cover some basics about Monte Carlo method and importance sampling, that will be useful in Chapter 3.

2.5.1 Monte Carlo Methods

Monte Carlo methods (Owen, 2013) are a class of randomized algorithms that estimate certain numerical quantities using random sampling. These methods are commonly used in a wide range of disciplines in science, engineering, and mathematics. One example application that we will use in this thesis is the estimation of numerical integration or probability expectation. These methods generally follow a similar pattern of: (**1**) generating inputs randomly from a probability distribution over a specific domain, (**2**) feeding these inputs into a deterministic algorithm that are designed based on the knowledge of the input probability distribution and the target numerical quantity that we seek to estimate, and (**3**) finally aggregating the results to improve the accuracy of the estimated quantity. Here, we give a simple example of estimating the numerical value of π by uniformly sampling points in a unit square and counting the number of points which fall into the quadrant inscribed by the unit square. The example is illustrated in Figure 2.4.

We can note from the example that as the number of randomly sampled points increases, the accuracy of the estimated quantity increases as well. In fact, Monte Carlo methods usually require a large number of randomly sampled inputs to achieve a reasonable accuracy, which will actually result in a high computational cost. There is a trade-off between estimation accuracy and computational cost.

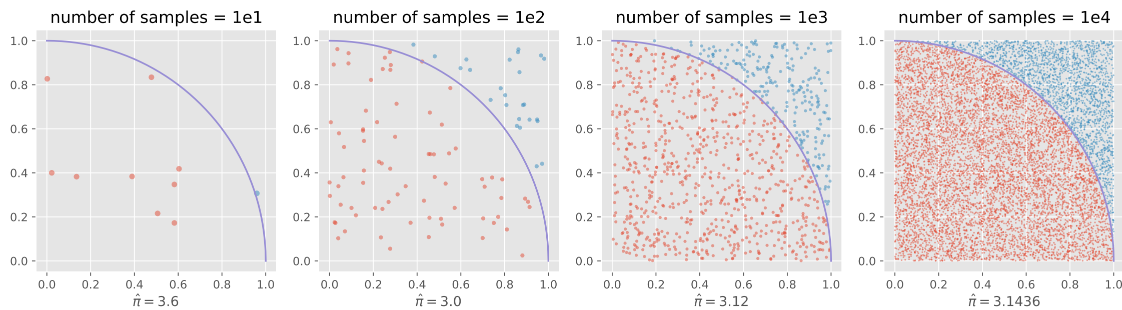

Figure 2.4: An example of using the Monte Carlo Method for estimating π. Uniformly sampling N points in $[0,1]^2$ and counting the number R of points that fall into the quadrant, leads to an estimate of π which is $\hat{\pi} = \frac{4 \times R}{N}$.

2.5.2 Importance Sampling

Importance Sampling (Kahn, 1950) is a technique used to improve the efficiency of Monte Carlo methods. It estimates a quantity of interest for a probability distribution, which is referred to the target distribution, using samples from a different distribution, which is referred as the proposal distribution.

In a Monte Carlo method, the target distribution $\mathcal{F}(x)$ is the the sampling distribution used in Monte Carlo methods to randomly sample inputs for deterministic algorithms. Sampling directly from the target distribution might be inefficient due to the difficulty of drawing samples from the target distribution or high variance requiring more samples for an accurate estimation. Instead of sampling directly from the target distribution $\mathcal{F}(x)$, importance sampling draw samples from the proposal distribution $\mathcal{G}(x)$. This distribution is chosen so that it is easier to sample for efficient sampling algorithms and will sample more frequently around the high impact regions – those that contribute more significantly to the quantity we are estimating to reduce the number of samples required for an accurate estimate. Because the samples are drawn from a different distribution rather than the target distribution, each sample needs to be weighted to correct for this discrepancy. The weight typically involves the ratio of the probability density of the target distribution to the density of the proposal distribution at each sample. Finally, the quantity of

interest is estimated by averaging the weighted samples. One typical example is the estimating of the expectation via Monte Carlo method using importance sampling:

$$\frac{1}{M}\sum_{i=1}^{M}\frac{\mathcal{F}(x_i)}{\mathcal{G}(x_i)}x_i$$

Here, we show an example of Monte Carlo method using importance sampling. If we are interested in estimating the expectation $E(X)$ where X is distributed according to a density function $\mathcal{F}(x)$ over domain $\mathbb{D} = [0, 1]$. We have

$$E(X) = \int_0^1 \mathcal{F}(x)x\,dx$$

However, direct computation of the quantity $E(X)$ might not be feasible, but we can estimate it using the Monte Carlo method. Directly sampling from $\mathcal{F}(x)$ might not be feasible and efficient, so we could choose a different proposal distribution. One example is the uniform distribution whose density is $\mathcal{U}(x)$ over domain $[0, 1]$

$$\frac{1}{M}\sum_{i=1}^{M}\frac{\mathcal{F}(x_i)}{\mathcal{U}(x_i)}x_i$$

Here, $\mathcal{U}(x) = 1$ for all x. While uniform distribution is easy to sample, it might not be sample efficient – a large number of samples may be required for an accurate estimation. The optimal proposal distribution to minimize the variance of estimate can be solved via

$$\mathcal{G}^* = \min_{\mathcal{G}} \operatorname{var}\left(X\frac{\mathcal{F}(X)}{\mathcal{G}(X)}\right)$$

However, solving this optimization problem or sampling from $\mathcal{G}^*(x)$ might be inefficient. In Chapter 3, we focus on proposal distributions that can be easily sampled from enabled by locality sensitive hashing and yet closely align with the target distribution $\mathcal{F}(x)$ to lower the variance.

2.6 Multi-Resolution Analysis

In this section, we will discuss some background related to Multi-Resolution Analysis (MRA), which will be useful in Chapter 5 and 6.

2.6.1 Multi-Resolution Analysis and Wavelets

Multi-Resolution Analysis (MRA) (Chui, 1992; Daubechies, 1992) is a mathematical framework used primarily in signal processing, particularly for analyzing signals at various resolutions. It forms the theoretical foundation for techniques such as wavelet transforms, which decompose signals into components that capture both time and frequency information. It reorganizes a signal into different resolutions, where the lower resolutions contain information summarizing global features of the signal, and higher ones capture fine-grained details of the signal. These strata are constructed iteratively, which we will briefly describe later. For simplicity, we will focus on signals that are 1-dimensional, but all of the following description extends to signals of arbitrary dimensions.

A wavelet transform (Mallat, 1999) decomposes a signal into different scales and locations represented by a set of scaled and translated copies of a *fixed* function. This fixed function $\mathcal{M}$ is called a mother wavelet, and the scaled and translated copies are called child wavelets specified by two factors, scale s and translation t.

$$\mathcal{M}_{s,t}(x) = \frac{1}{\sqrt{s}}\mathcal{M}(\frac{x-t}{s})$$

Here, s controls the "dilation" or inverse of the frequency of the wavelet, while t controls the location (e.g., time). These scaled/translated versions of mother wavelets play a key role in MRA. Given a choice of $\mathcal{M}$, the wavelet transform maps a function $\mathcal{F}$ to a function $\mathcal{G}$ of wavelet coefficients

$$\mathcal{G}(s,t) = \langle \mathcal{F}, \mathcal{M}_t^s \rangle = \int \mathcal{F}(x)\mathcal{M}_{s,t}(x)dx$$

which captures both frequency/scale and location information of $\mathcal{F}$.

2.6.2 Discrete Wavelet Transform

A Discrete Wavelet Transform (DWT) (Edwards, 1991) is a discretization of the wavelet transform for which the wavelets and signals are discretely sampled. For example, the discrete Haar wavelet $\mathcal{M}_{s,t}$ for $s \in \{1, 2, 4, \cdots, N/2\}$ and $t \in \{1, 2, \cdots, N/(2s)\}$ is defined as

$$[\mathcal{M}_{s,t}]_i = \begin{cases} \frac{1}{\sqrt{2s}} & \text{if } st - s < i \leqslant st \\ -\frac{1}{\sqrt{2s}} & \text{if } st < i \leqslant st + s \\ 0 & \text{otherwise} \end{cases}$$

Aside from how the DWT is defined, what a DWT essentially does is that it decomposes a signal into multiple signals in a nested subspace. Given a discretized signal $\mathbf{s}_0 \in \mathbb{R}^N$ with $N = 2^K$ for a constant K, one iteration of a wavelet analysis applied to $\mathbf{s}_0$ yields a pair of signals, $\mathbf{s}_1, \mathbf{h}_1 \in \mathbb{R}^{N/2}$. Each of these shorter signals is obtained by convolving $\mathbf{s}_0$ with a special filter and then downsampling by removing every other coefficient of that convolution product. The convolution filters, denoted $\mathbf{c}$ and $\mathbf{d}$, correspond to local smoothing and discrete differentiation, respectively. If we denote the downsampling operation by down, then the k-th iteration of a wavelet analysis yields $\mathbf{s}_k, \mathbf{h}_k \in \mathbb{R}^{N/2^k}$

$$\begin{aligned} \mathbf{s}_k &= \text{down}(\mathbf{s}_{k-1} * \mathbf{c}) \\ \mathbf{h}_k &= \text{down}(\mathbf{s}_{k-1} * \mathbf{d}) \end{aligned} \tag{2.3}$$

Here, $*$ is the convolution operator. After K iterations, we will have a sequence $\{\mathbf{s}_K, \mathbf{h}_K, \mathbf{h}_{K-1}, \mathbf{h}_{K-2}, \ldots, \mathbf{h}_1\}$. Figure 2.5 shows an illustration of (2.3). Concatenating this sequence yields a new signal in $\mathbb{R}^N$. Note that in this representation, all $\mathbf{s}_k$ have been discarded, for $k < K$.

Construction of the filters $\mathbf{c}$ and $\mathbf{d}$, which has been the subject of extensive research

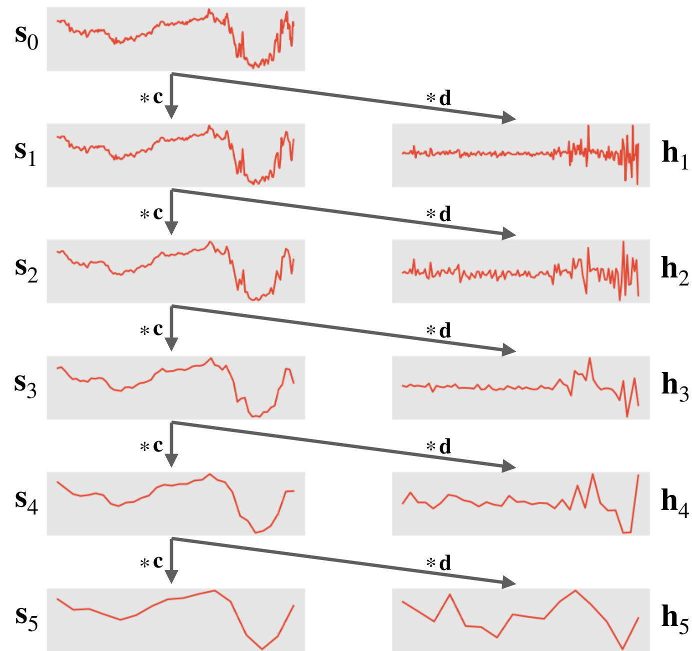

Figure 2.5: Illustration of the nested structure of the wavelet transform.

(Daubechies, 1992), guarantees that the wavelet analysis operator

$$\mathcal{W} : \mathbf{s}_0 \to \{\mathbf{s}_K, \mathbf{h}_K, \mathbf{h}_{K-1}, \mathbf{h}_{K-2}, \ldots, \mathbf{h}_1\}$$

is a linear isometry. That is,

$$\|\mathbf{s}_0\|^2 = \|\mathcal{W}(\mathbf{s}_0)\|^2$$

Thus, the reorganization of a signal under the action of $\mathcal{W}$ has a linear inverse, $\mathcal{W}^{-1} = \mathcal{W}^*$, which is the adjoint of $\mathcal{W}$. For a simple example, the filters $\mathbf{c}$ and $\mathbf{d}$ for the discrete Haar wavelet are defined as

$$\mathbf{c} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$
$$\mathbf{d} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

2.7 Gradient Estimates of Selection Operators

In this section, we will discuss some background about the gradient estimates of selection operators, which will be useful in Chapter 4. The common strategy for training deep learning models is gradient based optimization (Cauchy et al., 1847; Kiefer and Wolfowitz, 1952; Kingma and Ba, 2015). The Backpropagation (Rumelhart et al., 1986) is the most common algorithm for computing the gradient with respect to the objective function. When using the backpropagation algorithm, generally, each operator within deep learning models should be differentiable. However, not all operators are differentiable. For example, operators that involve selections might not be differentiable. Multiple surrogates or tricks (Bengio, 2013; Jang et al., 2017) have been proposed for these non-differentiable operators.

2.7.1 Binary Selections

One non-differentiable operator is the binary selection defined as

$$\mathcal{D}(x) = \begin{cases} 1 & \text{if } \frac{1}{1+\exp(-x)} > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

Here, $\frac{1}{1+\exp(-x)}$ is the sigmoid function mapping the input to a normalized Bernoulli probability between 0 and 1. Since (2.4) is a hard threshold function, it is not differentiable. A common strategy for estimate the derivative of (2.4) is the Straight-Through Estimator (Bengio, 2013). In Straight-Through Estimator, instead of performing the deterministic hard threshold, Bengio (2013) uses

$$\mathcal{S}(x, z) = \begin{cases} 1 & \text{if } z > \frac{1}{1+\exp(-x)} \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

where $z \sim \text{Uniform}[0, 1]$ is an uniform random variable. Then, an unbiased estimator of the derivative of the expectation of (2.5) is

$$\frac{\partial \widehat{E[\mathcal{S}(x, z)]}}{\partial x} = \mathcal{S}(x, z) - \frac{1}{1 + \exp(-x)}$$

Here, E is the expectation of the random variable input.

2.7.2 Performing Multiple Selections

Selecting from multiple choices is a common operation for adaptive computation, such as routing the input to different expert modules in mixture of experts (MoE) (Fedus et al., 2022). Given a selection score vector $\mathbf{p} \in \mathbb{R}^M$ over M choices, the selection is defined as

$$\mathcal{F}(\mathbf{p}) = \text{onehot}(\arg\max_i\, [\mathbf{p}]_i) \tag{2.6}$$

Here, $\text{onehot} : \mathbb{Z} \to \mathbb{R}^M$ is a function mapping the input k to the k-th vector, $\mathbf{e}_k$ in the standard basis. The entries of $\mathbf{e}_k$ are all zeros, except for k-th entry, which is 1. This operator is non-differentiable, but multiple differentiable surrogates are used in deep learning. For example, we can augment (2.6) with a differentiable relaxation of arg max

$$\mathcal{G}(\mathbf{p}) = \mathcal{F}(\mathbf{p}) \odot \text{softmax}(\mathbf{p}) \tag{2.7}$$

where $\odot$ is a element-wise multiplication. Then, the derivative can be calculated for the augmented component

$$\frac{\partial \mathcal{G}(\mathbf{p})}{\partial \mathbf{p}} = \mathcal{F}(\mathbf{p}) \odot \frac{\partial \text{softmax}(\mathbf{p})}{\partial \mathbf{p}}$$

Here, only the entry of $\mathbf{p}$ corresponding to the largest value will have a non-zero derivative. This surrogate only computes the derivative for $\text{softmax}(\mathbf{p})$. However, since $\mathcal{F}(\mathbf{p})$ also varies when $\mathbf{p}$ changes, it might not be ideal. Nonetheless, empirical evidence from (Fedus et al., 2022) and our deployment in Chapter 4 shows that (2.7) works reasonably well.

Sometimes we might not just select the largest score of $\mathbf{p}$, but prefer sampling from a categorical distribution defined by softmax$(\mathbf{p})$ to explore different choices. Sampling from softmax$(\mathbf{p})$ can be performed using Gumbel-Max trick (Maddison et al., 2014) defined as

$$\mathcal{S}(\mathbf{p}, \mathbf{g}) = \text{onehot}(\arg\max_i([\mathbf{p}]_i + [\mathbf{g}]_i)) \tag{2.8}$$

where entries of $\mathbf{g}$ are i.i.d sampled from Gumbel$(0, 1)$. The derivative of (2.8) can be estimated using

$$\frac{\partial \mathcal{S}(\mathbf{p}, \mathbf{g})}{\partial \mathbf{p}} = \frac{\partial \text{softmax}(\mathbf{p} + \mathbf{g})}{\partial \mathbf{p}}$$

Jang et al. (2017) calls this the Straight-Through Gumbel Estimator.

2.8 Hardware for Matrix Multiplication

In this section, we will discuss some background about the hardware requirements for GEneral Matrix Mulply (GEMM), which will be useful in Chapter 7. GEneral Matrix Multiply (GEMM) is a central operation in deep learning and corresponds to a large chunk of the compute footprint. It requires a large amount of additions and multiplications, and the algorithm for computing GEMM is optimized to leverage parallel computing. Hardware support for GEMM can greatly speed up the computation of deep learning models.

Graphics Processing Units (GPUs) have a large number of compute cores and a high-throughput memory system to provide massive parallelism and high memory bandwidth necessary for GEMM operations, so they are most commonly used in supporting the computation needs of deep learning models. Specialized hardware accelerators, such as the Tensor Core in GPUs (NVIDIA, a) and Tensor Processing Units (Jouppi et al., 2017), are specially designed hardware to further accelerate the computation of GEMM operations by implementing GEMM at the hardware level and provide specialized instruction sets to control the hardware level GEMM. These accelerators support GEMMs on multiple datatype including FP64, FP32, FP16,

INT8, INT4 (NVIDIA, a). Here, FP32 means 32-bit floating point, and INT8 means 8-bit integer. Usually, GEMMs using lower bit-width datatype will be faster and more efficient due to lower memory bandwidth requirement and simpler arithmetic. For example, the Tensor Core of NVIDIA's A100 (NVIDIA, a) can achieve 19.5, 156, 312, 624, and 1248 TeraFLOPs per second for FP64, FP32, FP16, INT8, and INT4 respectively. As a result, using lower bit-width is desired for better acceleration, but lower bit-width would also lower the precision of GEMMs, so there is a trade-off. The current mainstream choice is FP16 GEMMs (Micikevicius et al., 2018) for training of most deep learning models. Lower bit-width GEMMs, such as INT8 or INT4, are more common for inference (Banner et al., 2019; Nagel et al., 2019; Kim et al., 2021; Dettmers et al., 2022; Li and Gu, 2023; Xiao et al., 2023; Dettmers et al., 2022; Liu et al., 2023b,a; Lin et al., 2022; Li and Gu, 2023; Yuan et al., 2021; Ding et al., 2022; Li et al., 2023a), but leveraging lower bit-width GEMMs for training is also an active research direction (Wang et al., 2018b; Wu et al., 2018; Zhu et al., 2020; Wortsman et al., 2023). In Chapter 7, we will explore the use of low bit-width integers for both training and inference of Transformer models.

3 MEMORY LOOKUP BASED APPROXIMATION FOR EFFICIENT SELF-ATTENTION

As discussed in Chapter 1, self-attention makes Transformer models effective for understanding complex input sequences, but the quadratic cost of self-attention computation is a key bottleneck for Transformer models when processing long sequences. There are a number of active efforts devoted to reducing this cost (Beltagy et al., 2020; Zaheer et al., 2020; Kitaev et al., 2020; Chen et al., 2021a; Xiong et al., 2021; Lu et al., 2021; Peng et al., 2021; Choromanski et al., 2021; Wang et al., 2020).

In this chapter, we also seek to address the aforementioned issues, and our work is inspired by ideas of importance sampling via hashing-based sampling strategies (Spring and Shrivastava, 2017; Charikar and Siminelakis, 2017). We propose a Bernoulli based sampling scheme to approximate self-attention, which relies heavily on memory access (write and lookup) with minuscule compute footprint and scales linearly with the input sequence length. We view self-attention as a sum of individual tokens associated with Bernoulli random variables whose success probability is determined by the similarities among tokens. In principle, we can sample all Bernoulli random variables at once with a single hash. It turns out that the resultant strategy (You Only Sample Almost Once, YOSO-Attention) is more amenable to an efficient/backpropagation friendly implementation, and exhibits a favorable performance profile in experiments.

3.1 Locality Sensitive Hashing based Importance Sampling

Let $\mathbf{q}_1, \mathbf{q}_2, \cdots, \mathbf{q}_N; \mathbf{k}_1, \mathbf{k}_2, \cdots, \mathbf{k}_N; \mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_N$ be the rows of $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, respectively. As described in §2.2, Given a query $\mathbf{q}_i$, the self-attention output for $\mathbf{q}_i$ is defined as

$$\mathbf{y}_i = \sum_{j=1}^{N} \exp(\mathbf{q}_i^\top \mathbf{k}_j)\mathbf{v}_j \tag{3.1}$$

Self-attention can be thought of as integrating tokens over a softmax distribution, so a popular method of estimating this integral is importance sampling (Press et al., 2007) as described in §2.5. It is known (Press et al., 2007) that importance sampling can be directly used for the softmax distribution by drawing samples from a uniform distribution – which avoids sampling from the softmax distribution directly (which is harder). But this leads to a high variance estimate since the softmax distribution is usually concentrated in a small region.

Locality Sensitive Hashing based Importance Sampling. Consider the case when the angular distance between a key and a query is small. In this case, the similarity (between the key and the query) as well as the softmax probability will be large. When viewed through the lens of a nearest neighbor retrieval problem discussed in §2.4, the above property coincides with a large collision probability of high similarity key-query pairs, assuming that the neighbor retrieval is implemented via Locality Sensitive Hashing (LSH). Motivated by the link between softmax probability $\mathcal{P}$ and LSH collision probability $\mathcal{Q}$, Spring and Shrivastava (2017) and Charikar and Siminelakis (2017) suggest using LSH as an efficient sampler for low variance softmax estimators. Let us discuss two interesting results to properly contextualize our work.

(a) Spring and Shrivastava (2017) proposes approximating softmax by sampling a set, $\mathbb{S}$, a collection of neighboring keys for each query formed by the union of

colliding keys using M hash tables. The estimator is computed using

$$\frac{1}{|\mathbb{S}|}\sum_{j\in\mathbb{S}}\frac{\mathcal{P}(\mathbf{q}_i,\mathbf{k}_j)}{\mathcal{Q}(\mathbf{q}_i,\mathbf{k}_j)}\mathbf{v}_j$$

where $\mathbf{q}_i$ is a query vector, $\mathbf{k}_j, \mathbf{v}_j$ are key and value vectors in the sampling set $\mathbb{S}$, and $\mathcal{P}(\cdot,\cdot)$ and $\mathcal{Q}(\cdot,\cdot)$ are softmax probability and collision probability of given pairs. This procedure involves importance sampling without replacement, which leads to a dependency among the samples. Deduplication (avoiding double counting) requires memory to store keys in each hash table and runtime to deduplicate keys for each query. If the size of hash buckets is skewed, the (GPU) memory needs depend on the size of the hash bucket and the runtime depends on the size of $\mathbb{S}$.

(**b**) Charikar and Siminelakis (2017) provide a Hash based Estimator to simulate a proposal distribution for importance sampling via LSH, which can be easily applied in the context of softmax. For each hash table, a key is uniformly selected from the bucket that the query is hashed to, for simulating a draw from a proposal distribution. The estimate is computed as

$$\frac{1}{M}\sum_{k=1}^{M}\frac{\mathcal{P}(\mathbf{q}_i,\mathbf{k}_j)|\mathbb{H}_k(\mathbf{q}_i)|}{\mathcal{Q}(\mathbf{q}_i,\mathbf{k}_j)}\mathbf{v}_j,$$

where $|\mathbb{H}_k(\mathbf{q}_i)|$ denotes the size of hash bucket in the k-th hash table which $\mathbf{q}_i$ is hashed to. This simulates M samples drawn with replacement from the proposal distribution. However, the probability of one key being sampled depends not only on (i) the angular distance to the query but also (ii) the number of keys within the hash bucket, leading to a sampling dependency among all keys. Further, using it for self-attention causes a dependence between the sparsity in the softmax matrix and the number of hashes used. Specifically, the number of tokens that each query can attend to is bounded by the number of hashes: the procedure samples at most one distinct key for each hash table and so, it adds one additional nonzero to the softmax matrix, at most.

LSH-based Importance Sampling: practical considerations. While LSH-based importance sampling exploits the agreement between high probability $\mathcal{P}(\cdot,\cdot)$ and high collision probability $\mathcal{Q}(\cdot,\cdot)$, the alignment is not perfect. As discussed in §2.5, samples from the proposal distribution must be reweighted to compensate for the difference. Further, for different queries, the likelihood ratios between the softmax distribution and the proposal distribution with respect to a single key are different. Therefore, a reweighing has to be done *during* querying. Although maintaining hash tables for storing keys is not a major problem in general, the high memory cost for hash tables and computation time for reweighing noticeably influences efficiency when applied to self-attention. To **summarize**, directly applying LSH-based importance sampling in the deep learning context does not lead to an efficient self-attention scheme.

3.2 YOSO Attention

While the efficiency bottleneck can be alleviated through LSH-based importance sampling, these approaches are not very efficient on GPUs. Motivated by LSH-based importance sampling, we propose a sampling method for efficient self-attention via LSH-based Bernoulli sampling.

We start from LSH-based importance sampling and seek to address some of the aforementioned issues when it is deployed for approximating self-attention. Specifically, instead of using LSH to simulate sampling from a proposal distribution over tokens, we view attention as a *sum* of tokens associated with Bernoulli random variables. This modification relates better with LSH and less with LSH-based importance sampling – *the probability of one query colliding with a key is not based on other keys*. This strategy helps avoid the sampling dependency problem in LSH-based importance sampling and offers an opportunity to develop a strategy more amenable to GPUs.

Remark 3.1. *We assume that the input keys and queries of self-attention are unit length – to allow treating dot-product similarity in self-attention and cosine similarity in LSH*

similarly. This is simple using a construction described in Neyshabur and Srebro (2015): a variable τ is used to bound the squared ℓ_2 norm of all queries and keys and to reconstruct new unit length keys and queries while preserving their pairwise similarities:

Assume that the ℓ_2 norms of $\mathbf{q}_i$ and $\mathbf{k}_j$ are bounded by some $\sqrt{\tau}$ for all $i, j = 1, \cdots, N$. Then we can construct new queries and keys as follows:

$$\hat{\mathbf{q}}_i = \frac{1}{\sqrt{\tau}}[[\mathbf{q}_i]_1, \cdots, [\mathbf{q}_i]_D, \sqrt{\tau - \|\mathbf{q}_i\|_2^2}, 0]$$
$$\hat{\mathbf{k}}_j = \frac{1}{\sqrt{\tau}}[[\mathbf{k}_j]_1, \cdots, [\mathbf{k}_j]_D, 0, \sqrt{\tau - \|\mathbf{k}_j\|_2^2}]$$

Note that $\tau\hat{\mathbf{q}}_i^\top\hat{\mathbf{k}}_j = \mathbf{q}_i^\top\mathbf{k}_j$ while $\|\hat{\mathbf{q}}_i\|_2 = \left\|\hat{\mathbf{k}}_j\right\|_2 = 1$.

Then, when the vector norms are the same, the inner product in (3.1) becomes the cosine of the angle between the two vectors. So, we can work with the softmax (with angular distance) and derive our algorithm.

Self-Attention via Bernoulli Sampling. We aim to approximate self-attention, which uses a softmax matrix to capture the context dependency among tokens via their pairwise similarities. Assuming that we can represent this context dependency *directly* using collision probability $\mathcal{Q}(\cdot,\cdot)$, no reweighting is required if the proposal and target distributions are the same. This means that challenges discussed in LSH-based importance sampling do not exist. So the coincidence of softmax probability $\mathcal{P}(\cdot,\cdot)$ and LSH collision probability $\mathcal{Q}(\cdot,\cdot)$ makes $\mathcal{Q}(\cdot,\cdot)$ a sensible starting point for approximating self-attention. Specifically, to model dependency based on similarity, the collision probability aligns well with the exponential function in softmax in the domain of interest $[-1, 1]$ in Figure 3.1: both functions have positive zeroth, first and second order derivatives.

Note that (a) positive zeroth order derivative indicates that the dependency is positive, (b) positive first order derivative ensures that the dependency based on similarity is monotonic, and (c) positive second order derivative means that the attention weight will rapidly increase and dominate others as the similarity

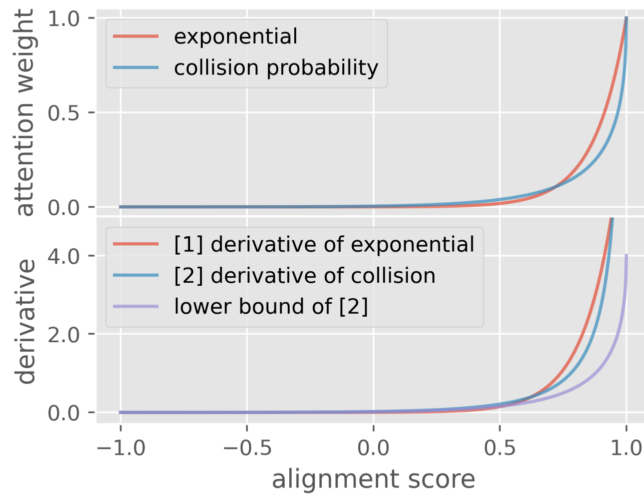

Figure 3.1: We compare attention weights using $\exp(\tau(x-1))$ with the collision probability of concatenating τ hyperplane hashes (Charikar, 2002) $(1-\arccos(x)/\pi)^{\tau}$ for $\tau = 8$. We plot $\exp(\tau(x-1))$ so that the range is between 0 and 1 but without changing the actual attention weights in softmax. We also plot the derivative of exponential function and of collision probability, as well as a lower bound we will use later during backpropagation.

increases. This leads us to hypothesize that a collision-based self-attention may be as effective as softmax-based self-attention. It can be formulated as,

$$\sum_{j=1}^{N} \mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)\mathbf{v}_j \tag{3.2}$$

where $\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)$ is a Bernoulli random variable where the success probability is given by the collision probability of $\mathbf{q}_i$ with the key $\mathbf{k}_j$. Hence, it can be determined by the similarity between $\mathbf{q}_i$ and $\mathbf{k}_j$.

In a single hash, each $\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)$ generates a realization to determine whether the corresponding token will be part of the attention output or not. Conceptually, when sampling from the softmax distribution, only one token is sampled as the attention output. In contrast, Bernoulli sampling determines whether each individual token

is a part of the attention output. In principle, to determine the context dependency among tokens, you only need to sample once (YOSO) using a single hash to generate *realizations of all Bernoulli random variables*, $\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)$ for $i, j \in \{1, 2, \ldots, N\}$. Specifically, when keys are hashed to a hash table using a single hash, the realization of $\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)$ for each query $\mathbf{q}_i$ will be 1 if $\mathbf{q}_i$ collides with $\mathbf{k}_j$, else it is 0. To our knowledge, using LSH collision probability to replace softmax dependencies for self-attention in this way has not been described before.

YOSO-Attention. By replacing softmax dependency with Bernoulli random variables and using LSH as an efficient sampler to estimate the success probability, we obtain an efficient self-attention (YOSO-Attention) to approximate softmax-based self-attention.

$$\text{YOSO}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathcal{B}(\mathbf{Q}, \mathbf{K})\mathbf{V} \tag{3.3}$$

where $\mathcal{B}(\mathbf{Q}, \mathbf{K})$ is the Bernoulli random matrix.

$$[\mathcal{B}(\mathbf{Q}, \mathbf{K})]_{i,j} = \mathcal{B}(\mathbf{q}_i, \mathbf{k}_j) = \mathbb{1}_{\mathcal{F}(\mathbf{q}_i)=\mathcal{F}(\mathbf{k}_j)}$$

where $\mathcal{F}$ is a hash function sampled from a LSH family. The expectation of $\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)$ is

$$E[\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)] = \left(1 - \frac{\arccos(\langle \mathbf{q}_i, \mathbf{k}_j \rangle)}{\pi}\right)^{\tau}$$

The variance of a Bernoulli random variable is simply:

$$\text{Var}[\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)] = E[\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)](1 - E[\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)])$$

While a single sample would work in estimating attention output, in practice, the actual output of YOSO-Attention can be the average of output from M samples to lower the estimation variance, where M is a small constant. The high-level overview of our method is demonstrated in Figure 3.2. For LSH, each sample (hash) is a space partitioning of the input space. The $\mathbf{v}_j$'s associated with $\mathbf{k}_j$'s in the same partition are summed together. The partitions give a coarse representation of $E[\text{YOSO}(\cdot, \mathbf{K}, \mathbf{V})]$. As M increases, the average of M representations converges to

$\mathbb{E}[\text{YOSO}(\cdot, \mathbf{K}, \mathbf{V})]$.

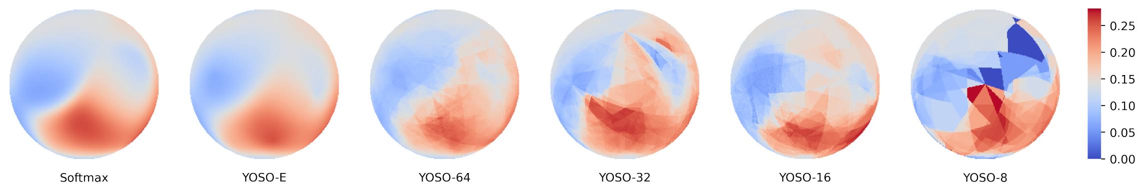

Figure 3.2: A high-level overview of YOSO-Attention. YOSO-m denotes YOSO-Attention with M samples (hashes) and YOSO-E denotes the expectation of YOSO over hash functions. Softmax denotes the softmax self-attention. We visualize YOSO-m, YOSO-E, and Softmax by randomly generating $\mathbf{K} \in \mathbb{R}^{32\times 3}, \mathbf{V} \in \mathbb{R}^{32\times 1}$ and using all unit vectors $\mathbf{q}_i \in \mathbb{S}^2$ to compute their output values on the surface of 3-dimensional sphere to demonstrate how YOSO-m approximates YOSO-E and the high similarity between YOSO-E and Softmax.

Remark 3.2. *Our proposed method enjoys multiple advantages explicitly noted in Performer (Choromanski et al., 2021), which is desired for self-attention: (**a**) The attention weights are always positive, which make it a robust self-attention mechanism. In YOSO, the attention weights are always in* $[0, 1]$*, which means that it is also numerically stable. (**b**) The variance goes to zero as attention weight approaches zero. In YOSO, the variance of attention weights are always upper bounded by the attention weights themselves, making the approximation error easily controllable.*

Normalizing Attention. In softmax self-attention, each row of the softmax matrix is normalized so that the dependencies sum up to 1. We discussed above how the pairwise query-key dependencies can be estimated using Bernoulli sampling. We now describe how to normalize the dependency in our method as softmax self-attention. We can first estimate the dependencies and then normalize them using the sum of estimated dependencies estimated by $\mathcal{B}(\mathbf{Q}, \mathbf{K})\mathbf{1}$ where $\mathbf{1}$ is a vector of all entries being 1. $\mathcal{B}(\mathbf{Q}, \mathbf{K})\mathbf{1}$ can be computed by (3.3) by plugging $\mathbf{1}$ into $\mathbf{V}$. To make the estimation of self-attention more efficient, we adopt a ℓ_2 normalization on the attention output, similar to use of ℓ_2 normalization for word embedding in Levy et al. (2015). Thus, attention outputs are invariant to scaling, $\mathcal{B}(\mathbf{Q}, \mathbf{K})\mathbf{1}$, under

ℓ_2 normalization. Therefore, we have,

$$\text{N-YOSO}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \ell_2(\mathcal{B}(\mathbf{Q}, \mathbf{K})\mathbf{V})$$

Empirically, as expected, we find that the ℓ_2 normalization does not affect the performance of our method.

3.2.1 LSH-based Bernoulli Sampling

Now, we discuss how to actually implement the idea of using Bernoulli sampling to approximate self-attention. While a standard LSH procedure can be used, maintaining hash tables to store keys is inefficient on a GPU – the GPU memory size required for the hash table cannot be predetermined and the workload might be skewed due to skewed bucket sizes. Due to how our Bernoulli sampling is set up, it turns out that simply saving the summation of values corresponding to hashed keys is sufficient (instead of storing a full collection of hashed keys).

Overview. An outline of our algorithm is shown in Figure 3.3. To compute $\mathbf{Y} = \mathcal{B}(\mathbf{Q}, \mathbf{K})\mathbf{V}$, the procedure proceeds as follows. We sample a hash function $\mathcal{F}$ and create a hash table $\mathbf{H} \in \mathbb{R}^{2^\tau \times D}$ representing 2^τ D-dimensional buckets. For each key $\mathbf{k}_j$, we add the value $\mathbf{v}_j$ to the bucket whose index is the hash code $\mathcal{F}(\mathbf{k}_j)$, denoted as $[\mathbf{H}]_{\mathcal{F}(\mathbf{k}_j)}$,

$$[\mathbf{H}]_{\mathcal{F}(\mathbf{k}_j)} \leftarrow [\mathbf{H}]_{\mathcal{F}(\mathbf{k}_j)} + \mathbf{v}_j$$

Note that the size of $\mathbf{H}$ is $O(2^\tau D)$ and is **independent** of which bucket keys are hashed. With all keys processed, for each query $\mathbf{q}_i$, we maintain an output vector $[\mathbf{Y}]_i$ initialized to 0. Then, we allocate the bucket in $\mathbf{H}$ using $\mathcal{F}(\mathbf{q}_i)$ and use $[\mathbf{H}]_{\mathcal{F}(\mathbf{q}_i)}$ as the attention output $[\mathbf{Y}]_i$ for $\mathbf{q}_i$. Therefore, each final output $[\mathbf{Y}]_i$ can be computed as,

$$[\mathbf{Y}]_i = \sum_{j=1}^{N} \mathbb{1}_{\mathcal{F}(\mathbf{q}_i)=\mathcal{F}(\mathbf{k}_j)} \mathbf{v}_j = \sum_{j=1}^{N} [\mathcal{B}(\mathbf{Q}, \mathbf{K})]_{i,j} \mathbf{v}_j$$

Remark 3.3. *The entire computation relies on a simple operation of finding vector values*

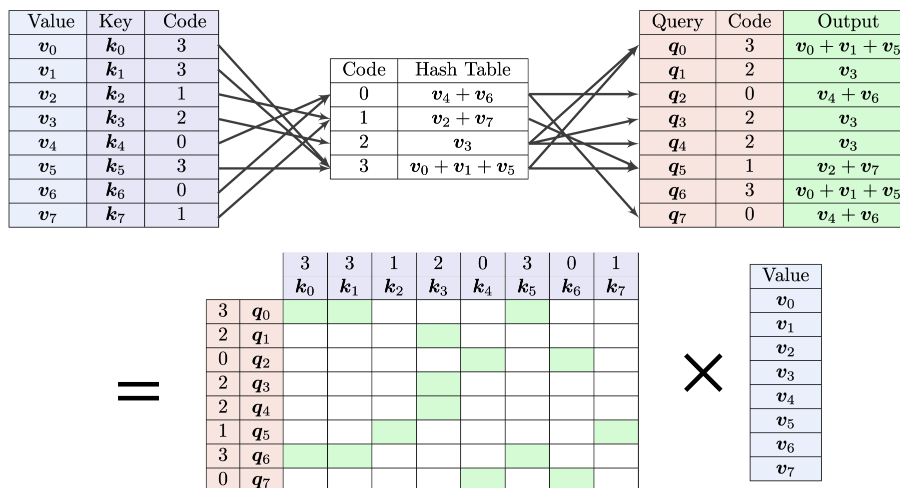

Figure 3.3: Overview of YOSO-Attention algorithm. The hash table stores the sum of values associated with hashed keys.

in certain memory locations and accumulating these vector values at the correct memory locations. This computation is entirely built upon memory access with a very small compute footprint.

Remark 3.4. *The memory and time complexity of this algorithm are* $O(M2^\tau D)$ *and* $O(NMD)$ *respectively, In addition, both time and memory are independent of the size of hash buckets. We can further improve the memory complexity to* $O(M2^\tau)$ *by reusing the hash table and processing a few dimensions each time without increasing the time complexity. The constant* τ *is defined in Remark 3.1 and relates to the maximal norm of query and key vectors, but we can simply set it as a hyperparameter (the length of hash code) that controls the decay rate of attention weights with respect to the angular distance between query and key.*

Speed-up. While not essential, we find that a fast random projection for computing the LSH hash code is beneficial, since this step takes a large portion of the overall runtime. As suggested by Andoni et al. (2015), we use the approximated random projection that we discussed in §2.4 to reduce time complexity to $O(NM\tau \log_2(D))$, allowing fast computation of hash codes. In Chapter 4, we will return to this idea again for efficient learnable projection.

3.3 Backpropagation

For training, we also need to show that backward propagation steps for YOSO-Attention are feasible. Here, we discuss this last component of YOSO-Attention which enables end-to-end efficient training.

For backpropagation, the gradient of the loss w.r.t. $\mathbf{V}$ can be estimated similar to (3.3),

$$\nabla_{\mathbf{V}} = ((1 - \frac{\arccos(\mathbf{Q}\mathbf{K}^\top)}{\pi})^\tau)^\top \nabla_{\text{YOSO}} \approx \mathcal{B}(\mathbf{K}, \mathbf{Q}) \nabla_{\text{YOSO}}$$

The gradients of the loss w.r.t. $\mathbf{Q}$ and $\mathbf{K}$ are similar, so we only provide the expression for $\mathbf{Q}$, and the case for $\mathbf{K}$ follows similarly.

$$\nabla_{\mathbf{Q}} = \left(\left(\nabla_{\text{YOSO}} \mathbf{V}^\top\right) \odot \tau \left(1 - \frac{\arccos(\mathbf{Q}\mathbf{K}^\top)}{\pi}\right)^{\tau-1} \oslash \left(\pi \sqrt{1 - (\mathbf{Q}\mathbf{K}^\top)^2}\right)\right) \mathbf{K} \quad (3.4)$$

where $\oslash, \odot$ are element-wise division and multiplication. One issue with the true gradient is that it goes to infinity as the alignment score between the query and the key approaches 1, which might lead to divergence. To avoid this numerical issue, we use a lower bound of the actual derivative of the collision probability,

$$\begin{aligned} \hat{\nabla}_{\mathbf{Q}} &= \left(\left(\nabla_{\text{YOSO}} \mathbf{V}^\top\right) \odot \frac{\tau}{2} (1 - \frac{\arccos(\mathbf{Q}\mathbf{K}^\top)}{\pi})^\tau \right) \\ &\approx \left(\left((\nabla_{\text{YOSO}} \mathbf{L}) \mathbf{V}^\top\right) \odot \frac{\tau}{2} \mathcal{B}(\mathbf{K}, \mathbf{Q}) \right) \mathbf{K} \end{aligned} \quad (3.5)$$

The empirical behavior is shown in Figure 3.1, and it can be efficiently estimated via a variation of LSH-based Bernoulli Sampling. Specifically, note that the approximation can be decomposed into sum of d LSH-based Bernoulli Sampling,

$$\left[\hat{\nabla}_{\mathbf{Q}}\right]_i = \sum_{l=1}^{D} [\nabla_{\text{YOSO}}]_{i,l} \sum_{j=1}^{N} \mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)([\mathbf{V}]_{j,l} \frac{\tau}{2} \mathbf{k}_j)$$

Since this calculation needs D runs of a subroutine whose complexity is $O(M2^\tau D)$ and $O(NMD)$ for memory and time respectively, its memory complexity is $O(M2^\tau D^2)$, and time complexity is $O(NMD^2)$. The D^2 term in the memory complexity can be eliminated by repeatedly using the same hash tables D^2 times without increasing runtime, which improves the memory complexity to $O(M2^\tau)$. The overall complexity of our method relative to softmax self-attention is shown in Table 3.1.

Time	Forward	Backward
Softmax	$O(N^2D)$	$O(N^2D)$
YOSO	$O(NM\tau\log_2(D) + NMD)$	$O(NMD^2)$
Memory	**Forward**	**Backward**
Softmax	$O(N^2)$	$O(N^2)$
YOSO	$O(NM\tau + M2^\tau)$	$O(M2^\tau)$

Table 3.1: Time/memory complexity of self-attention and YOSO-attention in forward/backward computation

3.3.1 Derivation of the Backpropagation scheme

When using expectation of LSH collision as attention weights, the attention output of one query $\mathbf{q}_i$ to keys $\mathbf{k}_j$ and associated values $\mathbf{v}_j$ for all $j \in \{1, 2, \cdots, N\}$ is defined as

$$[\mathbf{Y}]_i = \sum_{j=1}^{N} \left(1 - \frac{\arccos(\mathbf{q}_i^\top \mathbf{k}_j)}{\pi}\right)^\tau \mathbf{v}_j$$

Then, given the gradient of the loss L w.r.t. Y_i, denoted ∇_{Y_i}, the goal is to compute the gradient of the loss w.r.t. $\mathbf{q}_i$, denoted $\nabla_{\mathbf{q}_i}$. We start by computing the q-th entry of $\nabla_{\mathbf{q}_i}$:

$$\begin{aligned}
\frac{\partial L}{\partial [\mathbf{Q}]_{i,q}} &= \sum_{l=1}^{D} \frac{\partial L}{\partial [\mathbf{Y}]_{i,l}} \frac{\partial [\mathbf{Y}]_{i,l}}{\partial [\mathbf{Q}]_{i,q}} \\
&= \sum_{l=1}^{D} \frac{\partial L}{\partial [\mathbf{Y}]_{i,l}} \frac{\partial}{\partial [\mathbf{Q}]_{i,q}} \sum_{j=1}^{N} \left(1 - \frac{\arccos(\mathbf{q}_i^\top \mathbf{k}_j)}{\pi}\right)^\tau [\mathbf{V}]_{j,l}
\end{aligned} \tag{3.6}$$

Then we use

$$\frac{d}{dx}\left(1-\frac{\arccos(x)}{\pi}\right)^{\tau}=\frac{\tau(1-\frac{\arccos(x)}{\pi})^{\tau-1}}{\pi\sqrt{1-x^2}}$$

and plug it into (3.6),

$$\frac{\partial L}{\partial\,[\mathbf{Q}]_{i,q}}=\sum_{l=1}^{D}\frac{\partial L}{\partial\,[\mathbf{Y}]_{i,l}}\sum_{j=1}^{N}\frac{\tau\left(1-\frac{\arccos(\mathbf{q}_i^\top\mathbf{k}_j)}{\pi}\right)^{\tau-1}}{\pi\sqrt{1-(\mathbf{q}_i^\top\mathbf{k}_j)^2}}\,[\mathbf{K}]_{j,q}\,[\mathbf{V}]_{j,l}$$

After swapping the order of the two summations, (3.6) becomes

$$\frac{\partial L}{\partial\,[\mathbf{Q}]_{i,q}}=\sum_{j=1}^{N}\nabla_{[\mathbf{Y}]_i}^{\top}\mathbf{v}_j\frac{\tau\left(1-\frac{\arccos(\mathbf{q}_i^\top\mathbf{k}_j)}{\pi}\right)^{\tau-1}}{\pi\sqrt{1-(\mathbf{q}_i^\top\mathbf{v}_j)^2}}\,[\mathbf{K}]_{j,q}$$

Note that only $[\mathbf{K}]_{j,q}$ is different for different entries of $\nabla_{\mathbf{q}_i}$, so we can write it as

$$\nabla_{\mathbf{q}_i}=\sum_{j=1}^{N}\nabla_{[\mathbf{Y}]_i}^{\top}\mathbf{v}_j\frac{\tau\left(1-\frac{\arccos(\mathbf{q}_i^\top\mathbf{k}_j)}{\pi}\right)^{\tau-1}}{\pi\sqrt{1-(\mathbf{q}_i^\top\mathbf{v}_j)^2}}\mathbf{k}_j$$

The term $\nabla_{\mathbf{Q}}$ is the matrix form of above expression for $i\in\{1,2,\cdots,N\}$

$$\nabla_{\mathbf{Q}}=\left(\left(\nabla_{\text{YOSO}}\mathbf{V}^\top\right)\odot\tau\left(1-\frac{\arccos(\mathbf{Q}\mathbf{K}^\top)}{\pi}\right)^{\tau-1}\oslash\left(\pi\sqrt{1-(\mathbf{Q}\mathbf{K}^\top)^2}\right)\right)\mathbf{K}$$

Note that $\pi\sqrt{1-(\mathbf{Q}\mathbf{K}^\top)^2}$ approaches 0 as the similarity score between the query and the key approaches 1 – so to avoid numerical and stability issues, we use the fact that

$$\frac{1}{2}\left(1-\frac{\arccos(x)}{\pi}\right)\leqslant\frac{1}{\pi\sqrt{1-x^2}}\ \text{for}\ x\in[-1,1]$$

and define a lower bound to replace the actual gradient

$$\hat{\nabla}_{\mathbf{Q}}=\left(\left(\nabla_{\text{YOSO}}\mathbf{V}^\top\right)\odot\frac{\tau}{2}\left(1-\frac{\arccos(\mathbf{Q}\mathbf{K}^\top)}{\pi}\right)^{\tau}\right)\mathbf{K}$$

3.3.2 Alternative Procedure for Approximating Backpropagation

Above, we described a procedure to estimate (3.5), which uses LSH-based Bernoulli Sampling D times as a subroutine. The complexity of this procedure is linear w.r.t. sequence length N, which is desirable but the runtime can be large if D is set to be very large. An alternative described here based on additional assumptions, is linear with respect to D.

The gradient of L w.r.t. the i-th row of Q is written as

$$\hat{\nabla}_{\mathbf{q}_i} L = \sum_{j=1}^{N} \nabla_{[\mathbf{Y}]_i}^{\top} \mathbf{v}_j \mathcal{B}(\mathbf{q}_i, \mathbf{k}_j) \frac{\tau}{2} \mathbf{k}_j$$

Note that if $\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)$ is zero then the corresponding summation term does not need to be computed. The alternative procedure relies on the sparsity of attention matrices (which we have not exploited so far) to reduce the workload. The procedure is simple: it checks the value of $\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j)$ and only computes the summation term when $\mathcal{B}(\mathbf{q}_i, \mathbf{k}_j) = 1$.

For M samples, the procedure counts the number of times $\mathbf{q}_i$ and $\mathbf{k}_j$ collide, and only computes the summation term corresponding to $\mathbf{q}_i$ and $\mathbf{k}_j$ when at least one collision occurs. Therefore, the runtime is $O(\text{nnz}(\mathcal{B}(\mathbf{Q}, \mathbf{K}))(M+D))$ (counting number of success + computing nonzero terms). In the worst case, $\text{nnz}(\mathcal{B}(\mathbf{Q}, \mathbf{K})) = N^2$, it would be as expensive as dense matrix multiplications in complexity and even worse in practice due to a large memory latency resulting from indirect memory access. However, in practice, $\mathcal{B}(\mathbf{Q}, \mathbf{K})$ is generally sparse if τ is set sensibly. Further, the first procedure guarantees a linear complexity scaling of our method for extremely long sequences. As an improvement, one can dynamically select one from these two method based on runtime, then the time complexity is $O(\min(NMD^2, \text{nnz}(\mathcal{B}(\mathbf{Q}, \mathbf{K}))(M + D)))$.

Estimating backpropagation based on (3.4). To test the effect of using (3.5) instead of (3.4) for backpropagation, we developed a similar procedure to estimate (3.4). We only consider the gradient backpropagation through the non-zero entries of

$\mathcal{B}(\mathbf{Q},\mathbf{K})$, so the quantity $\nabla_{\mathbf{Q}}$ computed is as follows,

$$\left(\left(\nabla_{\text{YOSO}}\mathbf{V}^\top\right) \odot \tau\left(1-\frac{\arccos(\mathbf{Q}\mathbf{K}^\top)}{\pi}\right)^{\tau-1} \oslash \left(\pi\sqrt{1-(\mathbf{Q}\mathbf{K}^\top)^2}\right) \odot \mathbb{1}_{\mathcal{B}(\mathbf{Q},\mathbf{K})}\right)\mathbf{K}$$

where $\mathbb{1}_{\mathcal{B}(\mathbf{Q},\mathbf{K})}$ is an $n \times n$ matrix whose (i,j) entry is 1 if $[\mathcal{B}(\mathbf{Q},\mathbf{K})]_{i,j} \neq 0$ else it is 0. Then, we can use the sparsity of $\mathcal{B}(\mathbf{Q},\mathbf{K})$ to save compute steps. The procedure initializes $\nabla_{\mathbf{q}_i}$ to zero and checks if $\mathbf{q}_i$ and $\mathbf{k}_j$ collide in any of M hashes. If so, it computes the weight

$$w_{ij} = ([\nabla_{\text{YOSO}}]_i^\top \mathbf{v}_j)\frac{\tau(1-\frac{\arccos(\mathbf{q}_i^\top\mathbf{k}_j)}{\pi})^{\tau-1}}{\pi\sqrt{1-(\mathbf{q}_i^\top\mathbf{k}_j)^2+\epsilon}}$$

where a small ϵ is introduced to avoid a divide by zero error, and then accumulates $w_{ij}\mathbf{k}_j$ to $\nabla_{\mathbf{q}_i}$. This procedure has the same runtime complexity as the procedure above. The models trained using this procedure for backpropagation computation is denoted as *YOSO in the experiment section.

3.4 Experiments

In this section, we analyze YOSO experimentally and evaluate its performance. In the previous section, we assumed that queries and keys are unit length and described how to make the strategy work. In the experiments, we found that simply applying a ℓ_2 normalization on queries and keys and using τ as a hyperparameter does not degrade the performance and yet is more efficient to compute, so we use the simpler version in the experiments.

For empirical evaluations, we evaluate YOSO-Attention on the BERT language model pretraining followed by GLUE downstream tasks finetuning. Then, we compare our method with other efficient Transformer baselines using a small version of BERT and the LRA benchmark. As a sanity check, we also include YOSO-Attention (YOSO-E) where the expected attention weights are directly computed using colli-

sion probability. This represents the behavior of YOSO when the number of hashes tends to infinity. We verified that in all tasks, YOSO-E behaves similarly as softmax self-attention. Further, we demonstrate that the performance of YOSO-m (YOSO-Attention using m hashes) generally converges to YOSO-E as m increases. Also, training using the backpropagation estimate in (3.5) (denoted YOSO) converges in all tasks, but the backpropagation estimate based on (3.4) (denoted *YOSO) is slightly better. When compared to other efficient Transformer baselines, we show that our proposal performs favorably while maintaining high efficiency for both time and memory. Finally, we empirically verified that the approximation error of YOSO-m stays relatively flat as the sequence length increases.

3.4.1 Language Modeling

To evaluate YOSO, we follow the BERT language model pretraining procedure (Devlin et al., 2019) and evaluate the performance of our method on both intrinsic tasks and multiple downstream tasks in the GLUE benchmark.

BERT Pretraining. Following Devlin et al. (2019), the model is pretrained on BookCorpus (Zhu et al., 2015) and English Wikipedia. To evaluate the capacity of the model in capturing sentence level information, instead of using Next-Sentence-Prediction (NSP) as the sentence level loss as in the original BERT, we adapt the Sentence-Ordering-Prediction (SOP) from ALBERT (Lan et al., 2020) – this is more difficult compared to NSP. All models are trained with Mask-Language-Modeling (MLM) and SOP objectives. We use the same hyperparameters for pretraining as Devlin et al. (2019). However, to keep the compute needs manageable, all models are trained for 500K steps (batch size of 256).

Number of Hashes during Pretraining. Since the estimation variance decreases as the number of hashes increases, to evaluate the trade-off between efficiency and performance in YOSO, we test multiple hash settings: (*)YOSO-16, (*)YOSO-32, YOSO-64, and finally, YOSO-E (to simulate infinite hashes). We plot MLM validation perplexity and SOP validation loss curves of 512 length models pretrained with softmax self-attention and YOSO-Attention (Figure 3.4) and show the MLM

validation perplexity and SOP accuracy obtained in Table 3.2. The curves of YOSO-E agree with and slightly exceed softmax self-attention, indicating that YOSO is indeed as effective as self-attention. It is expected that as the number of hashes increase, the performance of YOSO will approach YOSO-E, as the approximation becomes more accurate. For both MLM and SOP, we confirm that YOSO is as effective as softmax self-attention.

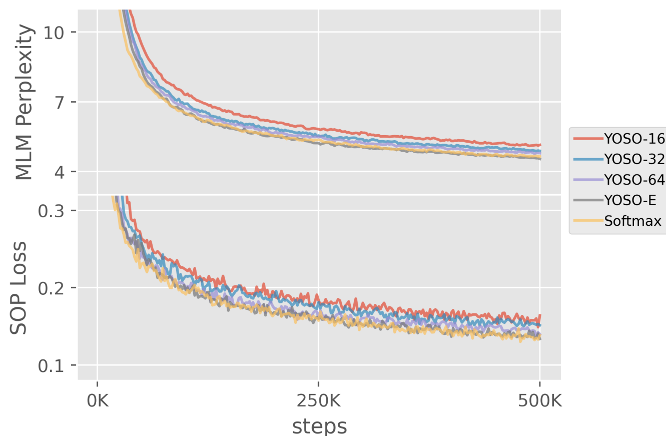

Figure 3.4: MLM and SOP result of 512 sequence length language model pretraining. YOSO-x means the model is pretrained with YOSO-Attention using x hashes with E being expectation.

Number of Hashes during Validation. YOSO-Attention is a stochastic model. To make the inference deterministic, as in dropout (Srivastava et al., 2014), ideally, we take the expectation as the output. However, directly computing the expectation involves a $O(n^2)$ cost, so we experiment with the effect of different hash settings in validation and simulate expectation as the number of hashes increases. We plot the MLM perplexity and SOP loss of the same pretrained models using different number of hashes on validation in Figure 3.5. We observe that as the number of hashes increases, the MLM perplexity and SOP loss generally decreases for all pretraining hash settings.

GLUE. We examined the effectiveness of our method on diverse downstream tasks and ask how YOSO compares with softmax self-attention even after finetuning. We

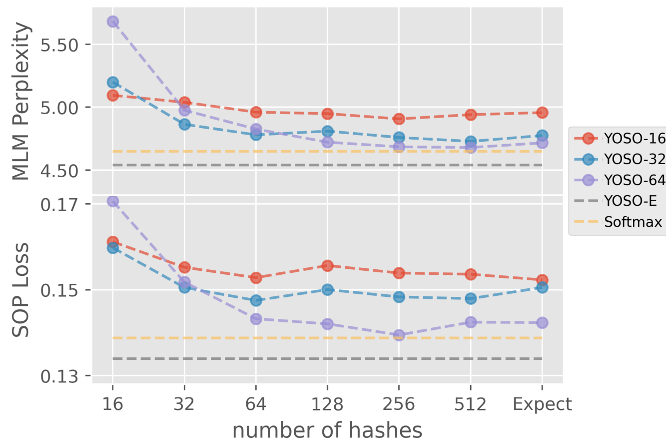

Figure 3.5: MLM and SOP result of pretrained pretraining when altering the number of hashes in inference.

finetuned all pretrained BERT-base model on MRPC (Dolan and Brockett, 2005), SST-2 (Socher et al., 2013), QNLI (Rajpurkar et al., 2016), QQP (Chen et al., 2018), and MNLI (Williams et al., 2018) tasks in the GLUE benchmarks and report their corresponding dev metrics. For large datasets including QNLI, QQP, and MNLI, due to extensive resource needs, we did not perform hyperparameter search, so we used a batch size of 32 and learning rate 3e-5 to update our model and finetune our models for 4 epochs. For MRPC and SST-2, we follow BERT finetuning to do a hyperparameter search with candidate batch size {8, 16, 32} and learning rate {2e-5, 3e-5, 4e-5, 5e-5} and select the best dev set result. Results are listed in Table 3.2. We observed that YOSO's performance on downstream tasks is comparable with softmax self-attention, and even shows slightly better results in some hash settings. Further, the downstream performance of YOSO generally increases with more hashes, providing an adjustable trade-off between efficiency and accuracy.

3.4.2 Performance considerations relative to baselines

We also evaluate how well our method performs compared to other efficient Transformer baselines. For the baselines, we compared YOSO with Nyströmformer (64

Method	MLM	SOP	MRPC	SST-2	QNLI	QQP	MNLI-m/mm
Softmax	4.65	94.2	88.3	91.1	90.3	87.3	82.4/82.4
YOSO-E	4.54	94.4	88.1	92.3	90.1	87.3	82.2/82.9
YOSO-64	4.79	94.2	88.1	91.5	89.5	87.0	81.6/81.6
YOSO-32	4.89	93.5	87.3	90.9	89.0	86.3	80.5/80.7
YOSO-16	5.14	92.8	87.1	90.7	88.3	85.3	79.6/79.5
*YOSO-32	4.89	93.5	87.6	91.4	90.0	86.8	80.5/80.9
*YOSO-16	5.02	93.4	87.7	90.8	88.9	86.7	80.6/80.5

Table 3.2: Dev set results on MLM and SOP pretraining and GLUE tasks for comparison to softmax self-attention in BERT-base setting. We report perplexity for MLM, F1 score for MRPC and QQP, and accuracy for others.

landmarks and 33 convolution size) (Xiong et al., 2021), Longformer (512 attention window size) (Beltagy et al., 2020), Linformer (256 projection dimensions) Wang et al. (2020), Reformer (2 hashes) (Kitaev et al., 2020), and Performer (256 random feature dimensions) (Choromanski et al., 2021) on a small version of BERT and LRA benchmark. The same model-specific hyperparameters are also used in efficiency profiles in Figure 3.6. Further, inspired by Nyströmformer, we also tested adding a depthwise convolution, which is referred as YOSO-C-x (x for the number of hashes). The experiment results indicate that depthwise convolution improves the performance of our method in some tasks.

BERT-Small. For BERT pretraining task, due to the large resource needs of running all baselines in BERT-base setting, we evaluate all methods in a BERT-small setting (4 layers, 512 dimensions, 8 heads) with 500K steps pretraining. Since the attention window size of Longformer is the same as the maximal sequence length of the input, it provides full self-attention, similar to softmax in this setting. Softmax self-attention achieves 7.05 MLM validation perplexity and 91.3% SOP validation accuracy on this task, and YOSO (with convolution) achieves 7.34 MLM validation perplexity and 89.6% SOP validation accuracy. Here, YOSO-C is comparable to softmax self-attention and Nyströmformer while it performs favorably relative to Reformer and Performer. For GLUE tasks, we found that 99% of instances in MRPC, SST-2, QNLI, QQP, and MNLI have sequence lengths less than 112. Since the chunked attention window in Reformer can capture full attention across all

tokens in this setting, we expect to see similar performance for Reformer as softmax self-attention. We provide results for QNLI, QQP, MNLI, and MLM and SOP pretraining tasks for all baselines in Table 3.3.

Method	MLM	SOP	QNLI	QQP	MNLI-m/mm
Softmax	7.05	91.3	87.7	86.0	79.1/79.5
Nyströmformer	7.60	90.2	84.3	84.0	75.5/76.0
Linformer	8.21	90.9	85.8	85.0	77.4/77.5
Reformer	8.57	89.0	86.9	86.2	78.2/78.6
Performer	11.59	88.9	82.8	83.7	73.8/74.6
YOSO-32	8.55	89.5	84.7	83.1	75.0/75.3
YOSO-C-32	7.72	89.7	85.1	83.9	75.8/75.8
*YOSO-32	8.43	89.1	84.9	84.4	76.7/77.0
*YOSO-C-32	7.34	89.6	84.9	84.7	77.2/77.2

Table 3.3: Dev set results on MLM and SOP pretraining and GLUE tasks for baseline comparisons.

LRA Benchmark. To evaluate the generalization of YOSO on diverse tasks and its viability on longer sequence tasks, we run our method on LRA benchmark (Tay et al., 2021) and compare it with standard Transformer as well as other efficient Transformer baselines. This benchmark consists of five tasks: Listops (Nangia and Bowman, 2018), byte-level IMDb reviews classfication (Text) (Maas et al., 2011), byte-level document matching (Retrieval) (Radev et al., 2013), pixel-level CIFAR-10 classification (image) (Krizhevsky et al., 2009), and pixel-level Pathfinder (Linsley et al., 2018). These tasks are designed to assess different aspects of an efficient Transformer and provide a comprehensive analysis of its generalization on longer sequence tasks.

Since the code release from Tay et al. (2021) only runs on TPUs, and the hyper-parameters are not known, we followed the experimental settings in Xiong et al. (2021). We include a model without self-attention, labeled “None", as a reference to show how much each baseline helps in modeling long sequences. The results are shown in Table 3.4. The performance of YOSO compared to softmax self-attention on LRA tasks is consistent with the results we reported for language modeling. For baseline comparisons, YOSO is comparable to Longformer and Nyströmformer

and outperforms all other baselines by 3% average accuracy across the five tasks. Further, with a depthwise convolution, YOSO outperforms all baselines. These results provide direct evidence for the applicability of YOSO on longer sequences.

Method Sequence Length	Listops 2K	Text 4K	Retrieval 4K	Image 1K	Pathfinder 1K	Avg
None	19.20	61.11	74.78	33.86	66.51	51.09
Softmax	37.10	65.02	79.35	38.20	74.16	58.77
YOSO-E	37.30	64.71	81.16	39.78	72.90	59.17
Nyströmformer	37.15	65.52	79.56	41.58	70.94	58.95
Longformer	37.20	64.60	80.97	39.06	73.01	58.97
Linformer	37.25	55.91	79.37	37.84	67.60	55.59
Reformer	19.05	64.88	78.64	43.29	69.36	55.04
Performer	18.80	63.81	78.62	37.07	69.87	53.63
YOSO-32	37.25	63.12	78.69	40.21	72.33	58.32
YOSO-C-16	37.40	64.28	77.61	44.67	71.86	59.16
*YOSO-16	37.20	62.97	79.02	40.50	72.05	58.35
*YOSO-C-16	37.35	65.89	78.80	45.93	71.39	59.87

Table 3.4: Test set accuracy of LRA tasks. Our proposed YOSO is comparable to Longformer and Nyströmformer and outperforms other baselines, and YOSO with a depthwise convolution outperforms all baselines.

3.4.3 Efficiency considerations relative to baselines

The overall thrust in efficient Transformer models is to have the same capacity as a standard Transformer while reducing the time and memory cost of self-attention. In this section, we profile the running time and memory consumption of our method as well as vanilla Transformer and other efficient Transformers for different sequence lengths. We use a Transformer model of 6 layers, 256 embedding dimension, 1024 hidden dimension, 4 attention heads and measure runtime and peak memory consumption using random inputs. To achieve the best efficiency for each baseline, for each method and each sequence length, we use the largest batch size we can fit into memory and run training for 10 steps and average the results to estimate the time and memory cost of a single instance. The experiments were performed on a single NVIDIA 2080TI. The result is shown in Figure 3.6. While Longformer scales linearly with respect to the sequence length, the benefit comes from longer

sequence, which is consistent to Beltagy et al. (2020). The profiling results indicate that our YOSO is able to scale efficiently with input sequence lengths. Further, the results suggest that our YOSO is highly efficient in terms of runtime and offers the highest efficiency in terms of memory compared to baselines.

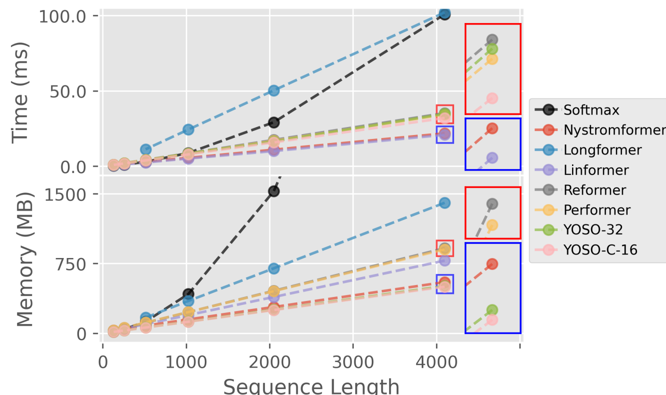

Figure 3.6: Running time and memory consumption results on various input sequence length. The reported values pertain to a single instance. Time is estimated by averaging total runtime and then dividing it by batch size, while memory is measured by dividing total memory consumption by batch size. Note that the trend lines are overlapping for several methods.

3.4.4 How large is the approximation error?

To assess the estimation error of YOSO, we generate attention matrices of YOSO using Q, K from a trained model and compare it against softmax self-attention. In Figure 3.8, visually, our method produces similar attention patterns as softmax self-attention. The estimation of attention matrix is more accurate as the number of hashes increases. Further, in the formulation of YOSO, each output of YOSO-Attention is a weighted sum of random variables as shown in (3.2); so one may suspect that as the sequence length increases, the variance of YOSO-Attention output might potentially increase. To assess the increase in variance, we use $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ from a trained model and measure the averaged angle between YOSO-E and YOSO-m for $M \in \{8, 16, 32, 64, 128\}$ for sequence lengths between 64 to 4096. Since YOSO

outputs unit vectors, only the vector direction is meaningful, we use the radian between outputs of YOSO-E and YOSO-m to assess the approximation error. The result is shown in Figure 3.7. The x-axis uses a logscale to verify that the approximation error increases at a much slower rate compared to the sequence length. One explanation is that most attention weights are near zero, see Figure 3.8. This means that the variance of the corresponding attention weights are near zero. In most cases, the size of the dependency (large attention weights) is relatively independent of the sequence length. As a result, the increase in sequence length does not introduce the same amount of error in approximation.

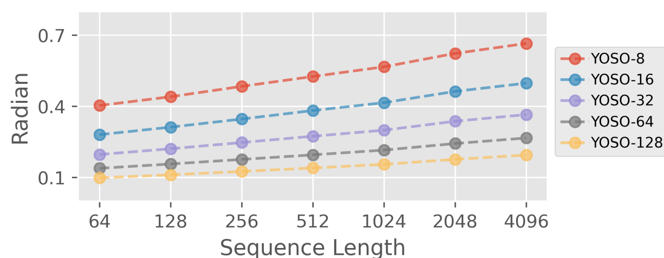

Figure 3.7: Averaged Radian between outputs of YOSO-E and YOSO-m for $m = 8, 16, 32, 64, 128$ for sequence length from 64 to 4096. The x axis is in logarithm scale, so we verify that the error only increase at a logarithm speed w.r.t. the sequence length.

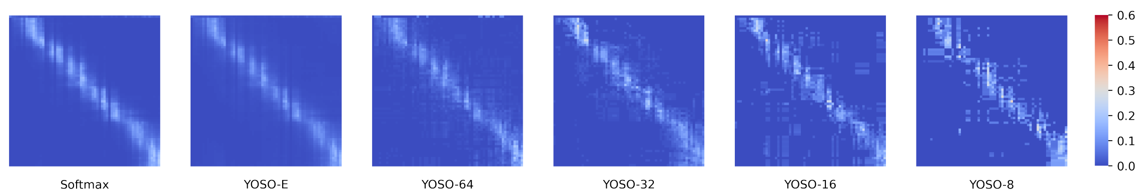

Figure 3.8: Attention matrices generated by Softmax and YOSO using the same input. We only visualize self-attention matrices for the first 64 tokens. Notice that the patterns are preserved well.

3.5 Summary

In this chapter, we presented an efficient self-attention mechanism, YOSO-Attention, which scales linearly in the number of tokens. Different from the common operations in deep learning like matrix multiplications, the operations in YOSO computation rely heavily on memory access, such as memory write and memory lookup, and use a very small compute footprint. Via a randomized sampling based scheme, YOSO approximates self-attention as a sum of individual tokens associated with Bernoulli random variables that can be sampled at once by a single hash, in principle. With specific modifications of LSH, YOSO-Attention can be efficiently deployed within a deep learning framework. We will develop this idea further in Chapter 4 A preliminary version of this chapter was published as (Zeng et al., 2021) and the codebase is available at `https://github.com/mlpen/YOSO`.

4 MEMORY LOOKUP BASED APPROXIMATION FOR COMPUTE-LITE FEEDFORWARD NETWORK

The Feedforward Network (FFN) is one of the two major building components of a Transformer block as we discussed in §2.2. FFNs are also essential components in almost all deep neural networks. They heavily rely on General Matrix Multiply (GEMM), which is extremely compute intensive, especially for large scale models common in the community today. Many alternatives Chen et al. (2020, 2021b); Fedus et al. (2022); Zhang et al. (2018); Moczulski et al. (2016) have been proposed to reduce the compute needs of FFNs. One popular line of work shows how to use Locality Sensitive Hashing (LSH) discussed in §2.4, to address the computational bottleneck of feed-forward via adaptive sparsity. For example, Slide Chen et al. (2020) uses LSH to retrieve a small subset of units that omit high activation via maximum inner product search (MIPS) discussed in §2.4 and only computes the outputs of these units, resulting in a sparse network. However, LSH poses certain difficulties, such as the need for a large number of hash functions, skewed workload to due imbalanced hash buckets, and the overhead for rehashing due to the constantly evolving parameters, which we will discuss in more detail.

In this chapter, motivated by the aforementioned limitations of LSH-based approaches in the context of FFN, we develop a formulation of end-to-end learnable memory lookup for FFNs: LookupFFN. Specifically, we propose to modify the formulation discussed in Chapter 3 and directly view the hash functions and hash tables as learnable modules. Projections within hash functions are handled via a specialized module based on the fast Hadamard transform discussed in §2.3, which may be of independent interest. We show that the skewness of bucket distribution becomes irrelevant in our proposal. Since there are no parameter matrices as in Chen et al. (2021b, 2020), rehashing can be completely avoided, and the gradient updates are performed directly on the hash functions and hash tables. The proposed formulation is differentiable, and no special optimization for the hash

modules is needed. In practice, LookupFFN can simply be integrated into common DNN models, optimization flows, and software frameworks. Meanwhile, since LookupFFN is a reformulation of YOSO algorithm in Chapter 3, the algorithm shares similar compute characteristics as YOSO: relying mostly on memory lookup with minuscule compute footprint, making it ideal for compute-lite hardware, such as CPUs.

In this chapter, we also move outside of the common GPU computing setting and explore the potential efficiency benefits of deploying this method on CPUs, which is growing in importance as evidenced by recent server chip announcements from IBM, Intel, AMD and ARM (Lichtenau et al., 2022; Intel; Nassif et al., 2022; AMD; Bhat, 2021) and academic efforts (Liu et al., 2019b; Mittal et al., 2022; Nori et al., 2021; Zhang et al., 2019). Some of the technical and business motivations include latency, security, privacy and the fact that modern data-intensive workloads have AI tasks embedded in a pipeline of non-AI tasks. Further, CPUs are a generic platform common across servers and clients faithfully serving the compute needs of businesses, which makes it attractive. And finally – it comes down to the cost of running the full workload. Unfortunately, CPU chips lack the computational intensity of raw-FLOPS compared to GPUs. On the positive side, CPUs provide tremendously large caches in the range of 128MB to 192MB, and even larger (Burd et al., 2022), which is currently under-utilized. Furthermore, such caches made out of SRAMs are more than an order of magnitude more energy efficient to access compared to DRAMs (DDR, GDDR, or HBM) (Jouppi et al., 2021; Horowitz, 2014), while providing $4\times$ access bandwidth increase [1]. This low compute and high memory capacity seems to align well with the compute characteristics of our LookupFFN formulation developed in this chapter.

Based on measurements and analytical calculations, we estimate $6\times$ (or more) reduction in FLOP compared to a vanilla FFN with almost the same accuracy. Even though our formulation requires somewhat large tables (16MB and more),

[1] AMD Zen2 for example, allows 64 bytes per cycle into each core: with 32 cores running 3.2 GHz that amounts to 6 TB/sec.

with careful algorithm design, we can make the access pattern somewhat cache-friendly – achieving nearly 80% L1 cache hit rate. In particular, hardware-managed caches work well, avoiding the need for excessive optimization for the software-managed shared-memory of a GPU. In practice, this means that we are able to reduce the energy consumption of 80% of access to SRAMs (14 pico joules or pj per 64-bit access), versus 300 to 450 pj for DRAM-based access. We show that, on contemporary hardware, for inference, we are 2.51× faster than a vanilla FFN. With new technology like 3D caches appearing in CPUs, we expect SRAM based bandwidth to increase even further, making LookupFFN integer factors faster as memory technology and packaging improvements continue. While this formulation has more memory lookups compared to GEMM-based FFN, the majority of the memory lookups are independent of each other. This means LookupFFN heavily relies on high memory throughput but has high tolerance to memory latency – making designing software (and potentially hardware) implementation easier.

4.1 Preliminaries

In this section, we describe some related LSH based approaches targeted to address the efficiency of FFN and their limitations, which inspire our work.

Given an input $\mathbf{X} \in \mathbb{R}^{N \times D}$, the FFN $\mathcal{F}(\mathbf{X})$ is a point-wise operation applied to each row of input matrix $\mathbf{X}$. Let $\mathbf{x} \in \mathbb{R}^{D}$ be any row of $\mathbf{X}$, T be the hidden dimension, and $\mathbf{W} \in \mathbb{R}^{T \times D}$ and $\mathbf{V} \in \mathbb{R}^{T \times D}$ be two parameter matrices in $\mathcal{F}(\cdot)$. Then, the output $\mathbf{y}$ of FFN is defined as

$$\mathbf{y} = \mathcal{F}(\mathbf{x}) = \sigma(\mathbf{x}\mathbf{W}^{\top})\mathbf{V} = \sum_{i=1}^{T} \sigma(\langle \mathbf{x}, [\mathbf{W}]_i \rangle)\, [\mathbf{V}]_i \tag{4.1}$$

The authors in Slide (Chen et al., 2020) observed that when σ is a softmax, the output of a FFN is dominated by only a few entries of $\sigma(\langle \mathbf{x}, [\mathbf{W}]_i \rangle)$ and proposed a sparse FFN, which uses LSH to perform a maximal inner product search (MIPS)

among $[\mathbf{W}]_i$ for large $\sigma(\langle \mathbf{x}, [\mathbf{W}]_i \rangle)$ terms. Only activations of the search results $\mathbb{S}(\mathbf{x})$ are computed to approximate the full softmax with a reduced compute burden,

$$\mathbf{y} \approx \sum_{i \in \mathbb{S}(\mathbf{x})} \sigma(\langle \mathbf{x}, [\mathbf{W}]_i \rangle) \, [\mathbf{V}]_i \tag{4.2}$$

Constructing $\mathbb{S}(\mathbf{x})$ requires a pre-processing step that hashes $[\mathbf{W}]_i$ into multiple hash tables, and a querying step that hashes $\mathbf{x}$ to these hash tables and collects all $[\mathbf{W}]_i$ from the buckets that $\mathbf{x}$ is hashed to. There are some problems with this construction. **Rehashing:** Since $[\mathbf{W}]_i$ are constantly updated while training, the hash tables need to be constantly updated or re-constructed, referred to as rehashing. **Large #-hashes:** The LSH relies on the randomness of hash functions, so a large number of hash functions are used to obtain accurate MIPS result resulting in high query time (see left plot of Figure 4.1). **Bucket skewness:** The LSH bucket distribution is skewed, so the number of $[\mathbf{W}]_i$ in different buckets are quite different and there is no control of how many $[\mathbf{W}]_i$ will be hashed into one bucket (see right plot of Figure 4.1). Therefore, $|\mathbb{S}(\mathbf{x})|$ varies for different inputs. This skewness makes the workload difficult to be parallelized.

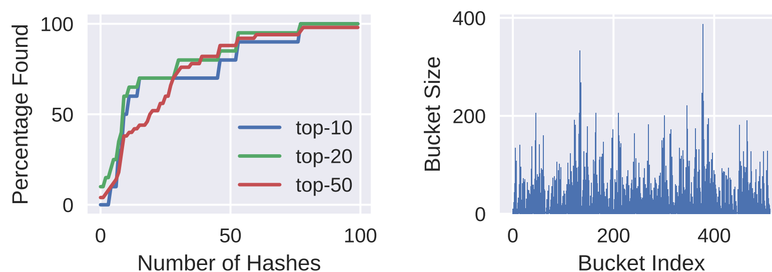

Figure 4.1: The left plot shows the number of hash functions used versus the percentage of top-x nearest neighbors found using these hashes. A large number of hash functions are needed for accurate MIPS result. The query time is linearly proportional to the number of hash functions. The right plot shows the bucket size of each bucket. It visualizes the bucket skewness issue.

Mongoose (Chen et al., 2021b) proposed a scheduler to reduce the frequency of rehashing and learnable hash functions to learn data-dependent hashing. So, the

number of hash functions can be reduced without sacrificing MIPS quality, thereby Chen et al. (2021b) partially reduces the **Rehashing** and **large #-hashes** issues but introduces an additional auxiliary learning component for learnable hashing. Further, the **bucket skewness** still persists. In Chapter 3, we described YOSO for approximating self-attention in Transformer models, which can be extended to approximating FFNs. We show that when σ is similar to the collision probability of LSH, instead of keeping track of $\mathbb{S}$ (as in Slide), one can store the summation of $[\mathbf{V}]_i$'s in hash buckets where $[\mathbf{W}]_i$ are hashed to, which is analogous to the LSH pre-processing step. Let $\mathcal{H}_k$ be a hash function, and $\mathbf{T}_k \in \mathbb{R}^{2^\tau \times D}$ be a hash table representing 2^τ D-dimensional buckets. Here, τ is the length of hash code as used in Chapter 3.

$$
\begin{aligned}
[\mathbf{T}_k]_j &= \frac{1}{H} \sum_{\mathcal{H}_k([\mathbf{W}]_i)=j} [\mathbf{V}]_i \\
\mathbf{y} &\approx \sum_{k=1}^{H} [\mathbf{T}_k]_{\mathcal{H}_k(\mathbf{x})}
\end{aligned}
\tag{4.3}
$$

Then, in the LSH querying step, we can directly estimate $\mathbf{y}$ by computing an average of one bucket of multiple $\mathbf{T}_k$ with a consistent compute cost. Therefore, the **bucket skewness** issue is solved. However, the estimation of YOSO relies on the randomness of $\mathcal{H}_k$, so it requires a large number of hash functions for a good estimate. Further, since $\mathbf{W}$ and $\mathbf{V}$ are evolving during training, $\mathbf{T}_k$ needs to be recomputed after every parameter update, which is inefficient. The **rehashing** and **large #-hashes** problems remain open.

None of the foregoing methods can resolve all issues. In particular, no method solves **rehashing** – all of them require rehashing when the parameters are updated. One of our goals is to completely eliminate the need for rehashing, and remove the dependency of workload on the bucket size. Further, we also hope to obtain a scheme that, if desired, can be trained end-to-end via back-propagation.

4.2 FFN as Lookups

Here, we present an end-to-end construction for differentiable table lookups as an efficient alternative to GEMM for FFNs where most operations are memory lookups.

4.2.1 Differentiable Lookup

To avoid the impact of skewed bucket distribution on efficiency, we start from (4.3) and attempt to adjust the formulation in the setting where it is used as a FFN. The randomness of $\mathcal{H}_k$ in (4.3) is the key ingredient of YOSO (Chapter 3), but at the same time, this randomness introduces the need for a large number of hash functions to get an accurate approximation. This issue must be handled. Separately, we try to completely avoid any pre-processing steps or rehashing for evolving parameters $\mathbf{W}$ and $\mathbf{V}$.

Main idea. Observe that for YOSO, the $\mathcal{H}_k$ is a partition of the $\mathbb{R}^d$ space and each hash table $\mathbf{T}_k$ is a coarse representation of $\mathcal{F}(\cdot)$ associated with $\mathcal{H}_k$. Whenever $\mathcal{H}_k$, $\mathbf{W}$, or $\mathbf{V}$ are updated, $\mathbf{T}_k$ needs to be updated. This is inefficient. But $\mathbf{T}_k$ is a coarse representation of a parameterized function, so we hypothesize that we might be able to directly optimize the coarse representation $\mathbf{T}_k$ and $\mathcal{H}_k$ to minimize the loss of the model. If possible, we also want to make it differentiable. If this is achieved, this strategy helps avoid any rehashing necessary in (4.2) and (4.3). Therefore, we consider the formulation

$$\xcancel{[\mathbf{T}_k]_j := \frac{1}{H} \sum_{\mathcal{H}_k([\mathbf{W}]_i)=j} [\mathbf{V}]_i} \qquad \mathbf{y} := \sum_{k=1}^{H} [\mathbf{T}_k]_{\mathcal{H}_k(\mathbf{x})} \tag{4.4}$$

where $\mathbf{T}_k$ and $\mathcal{H}_k$ are learnable modules. Here, the dependency of $\mathbf{T}_k$ on $\mathcal{H}_k, \mathbf{W}, \mathbf{V}$, as in (4.3), is removed. Figure 4.2 is a visualization of the difference comparing LSH-based FFNs. This decoupled dependency creates a problem in that the resultant formulation is not differentiable. In Chapter 3, we used the fact that (4.3) is an estimate of a differentiable function, and use the gradient of this function as an

estimate of the gradient of (4.3), however, this estimate relies on the randomness of $\mathcal{H}_k$ which is not available after decoupling in (4.4). So, the challenge is how we can train $\mathcal{H}_k$ and $\mathbf{T}_k$, and backpropagate to shallower layers.

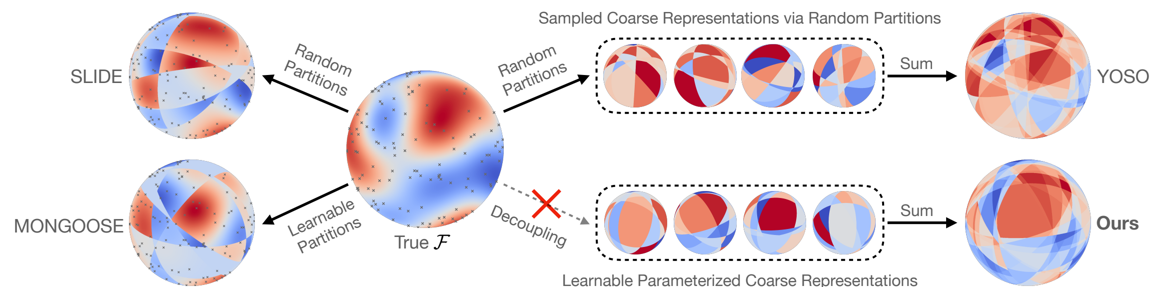

Figure 4.2: High level comparison of each method. The true $\mathcal{F}(\cdot) = \sum_{i=1}^{h} \sigma(\langle \cdot, [\mathbf{W}]_i \rangle) [\mathbf{V}]_i$ is constructed as a function in $\mathbb{S}^2$. Here, $[\mathbf{W}]_i \in \mathbb{S}^2$ and $[\mathbf{V}]_i \in \mathbb{R}$. The points $[\mathbf{W}]_i$ are marked in the left three figures. SLIDE, MONGOOSE, and YOSO try to construct an approximation of the true $\mathcal{F}(\cdot)$ via different uses of LSH partitions, so whenever $\mathcal{F}(\cdot)$ is updated, the LSH partitions need to be updated. Rather than approximating the function $\mathcal{F}$, our proposed method is plugged into a deep learning model and uses the back-propagated gradient to learn appropriate transformation similar to a vanilla FFN.

Making (4.4) differentiable again. To figure out a solution, we need to first dive into how $\mathcal{H}_k$ is computed. In Chapter 3, we use the hyperplane hash (Charikar, 2002) to compute the hash code. Specifically, define $\mathbf{z}_k = \mathbf{x}\mathbf{R}_k$, referred to as the "soft hash code", where $\mathbf{R}_k \in \mathbb{R}^{D \times \tau}$ is a random projection associated with $\mathcal{H}_k$ where τ is the length of binary representation of the hash code.

$$\mathcal{H}_k(\mathbf{x}) = \text{decimal}(\text{sign}(\mathbf{z}_k))$$

Here, decimal is a function that maps the binary representation $\{\pm 1\}^\tau$ to a decimal representation $\{0, \cdots, 2^\tau - 1\}$. This form does not directly suggest a method for back-propagation, but observe that $\mathcal{H}_k$ can be expressed as

$$\mathcal{H}_k(\mathbf{x}) = \arg\max_i (\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle)$$

where $\mathbf{S} \in \{\pm 1\}^{2^\tau \times \tau}$ is a structured matrix whose row vector

$$[\mathbf{S}]_i = \text{decimal}^{-1}(i)$$

is the binary representation of the integer i. An illustration of $\mathbf{S}$ is shown in Figure 4.3. While, by itself, this does not solve our problem, a common differentiable relaxation of arg max is the softmax activation, and the resultant formulation for (4.4) is,

$$\hat{y}^* := \sum_{k=1}^{H} \sum_{i=1}^{2^\tau} \frac{\exp(\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle) \, [\mathbf{T}_k]_i}{\sum_{j=1}^{2^\tau} \exp(\langle \mathbf{z}_k, [\mathbf{S}]_j \rangle)} \tag{4.5}$$

Then, by replacing the random matrix $\mathbf{R}_k$ with a learnable parameter matrix, this formulation makes $\mathcal{H}_k$ a learnable hash function and $\mathbf{T}_k$ a learnable coarse representation of a function in $\mathbb{R}^D$ in an end-to-end manner.

-1	-1	-1
-1	-1	1
-1	1	-1
-1	1	1
1	-1	-1
1	-1	1
1	1	-1
1	1	1

Figure 4.3: Illustration of structure matrix $\mathbf{S}$ for $\tau = 3$

Remaining difficulties and solutions. A naive implementation of this operation is extremely inefficient and has a runtime complexity of $\mathcal{O}(H2^\tau D)$, which is not practical. A common choice of efficient softmax approximations is to use a small subset of softmax numerators (we denote the set of corresponding indices as $\mathbb{N}(\mathbf{z}_k)$) to approximate the full softmax since the softmax is usually dominated by only a few entries within it (Spring and Shrivastava, 2017; Charikar and Siminelakis, 2017). Non-uniform sampling, such as LSH-based importance sampling, as described in Chapter 3, can be used to lower the estimation variance. However, we found that the structured matrix $\mathbf{S}$ used in (4.5) offers several properties that actually enables

efficient approximation. Due to the structure of $\mathbf{S}$, the denominator of (4.5) can be rewritten as

$$\sum_{j=1}^{2^\tau} \exp(\langle \mathbf{z}_k, [\mathbf{S}]_j \rangle) = \prod_{j=1}^{\tau} (\exp([\mathbf{z}_k]_j) + \exp(-[\mathbf{z}_k]_j))$$

which only involves a $\mathcal{O}(\tau)$ cost. For calculating the numerator, we use a simple non-uniform sampling scheme for a better approximation of the softmax with a small number of samples. Due to the structure of $\mathbf{S}$, we easily know the approximate sorting order of $\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle$ among different i. Specifically, note that

$$\begin{aligned} \arg\max_i(\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle) &= \text{decimal}(\text{sign}(\mathbf{z}_k)) \\ \arg\min_i(\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle) &= \text{decimal}(-\text{sign}(\mathbf{z}_k)) \end{aligned} \tag{4.6}$$

When the order of magnitudes for different entries of $\mathbf{z}_k$ are not too different, the $\|[\mathbf{S}]_i - \text{sign}(\mathbf{z}_k)\|_0$ term roughly indicates the magnitude of $\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle$. A smaller distance means a larger value.

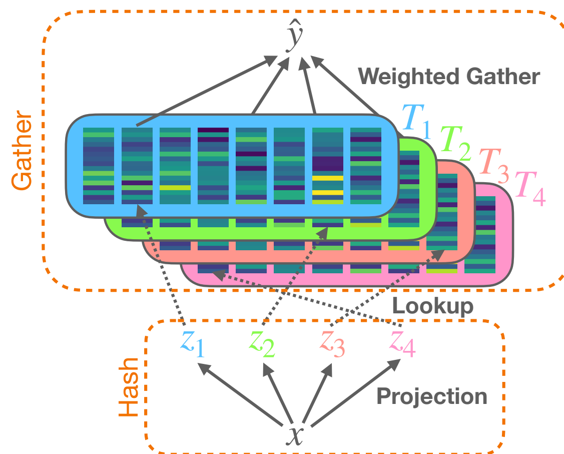

Figure 4.4: Illustration of LookupFFN operations.

Therefore, we use the approximation

$$\hat{\mathbf{y}} = \sum_{k=1}^{H} \sum_{i \in \mathbb{N}(\mathbf{z}_k)} \frac{\exp(\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle)\, [\mathbf{T}_k]_i}{\prod_{j=1}^{\tau} (\exp([\mathbf{z}_k]_j) + \exp(-[\mathbf{z}_k]_j))} \tag{4.7}$$

where $\mathbb{N}(\mathbf{z}_k)$ can be easily sampled according to ℓ_0 difference $\|[\mathbf{S}]_i - \text{sign}(\mathbf{z}_k)\|_0$. It is much easier compared to other non-uniform sampling based softmax approximations (Spring and Shrivastava, 2017; Charikar and Siminelakis, 2017) since we can sample large numerators based on the number of sign flips away from $\text{sign}(\mathbf{z}_k)$. Further, we empirically found that in most cases, just using the largest numerator, i.e., let

$$g(\mathbf{z}_k) := \arg\max_i (\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle) \tag{4.8}$$

computed via (4.6), then $\mathbb{N}(\mathbf{z}_k) = \{g(\mathbf{z}_k)\}$ is sufficient for performance. This is our default choice for experiments.

Two main operations. The proposed learnable lookup consists of two operations: **(a)** Hash: we compute multiple $\mathbf{z}_k = \mathbf{x}\mathbf{R}_k$ for $k = 1, 2, \cdots, h$. Then, **(b)** Gather: we use $g(\mathbf{z}_k)$ (defined in (4.8)) for memory lookup and calculate a weighted (based on $\mathbf{z}_k$) accumulation of the lookup results $[\mathbf{T}_k]_{g(\mathbf{z}_k)}$. This procedure is illustrated in Figure 4.4.

Remark 4.1. *While* (4.7) *might look unfamiliar, it is closely connected to two commonly used FFNs. When* σ *in* (4.1) *is the sigmoid activation, let* $\mathbf{z}_k = 0.5\langle \mathbf{x}, [\mathbf{W}]_k \rangle$*, we note that* (4.1) *can be rewritten as*

$$y = \sum_{k=1}^{H} \frac{\exp(\mathbf{z}_k)\, [\mathbf{V}]_k}{\exp(\mathbf{z}_k) + \exp(-\mathbf{z}_k)}$$

which is just a special case of (4.7) *with* $\tau = 1$*. When* σ *is a GELU* (*Hendrycks and Gimpel*, *2016*) *commonly used in Transformer models, let* $\mathbf{z}_k = 0.851\langle \mathbf{x}, [\mathbf{W}]_k \rangle$*, then a fast*

approximation of GELU can be written as

$$y = \sum_{k=1}^{H} \frac{1.175\mathbf{z}_k \exp(\mathbf{z}_k)\,[\mathbf{V}]_k}{\exp(\mathbf{z}_k) + \exp(-\mathbf{z}_k)}$$

This is again a special case of (4.7) *with* $\tau = 1$ *and an additional linear scaling* $1.175\mathbf{z}_k$*. This scaling can be incorporated in* (4.7) *by an extra term* $1.175\langle \mathbf{z}_k, [\mathbf{S}]_i \rangle$ *in the numerator.*

4.3 BH4: Efficient and Expressive Projection for Hashing

The problem. In practice, we compute the "soft hash code" $\mathbf{z}_k$ for multiple hash tables at once by computing $\mathbf{xR}$ where $\mathbf{R} \in \mathbb{R}^{d\times(h\tau)}$. Here, $\{\mathbf{z}_1, \cdots, \mathbf{z}_h\}$ are computed at once by partitioning $\mathbf{xR}$ into h τ-dimensional vectors (τ is the length of hash code). The time complexity for this projection is $\mathcal{O}(h\tau d)$. This is not desirable since it is compute heavy.

Some existing solutions yield unsatisfactory results. For simplicity, we assume $h\tau = d$ and d is power of 2. When computing hash codes in the LSH setting, a common efficient alternative is efficient random projections implemented via a fast Hadamard transform with $O(d\log(d))$ cost. For example, YOSO and Andoni et al. (2015) use

$$\mathbf{R} := \mathbf{D}_1\mathbf{H}\mathbf{D}_2\mathbf{H}\mathbf{D}_3\mathbf{H} \tag{4.9}$$

where $\mathbf{D}_i$ are matrices whose entries are $\{\pm 1\}$ for random sign flipping and $\mathbf{H}$ is Hadamard transform. A simple learnable extension would be to replace $\mathbf{D}_i$ with parameterized diagonal matrices. This belongs to a large family of structured efficient linear layers (SELLs) (Cheng et al., 2015; Le et al., 2013; Yang et al., 2015; Moczulski et al., 2016) For example, Moczulski et al. (2016) proposes a deep SELL,

named ACDC, to increase the representation power:

$$\mathbf{R} := \prod_{i=1}^{K} \mathbf{A}_i \mathbf{C} \mathbf{D}_i \mathbf{C}^{-1} \tag{4.10}$$

where $\mathbf{A}_i, \mathbf{D}_i$ are parameterized diagonal matrices and $\mathbf{C}$ is the discrete cosine transform and K is a hyper-parameter. A similar construction, but using the Hadamard transform would involve replacing $\mathbf{C}$ with $\mathbf{H}$. This can be viewed as a generalization of (4.9). To empirically evaluate the representation power of each efficient projection, we use a toy problem of approximating a randomly generated matrix using these ideas. The results are shown in Figure 4.5. We find that the representation power of (4.10) and its Hadamard transform variant for small K is extremely limited, but for large K, the efficiency is low and the optimization becomes difficult. We can verify this optimization difficulty from the fact that as K increases, the FLOP and parameter count increases, but the squared errors do not monotonically decrease.

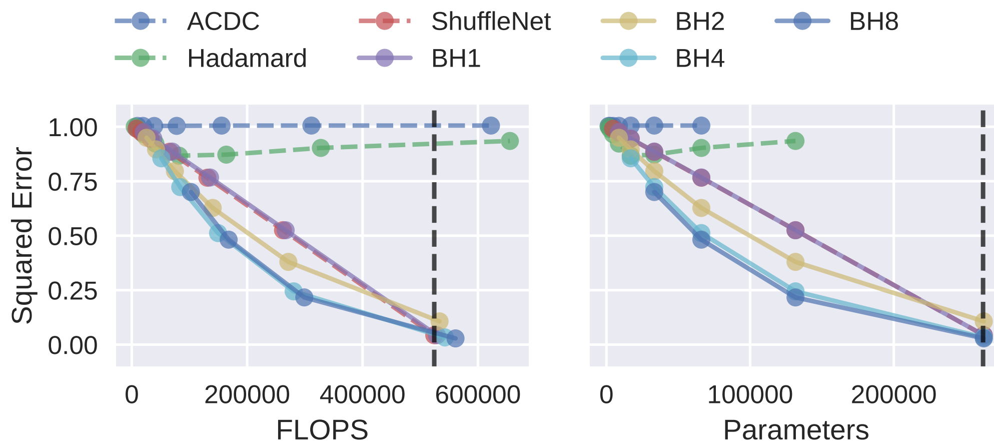

Figure 4.5: Approximation capacity vs FLOPs and parameters for each efficient projections. Hadamard denotes a variant of ACDC by replacing discrete cosine transform with Hadamard transform. The vertical dash lines are the FLOPs and parameters of the vanilla projection. Any results to the right of the vertical dashed lines are not meaningful, as there is no efficiency gain.

A simple yet highly effective scheme. To address the optimization difficulty for

large k, we propose that instead of scaling k, we can replace the diagonal matrices with their block diagonal counterparts, and scale the block size for the trade-off between efficiency and expressiveness.

$$\mathbf{R} := \prod_{i=1}^{M} \mathbf{B}_i \mathbf{H} \tag{4.11}$$

Here, $\mathbf{B}_i$'s are parameterized block diagonal matrices with an adjustable block size. We refer to (4.11) as BH{m} for different m. Figure 4.6 is a visualization of the projections discussed. We empirically verify that (4.11) with $M = 4$ has a better trade-off between the expressiveness and efficiency compared to other M values. When the FLOP or parameter counts are similar, a larger M does not increase its expressiveness, but a smaller M decreases its expressiveness.

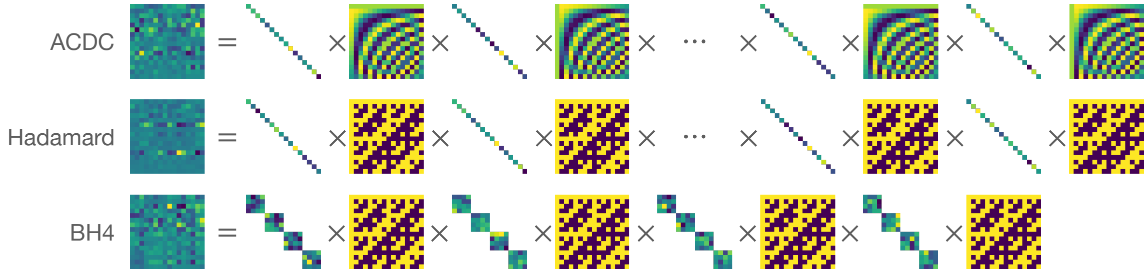

Figure 4.6: Visualization of efficient projections. ACDC and its Hadamard variant increase their capacity by increasing the depth. BH4 increases its capacity by increasing its block size.

Remark 4.2. *Grouped convolution followed by channel shuffling in ShuffleNet* (Zhang et al., 2018) *can be directly expressed as* **BF***: applying a block diagonal matrix followed by a structure transform* **F***. ShuffleNet can be viewed as a BH1. We empirically verified that BH1 has near-identical approximation accuracy as ShuffleNet as shown in Figure 4.5.*

4.4 Experiments

In this section, we will present our empirical results evaluating the benefits/limitations of replacing VanillaFFN with a LookupFFN in a Transformer, and conduct a

detailed performance profiling of LookupFFN.

Note. Slide and Mongoose report experiments on a two layer neural network whose last dimension is 200K (or even 670K) (Chen et al., 2020, 2021b). This is a synthetic setting to extract peak practical speedup of FFN in Slide/Mongoose because more than 99% of compute occurs in the final layer. In practical situations, such models are rare and might not reflect the actual workload of commonly used models, such as the Transformer. We are more interested in how these baselines and our method can be applied in more challenging commonly used DNN models, and what the corresponding performance impact is. As a result, we use the Transformer as a testbed to evaluate the effect of replacing VanillaFFN with baselines and our LookupFFN.

Outline. In §4.4.1, we compare LookupFFN's performance and FLOP reduction to baselines, and check that our formulation scales without difficulty to full size (12-layer) models. Then, in §4.4.2, to better understand its behavior, we perform an ablation study to study the effects of different hyper-parameters specific to lookup-based FFN. Finally, in §4.4.3, we analyze the performance characteristics for LookupFFN. Since our goal is to reduce the FLOP count, for comparison or even individual assessment of LookupFFN, we include the estimated FLOP count next to the model performance for each table (for comparisons). FLOPs are estimated as the number of floating operations of processing a single instance (a single token in the context of a Transformer).

Two variants are discussed in §4.2.1 corresponding to two different activations. To align well with the GELU activation used in Transformer, we use the linearly scaled variant of (4.7) with $\mathbb{N}(\mathbf{z}_k) = \{g(\mathbf{z}_k)\}$:

$$y = \sum_{k=1}^{H} \frac{\langle \mathbf{z}_k, [\mathbf{S}]_{g(\mathbf{z}_k)} \rangle \exp(\langle \mathbf{z}_k, [\mathbf{S}]_{g(\mathbf{z}_k)} \rangle) \, [\mathbf{T}]_{g(\mathbf{z}_k)}}{\prod_{j=1}^{\tau} (\exp([\mathbf{z}_k]_j) + \exp(-[\mathbf{z}_k]_j))}$$

Implementation Details. We used PyTorch (Paszke et al., 2019) for the majority of the implementation. On the GPU, our fast Hadamard Transform and weighted

gather operators are not supported by PyTorch so we implemented custom CUDA kernels to support the operators for training. For CPU, we implemented these kernels in C++ using OpenMP for inference which uses AVX2 vector instructions.

Evaluation Task. For empirical evaluations, we use RoBERTa language modeling pretraining (Liu et al., 2019a) as our evaluation tool to measure the method performance, since it is a challenging task. The models are pretrained using masked language modeling (Devlin et al., 2019) on the English Wikipedia corpus (Wikimedia, 2019). We pretrain each model for 250K steps with a batch size of 256, where each sequence is of length 512. We use an Adam optimizer with 1e-4 learning rate, 10,000 warm-up steps, and linear decay. To keep compute reasonable, we use RoBERTa with 4 layers and 512 embedding dimensions for model evaluation except one stress-test experiment checking the scaling behavior of LookupFFN to a full size RoBERTa-base.

4.4.1 Performance Comparison

Comparing to Baselines. We compare our method to Vanilla FFN, Slide (Chen et al., 2020), Mongoose (Chen et al., 2021b), and YOSO (Zeng et al., 2021) based FFNs discussed previously. Additionally, for comparison to more baselines, we include Switch Transformer (Fedus et al., 2022) and the grouped convolution + channel shuffling introduced in ShuffleNet (Zhang et al., 2018). Others have identified that the original implementation of Slide, which is implemented from scratch in C++, is difficult to be adopted (Chen et al., 2021b), and have propose optimized variants, which we use here (Chen et al., 2021b). Instead of each instance in a batch retrieving its own subset of weights, the union of the retrieved subsets is used for computation. This strategy is used to avoid an irregular workload due to the skewed bucket distribution, as discussed earlier. The size of $\mathcal{S}(\cdot)$ is larger for larger batches. We note that in a Transformer model, the effective batch size for a FFN is the number of sequences $\times$ the sequence length. The union of retrieved subsets will simply contain the entire set of weights. For a more reasonable size of set $\mathcal{S}(\cdot)$, we partition the effective batch into smaller mini-batches (128 tokens for Slide and

2048 tokens for Mongoose) and feed them into the FFN sequentially. This would severely increase the runtime of training on GPUs. The size of mini batch is set such that it is small enough but running the experiment is still feasible. Further, since the performance of Slide (Chen et al., 2020), Mongoose (Chen et al., 2021b), and YOSO (Zeng et al., 2021) based FFN largely depends on the frequency of rehashing, we perform rehashing after every parameter updates (this overhead is not counted towards FLOP) to ensure their optimal performance. The results are summarized in Table 4.1. Our method achieves lower perplexity using fewer FLOP compared to the baselines. Further, the FLOP count of our method can be significantly reduced with some loss in performance (but still better than the baselines except for the VanillaFFN) as shown in the last row of Table 4.1. Additional results are discussed in §4.4.2.

Method	h	τ	MFLOP	Log Perplexity
VanillaFFN	-	-	4.19	1.78
Switch Transformer	-	-	2.11	1.85
Channel Shuffle	-	-	2.10	1.96
Slide	-	-	1.32	1.98
Mongoose	-	-	3.21	1.87
YOSO	-	-	0.35	2.13
LookupFFN	256	8	1.38	1.74
	128	8	0.69	1.81

Table 4.1: Log perplexity of each baseline. (lower is better) LookupFFN was tested with two different hyper-parameter configurations specified in h and τ columns.

Downstream finetuning. Further, we evaluate the quality of the pretrained language models for VanillaFFN and our LookupFFN on MNLI downstream task (Williams et al., 2018) in the GLUE benchmark (Wang et al., 2018a). The result is shown in Table 4.2. We note that there is a small gap between our method and vanilla FFN, but the FLOP count of our method is much lower.

Scaling to Full Size Models. We check whether LookupFFN scales to a larger model, so we evaluate our method on a RoBERTa-base pretraining. All pretraining hyper-parameters remain the same as before. Due to the compute burden of training a full size model, we only perform one experiment comparing LookupFFN with

Method	h	τ	MFLOP	MNLI-m/mm
VanillaFFN	-	-	4.19	75.0/76.3
LookupFFN	256	4	0.82	74.1/74.7

Table 4.2: Downstream performance of RoBERTa-small models.

$h = 170$ and $\tau = 9$ to VanillaFFN. The results are shown in Table 4.3. Our method achieves 6.8× FLOP reduction while the log perplexity is only higher by 0.04 in a RoBERTa-base model.

Method	h	τ	MFLOP	Log Perplexity
VanillaFFN	-	-	9.44	1.37
LookupFFN	170	9	1.39	1.41

Table 4.3: Log perplexity when scaling to a RoBERTa-base model.

Memory use. Our LookupFFN requires more memory (for large h and τ) since we directly parameterize the hash tables, but we believe this is not a key issue. In a GPU setting, memory use is critical since the GPU memory is usually much more expensive and limited. On the other hand, CPU memory is much cheaper and larger compared to GPU memory.

4.4.2 Ablation Study

Reducing FLOP for Hash. We note that the projection in the Hash step has a complexity of $O(h\tau d)$ when using a vanilla dense projection and will generate the majority of FLOPs for our LookupFFN. In §4.3, we propose an efficient alternative, BH4 and verify that the block size of B_i has a direct impact on the representation power of BH4. But will this impact the final performance of a model? To evaluate the trade-off between efficiency and performance, we study the effect of block size of B_i on the log perplexity of RoBERTa. The results are summarized in the top table of Table 4.4. When using a vanilla projection, the FLOP for the Hash step accounts for 89% of the total FLOP. It is critical to reduce the FLOP need for this projection, else it becomes the main bottleneck. When using BH4, the performance

decreases and efficiency increases as the block size decreases, which is expected, but it is surprising that while offering a large FLOP reduction, the performance drop compared to the vanilla dense projection is quite small.

Type		Block Size	MFLOP Hash	MFLOP Gather	Log Perplexity
Dense		-	1.05	0.13	1.79
BH4		64	0.56	0.13	1.81
BH4		32	0.30	0.13	1.83
BH4		16	0.17	0.13	1.85
h	τ	$h\tau$	MFLOP		Log Perplexity
32	8	256	0.31		1.94
64	8	512	0.35		1.88
128	8	1024	0.69		1.81
256	8	2048	1.38		1.74
h	τ	$h\tau$	MFLOP Hash	MFLOP Gather	Log Perplexity
64	4	256	0.28	0.07	1.98
32	8	256	0.28	0.03	1.94
20	13	260	0.28	0.02	1.87

Table 4.4: Ablation study evaluating the effects of different hyper-parameters on model performance. The MFLOP columns in the top and button tables are broken down into two column showing the FLOP for Hash and Gather steps separately.

Scaling Effect of LookupFFN. The number of hash tables h and the length of the hash code τ (or log of table size) controls the scaling behavior. As a preliminary step, we first verify that our method can indeed be scaled for better accuracy by increasing the number of hash tables h. Therefore, we compare the model perplexity for different h when τ is fixed, as shown in Table 4.4 (middle). The model performance monotonically increases as h increases. When $h = 256$, our method achieves a lower perplexity compared to the VanillaFFN in Table 4.1. Then, we evaluate the effect of scaling τ. Instead of fixing h while scaling τ, we increase τ but at the same time reduce h such that $h\tau$ is roughly the same. As shown in Table 4.4 (bottom), this comparison reveals a surprising scaling effect of our method: when $h\tau$ is fixed (the FLOP of Hash step remains the same), by increasing τ (increasing the table size), we can reduce the number of hash tables and reduce the FLOP count for the Gather step while achieving better performance.

4.4.3 Throughput and Latency Study

In this subsection, we isolate the FFN from RoBERTa to examine the runtime and throughput of each style of FFN. We compare throughput of LookupFFN, Slide and Mongoose to VanillaFFN (YOSO is excluded since it does not have a CPU implementation at this time), then provide an in-depth analysis of LookupFFN's hardware-level behavior and potential for future throughput scalability. For all empirical results here, we use batch size 64 and sequence length 512, so the effective batch size is 32768 (side note: multi-head attention takes 274ms in this setting).

Latency Comparison. In order to demonstrate the latency improvement afforded by our LookupFFN for CPU inference, we compare runtime to alternatives. Table 4.5 shows the average per-iteration time for vanilla, Slide, and Mongoose-based FFN which is sized to match typical hyperparameters for a standard Transformer model on a modern AMD EPYC-7452 (Zen 2) 32-core Server. Our basic implementation for LookupFFN uses OpenMP without additional software-engineering optimization along with a naive implementation of BH4 and achieves 23% speedup over VanillaFFN. We did further optimization of both the hash function and gather operation (*opt1*), achieving 2.51× speedup over VanillaFFN. Overall we see that both Slide and Mongoose perform worse than vanilla, while our optimized LookupFFN provides good performance improvement.

Technique	Avg. Latency (ms)	Speedup
VanillaFFN	403	1.00×
SLIDE	428	0.94×
Mongoose	878	0.46×
LookupFFN	328	1.23×
LookupFFN (Opt1)	160	2.51×

Table 4.5: Average latency for LookupFFN compared to baselines.

Discrepancy between FLOP and Latency. We note that there is difference in the speed up between FLOP and Latency in Table 4.1 and Table 4.5. The latency not only depends on FLOP, but also the memory access pattern or arithmetic intensity. The GEMM used in VanillaFFN has a very structured memory access pattern and

its arithmetic intensity is high, but the gather operator used in LookupFFN has a more random memory access pattern that depends on the input and its arithmetic intensity is lower. As a result, while the FLOP of LookupFFN is drastically lower than VanillaFFN, the corresponding speed up in latency is not as drastic.

Performance analysis. To further elucidate how we are able to achieve speedup, we detail architectural level performance statistics in Table 4.6 We notice the speedup primarily is afforded by achieving a higher sustained cache bandwidth. The lower LLC miss rate suggests this is at least in part due to extracting more reuse from the LLC, thereby making internal cache management including MSHRs, and on-chip network traffic perform better.

LookupFFN		naive	opt1
Gather	Latency	100 ms	35 ms
	Compute Utilization	1.21%	3.53%
	Sustained L1 BW	79.7 GB/s	231.5 GB/s
	Sustained LLC BW	9.5 GB/s	52.5 GB/s
	L1 Miss %	11.87%	22.68%
	LLC Miss %	69.56%	12.77%
Hash Latency		208 ms	106 ms
Other Latency		20 ms	20 ms
Total Latency		328 ms	160 ms

Table 4.6: Analysis of performance characteristics for LookupFFN. We keep volume of work and working set constant for both so instruction count and FLOPs are constant.

Future Performance Opportunities. Future performance gains for LookupFFN are possible through a combination of software and upcoming hardware optimizations which we summarize in Figure 4.7. *naive* and *opt1* are our two configurations reported previously. Additional tuning of block size for the BH4 projection (64 to 16) could provide a 4× speedup for our Hash step (*hash-opt2*). Future cache technology such as 3D stacking (Wuu et al., 2022) will likely provide a rather generous boost in high-speed LLC cache capacity, enabling larger table size LookupFFN weights to be entirely cache-resident. In combination with this, careful cache optimization through the use of modern prefetching techniques (Georganas et al., 2018), as well

as hardware improvements such as Intel's wide 128 B L1 cache interface (Rotem et al., 2022) can improve hit rates and overall bandwidth. If 90% of bandwidth from a 128 B cache interface could be sustained, the Gather step can be improved by a factor of 35× (*gather-opt2*). Overall, these improvements could yield 8.49× improvement over VanillaFFN. Datatype precision reduction could potentially afford a further multiplicative runtime improvement of 2×, achieved by switching from float32 to float16 – however, VanillaFFN would also gain a 2× execution time reduction from float16.

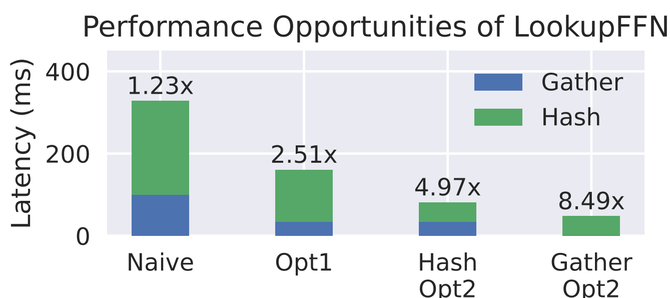

Figure 4.7: Future performance opportunities of LookupFFN. Each bar shows the time breakdown of LookupFFN by GatherAdd and Hash performance. Each version of LookupFFN is annotated with its speedup over Vanilla FFN.

Discussion of Compiler Techniques for FFN. Deep Learning specific compilers, such as TVM and XLA (built into TensorFlow) have been introduced, aiming to optimize operations such as a feed-forward network. We evaluated TVM and XLA on VanillaFFN to compare LookupFFN's latency to these two compiler frameworks. TVM performs 21% worse than PyTorch, with a latency of 488ms. Switching to TensorFlow gives 2.39× improvement over the PyTorch VanillaFFN, with XLA yielding an additional 37% performance boost (absolute latency of TF+XLA: 123ms). Our optimized LookupFFN already provides competitive performance compared to TensorFlow, and our additional hash optimization, when fully implemented/integrated, LookupFFN will provide performance improvement over TF+XLA.

4.5 Summary

In this chapter, we extended the formulation discussed in Chapter 3 and developed a memory lookup based algorithm for efficient FFNs. By viewing memory lookup as a selection problem, which can be approximated using softmax differentiable relaxation, combined with some efficient modifications of the softmax calculation, we developed a LookupFFN that is fully differentiable and can be trained end-to-end. We conclude with a contemplative remark followed by a practical comment. Given the rapid improvements in compute capabilities of GPUs we have seen over the last decade and a mature software support for different kernel shapes, there was little incentive for algorithm designers to use non-GEMM operations. In fact, any modern deep learning algorithm that is not heavily reliant on GEMM may have little chance of broader adoption, assuming it even sees the light of day. The ideas described here (and in Slide, Mongoose, YOSO), in some sense, lie at the other extreme. LookupFFN almost operates as if GEMM is forbidden. While compute capability improvements are slowing down, new memory technologies are already available and others in the development pipeline, we hypothesize that novel yet-undiscovered deep neural network architectures must hit a sweet-spot to delicately balance the trade-off between these two resources. Doing so also offers other benefits including potential energy savings. Now, specific to the model in this chapter, we expect that the LookupFFN benefits can translate to other models. This will complement the server chip developments and enable models to serve a broader cross-section of industries. A preliminary version of this chapter was published as (Zeng et al., 2023a) and the codebase is available at `https://github.com/mlpen/LookupFFN`.

5 MULTI-RESOLUTION BASED APPROXIMATION FOR EFFICIENT SELF-ATTENTION

In Chapter 3, we discussed an efficient construction for approximating self-attention. There are other alternatives being proposed in recent years. For instance, we may ask that self-attention has a pre-specified form of sparsity (for instance, diagonal), see Beltagy et al. (2020); Zaheer et al. (2020). Alternatively, we may model self-attention globally as a low-rank matrix, successfully utilized in Wang et al. (2020); Xiong et al. (2021); Choromanski et al. (2021). Progress has been brisk in improving these approximations: recent proposals have investigated a hybrid global+local strategy based on invoking a robust PCA style model (Chen et al., 2021a) and hierarchical (or H-) matrices (Zhu and Soricut, 2021).

Recall that the hybrid global+local intuition above has a rich classical treatment formalized under Multiresolution analysis (MRA) methods, and Wavelets are a prominent example (Mallat, 1999). While the use of wavelets for signal processing goes back at least three decades (if not longer), their use in machine learning especially for graph based datasets, as a numerical preconditioner, and even for matrix decomposition problems has seen a resurgence (Kondor et al., 2014; Ithapu et al., 2017; Gavish et al., 2010; Lee and Nadler, 2007; Hammond et al., 2011; Coifman and Maggioni, 2006). The extent to which classical MRA ideas (or even their heuristic forms) can guide efficiency in Transformers is largely unknown. Figure 5.1 shows that, given a representative self-attention matrix, only 10% of the coefficients are sufficient for a high fidelity reconstruction. At a minimum, we see that the hypothesis of evaluating a MRA-based self-attention within a Transformer model may have merit.

In this chapter, motivated by the observation in Figure 5.1, we explore classical Multi-Resolution Analysis (MRA) concepts, such as Wavelet, for approximating self-attention matrices. It turns out that modulo some small compromises (on the theoretical side), MRA-based self-attention shows excellent performance across the

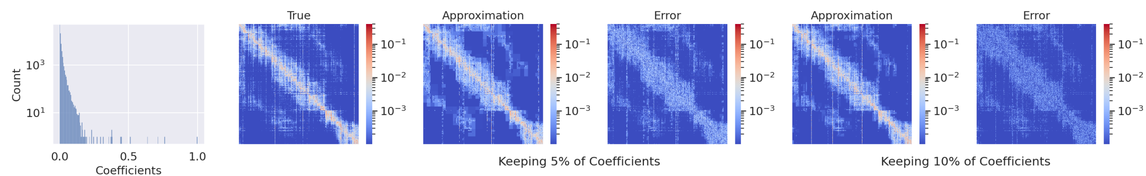

Figure 5.1: The left plot is the histogram of wavelet coefficients in log scale for 2D Haar basis. More than 95% of coefficients have a magnitude less than 0.005. The second plot is the true self-attention matrix $\mathbf{M} = \exp(\mathbf{QK}^\top)$. The third and fourth plots are the approximation and error by keeping top 5% of coefficients. The fifth and last plots are similar for keeping top 10% of coefficients. The errors $\left\|\hat{\mathbf{M}} - \mathbf{M}\right\|_F$ for {MRA, low rank, sparsity} by keeping 10% of {coefficients, ranks, nonzero entries}, are $0.30, 1.24, 0.39$, respectively. This shows performance benefits of MRA compared to low rank and sparsity. We discuss low rank and sparsity in more detail in §5.3.

board. It leads to an accurate approximation for the vanilla self-attention matrices, which is competitive with standard self-attention and outperforms most of baselines while maintaining significantly high time and memory efficiency on both short and long sequences.

5.1 MRA view of Self-attention

As in §2.2, an attention matrix $\mathbf{M}$ is defined as

$$\mathbf{M} = \exp(\mathbf{QK}^\top) = \exp(\mathbf{P})$$

where $\mathbf{Q}$ and $\mathbf{K}$ are queries and keys, and use $\mathbf{P}$ to denote the product of $\mathbf{QK}^\top$. To motivate the use of MRA, we used Figure 5.1 to check how a 2D Haar wavelet basis decomposes the target attention matrix into terms involving different scales and translations, and terms with larger coefficients suffice for a good approximation of $\mathbf{M}$. But the reader will notice that the calculation of the coefficients requires access to the full matrix $\mathbf{M}$. Our discussion below will start from a formulation which will still need the full matrix $\mathbf{M}$. Later, in §5.2, by exploiting the locality of $\mathbf{Q}$ and $\mathbf{K}$, we

will be able to derive an approximation with reduced complexity (without access to $\mathbf{M}$ or $\mathbf{P}$).

For simplicity, we assume that the sequence length N is a power of 2. Inspired by the Haar basis and its ability to adaptively approximate while preserving locality, we apply a pyramidal MRA scheme (Figure 5.2 Left). We consider a decomposition of $\mathbf{M}$ using a set of simpler unnormalized components $\mathbf{B}_{s,x,y} \in \mathbb{R}^{N \times N}$ defined as

$$[\mathbf{B}_{s,x,y}]_{i,j} = \begin{cases} 1 & \text{if } sx - s < i \leqslant sx, sy - s < j \leqslant sy \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

for $s \in \{1, 2, 4, \cdots, N\}$ and $x, y \in \{1, \cdots, N/s\}$. Here, $(sx - s, sx] \times (sy - s, sy]$ is the support of $\mathbf{B}_{s,x,y}$, and s represents the scale of the components, i.e., a smaller s denotes a higher resolution and vice versa. Also, (x, y) denotes the translation of the components.

Why not Haar basis? The main reason for using the form in (5.1) instead of a 2D Haar basis directly is implementation driven, and will be discussed shortly in Remark 5.2. For the moment, we can observe that (5.1) is an overcomplete frame for $\mathbb{R}^{N \times N}$. As shown in Figure 5.2, frame (5.1) has *one extra* scale (with support on a single entry) compared to the Haar basis (4 rows versus 3 rows). Except for this extra scale, (5.1) has the same support as the Haar basis at different scales. In addition, (5.1) provides scaled and translated copies of the "mother" component, similar to Haar.

Let $\mathbb{S} = \{\mathbf{B}_{s,x,y}\}$ be a set of components for the possible scales and translations, then we decompose

$$\mathbf{M} = \sum_{\mathbf{B}_{s,x,y} \in \mathbb{S}} \alpha_{s,x,y} \mathbf{B}_{s,x,y}$$

for some set of coefficients $\alpha_{s,x,y}$. Since (5.1) is overcomplete, the coefficients $\alpha_{s,x,y}$ are not unique. We specifically compute the coefficients $\alpha_{s,x,y}$ as follows. Let

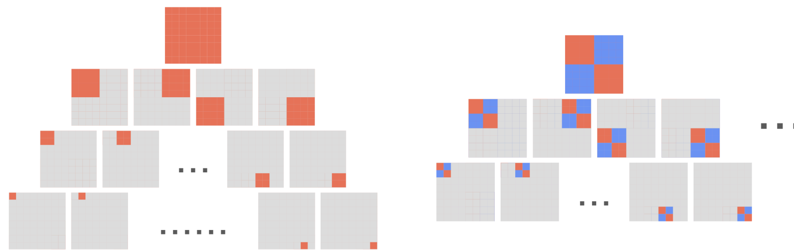

Figure 5.2: The left plot is the overcomplete frame defined in (5.1), which consists of 85 matrices for $N = 8$. The right plot is a 2D generalization of Haar basis, which consists of three groups of 21 self-similar matrices plus a constant matrix (all entries equal to $\frac{1}{N}$). The color red, blue, and gray means positive, negative, and 0, respectively. Notice that (5.1) does not include negative entries: it involves more components but makes the formulation simpler.

$\mathbf{E}_N = \mathbf{M}$ and

$$\begin{aligned} \alpha_{s,x,y} &= \frac{1}{s^2}\langle \mathbf{B}_{s,x,y}, \mathbf{E}_s \rangle \\ \mathbf{E}_{s/2} &= \mathbf{E}_s - \sum_{\mathbf{B}_{s,x,y} \in \mathbb{S}} \alpha_{s,x,y} \mathbf{B}_{s,x,y} \end{aligned} \tag{5.2}$$

Here, the $\mathbf{E}_{s/2}$ denotes the residuals of the higher frequencies. At each scale s, $\alpha_{s,x,y}$ is the optimal solution of the least squares problem minimizing $\left\|\mathbf{E}_{s/2}\right\|_F^2$. Intuitively, the approximation procedure starts from the coarsest approximation of $\mathbf{M}$ which only consists of the lowest frequency, then the procedure refines the approximation by adding residuals of higher frequencies.

Parsimony. We empirically observe that the coefficients of most components are near zero, so we can represent $\mathbf{M}$ with only a few components while maintaining the accuracy of approximation. Specifically, we can find a small subset $\mathbb{J} \subset \mathbb{S}$, the corresponding coefficients $\alpha'_{s,x,y}$ computed following (5.2), and the resulting approximation

$$\hat{\mathbf{M}}^* = \sum_{\mathbf{B}_{s,x,y} \in \mathbb{J}} \alpha'_{s,x,y} \mathbf{B}_{s,x,y} \tag{5.3}$$

with a negligible approximation error $\left\|\hat{\mathbf{M}}^* - \mathbf{M}\right\|_F$.

But $\hat{\mathbf{M}}^*$ in (5.3) does not suggest any interesting property of the approximation. But we can check an equivalent form of $\hat{\mathbf{M}}^*$. Denote the average of $\mathbf{M}$ over the support of $\mathbf{B}_{s,x,y}$ to be

$$\mu^*_{s,x,y} = \frac{1}{s^2}\langle \mathbf{B}_{s,x,y}, \exp(\mathbf{P})\rangle \tag{5.4}$$

It turns out that the entries of $\hat{\mathbf{M}}^*$ in (5.3) can be rewritten (Observation 5.1) as

$$\left[\hat{\mathbf{M}}^*\right]_{i,j} = \mu^*_{s,x,y} \tag{5.5}$$

where (s, x, y) is the index of $\mathbf{B}_{s,x,y} \in \mathbb{J}$ that has the smallest support region and is supported on (i, j). But if $(i, j) \notin \text{supp}(\mathbf{B}_{s,x,y})$ for all $\mathbf{B}_{s,x,y} \in \mathbb{J}$, then $\left[\hat{\mathbf{M}}^*\right]_{i,j} = 0$. Here, $\text{supp}(\cdot)$ denotes the support region of the input, or the set of indices of non-zero entries of the input. The (i, j) entry of $\hat{\mathbf{M}}^*$ is precisely approximated by the average of $\mathbf{M}$ over the smallest support region for $\mathbf{B}_{s,x,y} \in \mathbb{J}$ containing (i, j). In other words, the procedure uses the highest resolution possible for a given $\mathbb{J}$ as an approximation.

Observation 5.1. *We can re-write* $\hat{\mathbf{M}}^* = \sum_{\mathbf{B}_{s,x,y}\in\mathbb{J}} \alpha_{s,x,y}\mathbf{B}_{s,x,y}$ *as* $\left[\hat{\mathbf{M}}^*\right]_{i,j} = \mu^*_{s,x,y}$ *where* (s, x, y) *is the index of* $\mathbf{B}_{s,x,y} \in \mathbb{J}$ *that has the smallest support region and is supported on* (i, j).

Proof. We describe the details next. First, notice that at each scale s,

$$\mathbf{E}_{s/2} = \mathbf{E}_s - \sum_{\mathbf{B}_{s,x,y}\in\mathbb{J}} \alpha_{s,x,y}\mathbf{B}_{s,x,y}$$

If $(i, j) \notin \text{supp}(\mathbf{B}_{s,x,y})$ for all $\mathbf{B}_{s,x,y} \in \mathbb{J}$, then $\left[\sum_{\mathbf{B}_{s,x,y}\in\mathbb{J}} \alpha_{s,x,y}\mathbf{B}_{s,x,y}\right]_{i,j} = 0$ and thus $\left[\mathbf{E}_{s/2}\right]_{i,j} = [\mathbf{E}_s]_{i,j}$. Further, at each scale s, the supports of $B_{s,x,y}$ are disjoint, and there is exactly one $B_{s,x,y}$ whose support includes (i, j). Thus, if $\left[\mathbf{E}_{s/2}\right]_{i,j} = [\mathbf{E}_s]_{i,j}$, then $\left[\mathbf{E}_{s/2}\right]_{i',j'} = [\mathbf{E}_s]_{i',j'}$ for all $(i', j') \in \text{supp}(\mathbf{B}_{s,x,y})$. Also, for scale $s' < s$, if $\mathbf{B}_{s',x',y'}$ is supported on (i, j), then $\text{supp}(\mathbf{B}_{s',x',y'}) \subseteq \text{supp}(\mathbf{B}_{s,x,y})$. These observations are

useful for the derivation below.

Consider approximation on the (i, j) entry of $\hat{\mathbf{M}}^*$, and let $\mathbf{B}_{s_1,x_1,y_1}, \cdots, \mathbf{B}_{s_k,x_k,y_k} \in \mathbb{J}$ be all components that are supported on (i, j) and $s_1 < \cdots < s_k$. Then, $\mathbf{E}_n = \mathbf{M}$. Since $\mathbf{B}_{s_k,x_k,y_k}$ has the largest s_k and is supported on (i, j), by the above observations, for all $(i', j') \in \text{supp}(\mathbf{B}_{s_k,x_k,y_k})$,

$$[\mathbf{E}_{s_k}]_{i',j'} = [\mathbf{E}_{2s_k}]_{i',j'} = \cdots = [\mathbf{E}_n]_{i',j'} = [\mathbf{M}]_{i',j'}$$

Therefore,

$$\alpha_{s_k,x_k,y_k} = \frac{1}{s_k^2}\langle \mathbf{B}_{s_k,x_k,y_k}, \mathbf{E}_{s_k}\rangle = \frac{1}{s_k^2}\langle \mathbf{B}_{s_k,x_k,y_k}, \mathbf{M}\rangle = \mu^*_{s_k,x_k,y_k}$$

Then,

$$\left[\mathbf{E}_{s_k/2}\right]_{i',j'} = [\mathbf{E}_{s_k}]_{i',j'} - \left[\sum_{\mathbf{B}_{s_k,x_k,y_k} \in \mathbb{J}} (\alpha_{s_k,x_k,y_k})' B_{s_k,x_k,y_k}\right]_{i',j'} = [\mathbf{M}]_{i',j'} - \mu^*_{s_k,x_k,y_k}$$

Assume for all $(i', j') \in \text{supp}(\mathbf{B}_{s_{m+1},x_{m+1},y_{m+1}})$,

$$\left[\mathbf{E}_{s_{m+1}/2}\right]_{i',j'} = [\mathbf{M}]_{i',j'} - \mu^*_{s_{m+1},x_{m+1},y_{m+1}}$$

Then, again by the above observations,

$$[\mathbf{E}_{s_m}]_{i',j'} = \cdots = \left[\mathbf{E}_{s_{m+1}/2}\right]_{i',j'} = [\mathbf{M}]_{i',j'} - \mu^*_{s_{m+1},x_{m+1},y_{m+1}}$$

Therefore,

$$\begin{aligned}
\alpha_{s_m,x_m,y_m} &= \frac{1}{s_m^2}\langle \mathbf{B}_{s_m,x_m,y_m}, \mathbf{E}_{s_m}\rangle \\
&= \frac{1}{s_m^2}\sum_{(i'',j'')\in \text{supp}(\mathbf{B}_{s_m,x_m,y_m})} (\mathbf{M}_{i'',j''} - \mu^*_{s_{m+1},x_{m+1},y_{m+1}}) \\
&= \frac{1}{s_m^2}\sum_{(i'',j'')\in \text{supp}(\mathbf{B}_{s_m,x_m,y_m})} \mathbf{M}_{i'',j''} - \mu^*_{s_{m+1},x_{m+1},y_{m+1}} \\
&= \mu^*_{s_m,x_m,y_m} - \mu^*_{s_{m+1},x_{m+1},y_{m+1}}
\end{aligned}$$

Then,

$$\begin{aligned}
\left[\mathbf{E}_{s_m/2}\right]_{i',j'} &= [\mathbf{E}_{s_m}]_{i',j'} - \left[\sum_{\mathbf{B}_{s_m,x_m,y_m}\in \mathbb{J}} (\alpha_{s_m,x_m,y_m})'\mathbf{B}_{s_m,x_m,y_m}\right]_{i',j'} \\
&= \mathbf{M}_{i,j} - \mu^*_{s_{m+1},x_{m+1},y_{m+1}} - (\mu^*_{s_m,x_m,y_m} - \mu^*_{s_{m+1},x_{m+1},y_{m+1}}) \\
&= \mathbf{M}_{i,j} - \mu^*_{s_m,x_m,y_m}
\end{aligned}$$

By induction, for all $(i', j') \in \text{supp}(\mathbf{B}_{s_m,x_m,y_m})$,

$$\left[\mathbf{E}_{s_m/2}\right]_{i',j'} = \mathbf{M}_{i',j'} - \mu^*_{s_m,x_m,y_m}$$

holds for all s_m. Further, we know for $m < k$,

$$\begin{aligned}
\alpha_{s_k,x_k,y_k} &= \mu^*_{s_k,x_k,y_k} \\
\alpha_{s_m,x_m,y_m} &= \mu^*_{s_m,x_m,y_m} - \mu^*_{s_{m+1},x_{m+1},y_{m+1}}
\end{aligned}$$

Finally,

$$\begin{aligned}
(\hat{\mathbf{M}}^*)_{i,j} &= (\sum_{\mathbf{B}_{s,x,y}\in \mathbb{J}} \alpha_{s,x,y}\mathbf{B}_{s,x,y})_{i,j} \\
&= \mu^*_{s_k,x_k,y_k} + \sum_{m=1}^{k-1}(\mu^*_{s_m,x_m,y_m} - \mu^*_{s_{m+1},x_{m+1},y_{m+1}}) = \mu^*_{s_1,x_1,y_1}
\end{aligned}$$

And, (s_1, x_1, y_1) is the index of $\mathbf{B}_{s_1,x_1,y_1} \in \mathbb{J}$ that has the smallest support region and is supported on (i, j). This completes the proof.

□

The reader will notice that rewriting $\hat{\mathbf{M}}^*$ as (5.5) is possible due to our modifications to the Haar basis in (5.1).

Remark 5.2. *Consider using a Haar decomposition and let $\mathbb{L}$ be the subset of basis with nonzero coefficients. The approximation $\left[\hat{\mathbf{M}}_{Haar}\right]_{i,j}$ depends on* all *$\phi \in \mathbb{L}$ which are supported on (i, j). For example, in the worst case, the coefficients of all ϕ which are supported on (i, j) need to be nonzero to have $\left[\hat{\mathbf{M}}_{Haar}\right]_{i,j} = [\mathbf{M}]_{i,j}$. We find that a hardware friendly and efficient approximation scheme in this case is challenging. On the other hand, when using the decomposition (5.2) over the overcomplete frame (5.1), $\left[\hat{\mathbf{M}}^*\right]_{i,j}$ depends on only* one *$\mathbf{B}_{s,x,y} \in \mathbb{J}$ that has the smallest support region and is supported on (i, j). This makes constructing the set $\mathbb{J}$ easier and more flexible.*

5.2 A Practical Approximation scheme

Given that we now understand all relevant modules, we can focus on practical considerations. Notice that each $\mu^*_{s,x,y}$ requires averaging over $\mathcal{O}(s^2)$ entries of the matrix $\mathbf{M}$, so in the worst case, we would need access to the entire matrix $\mathbf{M}$ to compute all $\mu^*_{s,x,y}$ for $\mathbf{B}_{s,x,y} \in \mathbb{J}$. Nonetheless, suppose that we still compute all the coefficients $\alpha_{s,x,y}$ and then post-hoc truncate the small coefficients to construct the set $\mathbb{J}$. This approach will clearly be inefficient. In this section, we discuss two strategies where the main goal is efficiency.

5.2.1 Can we approximate $\mu^*_{s,x,y}$ quickly?

We first discuss calculating $\mu^*_{s,x,y}$. To avoid accessing the full matrix $\mathbf{M}$, instead of computing the average of exponential (5.4), we compute a lower bound (due to

convexity of exponential), i.e., exponential of average (5.6), as an approximation.

$$\mu_{s,x,y} = \exp\left(\frac{1}{s^2}\langle \mathbf{B}_{s,x,y}, \mathbf{P}\rangle\right) \tag{5.6}$$

We can verify that the expression in (5.6) can be computed efficiently as follows. Define $\mathbf{q}_{s,i}, \mathbf{k}_{s,j} \in \mathbb{R}^D$ where $\mathbf{q}_{1,i} = [\mathbf{Q}]_i$, $\mathbf{k}_{1,j} = [\mathbf{K}]_j$, and

$$\begin{aligned} \mathbf{q}_{s,i} &= \frac{1}{2}\mathbf{q}_{s/2,2i-1} + \frac{1}{2}\mathbf{q}_{s/2,2i} \\ \mathbf{k}_{s,j} &= \frac{1}{2}\mathbf{k}_{s/2,2j-1} + \frac{1}{2}\mathbf{k}_{s/2,2j} \end{aligned} \tag{5.7}$$

Interestingly, (5.6) is simply,

$$\mu_{s,x,y} = \exp(\langle \mathbf{q}_{s,x} \mathbf{k}_{s,y}\rangle)$$

Then, the approximation using (5.6) is,

$$\hat{\mathbf{M}}_{i,j} = \mu_{s,x,y}$$

where (s, x, y) is the same as (5.5) and $\hat{\mathbf{M}}_{i,j} = 0$ otherwise. Each $\mu_{s,x,y}$ only requires an inner product between one row of $\tilde{Q}_s$ and one row of $\tilde{K}_s$ and applying an exponential, so the cost of a single $\mu_{s,x,y}$ is $\mathcal{O}(1)$ when $\tilde{Q}_s$ and $\tilde{K}_s$ are provided. We will discuss the overall complexity in §5.2.4.

While efficient, this modification will incur a small amount of error. However, by using the property of **P** inherited from **Q** and **K**, we can quantify the error.

Lemma 5.3. *Assume for all* $(i_1, j_1), (i_2, j_2) \in$ *supp*$(\mathbf{B}_{s,x,y})$, $\left\|[\mathbf{Q}]_{i_1} - [\mathbf{Q}]_{i_2}\right\|_q \leqslant \beta_2$ *and* $\left\|[\mathbf{K}]_{j_1} - [\mathbf{K}]_{j_2}\right\|_q \leqslant \beta_2$ *and* $\left\|[\mathbf{Q}]_{i_1}\right\|_p, \left\|[\mathbf{Q}]_{i_2}\right\|_p, \left\|[\mathbf{K}]_{j_1}\right\|_p, \left\|[\mathbf{K}]_{j_2}\right\|_p \leqslant \beta_1$ *where* $\frac{1}{p} + \frac{1}{q} = 1$, *then* $[\mathbf{P}]_{i,j} \in [a, a + r]$ *where* $r = 2\beta_1\beta_2$ *for all* $(i, j) \in$ *supp*$(\mathbf{B}_{s,x,y})$ *and some* a, *and*

$$0 \leqslant \mu^*_{s,x,y} - \mu_{s,x,y} \leqslant c_r \mu_{s,x,y}$$

where $c_r = 1 + \exp(r) - 2\exp(r/2)$.

Proof.

$$\begin{aligned}[\mathbf{P}]_{i_1,j_1} - [\mathbf{P}]_{i_2,j_2} &= [\mathbf{Q}]_{i_1} [\mathbf{K}]_{j_1}^\top - [\mathbf{Q}]_{i_2} [\mathbf{K}]_{j_2}^\top \\ &= ([\mathbf{Q}]_{i_1} - [\mathbf{Q}]_{i_2} + [\mathbf{Q}]_{i_2}) [\mathbf{K}]_{j_1}^\top - [\mathbf{Q}]_{i_2} [\mathbf{K}]_{j_2}^\top \\ &= ([\mathbf{Q}]_{i_1} - [\mathbf{Q}]_{i_2}) [\mathbf{K}]_{j_1}^\top + [\mathbf{Q}]_{i_2} ([\mathbf{K}]_{j_1} - [\mathbf{K}]_{j_2})^\top\end{aligned}$$

Then by Hölder's inequality,

$$\begin{aligned}[\mathbf{P}]_{i_1,j_1} - [\mathbf{P}]_{i_2,j_2} &\leqslant \| [\mathbf{Q}]_{i_1} - [\mathbf{Q}]_{i_2} \|_q \| [\mathbf{K}]_{j_1} \|_p + \| [\mathbf{Q}]_{i_2} \|_p \| [\mathbf{K}]_{j_1} - [\mathbf{K}]_{j_2} \|_q \\ &\leqslant 2\beta_1\beta_2\end{aligned}$$

Therefore, $[\mathbf{P}]_{i,j} \in [a, a+r]$ where $r = 2\beta_1\beta_2$ and $a = \min_{(i',j') \in \mathrm{supp}(\mathbf{B}_{s,x,y})} [\mathbf{P}]_{i',j'}$. Then, by Jensen's inequality,

$$\mu_{s,x,y} = \exp\Big(\frac{1}{s^2} \sum_{(i,j)\in \mathrm{supp}(\mathbf{B}_{s,x,y})} [\mathbf{P}]_{i,j}\Big) \leqslant \frac{1}{s^2} \sum_{(i,j)\in \mathrm{supp}(\mathbf{B}_{s,x,y})} \exp([\mathbf{P}]_{i,j}) = \mu^*_{s,x,y}$$

Also, by Simic (2009), for a convex function f and $x_i \in [a, b]$ for all x_i,

$$\frac{1}{k}\sum_{i=1}^{k} f(x_i) - f\Big(\frac{1}{k}\sum_{i=1}^{k} x_i\Big) \leqslant f(a) + f(b) - 2f\Big(\frac{a+b}{2}\Big)$$

Therefore,

$$\begin{aligned}\mu^*_{s,x,y} - \mu_{s,x,y} &\leqslant \exp(a) + \exp(a+r) - 2\exp\Big(a + \frac{r}{2}\Big) \\ &\leqslant \Big(1 + \exp(r) - 2\exp\Big(\frac{r}{2}\Big)\Big)\mu_{s,x,y}\end{aligned} \tag{5.8}$$

Denote $C_r = 1 + \exp(r) - 2\exp(\frac{r}{2})$.

□

Lemma 5.3 suggests that the approximation error depends on the "spread" or numerical range r of values (range, for short) in the $[\mathbf{P}]_{i,j}$ entries within a region

$\text{supp}(\mathbf{B}_{s,x,y})$ and $\mu_{s,x,y}$. If r is small or $\mu_{s,x,y}$ is small, then the approximation error is small. The range of a region is influenced by properties of $\mathbf{Q}$ and $\mathbf{K}$. The range r is bounded by the norm and spread of $[\mathbf{Q}]_i$ and $[\mathbf{K}]_j$ for $(i,j) \in \text{supp}(\mathbf{B}_{s,x,y})$. This relies on the locality assumption that spatially nearby tokens should also be semantically similar which commonly holds in many applications. Of course, this can be avoided if needed – it is easy to reduce the spread of $[\mathbf{Q}]_i$ and $[\mathbf{K}]_j$ in local regions simply by permuting the order of $\mathbf{Q}$ and $\mathbf{K}$. For example, we can use Locality Sensitive Hashing (LSH) to reorder $[\mathbf{Q}]_i$ and $[\mathbf{K}]_j$ such that similar vectors are in nearby positions, e.g., see Kitaev et al. (2020). While the range r is data/problem dependent, we can control the range by using a smaller s since the range of a smaller region will be smaller. In the extreme case, when $s = 1$, the range is 0. So, this offers guidance that when $\mu_{s,x,y}$ is large, we should approximate the region at a higher resolution such that the range is smaller.

Remark 5.4. *The underlying assumption of diagonal attention structure from Longformer, Big Bird, and H-Transformer-1D is that tokens are highly dependent on the nearby tokens and* only *the nearby tokens, which is more important than attention w.r.t. distant tokens. This might appear similar to the locality assumption discussed earlier, but this is incorrect. Our locality does* not *assume that semantically similar tokens must be spatially close, i.e., we allow high and precise dependence on distant tokens.*

5.2.2 Can we construct $\mathbb{J}$ quickly?

So far, we have assumed that the set $\mathbb{J}$ is given which is not true in practice. We now place a mild restriction on the set $\mathbb{J}$ as a simplification heuristic. We allow each (i,j) entry of $\hat{\mathbf{M}}$ to be included in the support of exactly one $\mathbf{B}_{s,x,y} \in \mathbb{J}$. Consider a $\mathbf{B}_{s,x,y} \in \mathbb{J}$, if each entry of the support region of $\mathbf{B}_{s,x,y}$ is included in the support of some $\mathbf{B}_{s',x',y'} \in \mathbb{J}$ with a smaller s', then $\mathbf{B}_{s,x,y}$ can be safely removed from $\mathbb{J}$ without affecting the approximation, by construction. This restriction allows us to avoid searching for the $\mathbf{B}_{s,x,y}$ with the smallest s among multiple candidates. Then,

the overall approximation can be written as

$$\hat{\mathbf{M}} = \sum_{\mathbf{B}_{s,x,y} \in \mathbb{J}} \mu_{s,x,y} \mathbf{B}_{s,x,y}$$

Remark 5.5. *Under this restriction, $\mathbb{J}$ is a subset of an orthogonal basis of $\mathbb{R}^{N \times N}$.*

Mechanics of constructing $\mathbb{J}$. Now, we can discuss how the set $\mathbb{J}$ is constructed. Let us first consider the approximation error $\mathcal{E} = ||\hat{\mathbf{M}} - \mathbf{M}||_F^2$, by factoring out $\mu_{s,x,y}^2$,

$$\mathcal{E} = \sum_{\mathbf{B}_{s,x,y} \in \mathbb{J}} \mu_{s,x,y}^2 \sum_{(i,j) \in \mathrm{supp}(\mathbf{B}_{s,x,y})} \left(\frac{[\mathbf{M}]_{i,j}}{\mu_{s,x,y}} - 1 \right)^2 \tag{5.9}$$

Since the goal is to minimize the error $\mathcal{E}$, the optimal solution is to fix the computation budget $|\mathbb{J}|$ and solve the optimization problem which minimizes $\mathcal{E}$ over all possible $\mathbb{J}$. However, this might not be efficiently solvable.

Instead, we consider finding a good solution greedily. Consider the error $\mathcal{E}$. We can analyze the specific term to get an insight into how approximation error can be reduced. Note that

$$\log \frac{[\mathbf{M}]_{i,j}}{\mu_{s,x,y}} = [\mathbf{P}]_{i,j} - \frac{1}{s^2} \langle \mathbf{B}_{s,x,y}, \mathbf{P} \rangle$$

is the deviation of $[\mathbf{P}]_{i,j}$ from the mean of the support region, so the approximation error (5.9) is determined by $\mu_{s,x,y}$ and the deviation of $[\mathbf{P}]_{i,j}$ within the region, which coincides with the conclusion of Lemma 5.3. Computing this deviation would incur a $\mathcal{O}(s^2)$ cost, so we avoid using it as a criteria for constructing $\mathbb{J}$. We found that we can make a reasonable assumption that the deviation of $[\mathbf{P}]_{i,j}$ in a support of $\mathbf{B}_{s,x,y}$ for the *same* s are similar, and the deviation of a region for a smaller s is smaller. Then, a sensible heuristic is to use $\mu_{s,x,y}$ as a criteria such that if $\mu_{s,x,y}$ is large, then we must approximate the region using a higher resolution. The approximation procedure is described in Algorithm 1, and the approximation result is shown in Figure 5.3. Figure 5.4 shows a visualization of our approximation procedure using a linear scale and gives a better illustration of how approximation quality increases

as approximation procedure proceeds. Broadly, this approximation starts with a coarse approximation of a self-attention matrix, then for regions with a large $\mu_{s,x,y}$, we successively refines the regions to a higher resolution.

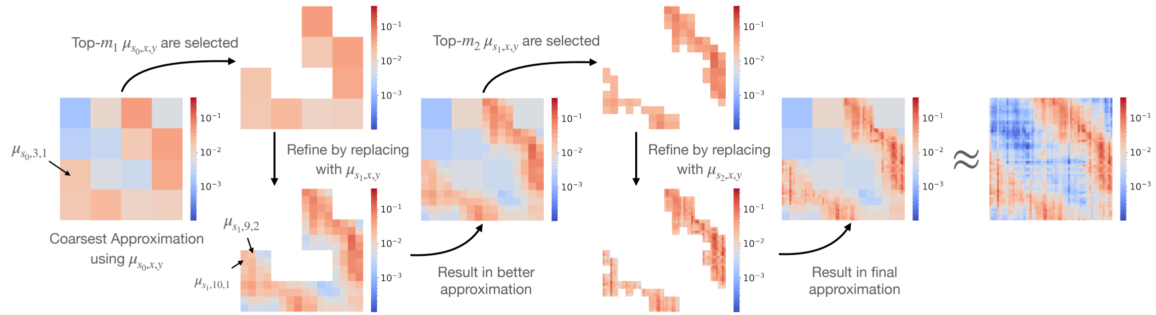

Figure 5.3: Illustration of our approximation scheme for $R = \{16, 4, 1\}$. A log scale is used for a better visualization.

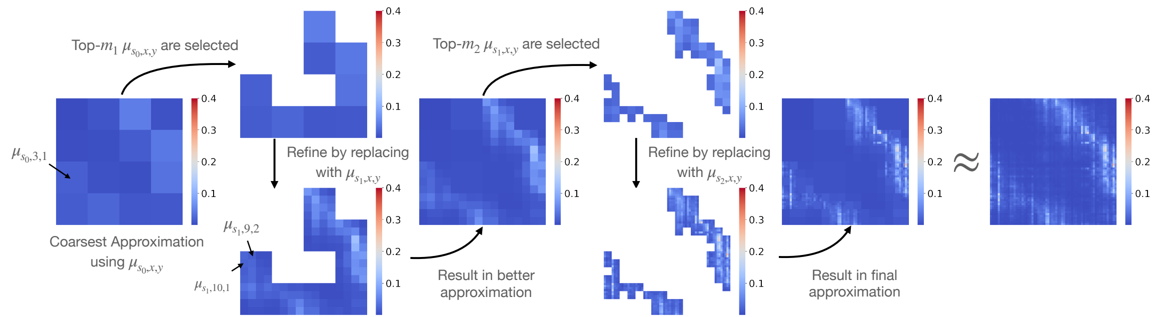

Figure 5.4: Illustration of approximation scheme for $R = \{16, 4, 1\}$ in linear scale.

With the approximation procedure in place, we can quantify the error of this multiresolution approximation. We only show the approximation error for $\mathcal{R} = \{b, 1\}$ for some b, but the analysis easily extends beyond $\mathcal{R} = \{b, 1\}$.

Proposition 5.6. *Let $\mathcal{R} = \{b, 1\}$ for some b and δ be the m_1-th largest $\mu_{b,x,y}$, assume for all $(i, j) \in supp(B^b_{x,y})$, $[\mathbf{P}]_{i,j} \in [a, a + r]$ for some a and $r > 0$, then*

$$\frac{\left\|\hat{\mathbf{M}} - \mathbf{M}\right\|_F}{\|\mathbf{M}\|_F} \leqslant \sqrt{\frac{(n^2 - m_1 b^2) c_{2r} \delta^2}{\sum_{i,j=1}^{n} \exp(2\,[\mathbf{P}]_{i,j})}}$$

where $c_{2r} = 1 + \exp(2r) - 2\exp(r)$

Algorithm 1 Constructing the set $\mathbb{J}$

Input: $\mathcal{R} = \{s_0, s_1, ..., s_k\}$ in descending order
Input: Budget m_i for each s_i for $i > 0$
Input: Initial $\mathbb{J}$ (empty or prespecified via priors)
Compute $\mu_{s_0,x,y}$ for all possible x, y and add $\mathbf{B}_{s_0,x,y}$ to $\mathbb{J}$
for $i = 1$ **to** $i = k$ **do**
  Pop m_i elements $\mathbf{B}_{s_{i-1},x,y}$ with the largest $\mu_{s_{i-1},x,y}$
  for each $\mathbf{B}_{s_{i-1},x,y}$ **do**
    Compute $\mu_{s_i,x',y'}$ for all $\text{supp}(\mathbf{B}_{s_i,x',y'}) \subseteq \text{supp}(\mathbf{B}_{s_{i-1},x,y})$
    Add $\mathbf{B}_{s_i,x',y'}$ to $\mathbb{J}$
  end for
end for
Output: $\mathbb{J}$

Proof. Similar to (5.8), for $\mu'_{s,x,y}$ defined in (5.11)

$$\begin{aligned}\mu'_{s,x,y} - \mu^2_{s,x,y} &\leqslant \exp(2a) + \exp(2a + 2r) - 2\exp(2a + r) \\ &\leqslant (1 + \exp(2r) - 2\exp(r))\mu^2_{s,x,y}\end{aligned}$$

Denote $c_{2r} = 1 + \exp(2r) - 2\exp(r)$, Then, the error by approximating a region $\text{supp}(\mathbf{B}_{s,x,y})$ with lower resolution $\mu_{s,x,y}$ is

$$\begin{aligned}&\sum_{(i,j)\in\text{supp}(\mathbf{B}_{s,x,y})} (\mu_{s,x,y} - \exp([\mathbf{P}]_{i,j}))^2 \\ =&\sum_{(i,j)\in\text{supp}(\mathbf{B}_{s,x,y})} (\mu^2_{s,x,y} + \exp(2\,[\mathbf{P}]_{i,j}) - 2\mu_{s,x,y}\exp([\mathbf{P}]_{i,j})) \\ =&s^2\mu^2_{s,x,y} + s^2\mu'_{s,x,y} - 2s^2\mu_{s,x,y}\mu^*_{s,x,y} \\ \leqslant&s^2\mu^2_{s,x,y} + s^2\mu'_{s,x,y} - 2s^2\mu^2_{s,x,y} \\ \leqslant&c_{2r}s^2\mu^2_{s,x,y}\end{aligned}$$

Since $\mu_{1,x,y} = \exp([\mathbf{P}]_{x,y})$, the approximation error only comes from terms $\mathbf{B}_{s,x,y} \in \mathbb{J}$

for $s > 1$. Therefore,

$$\begin{aligned}
\|\hat{\mathbf{M}} - \mathbf{M}\|_F^2 &= \sum_{\mathbf{B}_{b,x,y} \in \mathbb{J}} \sum_{(i,j) \in \text{supp}(B^b_{x,y})} (\mu_{b,x,y} - \exp([\mathbf{P}]_{i,j}))^2 \\
&= \sum_{B^b_{x,y} \in \mathbb{J}} C_{2r} b^2 \mu^2_{b,x,y} \\
&\leqslant \sum_{B^b_{x,y} \in \mathbb{J}} C_{2r} b^2 \delta^2 \\
&= (N^2 - m_1 b^2) c_{2r} \delta^2
\end{aligned}$$

Then, the relative error is

$$\frac{\|\hat{\mathbf{M}} - \mathbf{M}\|_F}{\|\mathbf{M}\|_F} \leqslant \sqrt{\frac{(N^2 - m_1 b^2) C_{2r} \delta^2}{\sum_{i,j=1}^{n} \exp(2\,[\mathbf{P}]_{i,j})}}$$

□

Proposition 5.6 again emphasizes the relation between the numerical range r of $\mathbf{P}$ and the quality of an approximation. With some knowledge of the range r and $\mu_{b,x,y}$, we can control the error using an appropriate budget m_1.

Remark 5.7. *The procedure shares some commonalities with the correction component of Geometric Multigrid methods (Saad, 2003; Hackbusch, 1985). Coarsening is similar to our low resolution approximation, but the prolongation step is different. Rather than interpolate the entire coarse grid to finer grids, our method replaces some regions of the coarse grid with its higher resolution approximation.*

5.2.3 How do we compute $\hat{\mathbf{M}}\mathbf{V}$?

We obtained an approximation $\hat{\mathbf{M}}$, but we should not instantiate this matrix (to avoid the $\mathcal{O}(n^2)$ cost). So, we discuss a simple procedure for computing $\hat{\mathbf{M}}\mathbf{V}$ without

constructing the $n \times n$ matrix. Define $\mathbf{v}_{s,j} \in \mathbb{R}^D$ where $\mathbf{v}_{1,j} = [\mathbf{V}]_j$ and

$$\mathbf{v}_{s,j} = \frac{1}{2}\mathbf{v}_{s/2,2j-1} + \frac{1}{2}\mathbf{v}_{s/2,2j}$$

similar to (5.7). Then, the steps follow Algorithm 2. We again start with multiplying coarse components of $\hat{\mathbf{M}}$ with $\mathbf{V}$, then successively add the multiplication of higher resolution components of $\hat{\mathbf{M}}$ and $\mathbf{V}$, and finally compute $\hat{\mathbf{M}}\mathbf{V}$.

Algorithm 2 Computing $\hat{\mathbf{M}}\mathbf{V}$

Input: $\mathcal{R} = \{s_0, s_1, ..., s_k\}$ in descending order
Input: $\mathbb{J}$ and all $\mathbf{v}_{s,j}$
Initialize $\mathbf{Y}_{s_{-1}} \in \mathbb{R}^{1 \times d}$ to be zero matrix
for $i = 0$ **to** $i = k$ **do**
  Duplicate rows of $\mathbf{Y}_{s_{i-1}}$ to create $\mathbf{Y}_{s_i} \in \mathbb{R}^{N/s_i \times D}$
  for each $\mathbf{B}_{s_i,x,y} \in \mathbb{J}$ **do**
    $[\mathbf{Y}_{s_i}]_x \leftarrow [\mathbf{Y}_{s_i}]_x + \mu_{s_i,x,y}\mathbf{v}_{s_i,y}$
  end for
end for
Output: $Y_{s_k} = \hat{\mathbf{M}}\mathbf{V}$

5.2.4 What is the overall complexity?

We have now described the overall procedure of our approximation approach. In this section, we analyze the complexity of our procedure.

We first need to compute $\mathbf{q}_{s,i}, \mathbf{k}_{s,j}, \mathbf{v}_{s,j}$ for $s \in \{2, 4, \cdots, N\}$ and $i, j \in \{1, \cdots, N/s\}$. Since each of $\mathbf{q}_{s,i}, \mathbf{k}_{s,j}, \mathbf{v}_{s,j}$ takes $\mathcal{O}(D)$, so the total cost of computing all possible $\mathbf{q}_{s,i}, \mathbf{k}_{s,j}, \mathbf{v}_{s,j}$ is simply

$$\mathcal{O}(\frac{N}{2}D + \frac{N}{4}D + \cdots + \frac{N}{N}D) = \mathcal{O}(ND)$$

Given all $\mathbf{q}_{s,i}, \mathbf{k}_{s,j}$, in Algorithm 1, there are $\mathcal{O}((N/s_0)^2)$ possible entries of $\mu_{s_0,x,y}$. And at scale s_i for $i > 0$, there are $\mathcal{O}(m_i(s_{i-1}/s_i)^2)$ entries of $\mu_{s_i,x,y}$ since there are $\mathcal{O}((s_{i-1}/s_i)^2)$ number of of $\mathbf{B}_{s_i,x',y'}$ satisfying $\text{supp}(\mathbf{B}_{s_i,x',y'}) \subseteq \text{supp}(\mathbf{B}_{s_{i-1},x,y})$

and there are m_i regions at scale s_{i-1} to be refined. Note that computing each $\mu_{s,x,y}$ takes $\mathcal{O}(1)$, and selecting top-k elements is linear in the input size. Therefore, combined with the fact that the cost of computing individual $\mu_{s,x,y}$ is $\mathcal{O}(D)$, the cost of constructing $\mathbb{J}$ is

$$\mathcal{O}((N/s_0)^2 D + \sum_{i=1}^{k} m_i (s_{i-1}/s_i)^2 D)$$

Once $\mathbb{J}$ is constructed, $\hat{\mathbf{M}}$ is simple since $\hat{\mathbf{M}}_{i,j} = \mu_{s,x,y}$ for a $\mathbf{B}_{s,x,y} \in \mathbb{J}$.

Finally, multiplying $\hat{\mathbf{M}}$ and $\mathbf{V}$ in Algorithm 2 takes

$$\mathcal{O}(ND + (N/s_0)^2 D + \sum_{i=1}^{k} m_i (s_{i-1}/s_i)^2 D)$$

also. The cost of creating a Y_{s_i} is $\mathcal{O}(N/s_i)$, so the cost of creating all $\mathbf{Y}_s$ for $s \in \{s_0, \cdots, s_k\}$ is

$$\mathcal{O}(\frac{N}{s_0} D + \cdots + \frac{N}{s_k} D) = \mathcal{O}(ND)$$

Then, for each $\mathbf{B}_{s,x,y} \in \mathbb{J}$, adding $\mu_{s,x,y} \mathbf{v}_{s,y}$ to $[\mathbf{Y}_s]_x$ takes $\mathcal{O}(D)$. The size of $\mathbb{J}$ is

$$\mathcal{O}((N/s_0)^2 D + \sum_{i=1}^{k} m_i (s_{i-1}/s_i)^2 D)$$

so the total complexity of Algorithm 2 is as stated.

Therefore, the total complexity of our approach is

$$\mathcal{O}(ND + (N/s_0)^2 D + \sum_{i=1}^{k} m_i (s_{i-1}/s_i)^2 D)$$

For example, when $\mathcal{R} = \{\sqrt{N}, 1\}$, the complexity becomes $\mathcal{O}(m_1 ND)$. The parameter m_1 adjusts the trade-off between approximation accuracy and runtime similar to other efficient methods, e.g., window size W in $\mathcal{O}(WND)$ for Longformer (Beltagy

et al., 2020) and projection size P in $\mathcal{O}(PND)$ for Linformer (Wang et al., 2020) or Performer (Choromanski et al., 2021).

5.3 Link to Sparsity and Low Rank

Low rank and sparsity are two popular directions for efficient self-attention. To explore the potential of these two types of approximations, we set aside the efficiency consideration and use the best possible methods for each type of approximation. Specifically, subject to $\left\|\hat{\mathbf{M}} - \mathbf{M}\right\|_F \leqslant \epsilon$, we use sparse approximation which minimizes $\left\|\hat{\mathbf{M}}\right\|_0$ by finding support on the largest entries of $\mathbf{M}$, and low rank approximation which minimizes $\text{rank}(\hat{\mathbf{M}})$ via truncated SVD. As shown in Figure 5.5, these two types of methods are limited for approximating self-attention. The low rank method requires superlinear cost to maintain the approximation accuracy and fails when the entropy of self-attention is smaller. In many cases, sparse approximation is sufficient, but in some cases when the self-attention matrices are less sparse and have larger entropy, the sparse approximation would fail as well. This motivates the use of sparse + low rank approximation. It can be achieved via robust PCA which decomposes approximation to $\hat{\mathbf{M}} = \mathbf{S} + \mathbf{L}$ by solving an optimization objective $\|\mathbf{S}\|_0 + \lambda\, \text{rank}(\mathbf{L})$. A convex relaxation $\|\mathbf{S}\|_1 + \lambda \|\mathbf{L}\|_*$ (Candès et al., 2011) is used to make the optimization tractable, but the cost of finding a good solution is still more than $O(N^2)$, which is not suitable for efficient self-attention. Scatterbrain (Chen et al., 2021a) proposes to combine an existing sparse attention with an existing low rank attention to obtain a sparse + low rank approximation and avoid the expensive cost of robust PCA.

Interestingly, a special form of our work offers an alternative to Scatterbrain's approach for sparse + low rank approximation. When $\mathcal{R} = \{b, 1\}$ for some b, our MRA-2 can be viewed as a sparse + low rank approximation. Specifically, let

$$\hat{\mathbf{M}}_b = \sum_{\mathbf{B}_{b,x,y} \in \mathcal{J}} \mu_{b,x,y} \mathbf{B}_{b,x,y}$$

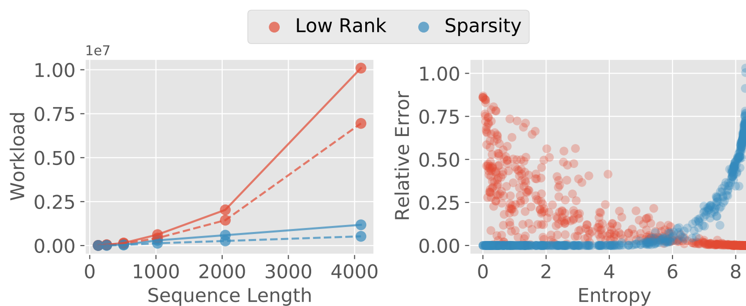

Figure 5.5: The solid and dashed lines in the left plot is the theoretical workload, without considering the overhead, necessary to get relative error less than 0.05 and 0.1, respectively, for different sequence length. Ideally, the workload should be linear in the sequence length. The right plot is the approximation error vs entropy of self-attention matrices, similar to Chen et al. (2021a), at 1/4 of the workload for standard self-attention (keeping 25% of the rank and nonzero entries for low rank and sparsity, respectively). Entropy of softmax is used as a proxy for the spread of attention. Relative error is defines as $\left\|\hat{\mathbf{D}}\hat{\mathbf{M}}\mathbf{V} - \mathbf{DMV}\right\|_F / \|\mathbf{DMV}\|_F$

for a resolution b, then $\hat{\mathbf{M}}_1 + \hat{\mathbf{M}}_b$ serves as a reasonably good solution for a relaxed version of sparse and low rank decomposition.

Let us consider an alternative relaxation of robust PCA objective,

$$\|\mathbf{S}\|_0 + \lambda \|\mathbf{L}\|_F \tag{5.10}$$

Note that $\|\mathbf{L}\|_F$ is easier to compute. And we have $\|\mathbf{L}\|_* \leqslant \sqrt{n}\,\|\mathbf{L}\|_F$ (Hu, 2015). In fact, Peng et al. (2018) shows that solutions obtained by minimizing $\|\mathbf{L}\|_*$ and $\|\mathbf{L}\|_F$ are two solutions of a low rank recovery problem and are identical in some situations. The optimal solution for objective (5.10) can be easily obtained. For $\epsilon = 0$, there exists a m such that the optimal S has support on the m largest entries of $\mathbf{M}$, and $\mathbf{L} = \mathbf{M} - \mathbf{S}$. However, for practical use, the recovered sparsity cannot be efficiently used on a GPU due to spatially scattered support. Further, the complexity is $O(N^2)$ since we found that we still need to find the largest entries of $\mathbf{M}$. Suppose we restrict $\mathbf{S}$ to be a block sparse matrix, namely, supported on a subset of $\{\text{supp}(\mathbf{B}_{b,x,y})\}_{x,y}$, then a GPU can exploit this block sparsity structure to

significantly accelerate computation. Further, the optimal $\mathbf{S}$ is supported on the regions with the largest $\mu'_{b,x,y}$ defined as

$$\mu'_{b,x,y} = \frac{1}{b^2} \sum_{(i,j) \in \text{supp}(\mathbf{B}_{b,x,y})} \exp(2\,[\mathbf{P}]_{i,j}) \tag{5.11}$$

which is similar to (5.4). As a result, similar to approximating (5.4) with (5.6), we use the lower bound $\mu^2_{b,x,y}$ as a proxy for (5.11). Then, the cost of locating support blocks is only $O(N^2/b^2)$. Consequently, the resulting solution S is supported on the regions with the largest $\mu_{b,x,y}$. Note that this S is exactly $\hat{\mathbf{M}}_1$ with an appropriate budget m. And the $\hat{\mathbf{M}}_b$ is a reasonable solution for $\mathbf{L}$ since $\left\|\hat{\mathbf{M}}_b\right\|_F$ is small and $\text{rank}(\hat{\mathbf{M}}_b) \leqslant n/b$.

We empirically evaluate the quality of this sparse solution. Kovaleva et al. (2019) showed that the BERT model (Devlin et al., 2019) has multiple self-attention patterns capturing different semantic information. We investigate the types of possible self-attention of a pretrained RoBERTa model (Liu et al., 2019a). And we show the optimal sparsity supports for self-attention matrices generated from RoBERTa-base with 4096 sequence in the top plots of Figure 5.6. Our MRA-2, as shown in the bottom plots of Figure 5.6, is able to find a reasonably good sparse solution for (5.10). Notice that while many self-attention matrices tend to be diagonally banded matrices, which Longformer and Big Bird can approximate well, they are not the only possible structures. A diagonally banded structure is not sufficient to approximate the last two self-attention patterns well.

5.4 Experiments

We performed a broad set of experiments to evaluate the practical performance profile of our MRA-based self-attention module. **First**, we compare our approximation accuracy with several other baselines. Then, we evaluate our method on the RoBERTa language model pretraining (Liu et al., 2019a) and downstream tasks on both short and long sequences. Finally, as is commonly reported in most evaluations

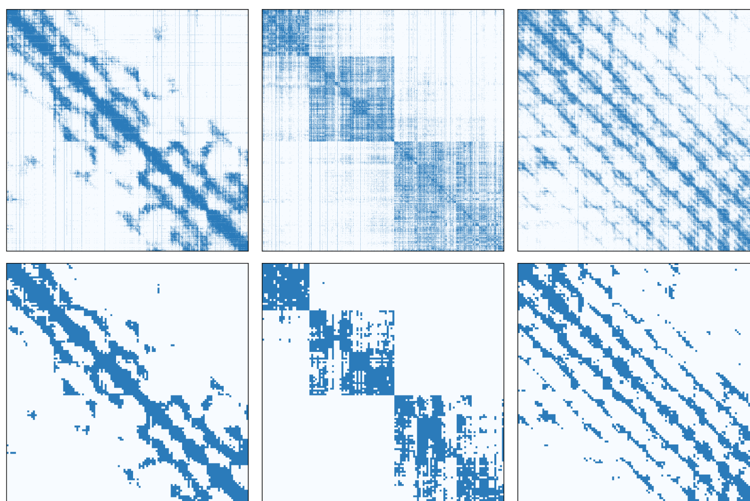

Figure 5.6: The top 3 plots are the optimal supports for 3 typical self-attention matrices at 80% sparsity. The bottom 3 plots are supports found via our MRA-2 at 80% sparsity.

of efficient self-attention methods, we discuss our evaluations on the Long Range Arena (LRA) benchmark (Tay et al., 2021). All hyperparameters are reported in Table 5.1.

Experiment	RoBERTa-base				RoBERTa-small				ImageNet
Task	MLM		MNLI	WikiHop	MLM		MNLI	WikiHop	
Sequence Length	512	4096	512	4096	512	4096	512	4096	1024
Num of Layers	12	12	12	12	4	4	4	4	4
Embedding Dim	768	768	768	768	128	128	128	128	128
Transformer Dim	768	768	768	768	384	384	384	384	128
Hidden Dim	3072	3072	3072	3072	1536	1536	1536	1536	512
Num of Heads	12	12	12	12	6	6	6	6	2
Head Dim	64	64	64	64	64	64	64	64	64
Batch Size	512	64	32	32	512	64	32	32	256
Learning Rate	5e-5	5e-5	3e-5	5e-5	1e-4	5e-5	3e-5	5e-5	5e-4
Num of Steps	20K	75K	4 epochs	15 epochs	150K	75K	10 epochs	25 epochs	300 epochs

Table 5.1: Hyperparameters for all experiments.

Efficiency Measurement. Since the efficiency is a core focus of efficient self-attention methods, time and memory efficiency is taken into account when evaluating performance. Whenever possible, we include runtime and memory consumption of a single instance for each method alongside the accuracy it achieves (in each table).

Since the models are exactly the same (except which self-attention module is used), we only profile the efficiency of one training step consumed by these modules. The efficiency is measured on a single Nvidia RTX 3090. We use the largest possible batch size for each method and average the measurements over multiple steps and batch sizes to get an accurate measurement of runtime and memory consumption of a single instance.

Baselines. For a rigorous comparison, we use an extensive list of baselines, including Linformer (Wang et al., 2020), Performer (Choromanski et al., 2021), Nyströmformer (Xiong et al., 2021), SOFT (Lu et al., 2021), YOSO (Zeng et al., 2021), Reformer (Kitaev et al., 2020), Longformer (Beltagy et al., 2020), Big Bird (Zaheer et al., 2020), H-Transformer-1D (Zhu and Soricut, 2021), and Scatterbrain (Chen et al., 2021a). Since Nystromformer, SOFT, and YOSO also have a variant which involves convolution, we perform evaluations for both cases. We use our multiresolution approximation with $R = \{32, 1\}$ for our method denoted in experiments as MRA-2. Further, we found that in tasks with limited dataset sizes, sparsity provides a regularization towards better performance. So, we include a MRA-2-s, which only computes

$$\hat{\mathbf{M}}_s = \sum_{\mathbf{B}_{1,x,y} \in \mathbb{J}} \mu_{1,x,y} \mathbf{B}_{1,x,y}$$

after finding $\mathbb{J}$. We use different method-specific hyperparameters for some methods to better understand their efficiency-performance trade off. **Takeaway**: These detailed comparisons suggest that our MRA-based self-attention offers favorable performance and efficiency among the baselines.

5.4.1 How good is the approximation accuracy?

We show that our method gives the best trade-off between approximation accuracy and efficiency by a significant margin compared to other baselines. The approximation accuracy of each method, compared to the standard self-attention, provides us a direct indication of the performance of approximation methods. To evaluate accuracy, we use 512 and 4096 length **Q**, **K**, and **V** from a pretrained model and

compute the relative error $\left\|\hat{\mathbf{D}}\hat{\mathbf{M}}\mathbf{V} - \mathbf{DMV}\right\|_F / \|\mathbf{DMV}\|_F$ where $\hat{\mathbf{D}}, \mathbf{D}$ are diagonal matrices to normalize $\hat{\mathbf{M}}$ and $\mathbf{M}$ for the normalization in softmax. As shown in Figure 5.7, our MRA-2(-s) has the lowest approximation error while maintaining the fastest runtime and smallest memory consumption by a large margin compared to other baselines in both short and long sequences.

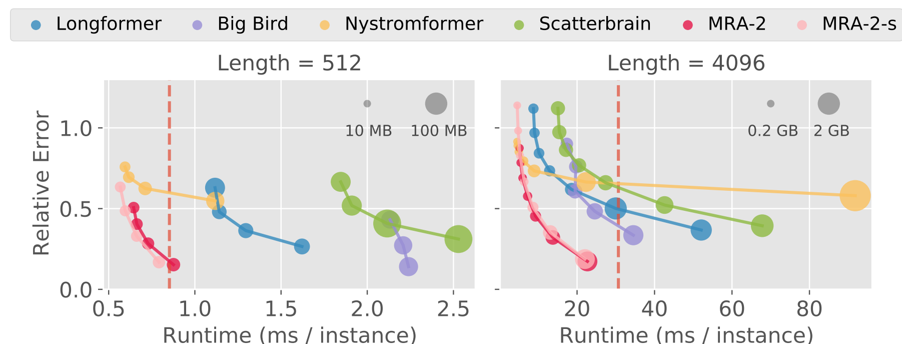

Figure 5.7: Approximation Error versus Runtime versus Memory. Red/dotted vertical line is the runtime of standard self-attention. Note that for any points to the right of the red vertical line, the approximation is slower than computing the true self-attention.

Next, we evaluate the effect of the spread (or entropy) of self-attention on the approximation for different methods. The result is shown in Figure 5.8. We see one limitation of low rank or sparsity-based schemes (discussed in §5.3 and Chen et al. (2021a)). Our MRA-2 performs well across attention instances with different entropy settings and significantly better than Scatterbrain (Chen et al., 2021a).

5.4.2 RoBERTa Language Modeling

Here, we use RoBERTa language modeling (Liu et al., 2019a) to assess the performance and efficiency trade off of our method and baselines. We use a pre-trained RoBERTa-base to evaluate the compatibility of each method with the existing Transformer models and overall feasibility for direct deployment. For fair comparisons, we also check the performance of models trained from scratch. Then, MNLI (Williams et al., 2018) is used to test the model's ability on downstream tasks.

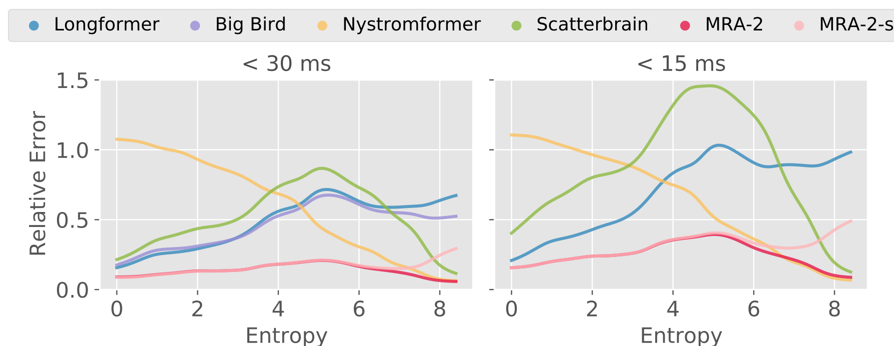

Figure 5.8: Entropy vs Approximation Error plots. Hyperparameters of each method is set such that the runtime is $< 30ms$ and $< 15ms$, resp. The fastest setting of Big Bird is still $> 15ms$, so omitted from the second plot. Note that the runtime of standard self-attention is roughly 30ms.

Further, we extend the 512 length models to 4096 length for a set of best performing methods and use the WikiHop (Welbl et al., 2018) task as an assessment on long sequence language models.

Standard Sequence Length. Since efficient self-attention approximates standard self-attention, we could simply substitute the standard self-attention of a trained model. This would allow us to minimize the training cost for new methods. To evaluate compatibility with the existing models, we use a pretrained 512 length RoBERTa-base model (Liu et al., 2019a) and replace its self-attention module with efficient alternatives and measure the validation Masked Language Modeling (MLM) accuracy. Then, we check accuracy after finetuning the model on English Wikipedia and Bookcorpus (Zhu et al., 2015). Eventually, we finetune the model on the downstream task MNLI (Williams et al., 2018).

Only a handful of schemes including Longformer, Big Bird, and MRA-2(-s) are fully compatible with pretrained models. Scatterbrain has a reasonable accuracy without further finetuning, but the training diverges when finetuning the model. The other methods cannot get a satisfactory level of accuracy. These statements also hold for the downstream finetuning results, shown in Table 5.2. Our method has the best performance among baselines for both MLM and MNLI. Meanwhile, it

has a much better time and memory efficiency.

Method	Time	Mem	MLM		MNLI	
	ms	MB	Before	After	m	mm
Transformer	**0.86**	71.0	**73.1**	**74.0**	**87.4**	**87.3**
Performer	1.29	62.8	6.8	63.1	32.7	33.0
Linformer	**0.74**	54.5	1.0	5.6	35.4	35.2
SOFT	**0.86**	34.0	10.9	25.0	32.7	33.0
SOFT + Conv	1.02	35.5	1.0	65.5	74.9	75.0
Nystromformer	**0.71**	34.8	17.2	68.2	35.4	35.2
Nystrom + Conv	0.88	37.2	1.4	70.9	85.1	84.6
YOSO	0.97	**29.8**	13.0	68.4	35.4	35.2
YOSO + Conv	1.20	**32.9**	3.0	69.0	83.2	83.1
Reformer	1.23	59.4	0.7	69.5	84.9	85.0
Longformer	1.30	43.3	66.0	71.2	85.6	85.4
	2.31	62.5	**71.9**	**73.2**	**87.0**	**87.1**
Big Bird	2.03	63.9	**71.6**	**73.3**	**87.1**	**87.0**
H-Transformer-1D	0.97	**29.3**	0.5	6.1	35.4	35.2
Scatterbrain	2.23	78.7	60.6	-	-	-
MRA-2	**0.73**	**28.1**	**68.9**	**73.1**	**86.8**	**87.1**
	0.86	34.3	**71.9**	**73.8**	**87.1**	**87.2**
MRA-2-s	**0.66**	**23.8**	**67.2**	72.8	**87.0**	**87.0**
	0.80	**29.1**	**71.8**	**73.8**	**87.4**	**87.4**

Table 5.2: Summary of 512 length RoBERTa-base models: runtime and memory efficiency, MLM accuracy, and MNLI accuracy. Unit for time (and memory) is ms (and MB). Before/After denotes accuracy before/after finetuning. The m/mm give the matched/mismatched MNLI. Some methods have more than one row for different model-specific hyperparameters. We divide measurements into three ranked groups for visualization (bold, normal, transparent).

Since many baselines are not compatible with the trained model weights (performance degrades when substituting the self-attention module), to make the comparison fair for all methods, we also evaluate models trained from scratch. Due to the large number of baselines we use, we train a small variant of RoBERTa on English Wikipedia and BookCorpus (Zhu et al., 2015) to keep the training cost reasonable. Then, we again finetune the model on downstream task (MNLI). Results are summarized in Table 5.3. Only a few methods (including ours) achieve both

good performance and efficiency.

Method	Time ms	Mem MB	MLM	MNLI m	MNLI mm
Transformer	**0.41**	35.47	**57.0**	72.7±0.6	73.8±0.2
Performer	0.63	31.38	48.6	69.8±0.4	70.5±0.1
Linformer	**0.35**	27.23	53.5	72.5±0.8	73.2±0.4
SOFT	0.43	17.02	42.8	63.8±2.2	64.7±2.6
SOFT + Conv	0.53	17.77	56.7	70.8±0.5	71.8±0.4
Nystromformer	**0.34**	17.40	53.1	71.4±0.6	72.0±0.3
Nystrom + Conv	0.45	18.60	**57.3**	73.0±0.4	73.9±0.6
YOSO	0.47	**14.91**	53.4	72.9±0.8	73.2±0.4
YOSO + Conv	0.58	**16.42**	**57.2**	72.5±0.4	72.9±0.5
Reformer	**0.39**	**16.43**	52.4	**73.7**±0.4	**74.6**±0.3
	0.61	29.65	55.6	**75.0**±0.2	**75.6**±0.3
Longformer	0.61	21.60	54.7	72.0±0.4	73.5±0.2
	1.10	31.44	**57.4**	**75.8**±0.5	**76.7**±0.6
Big Bird	1.02	31.91	**57.6**	**75.0**±0.5	**75.6**±0.6
H-Transformer-1D	0.47	**14.65**	43.7	62.9±2.7	63.4±3.9
Scatterbrain	1.04	78.66	20.5	42.6±8.1	43.4±9.5
MRA-2	**0.36**	**14.05**	56.4	**73.2**±0.2	**74.1**±0.5
	0.43	17.15	**57.3**	73.0±1.0	73.9±0.8
MRA-2-s	**0.31**	**11.93**	56.7	**73.6**±1.6	**74.3**±1.1
	0.38	**14.57**	**57.5**	**73.9**±0.6	**74.6**±0.8

Table 5.3: Summary of 512 length RoBERTa-small models. We also include a 95% error bar for experiments that have a small compute burden.

Longer Sequences. To evaluate the performance of our MRA-2(-s) on longer sequences, we extend the 512 length models to 4096 length. We extend the positional embedding and further train the models on English Wikipedia, Bookcorpus (Zhu et al., 2015), one third of Stories dataset (Trinh and Le, 2018), and one third of RealNews dataset (Zellers et al., 2019b) following (Beltagy et al., 2020). Then, the 4096 length models are finetuned on the WikiHop dataset (Welbl et al., 2018) to assess the performance of these models on downstream tasks. The results are summarized in Table 5.4 for base models and Table 5.5 for small models. Our MRA-2 is again one of the top performing methods with high efficiency among baselines. Note that the difference in WikiHop performance of Longformer (Beltagy et al., 2020) from the original paper is due to a much larger window size which has an

even slower runtime. Linformer (Wang et al., 2020) does not seem to be able to adapt the weights from its 512 length model to a 4096 model. It is interesting that the convolution in Nystromförmer (Xiong et al., 2021) seems to play an important role in boosting performance.

Method	Time (ms)	Mem (GB)	MLM	WikiHop
Transformer	30.88	3.93	**74.3**	**74.6**
Longformer	10.20	0.35	71.1	60.8
Big Bird	17.53	0.59	-	-
MRA-2	**7.03**	**0.28**	73.1	71.2
	9.25	0.38	**73.7**	**73.4**
MRA-2-s	**6.37**	**0.23**	73.0	71.8
	8.62	0.38	**73.8**	**74.1**

Table 5.4: Summary of 4096 length RoBERTa-base models. Since Big Bird is slow and we are not able to reduce its training time using multiple GPUs, we cannot can test Big Bird for 4096 sequence.

5.4.3 Long Range Arena

The Long Range Arena (LRA) (Tay et al., 2021) has been proposed to provide a lightweight benchmark to quickly compare the capability of long sequence modeling for Transformers. Due to a consistency issue and code compatibility of official LRA benchmark (see Issue-34, Issue-35, and Lee-Thorp et al. (2022)), we use the LRA code provided by Xiong et al. (2021) and follow exactly the same hyperparameter setting. The results are shown in Table 5.6. Our method has the best performance compared to others.

Caveats. A reader may ask why Longformer, Big Bird, and MRA-2-s perform better than standard Transformers (Vaswani et al., 2017) despite being approximations. The performance difference is most obvious on the image task. We also found that Longformer with a smaller local attention window (256, 128, 64) tends to offer better performance (39.52, 42.11, 42.71, respectively) on the image task. One reason is that standard self-attention needs larger datasets to compensate for its lack of locality bias (Xu et al., 2021; d'Ascoli et al., 2021). Hence, due to the small datasets

Method	Time (ms)	Mem (GB)	MLM	WikiHop
Transformer	15.36	1.96	**55.8**	**54.6**±1.6
Performer	5.13	0.24	23.2	43.7±0.6
Linformer	**2.85**	0.21	13.8	11.0±0.4
SOFT	**2.46**	**0.11**	25.9	14.0±8.6
	5.92	0.24	31.0	12.1±1.9
SOFT + Conv	**3.33**	**0.11**	52.8	30.8±29.3
Nystromformer	**2.38**	**0.11**	34.7	44.0±0.2
	4.34	0.27	46.8	46.0±0.8
Nystrom + Conv	**3.23**	**0.12**	53.1	**54.6**±0.8
YOSO	4.15	**0.12**	47.8	52.4±0.1
	5.07	0.17	49.9	52.8±0.5
YOSO + Conv	5.45	0.13	**55.1**	**53.2**±0.7
Reformer	5.04	0.24	52.2	**53.7**±0.9
Longformer	4.88	0.17	52.4	52.3±0.7
Big Bird	8.68	0.29	**54.4**	**54.3**±0.7
H-Transformer-1D	3.93	**0.12**	41.1	43.7±0.7
Scatterbrain	8.83	0.31	35.8	12.1±0.9
MRA-2	**3.43**	0.14	**54.2**	52.6±0.9
	4.52	0.19	**55.2**	**54.0**±0.9
MRA-2-s	**3.12**	**0.12**	**53.8**	51.8±0.9
	4.13	0.19	**55.1**	**53.6**±0.8

Table 5.5: Summary of 4096 length RoBERTa-small models.

(i.e., its lightweight nature), the LRA accuracy metrics should be interpreted with caution.

ImageNet. To test the performance on large datasets, we use ImageNet (Russakovsky et al., 2015) as a large scale alternative to CIFAR-10 (Krizhevsky et al., 2009) used in image task of LRA. Further, data augmentation is used to increase the dataset size. Like LRA, we focus on small models and use a 4-layer Transformer. Model specific hyperparameters are the same as the ones used on LRA. The results are shown in Table 5.7. MRA-2-s is the top performing approach. Standard self-attention and MRA-2 can clearly perform better on a large dataset.

Method	Listops	Text	Retrieval	Image	Pathfinder	Avg
Transformer	$37.1_{\pm 0.4}$	$\mathbf{65.2}_{\pm 0.6}$	$79.6_{\pm 1.7}$	$38.5_{\pm 0.7}$	$\mathbf{72.8}_{\pm 1.1}$	$58.7_{\pm 0.3}$
Performer	$36.7_{\pm 0.2}$	$65.2_{\pm 0.9}$	$79.5_{\pm 1.4}$	$38.6_{\pm 0.7}$	$71.4_{\pm 0.7}$	$58.3_{\pm 0.1}$
Linformer	$\mathbf{37.4}_{\pm 0.3}$	$57.0_{\pm 1.1}$	$78.4_{\pm 0.1}$	$38.1_{\pm 0.3}$	$67.2_{\pm 0.1}$	$55.6_{\pm 0.3}$
SOFT	$36.3_{\pm 1.4}$	$65.2_{\pm 0.0}$	$\mathbf{83.3}_{\pm 1.0}$	$35.3_{\pm 1.3}$	$67.7_{\pm 1.1}$	$57.5_{\pm 0.5}$
SOFT + Conv	$37.1_{\pm 0.4}$	$65.2_{\pm 0.4}$	$\mathbf{82.9}_{\pm 0.0}$	$37.1_{\pm 4.7}$	$68.1_{\pm 0.4}$	$58.1_{\pm 0.9}$
Nystromformer	$24.7_{\pm 17.5}$	$\mathbf{65.7}_{\pm 0.1}$	$\mathbf{80.2}_{\pm 0.3}$	$38.8_{\pm 2.9}$	$\mathbf{73.1}_{\pm 0.1}$	$56.5_{\pm 2.8}$
Nystrom + Conv	$30.6_{\pm 8.9}$	$\mathbf{65.7}_{\pm 0.2}$	$78.9_{\pm 1.2}$	$\mathbf{43.2}_{\pm 3.4}$	$69.1_{\pm 1.0}$	$57.5_{\pm 1.5}$
YOSO	$37.0_{\pm 0.3}$	$63.1_{\pm 0.2}$	$78.3_{\pm 0.7}$	$40.8_{\pm 0.8}$	$\mathbf{72.9}_{\pm 0.6}$	$58.4_{\pm 0.3}$
YOSO + Conv	$\mathbf{37.2}_{\pm 0.5}$	$64.9_{\pm 1.2}$	$78.5_{\pm 0.9}$	$\mathbf{44.6}_{\pm 0.7}$	$69.5_{\pm 3.5}$	$\mathbf{59.0}_{\pm 1.1}$
Reformer	$18.9_{\pm 2.4}$	$64.9_{\pm 0.4}$	$78.2_{\pm 1.6}$	$\mathbf{42.4}_{\pm 0.4}$	$68.9_{\pm 1.1}$	$54.7_{\pm 0.2}$
Longformer	$37.2_{\pm 0.3}$	$64.1_{\pm 0.1}$	$79.7_{\pm 1.1}$	$\mathbf{42.6}_{\pm 0.1}$	$70.7_{\pm 0.8}$	$\mathbf{58.9}_{\pm 0.1}$
Big Bird	$\mathbf{37.4}_{\pm 0.3}$	$64.3_{\pm 1.1}$	$79.9_{\pm 0.1}$	$40.9_{\pm 1.1}$	$\mathbf{72.6}_{\pm 0.7}$	$\mathbf{59.0}_{\pm 0.3}$
H-Transformer-1D	$30.4_{\pm 8.8}$	$\mathbf{66.0}_{\pm 0.2}$	$\mathbf{80.1}_{\pm 0.4}$	$\mathbf{42.1}_{\pm 0.8}$	$70.7_{\pm 0.1}$	$57.8_{\pm 1.8}$
Scatterbrain	$\mathbf{37.5}_{\pm 0.1}$	$64.4_{\pm 0.3}$	$79.6_{\pm 0.1}$	$38.0_{\pm 0.9}$	$54.8_{\pm 7.8}$	$54.9_{\pm 1.4}$
MRA-2	$37.2_{\pm 0.3}$	$\mathbf{65.4}_{\pm 0.1}$	$79.6_{\pm 0.6}$	$39.5_{\pm 0.9}$	$\mathbf{73.6}_{\pm 0.4}$	$\mathbf{59.0}_{\pm 0.3}$
MRA-2-s	$\mathbf{37.4}_{\pm 0.5}$	$64.3_{\pm 0.8}$	$\mathbf{80.3}_{\pm 0.1}$	$41.1_{\pm 0.4}$	$\mathbf{73.8}_{\pm 0.6}$	$\mathbf{59.4}_{\pm 0.2}$

Table 5.6: Test set accuracy of LRA tasks. Since the benchmark consist of multiple tasks with different sequence length, we do not include the efficiency components in the table.

Method	Time (ms)	Mem (MB)	Top-1	Top-5
Transformer	1.24	45.5	48.7	73.7
Reformer	1.14	19.1	39.6	65.5
Longformer	1.12	13.7	**49.1**	73.9
H-Transformer-1D	1.03	**9.8**	48.7	**73.9**
MRA-2	**1.00**	11.8	48.9	73.6
MRA-2-s	**0.98**	**9.7**	**49.2**	**73.9**

Table 5.7: Summary of ImageNet results trained on 4-layer Transformers. We reports both top-1 and top-5 accuracy.

5.5 Summary

In this chapter, we showed that Multi-Resolution Analysis (MRA) provides fresh ideas for efficiently approximating self-attention, which subsumes many piecemeal approaches in the literature. We expect that exploiting the links to MRA will allow leveraging a vast body of technical results developed over many decades. But we show that there are tangible practical benefits available immediately. When some consideration is given to which design choices or heuristics for a MRA-based self-attention scheme will interface well with mature software stacks and modern hardware, we obtain a procedure with strong advantages across both *performance/accuracy and efficiency*. Further, our implementation can be directly plugged into existing Transformers, a feature missing in some existing efficient transformer implementations. Finally, we should note the lack of integrated software support for MRA as well as our specialized model in current deep learning libraries. Overcoming this limitation required implementing custom CUDA kernels for some generic block sparsity operators. Therefore, extending our algorithm for other use cases may involve reimplementing the kernel. We hope that with broader use of MRA-based methods, the software support will improve thereby reducing this implementation barrier. A preliminary version of this chapter was published as (Zeng et al., 2022) and the codebase is available at `https://github.com/mlpen/mra-attention`.

6 MULTI-RESOLUTION BASED APPROXIMATION FOR FASTER TRANSFORMER BLOCK

Despite recent works, including the ones described in Chapter 3 and 5, devoted to reducing the quadratic cost $O(LN^2D)$ of self-attention computation in the last few years, we find that this is not sufficient to run Transformer models on ultra long sequences (e.g., >16K tokens) efficiently. Existing self-attention mechanisms often reduce the quadratic cost of self-attention to linear. But so far, most experiments for efficient self-attentions report sequence lengths of up to 4K, with some exceptions (Beltagy et al., 2020; Zaheer et al., 2020; Guo et al., 2022). Beyond 4K, the linear cost (relative to N) for both computing efficient attention and FFN makes the cost prohibitive, especially for large models. For example, although LongT5 (Guo et al., 2022) can train on sequence lengths of up to 16K tokens with an efficient self-attention and shows promising results for longer sequences, it is slower and needs a sizable amount of compute (for example, see Figure 6.1). Similar to efficient self-attention with linear cost attention, (Wu et al., 2022) and (Bulatov et al., 2022) do not try to reduce the linear cost (on sequence length) for the feedforward network, so the computation will still be expensive for processing ultra long sequences.

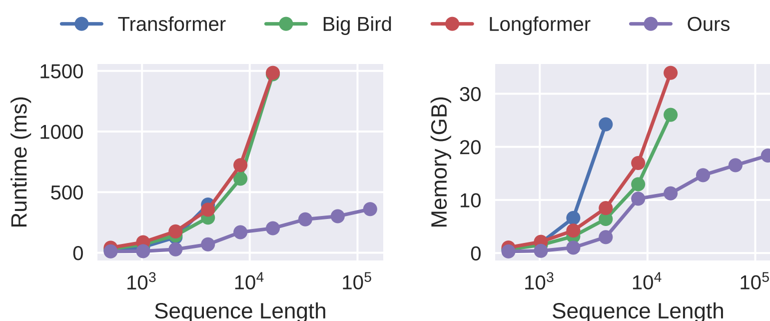

Figure 6.1: Model efficiency of processing one sequence on a NVIDIA A100 as sequence length increases (note logarithm x axis).

As a result, applications such as answering questions based on a book or summa-

rizing a scientific article, which require Transformers to process the entire book or article, are still inefficient or infeasible. To address this issue, special care must be taken with respect to this linear "N" term to efficiently and fully utilize Transformers' ability to learn long range dependency.

In this chapter, we seek to reduce the overall cost (both Multi-Head Attention (MHA) and Feedforward Network (FFN)) of processing ultra long sequences. Specifically, by exploiting the fact that in many tasks, only a small subset of special tokens, which we call VIP-tokens, are most relevant to the tasks, we propose a compression scheme which selectively compresses the sequence based on their impact on approximating the representation of the VIP-tokens via inspiration from the Multi-Resolution approximation that we exploited in Chapter 5.

(1) Focus on what we need for a task: VIP-token centric compression (VCC). We hypothesize/find that in many tasks where Transformers are effective, only a small subset of tokens, which we refer to as VIP-tokens, are relevant to the final output (and accuracy) of a Transformer. If these tokens had been identified somehow, we could preserve this information in its entirety and only incur a moderate loss in performance. Now, *conditioned* on *these* specific VIP-tokens, an aggressive compression on the other *non-VIP-tokens*, can serve to reduce (and often, fully recover) the loss in performance while dramatically decreasing the sequence length. This compression must leverage information regarding the VIP-tokens, with the goal of improving the approximation of the representation of the VIP-tokens. In other words, a high-fidelity approximation of the entire sequence is unnecessary. Once this “selectively compressed” input passes through a Transformer layer, the output sequence is decompressed to the original full sequence allowing the subsequent blocks to access the full sequence.

(2) Specialized data structure for compression/decompression. A secondary, but important practical issue, is reducing the overhead when compressing/decompressing the input/output sequences internally in the network. Ignoring this problem will impact efficiency. We give a simple but specialized data structure to maintain the hidden states of the intermediate blocks, where the compression

can be easily accessed from the data structure, and explicit decompression can be avoid by updating the data structure: the sequence is never fully materialized in intermediate blocks.

Practical contributions. Apart from the algorithmic modules above, we show that despite an aggressive compression of the input sequences, we achieve better/competitive performance on a broad basket of long sequence experiments. Compared to baselines, we get much better runtime/memory efficiency. We show that it is now practical to run **standard** Transformer models on sequences of 128K token lengths, with consistent performance benefits (and **no** complicated architecture changes).

6.1 Notations

Here, we define some notations for later discussion. As before, let $\mathbf{X} \in \mathbb{R}^{N \times D}$ be be the input for a Transformer encoder block. As discussed in §2.2, the output of this encoder block, $\mathbf{X}_{new}$, is defined as

$$\mathbf{X}_{new} = \mathcal{F}(\mathcal{A}(\mathbf{X}) + \mathbf{X}) + \mathcal{A}(\mathbf{X}) + \mathbf{X}$$

using $\mathcal{A}(\mathbf{X})$ as shorthand for $\mathcal{A}(\cdot, \cdot, \cdot)$, which is a MHA with $\mathbf{X}$ as input for queries, keys, and values. Here, $\mathcal{F}(\cdot)$ is a FFN. The simplified $\mathcal{A}$ can be expressed as:

$$\mathcal{A}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \exp(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}.$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are queries, keys, and values for MHA.

We define $\mathcal{G}(\cdot)$ be a placeholder for all heavy computations in the Transformer layer above:

$$\mathcal{G}(\mathbf{X}) := \mathcal{F}(\mathcal{A}(\mathbf{X}) + \mathbf{X}) + \mathcal{A}(\mathbf{X}).$$

We can verify that the output of a Transformer block is,

$$\mathbf{X}_{new} = \mathcal{G}(\mathbf{X}) + \mathbf{X}.$$

As discussed in §1.2, the overall complexity is $\mathcal{O}(LN^2D + LND^2)$.

6.2 VIP-Token Centric Compression (VCC)

In this chapter, our main goal is to reduce the dependency on N (but *not* by modifying Transformer internals). To do this, we describe a scheme that compresses the input sequence of a Transformer block and decompresses the output sequence, resulting in a model whose complexity is $\mathcal{O}(LRD^2 + LR^2D + LR\log(N_c)D + LRN_pD + ND)$. Here, R is the length of the compressed sequence, N_p is the number of VIP-tokens described shortly, and N_c is the size of non-VIP/remaining tokens. So, we have $N_p + N_c = N$ and assume $N_p \ll R \ll N$.

Parsing the complexity term: Let us unpack the term to assess its relevance. The first two terms $\mathcal{O}(LRD^2 + LR^2D)$ pertain to the cost for a Transformer, while the remaining terms are the overhead of compression and decompression. The term $\mathcal{O}(LR\log(N_c)D + LRN_pD)$ is the overhead of compression and updating our data structure at each block. The $\mathcal{O}(ND)$ term corresponds to pre-processing which involves converting the hidden states into our data structure and the post-processing steps to recover the hidden states from the data structure. Note that unlike the dependence on N for vanilla Transformers, this $\mathcal{O}(ND)$ is incurred only at the input/output stage of the Transformer, but **not** at any intermediate blocks.

High level design choices. We use the *standard* Transformer blocks with a *standard* feed-forward network (which results in D^2 in the first term) and *standard* quadratic cost self-attention (which gives the R^2 factor in the second term). Why? These choices help isolate the effect of incorporating their efficient counterparts. The proposed algorithm operates on the *input/output of each Transformer block* leaving the Transformer module itself unchanged. Therefore, our goals are distinct from the

literature investigating efficient self-attention and efficient feed-forward networks. This is because one can replace these two vanilla modules with *any other* efficient alternatives to further reduce the R^2 and D^2 terms directly. Despite these quadratic terms, our approach is faster than baselines, discussed in §6.5.

We will first describe our general idea, as shown in Figure 6.2, which uses VIP-tokens to guide the compression/decompression of the input/output of a Transformer block so that it only needs to process the compressed sequence. Then, we will discuss an instantiation of the compression process, by adapting a multi-resolution analysis technique (discussed in §6.3). Lastly, we will introduce a data structure which allows more efficient compression/decompression, discussed in §6.4.

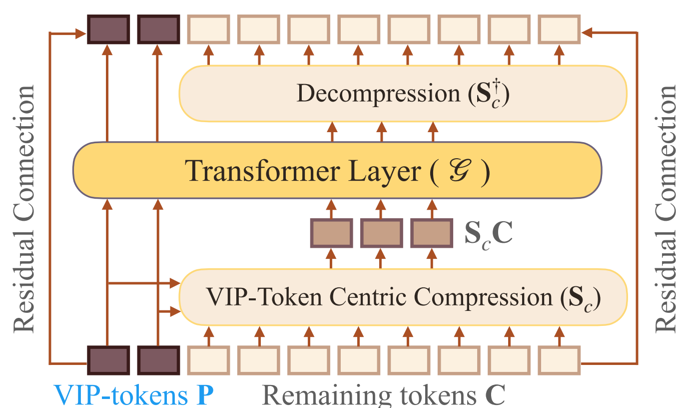

Figure 6.2: Diagram that illustrates a Transformer layer with VIP-token centric sequence compression.

6.2.1 Elevating the Importance of a Few Tokens: VIP-Tokens

Let us start with the simplest compression, which identifies a linear transformation $\mathbf{S} \in \mathbb{R}^{R \times N}$ which acts on the input, resulting in a smaller representation $\mathbf{SX} \in \mathbb{R}^{R \times D}$. Of course, a smaller R implies that more information about $\mathbf{X}$ is lost. But we find that in many tasks, only the embedding representations of *a few* tokens drive the final prediction: we refer to these tokens as *VIP-tokens*.

Examples of VIP-tokens: Observe that only the embedding outputs of masked

tokens in masked language modeling Devlin et al. (2019) and the CLS token in sequence classification Devlin et al. (2019); Dosovitskiy et al. (2021) are/is used for prediction. In question answering, only the questions and possible answers associated with the questions are used for prediction. It is important to note that the masked tokens, CLS tokens, and question tokens are **(1)** defined by the tasks and **(2)** *known* to the model (although the embedding representation of these tokens are unknown). These VIP-tokens can be viewed as a task or question that is given to the model. The model can process the sequence with a specific goal in mind so that the model can skip/skim less relevant segments. Our general principle involves choosing *a set of tokens* as the VIP-tokens that **(1)** are important to the specific task goals and **(2)** easily pre-identifiable by the user.

Caveats. Not all important tokens can be pre-identified. For example, the tokens in the correct answer span in answer span prediction are also important to the specific goals, but are difficult to pre-identify, so only the question tokens (and not the answer tokens) are used as VIP-tokens. We assume that any other tokens that are relevant for prediction should have high dependency with these VIP-tokens. For example, the answer tokens should have high dependency (in self-attention) with respect to the question tokens.

VIP-tokens occupy the front seats. VIP-tokens can occur anywhere within a sequence. But we can re-order the sequence as well as the positional encodings so that VIP-tokens are always at the *head of sequence* to make analysis/implementation easier. With this layout, let $\mathbf{P} \in \mathbb{R}^{N_p \times D}$ be the VIP-tokens and $\mathbf{C} \in \mathbb{R}^{N_c \times D}$ be the non-VIP/remaining tokens, $\mathbf{X}$ can be expressed as

$$\mathbf{X} = \begin{bmatrix} \mathbf{P} \\ \mathbf{C} \end{bmatrix} \tag{6.1}$$

This is possible since the Transformer model is permutation invariant when permuting positional encodings (embeddings or IDs) along with tokens. This re-ordering is performed only once for the input of the Transformer model, then the outputs generated by the model are rearranged to their original positions. Re-ordering

makes the analysis, implementation and presentation of our method much clearer and simpler. In fact, placing VIP tokens at the end of the sequence can also serve the same purpose.

From the above discussion, it is clear that one needs to make sure that after compressing the input tokens $\mathbf{X}$, the VIP-tokens must still stay (more or less) the same, and the compression matrix $\mathbf{S}$ must be *VIP-token dependent*. We hypothesize that such *VIP-token dependent* compression matrices require a much smaller dimension R, compared to *VIP-token agnostic* compression matrices.

6.2.2 VIP-Token Centric Compression (VCC): An Initial Proposal

For a Transformer block, let $\mathbf{X}$ denote its input matrix. Express the output of this block as follows:

$$\mathbf{X}_{new} = \mathbf{S}^{\dagger}\mathcal{G}(\mathbf{S}\mathbf{X}) + \mathbf{X} \tag{6.2}$$

where $\mathbf{S} \in \mathbb{R}^{R \times N}$ is a *compression* matrix compressing $\mathbf{X}$ to a smaller representation and $\mathbf{S}^{\dagger}$ is the pseudo inverse for *decompression*. With the layout in (6.1), we can write (6.2) as

$$\begin{bmatrix} \mathbf{P}_{new} \\ \mathbf{C}_{new} \end{bmatrix} = \mathbf{S}^{\dagger}\mathcal{G}\left(\mathbf{S}\begin{bmatrix} \mathbf{P} \\ \mathbf{C} \end{bmatrix}\right) + \begin{bmatrix} \mathbf{P} \\ \mathbf{C} \end{bmatrix} \tag{6.3}$$

where $\mathbf{P}_{new}$ and $\mathbf{C}_{new}$ are the new embeddings for $\mathbf{P}$ and $\mathbf{C}$.

Always reserve seats for VIP-tokens. What is a useful structure of $\mathbf{S}$? Since $\mathbf{P}_{new}$ is the embedding output for the VIP-tokens $\mathbf{P}$, we want them to be fully preserved. To achieve this, we impose the following structure on $\mathbf{S}$ and $\mathbf{S}^{\dagger}$:

$$\mathbf{S} = \begin{bmatrix} \mathbf{I}_{N_p} & 0 \\ 0 & \mathbf{S}_c \end{bmatrix} \qquad \mathbf{S}^{\dagger} = \begin{bmatrix} \mathbf{I}_{N_p} & 0 \\ 0 & \mathbf{S}_c^{\dagger} \end{bmatrix}. \tag{6.4}$$

The rearrangement simply says that we will avoid compressing $\mathbf{P}$. But rewriting it in this way helps us easily unpack (6.3) to check the desired functionality of $\mathbf{S}_c$.

Prioritize information in VIP-tokens. Our goal is to ensure that $\mathbf{P}_{new}$ generated from the compressed sequence in (6.3) will be similar to its counterpart from the uncompressed sequence. Let us check (6.3) using the compression matrix $\mathbf{S}$ defined in (6.4) first. We see that

$$\begin{bmatrix} \mathbf{I}_{N_p} & 0 \\ 0 & \mathbf{S}_c^\dagger \end{bmatrix} \mathcal{G}\left(\begin{bmatrix} \mathbf{P} \\ \mathbf{S}_c \mathbf{C} \end{bmatrix} \right) = \begin{bmatrix} \mathcal{F}(\mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX}) + \mathbf{P}) + \mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX}) \\ \mathbf{S}_c^\dagger \mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}) + \mathbf{S}_c^\dagger \mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) \end{bmatrix} \tag{6.5}$$

The orange color identifies terms where $\mathbf{P}_{new}$ interacts with other compression-related terms $\mathbf{C}$ and/or $\mathbf{S}_c$. We primarily care about $\mathbf{P}_{new}$ in (6.3), so the first (orange) row in (6.5) is the main concern. We see that $\mathbf{P}_{new}$ only depends on the compressed $\mathbf{SX}$ via $\mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX})$. We can further unpack,

$$\mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX}) = \exp(\mathbf{P}\mathbf{X}^\top \mathbf{S}^\top)\mathbf{SX} = \exp(\mathbf{P}\mathbf{P}^\top)\mathbf{P} + \exp(\mathbf{P}\mathbf{C}^\top \mathbf{S}_c^\top)\mathbf{S}_c\mathbf{C}.$$

Again, $\mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX})$ depends on $\mathbf{C}$ and $\mathbf{S}_c$ via the second (orange) term. Normalization in softmax is omitted for simplicity of discussion. This helps us focus on the key term that matters: $\exp(\mathbf{P}\mathbf{C}^\top \mathbf{S}_c^\top)\mathbf{S}_c\mathbf{C}$. As long as the following approximation using $\mathbf{S}_c$ is good

$$\exp(\mathbf{P}\mathbf{C}^\top \mathbf{S}_c^\top)\mathbf{S}_c \approx \exp(\mathbf{P}\mathbf{C}^\top), \tag{6.6}$$

we will obtain a good approximation of $\mathbf{P}_{new}$. Our remaining task is to outline a scheme of finding a compression matrix $\mathbf{S}_c$ such that this criterion can be assured.

6.3 A Specific Instantiation via Multi-Resolution Compression

What should be the mechanics of our compression such that (6.6) holds? In general, to get $\mathbf{S}_c$, we can use any sensible data driven sketching idea which minimizes the error of (6.6).

High level idea. Ideally, an efficient scheme for constructing $\mathbf{S}_c$ should operate as

follows. If some regions of the sequence $\mathbf{C}$ have a negligible impact on (6.6) (via the orange terms above), the procedure should compress the regions aggressively. If other regions are identified to have a higher impact on (6.6) (again due to the orange terms above), the procedure should scan these regions more carefully for a more delicate compression. This suggests that procedurally a coarse-to-fine strategy may work. For example, multi-resolution analysis discussed in Chapter 5 does help in approximating self-attention matrices in Transformers, but the formulation in Chapter 5 cannot be easily written in a form similar to (6.6), making it incompatible with our design. Nonetheless, we derive an analogous form in §6.3.1 that can be represented in a similar form as (6.6) and gives a strategy for obtaining $\mathbf{S}_c\mathbf{C}$.

Specifically, let us define a compressed representation (via averaging) of the x-th s-length segment of sequence $\mathbf{C}$: $\mathbf{c}_{s,x} \in \mathbb{R}^D$

$$\mathbf{c}_{s,x} := \frac{1}{s} \sum_{sx-s<i\leqslant sx} [\mathbf{C}]_i \tag{6.7}$$

where $s \in \{k^0, k^1, k^2, \cdots, N_c\}$ assuming N_c is a power of k and $x \in \{1, 2, \cdots, N_c/s\}$. We fix the increment ratio $k = 2$ for simplicity of discussion. The s represents the resolution of the approximation: it represents the number of non-VIP token embeddings being averaged into a vector $\mathbf{c}_{s,x}$. Higher s (e.g., $s = 8$ in $\mathbf{c}_{8,1}$ in Figure 6.3) means lower resolution and heavier compression of the corresponding segment. The x represents the location of the s-length segment within the sequence $\mathbf{C}$. In our scheme, we compress the sequence $\mathbf{C}$ and use a set of $\mathbf{c}_{s,x}$ for some selected s's and x's as the rows of the compressed $\mathbf{S}_c\mathbf{C}$ as seen in Figure 6.3. The sequence $\mathbf{C}$ is broken into multiple segments of different lengths, then each segment is compressed into a vector $\mathbf{c}_{s,x}$.

Procedurally, as shown in Figure 6.3, our scheme starts with the heaviest compression and progressively refines certain segments of $\mathbf{C}$ guided by the VIP-tokens $\mathbf{P}$. The scheme starts with the heaviest compression that treats $\mathbf{C}$ as a N_c-length segment and compresses it to a single $\mathbf{c}_1^{N_c}$. Then, starting with $s = N_c$ (root node), the procedure **(1)** computes the averaged attention scores between VIP-tokens

$\mathbf{P}$ and $\mathbf{c}_x^s$'s for different x's (averaged over all attention heads and all VIP-tokens; only one $\mathbf{c}_1^{N_c}$ at level $s = N_c$). We note that the attention scores are obtained by extracting attention matrices from the MHA module of the current Transformer layer when using $\mathbf{P}$ as queries and $\mathbf{c}_x^s$'s as keys. Then, it **(2)** splits the s-length segments, represented as $\mathbf{c}_x^s$, with higher averaged attention scores (one segment is split in Figure 6.3 but we might split more segments, again only one $\mathbf{c}_{N_c,1}$ at level $s = N_c$) into $(s/2)$-length sub-segments: the corresponding $\mathbf{c}_{s/2,x}$ (6.7) of each sub-segment is computed for finer representation. Then, at next level for $s = N_c/2$, the same procedure proceeds. This process continues until the sub-segments have length 1. We note that this procedure is guided by the VIP-tokens $\mathbf{P}$ and designed to maximally reduce the error of approximating (6.6). No additional learnable parameters are introduced for this scheme.

Figure 6.3: Illustration of multi-resolution compression. $n_c = 8$. Purple line: compute attention scores between $\mathbf{P}$ and different segments. Green arrow: segment with higher attention score is split into two sub-segments. Accordingly, $\mathbf{S}_c\mathbf{C} = \begin{bmatrix}\mathbf{c}_1^2 & \mathbf{c}_3^1 & \mathbf{c}_4^1 & \mathbf{c}_2^4\end{bmatrix}^\top$ is constructed.

6.3.1 Deviation of Multi-Resolution Compression

In this subsection, we describe the technical details of a modified formulation of Chapter 5 to construct $\mathbf{S}_c\mathbf{C}$ and corresponding $\mathbf{S}_c$ satisfying good approximation of

$$\exp(\mathbf{P}\mathbf{C}^\top\mathbf{S}_c^\top)\mathbf{S}_c \approx \exp(\mathbf{P}\mathbf{C}^\top). \tag{6.8}$$

6.3.1.1 Basic Problem Setup

Let $\mathbf{b}_{s,x} \in \mathbb{R}^{N_c}$ be a multi-resolution component defined as

$$[\mathbf{b}_{s,x}]_i := \begin{cases} \frac{1}{s} & \text{if } sx - s < i \leqslant sx \\ 0 & \text{otherwise} \end{cases} \tag{6.9}$$

for $s \in \{k^0, k^1, k^2, \cdots, N_c\}$ assuming N_c is a power of k (we assume $k = 2$ for simiplicity). Here, s and x represent the scaling and translation (similar to the concepts in wavelet basis) of the component, respectively. Figure 6.4 is the visualization of $\mathbf{b}_{s,x}$.

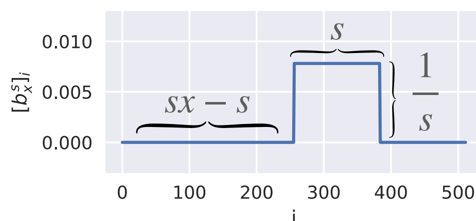

Figure 6.4: Visualization of $\mathbf{b}_{s,x}$ for a specific scaling s and translation x.

Then, any 1D signal $\mathbf{f} \in \mathbb{R}^{N_c}$ can be represented as a linear combination of $\mathbf{b}_{s,x}$:

$$\mathbf{f} = \sum_{s,x} c_{s,x}\mathbf{b}_{s,x}$$

where $c_{s,x}$ are the coefficients for the linear combination. For a signal with multi-resolution structure (that is, signal has high frequency in some regions and has low

frequency in other regions), we can find an approximation $\hat{\mathbf{f}}^*$ that can be expressed as a *sparse* linear combination where most coefficients are zeros, as shown in Figure 6.5.

$$\mathbf{f} \approx \hat{\mathbf{f}}^* := \sum_{\mathbf{b}_{s,x} \in \mathbb{J}} c_{s,x} \mathbf{b}_{s,x} \tag{6.10}$$

We denote $\mathbb{J}$ as the set of major components $\mathbf{b}_{s,x}$ corresponding to the large coefficients, that is, $\mathbb{J} := \{\mathbf{b}_{s,x} \mid |c_{s,x}| \text{ being large}\}$. Since the set of all possible $\mathbf{b}_{s,x}$ is an over-complete dictionary, there are multiple possible linear combinations. To reduce the search space of the best set $\mathbb{J}$, we place a mild restriction on the set $\mathbb{J}$:

$$\left[\sum_{\mathbf{b}_{s,x} \in \mathbb{J}} \mathbf{b}_{s,x} \right]_i \neq 0 \quad \forall i \qquad \langle \mathbf{b}_{s,x}, \mathbf{b}_{s',x'} \rangle = 0 \quad \forall \mathbf{b}_{s,x}, \mathbf{b}_{s',x'} \in \mathbb{J}, \mathbf{b}_{s,x} \neq \mathbf{b}_{s',x'} \tag{6.11}$$

The conditions state that each entry of signal $\mathbf{f}$ is included in the support region of exactly one component in $\mathbb{J}$. With these tools, we will first describe the approximation when $\mathbb{J}$ is given, then discuss how the approximation connects the set $\mathbb{J}$ to our target $\mathbf{S}_c$ and $\mathbf{S}_c\mathbf{C}$. Finally, we will discuss how to construct this $\mathbb{J}$.

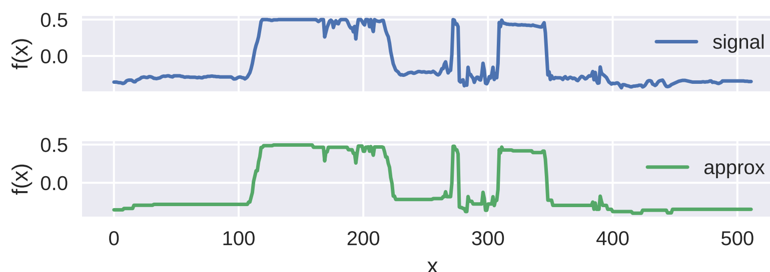

Figure 6.5: An example of approximating an 1D signal using a truncated wavelet transform with components defined in (6.9). It uses a set $\mathbb{J}$ of size 79 to represent a signal in $\mathbb{R}^{512}$.

6.3.1.2 Plugging Our Problem into the Setup

Chapter 5 shows that the self-attention matrix $\exp(\mathbf{X}\mathbf{X}^\top)$ has the multi-resolution structure discussed above. Since $\exp(\mathbf{P}\mathbf{C}^\top)$ is a sub-matrix of $\exp(\mathbf{X}\mathbf{X}^\top)$, we conjec-

ture that the multi-resolution structure also holds in $\exp(\mathbf{P}\mathbf{C}^\top)$. As a result, we can find a sparse combination of $\mathbf{b}_{s,x}$ to represent rows of $\exp(\mathbf{P}\mathbf{C}^\top)$.

Claim 6.1. *Given the set* $\mathbb{J}$ *satisfying restriction* (6.11)*, we can define an approximation of the* i*-th row of* $\exp(\mathbf{P}\mathbf{C}^\top)$ *similar to* (6.10) *as illustrated in Figure 6.5*

$$\left[\widehat{\exp(\mathbf{P}\mathbf{C}^\top)}^*\right]_i := \sum_{\mathbf{b}_{s,x}\in\mathbb{J}} c_{s,x}\mathbf{b}_{s,x}$$

where $c_{s,x}$ *is the optimal solution that minimizes*

$$\left\|\left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i - \left[\widehat{\exp(\mathbf{P}\mathbf{C}^\top)}^*\right]_i\right\|_2^2$$

Then, the approximation can be written as:

$$\left[\widehat{\exp(\mathbf{P}\mathbf{C}^\top)}^*\right]_{i,j} = \langle\left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i, \mathbf{b}_{s,x}\rangle \tag{6.12}$$

where $\mathbf{b}_{s,x} \in \mathbb{J}$ *is the component that is supported on* j *(a.k.a.* $[\mathbf{b}_{s,x}]_j \neq 0$ *and there is exactly one* $\mathbf{b}_{s,x} \in \mathbb{J}$ *satisfy this condition due to restriction* (6.11)*).*

Proof. If $\mathbb{J}$ is given, let $\mathbf{B} \in \mathbb{R}^{N_c\times|\mathbb{J}|}$ be a matrix whose columns are elements $\mathbf{b}_{s,x} \in \mathbb{J}$ and let $\mathbf{c} \in \mathbb{R}^{|\mathbb{J}|}$ be a vector whose entries are the corresponding $c_{s,x}$:

$$\mathbf{B} := \begin{bmatrix}\mathbf{b}_{s_1,x_1} & \mathbf{b}_{s_2,x_2} & \cdots & \mathbf{b}_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}}\end{bmatrix}$$
$$\mathbf{c} := \begin{bmatrix}c_{s_1,x_1} & c_{s_2,x_2} & \cdots & c_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}}\end{bmatrix}^\top$$

then the approximation can be expressed as

$$\left[\widehat{\exp(\mathbf{P}\mathbf{C}^\top)}^*\right]_i = \sum_{\mathbf{b}_{s,x}\in\mathbb{J}} c_{s,x}\mathbf{b}_{s,x} = \mathbf{B}\mathbf{c}$$

If we solve for

$$\mathbf{c} := \arg\min_{\mathcal{F}} \left\|\left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i - \mathbf{B}\mathcal{F}\right\|$$

then

$$\mathbf{c} = (\mathbf{B}^\top \mathbf{B})^{-1} \mathbf{B}^\top \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i$$

Due to the restriction (6.11), the columns of $\mathbf{B}$ are orthogonal, so $\mathbf{B}^\top \mathbf{B}$ is a diagonal matrix:

$$\mathbf{B}^\top \mathbf{B} = \begin{bmatrix} 1/s_1 & & & \\ & 1/s_2 & & \\ & & \cdots & \\ & & & 1/s_{|\mathbb{J}|} \end{bmatrix}$$

We can also write down $\mathbf{B}^\top \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i$

$$\mathbf{B}^\top \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i = \begin{bmatrix} \langle \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i, \mathbf{b}_{s_1,x_1} \rangle \\ \langle \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i, \mathbf{b}_{s_2,x_2} \rangle \\ \cdots \\ \langle \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i, \mathbf{b}_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}} \rangle \end{bmatrix}$$

Putting everything together, we have

$$\mathbf{B}\mathbf{c} = \begin{bmatrix} s_1 \mathbf{b}_{s_1,x_1} & s_2 \mathbf{b}_{s_2,x_2} & \cdots & s_{|\mathbb{J}|} \mathbf{b}_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}} \end{bmatrix} \begin{bmatrix} \langle \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i, \mathbf{b}_{s_1,x_1} \rangle \\ \langle \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i, \mathbf{b}_{s_2,x_2} \rangle \\ \cdots \\ \langle \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i, \mathbf{b}_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}} \rangle \end{bmatrix} \tag{6.13}$$

We note that $s\mathbf{b}_{s,x}$ simply re-scales the entry of $\mathbf{b}_{s,x}$ such that any non-zero entry becomes 1. Then, let us consider the j-th entry of $\mathbf{B}\mathbf{c}$. Due to the restriction (6.11), we have exactly one $\mathbf{b}_{s,x} \in \mathbb{J}$ whose support region contains j, so the j-th row of the first matrix on the right hand side of (6.13) contains exactly one 1 and the remaining entries are 0. Therefore, we have

$$\left[\widehat{\exp(\mathbf{P}\mathbf{C}^\top)}^*\right]_{i,j} = [\mathbf{B}\mathbf{c}]_j = \langle \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i, \mathbf{b}_{s,x} \rangle$$

where $\mathbf{b}_{s,x} \in \mathbb{J}$ is the component that is supported on j, which concludes our proof.

□

6.3.1.3 Efficient Approximation

Note that computing (6.12) for all j would require access to the entire $\left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i$. We exploit the same strategy as described in Chapter 5, so the exponential of the inner product is used as an approximation to the inner product of the exponential.

$$\left[\widehat{\exp(\mathbf{P}\mathbf{C}^\top)}\right]_{i,j} := \exp(\langle\left[\mathbf{P}\mathbf{C}^\top\right]_i, \mathbf{b}_{s,x}\rangle) \tag{6.14}$$

We note that $\langle\left[\mathbf{P}\mathbf{C}^\top\right]_i, \mathbf{b}_{s,x}\rangle$ is the local average of the support region of $\mathbf{b}_{s,x}$, which is also the x-th s-length segment of sequence $\left[\mathbf{P}\mathbf{C}^\top\right]_i$:

$$\langle[\mathbf{P}\mathbf{C}^\top]_i, \mathbf{b}_{s,x}\rangle = \frac{1}{s}\sum_{[\mathbf{b}_{s,x}]_j \neq 0}\left[\mathbf{P}\mathbf{C}^\top\right]_{i,j}$$

By using some arithmetic manipulations, (6.14) can be efficiently computed

$$\begin{aligned}\exp(\langle[\mathbf{P}\mathbf{C}^\top]_i, \mathbf{b}_{s,x}\rangle) &= \exp(\frac{1}{s}\sum_{[\mathbf{b}_{s,x}]_j \neq 0}\left[\mathbf{P}\mathbf{C}^\top\right]_{i,j}) = \exp(\frac{1}{s}\sum_{(\mathbf{b}_{s,x})_j \neq 0}\langle[\mathbf{P}]_i, [\mathbf{C}]_j\rangle) \\ &= \exp(\langle[\mathbf{P}]_i, \frac{1}{s}\sum_{[\mathbf{b}_{s,x}]_j \neq 0}[\mathbf{C}]_j\rangle) = \exp(\langle[\mathbf{P}]_i, \mathbf{c}_{s,x}\rangle)\end{aligned} \tag{6.15}$$

where $\mathbf{c}_{s,x}$ is defined in (6.7):

$$\mathbf{c}_{s,x} := \frac{1}{s}\sum_{sx-s<i\leqslant sx}[\mathbf{C}]_i = \mathbf{b}_{s,x}\mathbf{C} \tag{6.16}$$

We note that $\mathbf{c}_{s,x}$ is the local average of the x-th s-length segment of sequence $\mathbf{C}$. The $\mathbf{c}_{s,x}$ can be efficiently computed via

$$\mathbf{c}_x^{2s} = \frac{1}{2}\mathbf{c}_{2x-1}^s + \frac{1}{2}\mathbf{c}_{2x}^s \qquad \mathbf{c}_x^1 = [\mathbf{C}]_x \tag{6.17}$$

Claim 6.2. *Given the set $\mathbb{J}$ satisfying restriction (6.11), let $\mathbf{S}_c$ be be a matrix whose rows are elements $\mathbf{b}_{s,x} \in \mathbb{J}$*

$$\mathbf{S}_c = \begin{bmatrix} \mathbf{b}_{s_1,x_1} \\ \mathbf{b}_{s_2,x_2} \\ \cdots \\ \mathbf{b}_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}} \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} s_1 & & & \\ & s_2 & & \\ & & \cdots & \\ & & & s_{|\mathbb{J}|} \end{bmatrix}$$

Then,

$$\exp(\mathbf{P}\mathbf{C}^\top \mathbf{S}_c^\top)\mathbf{D}\mathbf{S}_c = \widehat{\exp(\mathbf{P}\mathbf{C}^\top)}$$

where $\widehat{\exp(\mathbf{P}\mathbf{C}^\top)}$ is defined as (6.14).

Proof. Consider the i-th row of $\exp(\mathbf{P}\mathbf{C}^\top \mathbf{S}_c^\top)$,

$$\begin{aligned} \left[\exp(\mathbf{P}(\mathbf{S}_c\mathbf{C})^\top)\right]_i &= \exp([\mathbf{P}]_i\,(\mathbf{S}_c\mathbf{C})^\top) \\ &= \exp([\mathbf{P}]_i \begin{bmatrix} \mathbf{c}_{s_1,x_1} & \mathbf{c}_{s_2,x_2} & \cdots & \mathbf{c}_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}} \end{bmatrix}) \\ &= \begin{bmatrix} \exp(\langle [\mathbf{P}]_i\,, \mathbf{c}_{s_1,x_1}\rangle) & \cdots & \exp(\langle [\mathbf{P}]_i\,, \mathbf{c}_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}}\rangle) \end{bmatrix} \end{aligned}$$

Then, we have

$$\left[\exp(\mathbf{P}(\mathbf{S}_c\mathbf{C})^\top)\mathbf{D}\mathbf{S}_c\right]_i = \begin{bmatrix} \exp(\langle [\mathbf{P}]_i\,, \mathbf{c}_{s_1,x_1}\rangle) & \cdots & \exp(\langle [\mathbf{P}]_i\,, \mathbf{c}_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}}\rangle) \end{bmatrix} \begin{bmatrix} s_1\mathbf{b}_{s_1,x_1} \\ \cdots \\ s_{|\mathbb{J}|}\mathbf{b}_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}} \end{bmatrix} \tag{6.18}$$

We note that $s\mathbf{b}_{s,x}$ simply re-scales the entry of $\mathbf{b}_{s,x}$ such that any non-zero entry becomes 1. Then, let us consider j-th entry of $\left[\exp(\mathbf{P}\mathbf{C}^\top \mathbf{S}_c^\top)\mathbf{D}\mathbf{S}_c\right]_i$. Due to the restriction (6.11), we have exactly one $\mathbf{b}_{s,x} \in \mathbb{J}$ whose support region contains j, so the j-th column of the second matrix in the right hand side of (6.18) contains

exactly one 1 and the remaining entries are 0. Therefore, we have

$$\left[\exp(\mathbf{P}(\mathbf{S}_c\mathbf{C})^\top)\mathbf{D}\mathbf{S}_c\right]_{i,j} = \exp(\langle[\mathbf{P}]_i, \mathbf{c}_{s,x}\rangle) = \exp(\langle[\mathbf{P}\mathbf{C}^\top]_i, \mathbf{b}_{s,x}\rangle) = \left[\widehat{\exp(\mathbf{P}\mathbf{C}^\top)}\right]_{i,j}$$

where $\mathbf{b}_{s,x} \in \mathbb{J}$ is the component that is supported on j. The second equality is based on (6.15).

□

Claim 6.3. *If $\mathbf{S}_c$ and $\mathbf{D}$ are defined as Claim 6.2, the pseudo inverse of $\mathbf{S}_c$ is simply $\mathbf{S}_c^\dagger = \mathbf{S}_c^\top\mathbf{D}$, so each row of $\mathbf{S}_c^\dagger$ and $\mathbf{S}^\dagger$ contain exactly one 1 (so the number of nonzero entries of $\mathbf{S}_c^\dagger$ and $\mathbf{S}^\dagger$ are N_c and n respectively).*

Proof. Since each row of $\mathbf{S}_c$ is some $\mathbf{b}_{s,x} \in \mathbb{J}$, due to the restriction (6.11), for $i \neq j$,

$$\left[\mathbf{S}_c\mathbf{S}_c^\top\mathbf{D}\right]_{i,i} = \langle[\mathbf{S}_c]_i, [\mathbf{D}\mathbf{S}_c]_i\rangle = \langle\mathbf{b}_{s_i,x_i}, s_i\mathbf{b}_{s_i,x_i}\rangle = s_i\frac{1}{s_i} = 1$$
$$\left[\mathbf{S}_c\mathbf{S}_c^\top\mathbf{D}\right]_{i,j} = \langle[\mathbf{S}_c]_i, [\mathbf{D}\mathbf{S}_c]_j\rangle = \langle\mathbf{b}_{s_i,x_i}, s_j\mathbf{b}_{s_j,x_i}\rangle = s_j 0 = 0$$

As a result, $\mathbf{S}_c\mathbf{S}_c^\top\mathbf{D} = \mathbf{I}$. Further, $\mathbf{S}_c^\top\mathbf{D}\mathbf{S}_c$ is a symmetric matrix. So, all Moore-Penrose conditions are verified. $\mathbf{S}_c^\dagger = \mathbf{S}_c^\top\mathbf{D}$.

From the restriction (6.11), we have every column of $\mathbf{S}_c$ contains exactly a non-zero entry. Also,

$$\mathbf{S}_c^\dagger = \mathbf{S}_c^\top\mathbf{D} = \begin{bmatrix} s_1\mathbf{b}_{s_1,x_1} & s_2\mathbf{b}_{s_2,x_2} & \cdots & s_{|\mathbb{J}|}\mathbf{b}_{s_{|\mathbb{J}|},x_{|\mathbb{J}|}} \end{bmatrix}$$

Since the non-zero entry of $\mathbf{b}_{s,x}$ is simply $\frac{1}{s}$ by definition, $s\mathbf{b}_{s,x}$ simply re-scales the entry of $\mathbf{b}_{s,x}$ such that any non-zero entry becomes 1. As a result, each row of $\mathbf{S}_c^\dagger$ has exactly one 1. Also, by the relation between $\mathbf{S}$ and $\mathbf{S}_c$:

$$\mathbf{S} = \begin{bmatrix} \mathbf{I}_{N_p} & 0 \\ 0 & \mathbf{S}_c \end{bmatrix} \qquad \mathbf{S}^\dagger = \begin{bmatrix} \mathbf{I}_{N_p} & 0 \\ 0 & \mathbf{S}_c^\dagger \end{bmatrix}$$

each row of $\mathbf{S}^\dagger$ has exactly one 1.

□

At the end, the approximation

$$\widehat{\exp(\mathbf{P}\mathbf{C}^\top)} = \exp(\mathbf{P}\mathbf{C}^\top\mathbf{S}_c^\top)\mathbf{D}\mathbf{S}_c \approx \exp(\mathbf{P}\mathbf{C}^\top) \tag{6.19}$$

does not look exactly as (6.8), but we can insert a simple diagonal matrix $\mathbf{D}$ to the formulation (6.8) and make the whole thing work.

6.3.1.4 How to Construct $\mathbb{J}$ for $\mathbf{S}_c$ and $\mathbf{S}_c\mathbf{C}$?

The derivation so far assumes access to $\mathbb{J}$, but in practice, we have no knowledge of $\mathbb{J}$ and need to construct $\mathbb{J}$ that leads to good approximation. With the approximation scheme in place, we can now analyze the approximation error, which will be leveraged later to find a reasonable set of components $\mathbb{J}$. The approximation error of i-th row of $\exp(\mathbf{P}\mathbf{C}^\top)$ can be expressed as

$$\begin{aligned}
\mathcal{E}_i &:= \left\| \left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_i - \left[\widehat{\exp(\mathbf{P}\mathbf{C}^\top)}\right]_i \right\|_F^2 \\
&= \sum_{j=1}^{N_c} \left(\left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_{i,j} - \left[\widehat{\exp(\mathbf{P}\mathbf{C}^\top)}\right]_{i,j} \right)^2 \\
&= \sum_{\mathbf{b}_{s,x}\in\mathbb{J}} \sum_{[\mathbf{b}_{s,x}]_j\neq 0} \left(\left[\exp(\mathbf{P}\mathbf{C}^\top)\right]_{i,j} - \exp(\langle \left[\mathbf{P}\mathbf{C}^\top\right]_i, \mathbf{b}_{s,x}\rangle) \right)^2 \\
&= \sum_{\mathbf{b}_{s,x}\in\mathbb{J}} \exp(\langle \left[\mathbf{P}\mathbf{C}^\top\right]_i, \mathbf{b}_{s,x}\rangle) \sum_{(\mathbf{b}_{s,x})_j\neq 0} \left(\exp(\left[\mathbf{P}\mathbf{C}^\top\right]_{i,j} - \langle \left[\mathbf{P}\mathbf{C}^\top\right]_i, \mathbf{b}_{s,x}\rangle) - 1\right)^2 \\
&= \sum_{\mathbf{b}_{s,x}\in\mathbb{J}} \exp(\langle [\mathbf{P}]_i, \mathbf{c}_{s,x}\rangle) \sum_{(\mathbf{b}_{s,x})_j\neq 0} \left(\exp(\langle [\mathbf{P}]_i, [\mathbf{C}]_j\rangle - \langle [\mathbf{P}]_i, \mathbf{c}_{s,x}\rangle) - 1\right)^2 \\
&= \sum_{\mathbf{b}_{s,x}\in\mathbb{J}} \exp(\langle [\mathbf{P}]_i, \mathbf{c}_{s,x}\rangle) \sum_{(\mathbf{b}_{s,x})_j\neq 0} \left(\exp(\langle [\mathbf{P}]_i, [\mathbf{C}]_j - \mathbf{c}_{s,x}\rangle) - 1\right)^2
\end{aligned}$$

The second equality and fourth equality are due to (6.11) and (6.15). The approximation error is governed by two components multiplied together: attention score between $[\mathbf{P}]_i$ and the local average $\mathbf{c}_{s,x}$ of the x-th s-length segment of sequence $\mathbf{C}$ and the inner product of $[\mathbf{P}]_i$ with the amount of deviation of $[\mathbf{C}]_j$ from its local

average $\mathbf{c}_{s,x}$. When $s = 1$, the deviation is simply zero:

$$\mathbf{c}_x^1 = [\mathbf{C}]_x .$$

It is reasonable to assume that the deviation $[\mathbf{C}]_j - \mathbf{c}_{s,x}$ is smaller if s is smaller.

Note that this approximation error is similar to the error of MRA-Attention discussed in Chapter 5, so a similar algorithm for constructing $\mathbb{J}$ can be derived following the algorithm discussed in Chapter 5: when $\exp(\langle [\mathbf{P}]_i , \mathbf{c}_{s,x} \rangle)$ is large, we should approximate the x-th s-length segment of $\mathbf{C}$ with higher resolution (by splitting the segment to shorter sub-segments and using finer approximation). This heuristic describes the selection criteria for one row of $\exp(\mathbf{P}\mathbf{C}^\top)$, which corresponds to a single VIP-token, for multiple rows of $\exp(\mathbf{P}\mathbf{C}^\top)$ (for multiple VIP-tokens), we simply use

$$\mu_{s,x} = \sum_{i=1}^{n_p} \exp(\langle [\mathbf{P}]_i , \mathbf{c}_{s,x} \rangle) \tag{6.20}$$

as selection criteria since $\mathbb{J}$ is shared by all VIP-tokens.

The construction of $\mathbb{J}$ is described in Algorithm 3. This algorithm describes the same procedure as the Figure 6.3. The $\mathbf{b}_{s,x}$'s in $\mathbb{J}$ are the rows of $\mathbf{S}_c$, and the corresponding $\mathbf{c}_{s,x}$'s (6.16) are the rows of $\mathbf{S}_c\mathbf{C}$. The budgets $h_2, h_4, \cdots, h_{N_c}$ required by Algorithm 3 is used determine the number of components at each resolution that will be added to $\mathbb{J}$. Specifically, there are $2h_{2s} - h_s$ number of components $\mathbf{b}_x^s$ for $s \neq 1$ based on simple calculations. We can choose budgets such that the final size of $\mathbb{J}$ is $R - N_p$ to make the length of compressed sequence to be R.

6.3.1.5 How Good is This Approximation?

At high level, the compression $\mathbf{S}_c$ performs more compression on tokens that are not relevant to the VIP-tokens and less compression to tokens that are important to the VIP-tokens. We will now discuss it in more detail. Since each row of $S^\dagger$ contain exactly one 1 as stated in Claim 6.3, $S^\dagger$ can commute with $\mathcal{F}$, so in summary, we

Algorithm 3 Constructing $\mathbb{J}$

Input: VIP-tokens $\mathbf{P}$ and $\mathbf{c}_x^s$ for all s and x
Input: h_s: number of s-length segments to refine for each $s \in \{2, 4, \cdots, n_c\}$
Initialize empty $\mathbb{J}$
Compute $\mu_1^{n_c}$ (6.20) and add $\mathbf{b}_1^{n_c}$ (6.9) to $\mathbb{J}$ (compute root node)
for $s \leftarrow n_c, n_c/2, \cdots, 2$ **do**
  Pop h_s elements $\mathbf{b}_x^s$ with the largest μ_x^s (6.20) (select nodes with higher attention scores)
  for each $\mathbf{b}_x^s$ **do**
    Compute $\mu_{2x-1}^{s/2}, \mu_{2x}^{s/2}$ (6.20) and add $\mathbf{b}_{2x-1}^{s/2}, \mathbf{b}_{2x}^{s/2}$ (6.9) to $\mathbb{J}$ (split selected nodes)
  end for
end for
Output: $\mathbb{J}$

can write the approximation of the computation of a Transformer layer as

$$\begin{aligned}
\mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX}) &= \exp(\mathbf{P}\mathbf{P}^\top)\mathbf{P} + \exp(\mathbf{P}\mathbf{C}^\top\mathbf{S}_c^\top)\mathbf{D}\mathbf{S}_c\mathbf{C} \\
\mathbf{S}_c^\dagger\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) &= \mathbf{S}_c^\dagger \exp(\mathbf{S}_c\mathbf{C}\mathbf{P}^\top)\mathbf{P} + \mathbf{S}_c^\dagger \exp(\mathbf{S}_c\mathbf{C}\mathbf{C}^\top\mathbf{S}_c^\top)\mathbf{D}\mathbf{S}_c\mathbf{C} \\
\begin{bmatrix} \mathbf{P}_{new} \\ \mathbf{C}_{new} \end{bmatrix} &= \begin{bmatrix} \mathcal{F}(\mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX}) + \mathbf{P}) + \mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX}) \\ \mathcal{F}(\mathbf{S}_c^\dagger\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c^\dagger\mathbf{S}_c\mathbf{C}) + \mathbf{S}_c^\dagger\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) \end{bmatrix} + \begin{bmatrix} \mathbf{P} \\ \mathbf{C} \end{bmatrix}
\end{aligned} \tag{6.21}$$

Note that $\mathbf{D}$ is added as discussed in (6.19).

There are four main approximation components (purple) in (6.21). Taking the fact that $\mathbf{D}\mathbf{S}_c = (\mathbf{S}_c^\dagger)^\top$, all of these approximations are row or column space multi-resolution approximations governed by the $\mathbf{S}_c$ matrix. High attention weight implies higher dependency, and the procedure in Algorithm 3 refines regions with large attention weights with higher resolutions. Therefore, the token embedding in $\mathbf{C}$ that have a higher dependency with $\mathbf{P}$ are better approximated. The output $\mathbf{P}_{new}$ is well approximated by design since the approximation preserves the higher frequency components of the subset of rows of $\mathbf{C}$ that has a high impact on the output $\mathbf{P}_{new}$. Further, the output in $\mathbf{C}_{new}$ corresponding to the subset of rows of $\mathbf{C}$ that have a higher dependency with the VIP-tokens will have a better approximation than the remaining rows of $\mathbf{C}$. This property addresses the issue that some tokens

with unknown locations are also relevant to the final prediction of a Transformer in some tasks. For example, in question answering tasks, candidate answers are usually expected to have a large dependency with question tokens (VIP-tokens), so they are approximated well as well. This approximation property is exactly what we need.

6.4 Efficient Data Structure for De/compression

By employing the procedure in §6.3 illustrated in Figure 6.3, we can find the compressed $\mathbf{S}_c\mathbf{C}$ with an $\mathcal{O}(N_c D + RN_p D)$ cost at each layer. The main cost $\mathcal{O}(N_c D)$ is due to computing $\mathbf{c}_{s,x}$ defined in (6.7) for all resolution s and location x by using recursive relation from the bottom up:

$$\mathbf{c}_{2s,x} = \frac{1}{2}\mathbf{c}_{s,2x-1} + \frac{1}{2}\mathbf{c}_{s,2x} \qquad \mathbf{c}_{1,x} = [\mathbf{C}]_x \tag{6.22}$$

We find that these steps could introduce a large overhead. Further, note that if we decompress (apply $\mathbf{S}^\dagger$ to) the output of $\mathcal{G}$ for the compressed sequence as in (6.3), the cost is $\mathcal{O}(ND)$ since the number of nonzero entries in $\mathbf{S}^\dagger$ is n (more details in §6.4.2). As a solution, we now introduce a data structure $\mathcal{T}(\cdot)$ for storing $\mathbf{C}$ and $\mathbf{C}_{new}$, as shown in Figure 6.6, which enables efficient computation of $\mathbf{c}_{s,x}$ and eliminates explicit decompression. We note that this data structure is only possible due to the specific structure of $\mathbf{S}_c\mathbf{C}$ constructed in §6.3. Specifically, $\mathcal{T}(\mathbf{C})$ stores $\mathbf{c}_1^{N_c}$ and $\Delta\mathbf{c}_{s,x}$ defined as

$$\Delta\mathbf{c}_{s,x} := \mathbf{c}_{2s,\lceil x/2 \rceil} - \mathbf{c}_{s,x} \tag{6.23}$$

for every resolution $s \neq N_c$ and location x. Similarly, $\mathcal{T}(\mathbf{C}_{new})$ stores $\mathbf{c}^{new}_{N_c,1}$ and $\Delta\mathbf{c}^{new}_{s,x}$ where $\mathbf{c}^{new}_{s,x}$ and $\Delta\mathbf{c}^{new}_{s,x}$ are defined similar to (6.7) and (6.23) but using $\mathbf{C}_{new}$ instead of $\mathbf{C}$.

Then, given $\mathcal{T}(\mathbf{C})$, any $\mathbf{c}_{s,x}$ can be retrieved efficiently in $\mathcal{O}(\log(N_c)D)$ cost via

recursion:

$$\mathbf{c}_{s,x} = \mathbf{c}_{2s,\lceil x/2 \rceil} - \Delta\mathbf{c}_{s,x} = \mathbf{c}_{4s,\lceil x/4 \rceil} - \Delta\mathbf{c}_{2s,\lceil x/2 \rceil} - \Delta\mathbf{c}_{s,x} = \cdots \tag{6.24}$$

The only reason we need to decompress $\mathbf{S}^\dagger$ is that we need to obtain new representation $\mathbf{C}_{new}$ (no decompression is needed for $\mathbf{P}_{new}$ since $\mathbf{P}$ is uncompressed). Suppose we have $\mathcal{T}(\mathbf{C}_{new})$, then we have an alternative way of getting $\mathbf{C}_{new}$ similar to (6.24) (note that $\mathbf{c}^{new}_{1,x} = [\mathbf{C}_{new}]_x$) without explicit decompression. The key benefit of this data structure is that we can obtain $\mathcal{T}(\mathbf{C}_{new})$ by changing some nodes in $\mathcal{T}(\mathbf{C})$. This only needs updating $\mathcal{O}(R)$ nodes, and each update takes $\mathcal{O}(D)$ cost.

An example. We show a $\mathcal{T}(\mathbf{C})$ for $N_c = 8$ in Figure 6.6. Let

$$\mathbf{S}_c\mathbf{C} = \begin{bmatrix} \mathbf{c}_{2,1} & \mathbf{c}_{1,3} & \mathbf{c}_{1,4} & \mathbf{c}_{4,2} \end{bmatrix}^\top$$

as in Figure 6.3. Since the segment $\mathbf{c}_{2,1}$ is not split into sub-segments $\mathbf{c}_{1,1}$ and $\mathbf{c}_{1,2}$, we have (details in §6.4.1):

$$\mathbf{c}^{new}_{1,1} - \mathbf{c}_{1,1} = \mathbf{c}^{new}_{1,2} - \mathbf{c}_{1,2} = \mathbf{c}^{new}_{2,1} - \mathbf{c}_{2,1} \tag{6.25}$$

By rearranging (6.25), we can verify that $\Delta\mathbf{c}^{new}_{1,1}, \Delta\mathbf{c}^{new}_{1,2}$ in $\mathcal{T}(\mathbf{C}_{new})$ stays the same as $\Delta\mathbf{c}_{1,1}, \Delta\mathbf{c}_{1,2}$ in $\mathcal{T}(\mathbf{C})$ and thus do not need to be updated:

$$\begin{aligned}
\Delta\mathbf{c}^{new}_{1,1} &= \mathbf{c}^{new}_{2,1} - \mathbf{c}^{new}_{1,1} = \mathbf{c}_{2,1} - \mathbf{c}_{1,1} = \Delta\mathbf{c}_{1,1} \\
\Delta\mathbf{c}^{new}_{1,2} &= \mathbf{c}^{new}_{2,1} - \mathbf{c}^{new}_{1,2} = \mathbf{c}_{2,1} - \mathbf{c}_{1,2} = \Delta\mathbf{c}_{1,2}
\end{aligned}$$

Further, we can verify that only the green nodes in Figure 6.6 will be updated. These nodes correspond to the nodes in Figure 6.3 that have been traversed. In summary, for each row $\mathbf{c}^s_x$ of $\mathbf{S}_c\mathbf{C}$ (a leaf node in Figure 6.3), only the node storing $\Delta(\mathbf{c})^s_x$ and its ancestor nodes in $\mathcal{T}(\mathbf{C})$ must be updated, so the total number of nodes (including their ancestors) being updated is $\mathcal{O}(r)$. Next, we can update the nodes as follows: first, we get representations $\mathbf{c}^{new}_{2,1}, \mathbf{c}^{new}_{1,3}, \mathbf{c}^{new}_{1,4}, \mathbf{c}^{new}_{4,2}$ by feeding $\mathbf{S}_c\mathbf{C}$ into Transformer layer (details in §6.4.1). At level $s = 1$, given $\mathbf{c}^{new}_{1,3}$ and $\mathbf{c}^{new}_{1,4}$, we **(1)**

compute $\mathbf{c}_{2,2}^{new}$ via (6.22), and then **(2)** compute $\Delta\mathbf{c}_{1,3}^{new}$ and $\Delta\mathbf{c}_{1,4}^{new}$ via (6.23). The last two values are the new values for $\Delta\mathbf{c}_{1,3}$ and $\Delta\mathbf{c}_{1,4}$ in $\mathcal{T}(\mathbf{C})$. At level $s = 2$, given $\mathbf{c}_{1,2}^{new}$ and $\mathbf{c}_{2,2}^{new}$ computed at previous level, we apply similar procedure to obtain $\mathbf{c}_{4,1}^{new}, \Delta\mathbf{c}_{1,2}^{new}, \Delta\mathbf{c}_{2,2}^{new}$, and the last two values are used to update two nodes in $\mathcal{T}(\mathbf{C})$. It becomes apparent that each node update takes $\mathcal{O}(D)$ cost. Putting it together: the complexity of modifying $\mathcal{T}(\mathbf{C})$ to $\mathcal{T}(\mathbf{C}_{new})$ is $\mathcal{O}(RD)$.

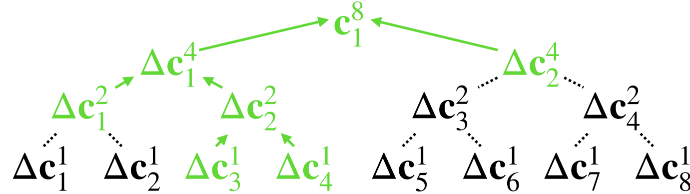

Figure 6.6: Proposed data structure $\mathcal{T}(\mathbf{C})$

By maintaining this data structure, we never need to materialize the entire $\mathbf{C}$ or $\mathbf{C}_{new}$ in any intermediate layer, but instead we use (6.24) to construct the rows of $\mathbf{S}_c\mathbf{C}$ and perform updates to $\mathcal{T}(\mathbf{C})$ to obtain $\mathbf{C}_{new}$ (represented as $\mathcal{T}(\mathbf{C}_{new})$) at each intermediate layer. At the output of a Transformer, $\mathbf{C}_{new}$ is materialized from $\mathcal{T}(\mathbf{C}_{new})$ at a $\mathcal{O}(N_c D)$ cost via the recursion (6.24) from the bottom up.

6.4.1 Details of Efficient Data Structure

In this subsection, we describe some technical details of the data structure $\mathcal{T}(\cdot)$.

6.4.1.1 Why Equation (6.25) holds?

Claim 6.4. *Given the set $\mathbb{J}$ satisfying restriction (6.11), if $\mathbf{b}_{s,x} \in \mathbb{J}$, then $\mathbf{c}_{s,x}^{new} - \mathbf{c}_{s,x} = \mathbf{c}_{s',x'}^{new} - \mathbf{c}_{s',x'}$ for all $\mathbf{b}_{s',x'}$ satisfying the support of $\mathbf{b}_{s',x'}$ is contained in the support of $\mathbf{b}_{s,x}$ (the x'-th s'-length segment of $\mathbf{C}$ is a sub-segment of the x-th s-length segment of $\mathbf{C}$).*

Proof. To simplify the notations a bit, without loss of generality, we assume $x = 1$.

Then, for $i \leqslant s$, consider $\mathbf{c}_{1,i}^{new}$:

$$\begin{aligned}\mathbf{c}_{1,i}^{new} = [\mathbf{C}_{new}]_i &= \left[\mathbf{S}_c^{\dagger}\mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}) + \mathbf{S}_c^{\dagger}\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX})\right]_i + [\mathbf{C}]_i \\ &= \left[\mathbf{S}_c^{\dagger}\right]_i \mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}) + \left[\mathbf{S}_c^{\dagger}\right]_i \mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{c}_{1,i}\end{aligned}$$

By Claim 6.3, $\mathbf{S}_c^{\dagger} = \mathbf{S}_c^{\top}\mathbf{D}$ and i-th row of $\mathbf{S}_c^{\dagger}$ contains exactly a 1. The column that contains 1 in the i-th row of $\mathbf{S}_c^{\dagger}$ is exactly $s\mathbf{b}_1^s$ since i is contained in the support of exactly one components in $\mathbb{J}$ due to the restriction (6.11). Denote this column index as j, then

$$\mathbf{c}_{1,i}^{new} = [\mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C})]_j + [\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX})]_j + \mathbf{c}_{1,i}$$

Note that this holds for all $i \leqslant s$. As a result, for $i, i' \leqslant s$,

$$\mathbf{c}_{1,i}^{new} - \mathbf{c}_{1,i} = [\mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C})]_j + [\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX})]_j = \mathbf{c}_{1,i'}^{new} - \mathbf{c}_{1,i'}$$

Then,

$$\begin{aligned}\mathbf{c}_{2,\lceil i/2\rceil}^{new} - \mathbf{c}_{2,\lceil i/2\rceil} &= \frac{1}{2}\mathbf{c}_{1,2\lceil i/2\rceil-1}^{new} + \frac{1}{2}\mathbf{c}_{1,2\lceil i/2\rceil}^{new} - \frac{1}{2}\mathbf{c}_{1,2\lceil i/2\rceil-1} - \frac{1}{2}\mathbf{c}_{1,2\lceil i/2\rceil} \\ &= \frac{1}{2}(\mathbf{c}_{1,2\lceil i/2\rceil-1}^{new} - \mathbf{c}_{1,2\lceil i/2\rceil-1}) + \frac{1}{2}(\mathbf{c}_{1,2\lceil i/2\rceil}^{new} - \mathbf{c}_{1,2\lceil i/2\rceil}) \\ &= \mathbf{c}_{1,2\lceil i/2\rceil-1}^{new} - \mathbf{c}_{1,2\lceil i/2\rceil-1}\end{aligned}$$

The rest follows from induction. This shows why Equation (6.25) holds.

□

6.4.1.2 How do we get $\mathbf{c}_{2,1}^{new}, \mathbf{c}_{1,3}^{new}, \mathbf{c}_{1,4}^{new}, \mathbf{c}_{4,2}^{new}$ if $\mathbf{S}_c\mathbf{C} = \begin{bmatrix}\mathbf{c}_{2,1} & \mathbf{c}_{1,3} & \mathbf{c}_{1,4} & \mathbf{c}_{4,2}\end{bmatrix}^{\top}$?

Claim 6.5. *We have*

$$\mathbf{S}_c\mathbf{C}_{new} = \mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}) + \mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}.$$

And the updated representation $\mathbf{c}_{s,x}^{new}$ *of the corresponding* $\mathbf{c}_{s,x}$ *(a row of* $\mathbf{S}_c\mathbf{C}$*) is the corresponding row of* $\mathbf{S}_c\mathbf{C}_{new}$*.*

Proof. By definition,

$$\mathbf{C}_{new} = \mathbf{S}_c^\dagger \mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}) + \mathbf{S}_c^\dagger \mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{C}$$

Then,

$$\begin{aligned}\mathbf{S}_c\mathbf{C}_{new} &= \mathbf{S}_c\mathbf{S}_c^\dagger \mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}) + \mathbf{S}_c\mathbf{S}_c^\dagger \mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C} \\ &= \mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}) + \mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}\end{aligned}$$

Since the $\mathbf{S}_c$ is the same for $\mathbf{S}_c\mathbf{C}_{new}$ and $\mathbf{S}_c\mathbf{C}$, the second statement follows. □

6.4.1.3 Algorithm for Modifying $\mathcal{T}(\mathbf{C})$ into $\mathcal{T}(\mathbf{C}_{new})$

In this section, we describe the exact algorithm to update $\mathcal{T}(\mathbf{C})$ into $\mathcal{T}(\mathbf{C}_{new})$. The pseudo code is described in Algorithm 4 where $\mathbf{c}_{s,x}$ is computed via

$$\mathbf{c}_{s,x} = \mathbf{c}_{2s,\lceil x/2 \rceil} - \Delta\mathbf{c}_{s,x} = \mathbf{c}_{4s,\lceil x/4 \rceil} - \Delta\mathbf{c}_{2s,\lceil x/2 \rceil} - \Delta\mathbf{c}_{s,x} = \cdots \tag{6.26}$$

We use the term "dirty" in Algorithm 4 to indicate the node needs to be handled due to node updates. This term is commonly used in computer cache implementations to indicate that the data of a specific location has been updated and needs to be accounted for.

6.4.2 Complexity Analysis

In this section, we will discuss the detailed complexity analysis of our proposed method. The overall complexity of our proposed method is $\mathcal{O}(LRD^2 + LR^2D + LR\log(N_c)D + LRN_pD + ND)$ when using the proposed efficient data structure.

Algorithm 4 Computation of one Transformer layer with $\mathcal{T}(\mathbf{C})$

Input: VIP-tokens $\mathbf{P}$ and data structure $\mathcal{T}(\mathbf{C})$
Use Algorithm 3 to construct $\mathbb{J}$ but use (6.26) to retrieve $\mathbf{c}_{s,x}$ from $\mathcal{T}(\mathbf{C})$
Construct $\mathbf{S}_c, \mathbf{S}_c\mathbf{C}$ associated with $\mathbb{J}$ using Claim 6.2
Compute

$$\begin{bmatrix} \mathbf{P}_{new} \\ \mathbf{S}_c\mathbf{C}_{new} \end{bmatrix} = \begin{bmatrix} \mathcal{F}(\mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX}) + \mathbf{P}) + \mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX}) + \mathbf{P} \\ \mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}) + \mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C} \end{bmatrix}$$

Set $\mathcal{T}(\mathbf{C}_{new}) \leftarrow \mathcal{T}(\mathbf{C})$
for $s \leftarrow 1, 2, 4, \cdots n_c/2$ **do**
  for $\mathbf{b}_{s,x} \in \mathbb{J}$ **do**
    Locate row location $\mathbf{b}_{s,x}$ in $\mathbf{S}_c$, refer the index as j
    Compute $\mathbf{c}_{s,x}^{new} = [\mathbf{S}_c\mathbf{C}_{new}]_j$
    Mark $\Delta\mathbf{c}_{s,x}^{new}$ dirty
  end for
  for dirty $\Delta\mathbf{c}_{s,x}^{new}$ **do**
    Compute $\mathbf{c}_{2s,\lceil x/2 \rceil}^{new}$ and update $\Delta\mathbf{c}_{s,x}^{new}$
    Mark $\Delta\mathbf{c}_{2s,\lceil x/2 \rceil}^{new}$ dirty
  end for
end for
Update $\mathbf{c}_{N_c,1}^{new}$
Output: VIP-tokens $\mathbf{P}_{new}$ and data structure $\mathcal{T}(\mathbf{C}_{new})$

6.4.2.1 Preparing Input Sequence to $\mathcal{T}(\mathbf{C})$: $\mathcal{O}(ND)$

At the first layer, we need to permute the rows of $\mathbf{X}$ into $[\mathbf{P}; \mathbf{C}]$, which takes $\mathcal{O}(ND)$ cost. Then, we process $\mathbf{C}$ into $\mathcal{T}(\mathbf{C})$. This requires **(1)** computing $\mathbf{c}_x^s$ defined in (6.17). $\mathbf{c}_x^1 = [\mathbf{C}]_x$, so no compute is needed. With all $\mathbf{c}_x^1$ given, computing all $\mathbf{c}_x^2$ takes $\mathcal{O}(N_cD/2)$. With all $\mathbf{c}_x^2$ given, computing all $\mathbf{c}_x^4$ takes $\mathcal{O}(N_cD/4)$... So, the cost is

$$\mathcal{O}(N_cD/2 + N_cD/4 + \cdots + D) = \mathcal{O}(N_cD).$$

Then **(2)** computing $\Delta\mathbf{c}_x^s$ for all s and x. Computing each $\Delta\mathbf{c}_x^s$ takes $\mathcal{O}(D)$ when given $\mathbf{c}_x^s$ and $\mathbf{c}_{\lceil x/2 \rceil}^{2s}$. The amount of cost is the same as the number of nodes in the tree $\mathcal{T}(\mathbf{C})$, so the cost is $\mathcal{O}(N_cD)$. Note that $N_c < N$, so the overall complexity of the above operations is $\mathcal{O}(ND)$.

6.4.2.2 Constructing $\mathbb{J}, \mathbf{S}_c, \mathbf{S}_c\mathbf{C}$: $\mathcal{O}(LR\log(N_c)D + LRN_pD)$

We can analyze the complexity of constructing $\mathbb{J}$ using Algorithm 3. There is only one possible $\mu_x^{N_c}$. Then for each s, there are $2h_s$ number of $\mu_x^{s/2}$ being computed since there are 2 components $\mathbf{b}_x^{s/2}$ for each $\mathbf{b}_{x'}^s$. As a result, we need to compute $\mathcal{O}(1 + \sum_s 2h_s)$ number of $\mu_x^{s/2}$. When $\mathbf{c}_x^{s/2}$ is given, the cost of computing a $\mu_x^{s/2}$ is $\mathcal{O}(N_pD)$, so the overall cost of constructing $\mathbb{J}$ is $\mathcal{O}((1 + \sum_s 2h_s)N_pD)$.

Further, at each s, the size of $\mathbb{J}$ is increased by h_s since h_s segments are split into $2h_s$ sub-segments, so the size of $\mathbb{J}$ is $\mathcal{O}(\sum_s h_s)$. Since $\mathbf{S}_c \in \mathbb{R}^{(R-N_p)\times N}$ and $|\mathbb{J}| = R - N_p$ as discussed in §6.3.1.4, $\mathcal{O}(R - N_p) = \mathcal{O}(\sum_s h_s)$. We use $\mathcal{O}(R)$ for simplicity instead of $\mathcal{O}(R - N_p)$. As a result, the overall cost of constructing $\mathbb{J}$ is $\mathcal{O}(RN_pD)$.

The above cost assumes $\mathbf{c}_x^{s/2}$ is given. If we compute all possible $\mathbf{c}_x^{s/2}$ using (6.17), the cost will be $\mathcal{O}(N_cD)$ as analyzed in §6.4.2.1. However, if we employ the proposed data structure, each $\mathbf{c}_x^{s/2}$ can be retrieved in at most $\mathcal{O}(\log(N_c)D)$ time by recursively computing (6.26). Since we need to retrieve $\mathcal{O}(1 + \sum_s 2h_s) = \mathcal{O}(R)$ number of $\mathbf{c}_x^s$, the complexity of computing necessary $\mathbf{c}_x^s$ is $\mathcal{O}(R\log(N_c)D)$.

As a result, the complexity of constructing $\mathbb{J}$ is $\mathcal{O}(RN_pD + R\log(N_c)D)$ at each layer. When summing the cost over all layers, the complexity is $\mathcal{O}(LRN_pD + LR\log(N_c)D)$.

By Claim 6.2, the rows of $\mathbf{S}_c$ and $\mathbf{S}_c\mathbf{C}$ are simply the $\mathbf{b}_x^s \in \mathbb{J}$ and the corresponding $\mathbf{c}_x^s$, which are already computed during the construction of $\mathbb{J}$, so we essentially can get these $\mathbf{S}_c$ and $\mathbf{S}_c\mathbf{C}$ for free.

6.4.2.3 Computation within a Transformer Layer: $\mathcal{O}(LRD^2 + LR^2D)$

At each layer, we need to compute

$$\begin{bmatrix} \mathbf{P}_{new} \\ \mathbf{S}_c\mathbf{C}_{new} \end{bmatrix} = \begin{bmatrix} \mathcal{F}(\mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX}) + \mathbf{P}) + \mathcal{A}(\mathbf{P}, \mathbf{SX}, \mathbf{SX}) + \mathbf{P} \\ \mathcal{F}(\mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C}) + \mathcal{A}(\mathbf{S}_c\mathbf{C}, \mathbf{SX}, \mathbf{SX}) + \mathbf{S}_c\mathbf{C} \end{bmatrix} \tag{6.27}$$

for updating $\mathcal{T}(\mathbf{C})$. This is the part of a Transformer layer that requires heavy computation. It can be verified that the complexity of a Transformer layer is $\mathcal{O}(ND^2+$

$N^2D)$ for a input sequence of length N. Now a compressed sequence of length R is fed into a Transformer layer, the cost is simply $\mathcal{O}(RD^2 + R^2D)$. We note that there is an additional re-scaling to plug $\mathbf{D}$ into $\exp(\mathbf{P}\mathbf{C}^\top\mathbf{S}_c^\top)\mathbf{D}\mathbf{S}_c$ during multi-head attention computation discussed in 6.19. However, the additional cost of applying $\mathbf{D}$ is $\mathcal{O}(RD)$, which does not change the complexity. When summing the cost of all layers, the overall complexity is $\mathcal{O}(LRD^2 + LR^2D)$.

6.4.2.4 Updating $\mathcal{T}(\mathbf{C})$ into $\mathcal{T}(\mathbf{C}_{new})$: $\mathcal{O}(LRD)$

Once (6.27) is computed, we need to change $\mathcal{T}(\mathbf{C})$ into $\mathcal{T}(\mathbf{C}_{new})$. The cost of changing $\mathcal{T}(\mathbf{C})$ into $\mathcal{T}(\mathbf{C}_{new})$ is $\mathcal{O}(RD)$ as analyzed in §6.4. For more specific analysis, let us take a look at the first three iterations:

(1) At the first iteration, there are $\mathcal{O}(2h_2)$ number of $(\mathbf{c}_{new})^1_x$ to be computed at the first inner for loop, and there are $\mathcal{O}(2h_2)$ number of $\Delta(\mathbf{c}_{new})^1_x$ to be updated in the second inner for loop. Additional $\mathcal{O}(h_2)$ number of $\Delta(\mathbf{c}_{new})^2_{\lceil x/2 \rceil}$ are masked dirty.

(2) At the second iteration, there are $\mathcal{O}(2h_4)$ number of $(\mathbf{c}_{new})^2_x$ to be computed at the first inner for loop, and there are $\mathcal{O}(2h_4 + h_2)$ number of $\Delta(\mathbf{c}_{new})^2_x$ to be updated in the second inner for loop. The second term is due to the dirty $\Delta(\mathbf{c}_{new})^2_{\lceil x/2 \rceil}$ from the first iteration. An additional $\mathcal{O}(h_4 + \frac{h_2}{2})$ number of $\Delta(\mathbf{c}_{new})^4_{\lceil x/2 \rceil}$ are masked dirty.

(3) At the third iteration, there are $\mathcal{O}(2h_8)$ number of $(\mathbf{c}_{new})^4_x$ to be computed at the first inner for loop, and there are $\mathcal{O}(2h_8 + h_4 + \frac{h_2}{2})$ number of $\Delta(\mathbf{c}_{new})^4_x$ to be updated in the second inner for loop. The second and third term is due to the dirty $\Delta(\mathbf{c}_{new})^4_{\lceil x/2 \rceil}$ from the second iteration. An additional $\mathcal{O}(h_8 + \frac{h_4}{2} + \frac{h_2}{4})$ number of $\Delta(\mathbf{c}_{new})^8_{\lceil x/2 \rceil}$ are masked dirty.

It becomes apparent that if we sum over the number of computes of $(\mathbf{c}_{new})^s_x$ and updates of $\Delta(\mathbf{c}_{new})^s_x$, the total number is $\mathcal{O}(\sum_s 2h_s + 2\sum_s \sum_{j=1}^{\log(s)} \frac{h_s}{2^j}) = \mathcal{O}(\sum_s h_s +$

$\sum_s h_s) = \mathcal{O}(R)$. Since each compute and update takes $\mathcal{O}(D)$ cost, the overall complexity of changing $\mathcal{T}(\mathbf{C})$ into $\mathcal{T}(\mathbf{C}_{new})$ is $\mathcal{O}(RD)$. When summing the cost of all layers, the overall complexity is $\mathcal{O}(LRD)$.

6.4.2.5 Materializing C_{new} from $\mathcal{T}(\mathbf{C}_{new})$ at the Last Layer: $\mathcal{O}(ND)$

At the output of the last layer, we can **(1)** compute all $\mathbf{c}^{new}_{N_c/2,x}$ via (6.26) paying a cost of $\mathcal{O}(2d)$, **(2)** compute $\mathbf{c}^{new}_{N_c/4,x}$ via (6.26) paying a cost of $\mathcal{O}(4D)$... until all $\mathbf{c}^{new}_{1,x}$ are computed. Then, $[\mathbf{C}_{new}]_x = c_{1,x}$ is materialized from $\mathcal{T}(\mathbf{C}_{new})$ for a total cost of

$$\mathcal{O}(D + 2D + 4D + \cdots + N_c D) = \mathcal{O}(N_c D).$$

Lastly, undoing the permutation so that $[\mathbf{P}_{new}; \mathbf{C}_{new}]$ are re-ordered to the original positions has a complexity of $\mathcal{O}(ND)$. As a result, the overall complexity is $\mathcal{O}(ND)$.

In summary, the overall complexity of our method is

$$\mathcal{O}(LRD^2 + LR^2D + LR\log(N_c)D + LRN_pD + ND)$$

6.5 Experiments

We performed a broad set of experiments to empirically evaluate the performance of our proposed compression. We evaluate our method on both encoder-only and encoder-decoder architecture types. We compare our method with baselines on a large list of question answering and summarization tasks, where we find that long sequences occur most frequently. Then, we study the model performance of scaling to ultra long sequences enabled by our method. While the major application of focus is in the context of NLP tasks, we also evaluate our method on non-language tasks.

For ease of implementation and hyperparameter selection, we restrict the rows of $\mathbf{S}_c\mathbf{C}$ to have exactly two resolutions for experiments. Specifically, for a pre-defined increment ratio k, we split and refine all segments $\mathbf{c}_{s,x}$ with $s > k$ to k-length

sub-segments, and select h (pre-defined) k-length segments to further split into 1-length sub-segments. So, the rows of $\mathbf{S}_c\mathbf{C}$ would consist of $(N_c/k - h)$ of $\mathbf{c}_{k,x}$ and hk of $\mathbf{c}_{1,x}$ for some x. To simplify the implementation, we only use the proposed compression in the encoder, and use the vanilla computation in the decoder of encoder-decoder models.

Further, we found a few layers of standard Transformer layers to pre-process tokens helps the performance. Therefore, in the initial stage of a Transformer, we segment input sequence into multiple 512-length segments. For each segment, we use vanilla computation in the first 4 layers (for base models and 6 layers for larger models) of a Transformer. Then, for the remaining layers, segments are concatenated back into one sequence and processed using our proposed compression. There is *no communication* among any segments, so the downstream tasks cannot be solved by these first 4 transformer layers alone, and the initial stage is used just for getting a reasonable representation for the compression to operate on.

6.5.1 Language Modeling

Approximation Quality of VIP-Tokens. We empirical measured the approximation quality of our VIP centric strategy compared to random strategy (the tree growth in Figure 6.3 is not guided by VIP-tokens, but is random) and lazy strategy (each k-length segment is compressed to a token). $\mathbf{P}_{new}$ is the approximated representation of VIP tokens computed with compression and let $\mathbf{P}^*_{new}$ be the ground truth representation of VIP tokens computed without compression. We measure the relative error (defined as $\|\mathbf{P}_{new} - \mathbf{P}^*_{new}\|_F / \|\mathbf{P}^*_{new}\|_F$) and correlation coefficient between $\mathbf{P}_{new}$ and $\mathbf{P}^*_{new}$. As shown in Table 6.1, we can verify that the proposed procedure indeed improves the approximation quality of VIP-tokens.

Compression	Error	Correlation
Random	0.403	0.919
Lazy	0.528	0.869
VIP centric	0.137	0.991

Table 6.1: Approximation quality.

Encoder-Only Models. For encoder-only architecture, we compare our method with RoBERTa (Liu et al., 2019a) and three strong baselines: Longformer (Beltagy et al., 2020), Big Bird (Zaheer et al., 2020), and MRA Attention (Zeng et al., 2022). We use HotpotQA (Yang et al., 2018), QuALITY (Pang et al., 2022), and WikiHop (Welbl et al., 2018) to assess the language models. HotpotQA is an answer span extraction task, while QuALITY and WikiHop are multi-choice question answering tasks. We set questions and multi-choice answers (for QuALITY and WikiHop) as VIP-tokens.

As shown in Table 6.2, we verify that our method is consistently better compared to Longformer and Big Bird. Our method obtains better accuracy in QuALITY and WikiHop compared to 4K length RoBERTa model, but it is a bit worse than 4k length RoBERTa model on HotpotQA. More pretraining helps close the gap. We also use WikiHop to experiment with method specific hyperparameters (such as block size in Big Bird, window size in Longformer, and compression size R in our method). As shown in Figure 6.7, our runtime efficiency frontier is consistently better than the baselines. The key **takeaway** is that our method has a much better runtime efficiency than baselines that have the same sequence length without sacrificing its model performance. Further, we note that our method can be scaled to larger models for accuracy improvement.

Method	Size	Length	HotpotQA			QuALITY		WikiHop	
			Time	EM	F1	Time	Accuracy	Time	Accuracy
RoBERTa	base	512	19.9	35.1	44.9	21.2	39.0	19.6	67.6
RoBERTa	base	4K	422.3	62.2	76.1	403.2	39.5	414.1	75.2
Big Bird	base	4K	297.9	59.5	73.2	307.0	38.5	293.3	74.5
Longformer	base	4K	371.0	59.9	73.6	368.0	27.9	369.7	74.3
MRA Attention	base	4K	203.5	63.4	77.0	200.5	38.7	199.2	76.1
Ours	base	4K	114.6	60.9	74.6	126.4	39.6	108.0	75.9
Ours-150k	base	4k	114.6	60.7	74.1	126.4	39.4	108.0	76.1
Ours*	base	4K	114.6	61.4	75.0	125.7	39.5	108.0	76.1
Ours*	large	4K	285.8	66.7	80.0	390.8	41.8	394.3	79.6

Table 6.2: Dev set results for encoder-only models.

Encoder-Decoder Models. We compare our method with T5 (Raffel et al., 2020), LongT5 (Guo et al., 2022), and LED (Beltagy et al., 2020). We note that LED-

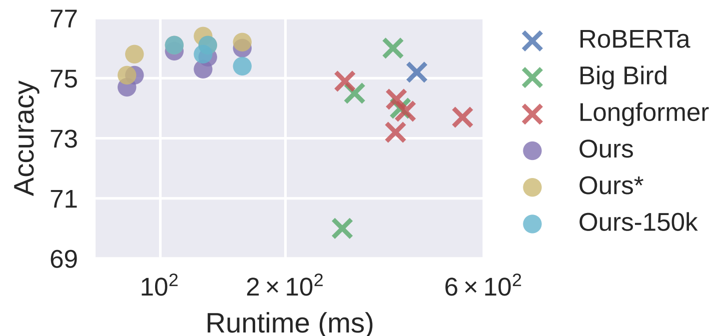

Figure 6.7: Model runtime vs WikiHop dev accuracy when using different model specific hyperparameters

base has 6 encoder/decoder layers compared to 12 encoder/decoder layers in base models of other methods, its accuracy is usually lower. So, LED is evaluated in limited tasks. We use HotpotQA (Yang et al., 2018), WikiHop (Welbl et al., 2018), CNN/Dailymail (See et al., 2017), MediaSum (Zhu et al., 2021), Arxiv (Cohan et al., 2018), and GovReport (Huang et al., 2021), SummScreenFD (Chen et al., 2022), QMSum (Zhong et al., 2021), NarrativeQA (Kočiský et al., 2018), Qasper (Dasigi et al., 2021), QuALITY (Pang et al., 2022), ContractNLI (Koreeda and Manning, 2021) from SCROLLS benchmark (Shaham et al., 2022), and MultiNews (Fabbri et al., 2019) to assess the language models. For question answering tasks, we set questions and multi-choice answers (for QuALITY and WikiHop) as VIP-tokens in our method. For query-based summarization, such as QMSum, we use the query as VIP-tokens in our method. For general summartization tasks, we prepend a "summarize:" in each instance and use it as VIP-tokens in our method. The results are shown in Table 6.3 and Table 6.4. Our method achieves matching or better performance in most tasks compared to T5, LongT5, and LED with much higher efficiency. Further, the performance monotonically increases with the model size, so our method can scale to larger models.

FLOPs. Some readers might be interested in the FLOP efficiency of each method, so we provide FLOP profiling results for some experiments. The automatic tools that we used for calculating FLOPs is deepspeed's FlopsProfiler. We note that the

Method	Size	# Param	Length	WikiHop Runtime	WikiHop EM	WikiHop F1	HotpotQA Runtime	HotpotQA EM	HotpotQA F1	CNN/Dailymail Runtime	CNN/Dailymail R-1	CNN/Dailymail R-2	CNN/Dailymail R-L	MediaSum Runtime	MediaSum R-1	MediaSum R-2	MediaSum R-L
T5	base	223M	512	25.7 / 20.5	66.7	69.1	26.3 / 20.5	34.1	44.4	40.0 / 20.5	43.3	20.5	40.4	39.9 / 20.5	30.7	14.5	28.1
T5	base	223M	4K	594.3 / 553.7	76.2	78.1	594.3 / 550.6	64.2	77.5	614.4 / 549.4	43.8	20.9	41.0	613.5 / 552.9	34.9	17.2	31.9
LongT5	base	248M	4K	270.7 / 233.9	72.7	74.8	271.3 / 233.7	62.3	75.7	291.6 / 234.9	43.3	20.6	40.5	287.3 / 229.5	34.9	17.3	32.0
LED	base	162M	4K	236.6 / 222.9	70.0	72.4	237.4 / 222.9	55.1	67.9	249.4 / 221.8	43.3	20.0	40.5	- / -	-	-	-
Ours	base	223M	4K	181.7 / 148.1	76.7	78.4	155.4 / 127.4	64.5	77.7	195.8 / 139.9	43.6	20.7	40.7	196.7 / 140.2	34.8	17.3	31.9
T5	large	738M	512	83.5 / 67.0	69.1	71.4	84.1 / 67.0	36.9	47.8	124.6 / 67.0	43.8	20.7	40.9	124.5 / 67.0	31.9	15.5	29.1
T5	large	738M	4K	1738.7 / 1601.0	79.1	80.7	1598.1 / 1598.1	68.0	81.3	1824.8 / 1600.4	44.3	21.0	41.4	- / -	-	-	-
Ours	large	738M	4K	561.4 / 460.6	79.0	80.6	485.3 / 382.8	67.8	81.0	608.1 / 433.8	44.4	21.4	41.5	609.7 / 434.4	35.8	18.2	32.8
Ours	3b	3B	4K	1821.5 / 1441.2	80.8	82.3	1547.7 / 1197.1	70.2	83.2	1930.7 / 1364.8	44.8	21.5	41.9	1930.7 / 1364.8	36.3	18.5	33.3

Method	Size	# Param	Length	Qasper Runtime	Qasper EM	Qasper F1	QuALITY Runtime	QuALITY EM	QuALITY F1	Arxiv Runtime	Arxiv R-1	Arxiv R-2	Arxiv R-L	SummScreenFD Runtime	SummScreenFD R-1	SummScreenFD R-2	SummScreenFD R-L
T5	base	223M	512	31.8 / 20.5	10.8	16.4	29.3 / 20.5	33.6	47.3	59.0 / 20.5	28.9	8.6	25.6	59.1 / 20.5	27.0	4.8	23.5
T5	base	223M	4K	608.2 / 551.7	13.2	29.1	596.3 / 551.2	34.7	47.4	645.4 / 549.1	44.4	18.4	39.9	647.9 / 551.1	31.6	6.8	27.6
LongT5	base	248M	16K	1628.5 / 1421.3	16.2	33.4	1633.1 / 1439.7	35.8	48.5	1699.7 / 1370.4	48.5	21.7	43.7	1763.4 / 1427.8	33.1	7.3	28.5
LED	base	162M	16K	- / -	-	-	- / -	-	-	1055.8 / 923.6	47.8	20.6	43.2	- / -	-	-	-
Ours	base	223M	16K	538.3 / 391.6	16.0	30.8	557.1 / 419.2	36.5	48.7	672.8 / 392.1	48.5	21.4	43.9	670.5 / 390.9	33.1	7.3	28.6
T5	large	738M	512	101.9 / 66.4	11.3	17.0	95.8 / 67.1	35.3	49.0	182.2 / 67.1	30.5	9.1	27.1	180.9 / 66.5	28.3	4.9	24.9
T5	large	738M	4K	- / -	-	-	1760.5 / 1596.4	37.8	50.5	1901.5 / 1598.8	46.0	19.4	41.4	- / -	-	-	-
Ours	large	738M	16K	1679.6 / 1120.2	16.3	33.7	1753.6 / 1210.7	40.3	52.5	1959.1 / 1111.0	49.5	22.2	44.7	1957.1 / 1109.2	34.3	7.6	29.6
Ours	3b	3B	16K	6165.4 / 4637.3	19.0	38.2	6398.8 / 4962.7	45.2	56.0	7676.3 / 4642.2	49.8	22.4	45.0	7641.5 / 4631.3	34.7	7.8	30.1

Method	Size	# Param	Length	ContractNLI Runtime	ContractNLI EM	ContractNLI F1	NarrativeQA Runtime	NarrativeQA EM	NarrativeQA F1	GovReport Runtime	GovReport R-1	GovReport R-2	GovReport R-L	QMSum Runtime	QMSum R-1	QMSum R-2	QMSum R-L
T5	base	223M	512	24.0 / 20.5	73.5	73.5	26.8 / 20.5	2.0	11.3	59.1 / 20.5	40.5	14.8	38.2	43.5 / 20.5	30.2	8.0	26.5
T5	base	223M	4K	579.0 / 551.6	86.8	86.8	593.4 / 547.6	3.8	13.3	648.3 / 551.5	54.0	25.2	51.4	620.2 / 551.5	31.1	8.2	27.4
LongT5	base	248M	16K	1564.2 / 1462.5	85.1	85.1	1541.7 / 1370.2	5.2	15.6	1726.4 / 1387.7	55.8	27.9	53.2	1721.4 / 1450.7	35.7	11.7	31.4
Ours	base	223M	16K	484.2 / 393.1	87.0	87.0	518.2 / 394.4	5.0	15.8	674.0 / 391.6	55.2	27.1	52.6	623.1 / 396.5	31.8	8.8	27.9
T5	large	738M	512	78.1 / 67.1	74.3	74.3	- / -	-	-	180.9 / 67.0	43.3	16.2	41.1	136.4 / 67.1	31.7	8.1	27.6
T5	large	738M	4K	1702.4 / 1601.2	87.2	87.2	- / -	-	-	- / -	-	-	-	- / -	-	-	-
Ours	large	738M	16K	1440.6 / 1122.6	87.8	87.8	1551.7 / 1133.9	6.6	18.7	1955.5 / 1113.8	56.3	28.0	53.8	1816.4 / 1134.6	34.8	10.4	30.7
Ours	3b	3B	16K	5850.2 / 4665.9	88.5	88.5	6055.4 / 4659.4	8.2	21.2	7668.2 / 4642.7	56.9	28.5	54.3	7146.7 / 4655.6	35.7	10.9	31.1

Table 6.3: Dev set results for encoder-decoder models. The left / right values of runtime columns are the runtime for the entire model / the encoder.

Method	Size	# Param	Length	MultiNews			
				Runtime	R-1	R-2	R-L
T5	base	223M	512	59.2 / 20.5	42.5	15.3	39.0
T5	base	223M	4K	651.2 / 551.8	46.4	18.2	42.6
LongT5	base	248M	8K	721.7 / 550.6	46.7	18.3	42.9
LED	base	162M	8K	526.5 / 454.2	46.6	17.8	42.7
Ours	base	223M	8K	377.0 / 224.6	46.4	18.1	42.7
T5	large	738M	512	180.8 / 67.0	43.4	15.6	39.8
Ours	large	738M	8K	1140.3 / 651.5	48.2	19.2	44.2
Ours	3b	3B	8K	4094.5 / 2696.0	48.9	19.4	44.7

Table 6.4: Dev results for encoder-decoder models on MultiNews.

FLOP profiling tools do not always work and may throw errors or produce incorrect results when the code contains custom CUDA kernels. The results are summarized in Table 6.5 and Table 6.6. $\times$ in some entries means the profiler throws errors and cannot estimate the FLOPs numbers. Also, the profiler throws errors when we just profile the encoders of encoder-decoder models, so the FLOP numbers in Table 6.6 are for entire models. As shown in the tables, our method has the lowest FLOPs.

Method	Size	Length	WikiHop		
			Runtime	GFLOPs	Accuracy
RoBERTa	base	512	19.6	-	67.6
RoBERTa	base	4K	414.1	1317	75.2
Big Bird	base	4K	293.3	790	74.5
Longformer	base	4K	369.7	$\times$	74.3
MRA Attention	base	4K	199.2	697*	76.1
Ours	base	4K	108.0	526	75.9

Table 6.5: Dev accuracy and FLOPs for encoder-only models. *: some calculations are not captured by profiler, so value is underestimated.

However, we note that FLOP numbers do not always reflect practical latency reductions. Memory bandwidth and latency (which are not captured by FLOP numbers) also play an important role in the overall latency. For example, sparse matrix multiplication (with unstructured sparsity) usually has a much lower FLOP count than dense matrix multiplication. But, the latency reduction is only possible when sparsity is at least 95% or more (depending on the implementation) since sparse matrix multiplication is a memory bandwidth bounded operator.

Method	Size	Length	WikiHop			
			Runtime	GFLOPs	EM	F1
T5	base	512	25.7 / 20.5	120	66.7	69.1
T5	base	4K	594.3 / 553.7	1449	76.2	78.1
LongT5	base	4K	270.7 / 233.9	948	72.7	74.8
LED	base	4K	236.6 / 222.9	×	70.0	72.4
Ours	base	4K	181.7 / 148.1	663	76.7	78.4
Method	**Size**	**Length**	**ContractNLI**			
			Runtime	GFLOPs	EM	F1
T5	base	512	24.0 / 20.5	112	73.5	73.5
T5	base	4K	579.0 / 551.6	1437	86.8	86.8
LongT5	base	16K	1564.2 / 1462.5	4442	85.1	85.1
Ours	base	16K	484.2 / 393.1	1933	87.0	87.0

Table 6.6: Dev results and FLOPs for encoder-decoder models.

Scaling to Longer Sequences. The prior experiments limit the sequence length to at most 4K or 16K since the baselines can only be scaled up to these lengths. However, our method can be scaled to much longer sequences. We note that NarrativeQA (Kočiský et al., 2018) is an ideal testbed as shown in dataset statistics in Table 6.9. The results are shown in Table 6.7. The left / middle / right values of runtime column are for the entire model / the encoder / the last 8 layers (out of 12 layers) that uses our compression. The performance monotonically increases as sequence length increases. We note that for sequence length 64K, the performance of model with $k = 64$ is lower than the model with $k = 16$. We suspect that since the results are finetuned from the same model that is pretrained with $k = 16$, the large gap between the two different k's may have a negative impact on finetuning performance. Nevertheless, the performance is still higher than 32K length models.

Length	Runtime (ms)	k	h	EM	F1
16K	518.2 / 394.4 / 162.4	16	90	5.9	16.6
32K	946.8 / 671.6 / 212.6	32	55	6.6	17.5
32K	1027.9 / 751.0 / 298.0	16	90	6.4	17.5
64K	1848.7 / 1177.2 / 254.8	64	30	7.2	18.4
64K	2244.8 / 1574.2 / 659.4	16	90	7.5	19.3
128K	6267.8 / 5125.9 / 1902.2	16	90	8.0	19.6

Table 6.7: Dev results of NarrativeQA on base model when scaling sequence length from 16K to 128K.

Why focus on 4K - 128K lengths? We believe that the computation required

by standard Transformers for processing shorter sequences is not an efficiency bottleneck. As a result, we do not profile the performance of our method for smaller length sequences, since the standard Transformers are sufficiently fast in this case. Further, while our model can be applied to shorter sequences, we suspect that for shorter sequences, there may be less irrelevant information for VIP-tokens. So compressing the irrelevant information will not offer a meaningful speed up. This is a limitation as the compression works better when there is more compressible information. We have only pushed the sequence lengths to 128K since this length was sufficient to cover a majority of sequence lengths encountered in long sequence tasks (for example, our model is able to process an entire book at once).

6.5.2 Non-Language Tasks

While the focus of this method is on language tasks, the overall design does not limit its application to other tasks as long as the assumption ("a subset of tokens are disproportionately responsible for the model prediction" and VIP-tokens can be reasonably selected) holds or partially holds for the tasks. As a result, we also tested our method on the Long Range Arena (LRA) benchmark (Tay et al., 2021) and obtained very promising performance among all baselines compared in Zeng et al. (2022). We get slightly better performance than the top performing baselines presented in MRA-attention, but note that we are not trying to show that our method outperforms other baselines but to verify that our method can indeed be applied to other tasks.

Note that there are multiple implementations and hyperparameters used in the LRA benchmark, and comparisons across different implementations and hyperparameters is awkward. We use the same implementation and same hyperparameters as Zeng et al. (2022). The VIP-token selection of our method in LRA experiments is quite easy. We simply use the prepended CLS token as the only VIP token since it is responsible for the final model classification. The results are shown in Table 6.8. All results except for ours are directly cited from Zeng et al. (2022). Since LRA consists of a synthetic task (ListOps), language tasks (Text, Retrieval), and vision tasks

(Image, Pathfinder), these results can serve as preliminary evidence indicating the potential applications of our method on other non-language tasks.

Method	Listops	Text	Retrieval	Image	Pathfinder	Avg
Transformer	$37.1_{\pm 0.4}$	$65.2_{\pm 0.6}$	$79.6_{\pm 1.7}$	$38.5_{\pm 0.7}$	$72.8_{\pm 1.1}$	$58.7_{\pm 0.3}$
Performer	$36.7_{\pm 0.2}$	$65.2_{\pm 0.9}$	$79.5_{\pm 1.4}$	$38.6_{\pm 0.7}$	$71.4_{\pm 0.7}$	$58.3_{\pm 0.1}$
Linformer	$37.4_{\pm 0.3}$	$57.0_{\pm 1.1}$	$78.4_{\pm 0.1}$	$38.1_{\pm 0.3}$	$67.2_{\pm 0.1}$	$55.6_{\pm 0.3}$
SOFT	$36.3_{\pm 1.4}$	$65.2_{\pm 0.0}$	$83.3_{\pm 1.0}$	$35.3_{\pm 1.3}$	$67.7_{\pm 1.1}$	$57.5_{\pm 0.5}$
SOFT + Conv	$37.1_{\pm 0.4}$	$65.2_{\pm 0.4}$	$82.9_{\pm 0.0}$	$37.1_{\pm 4.7}$	$68.1_{\pm 0.4}$	$58.1_{\pm 0.9}$
Nystromformer	$24.7_{\pm 17.5}$	$65.7_{\pm 0.1}$	$80.2_{\pm 0.3}$	$38.8_{\pm 2.9}$	$73.1_{\pm 0.1}$	$56.5_{\pm 2.8}$
Nystrom + Conv	$30.6_{\pm 8.9}$	$65.7_{\pm 0.2}$	$78.9_{\pm 1.2}$	$43.2_{\pm 3.4}$	$69.1_{\pm 1.0}$	$57.5_{\pm 1.5}$
YOSO	$37.0_{\pm 0.3}$	$63.1_{\pm 0.2}$	$78.3_{\pm 0.7}$	$40.8_{\pm 0.8}$	$72.9_{\pm 0.6}$	$58.4_{\pm 0.3}$
YOSO + Conv	$37.2_{\pm 0.5}$	$64.9_{\pm 1.2}$	$78.5_{\pm 0.9}$	$44.6_{\pm 0.7}$	$69.5_{\pm 3.5}$	$59.0_{\pm 1.1}$
Reformer	$18.9_{\pm 2.4}$	$64.9_{\pm 0.4}$	$78.2_{\pm 1.6}$	$42.4_{\pm 0.4}$	$68.9_{\pm 1.1}$	$54.7_{\pm 0.2}$
Longformer	$37.2_{\pm 0.3}$	$64.1_{\pm 0.1}$	$79.7_{\pm 1.1}$	$42.6_{\pm 0.1}$	$70.7_{\pm 0.8}$	$58.9_{\pm 0.1}$
Big Bird	$37.4_{\pm 0.3}$	$64.3_{\pm 1.1}$	$79.9_{\pm 0.1}$	$40.9_{\pm 1.1}$	$72.6_{\pm 0.7}$	$59.0_{\pm 0.3}$
H-Transformer-1D	$30.4_{\pm 8.8}$	$66.0_{\pm 0.2}$	$80.1_{\pm 0.4}$	$42.1_{\pm 0.8}$	$70.7_{\pm 0.1}$	$57.8_{\pm 1.8}$
Scatterbrain	$37.5_{\pm 0.1}$	$64.4_{\pm 0.3}$	$79.6_{\pm 0.1}$	$38.0_{\pm 0.9}$	$54.8_{\pm 7.8}$	$54.9_{\pm 1.4}$
MRA-2	$37.2_{\pm 0.3}$	$65.4_{\pm 0.1}$	$79.6_{\pm 0.6}$	$39.5_{\pm 0.9}$	$73.6_{\pm 0.4}$	$59.0_{\pm 0.3}$
MRA-2-s	$37.4_{\pm 0.5}$	$64.3_{\pm 0.8}$	$80.3_{\pm 0.1}$	$41.1_{\pm 0.4}$	$73.8_{\pm 0.6}$	$59.4_{\pm 0.2}$
Ours	$37.3_{\pm 0.5}$	$65.3_{\pm 0.2}$	$80.9_{\pm 0.5}$	$41.3_{\pm 1.7}$	$74.6_{\pm 1.5}$	$59.9_{\pm 0.9}$

Table 6.8: Test set accuracy of LRA tasks.

6.5.3 Experiment Details

We run all experiments on NVIDIA A100 GPUs. The runtimes presented in this section are measured runtimes of a complete training step (including both forward and backward). For each method, we use the largest batch size that can fit into a 80GB A100 and measure the average latency of 10 steps. Then, the average latency is divided by the batch size to get the estimated runtime for a single instance. Via this procedure, we seek to measure the peak efficiency of each method when the GPU is at the highest possible utilization.

Dataset statistics and hyperparameters. The statistics of the sequence lengths of instances in each dataset are summarized in Table 6.9. The hyperparameters of all experiments are summarized in Table 6.10. When there are multiple values in an entry, it means we perform a hyperparameter search on these values. The amount of search is determined by the size of datasets. If a dataset is relatively large, we only search the learning rate. If a dataset is small, we include batch size and the

number of epochs in search. For all tasks, if the sequence lengths are longer than the model length m, the sequences will be truncated and only the first m tokens will be used. For encoder-decoder models, we use greedy decoding in sequence generations for simplicity. The maximal decoder output length, specified in Table 6.10, is set such that the maximal length covers the output lengths of more than 99% of instances. When the length needed for covering 99% of instances is greater than 512, we just set the maximal decoder output length to 512.

Pretraining. We use a filtered The Pile dataset (Gao et al., 2020) for all pretraining runs. Since we use public pretrained tokenizers, we want to enable the distribution of pretraining corpus to align well with the distribution of corpus used to create the tokenizers. As a result, we use tokens per byte as a proxy for alignment of distributions and filter out PubMed Central, ArXiv, Github, StackExchange, DM Mathematics (Saxton et al., 2019), Ubuntu IRC, EuroParl (Koehn, 2005), Youtube-Subtitles, and Enron Emails (Klimt and Yang, 2004) components, which have tokens per byte greater than 0.3. Then, the remaining corpus of The Pile dataset is used for pretraining.

For encoder-only models, we pretrain RoBERTa for 750K steps. A batch consists of 8,192 sequences of 512 length. The masking ratio for masked language modeling (MLM) is 15%. Then, 4K length models are continuously pretrained from the RoBERTa checkpoints for 300k steps. The positional embeddings are extended by duplicating the pretrained 512 positional embedding multiple times. For 4K length RoBERTa, Longformer, Big Bird and MRA Attention, the batch size is 64, and the masking ratio is 15%. With 15% masking ratio, there are roughly 616 masked tokens scattering in the sequences. We find that using 616 scattered masked tokens as VIP tokens for 4,096 length sequences might not be ideal for VIP-token centric compression, so we use masking ratio 7.5% and batch size 128 for our method. The number of masked tokens per sequence is reduced, and the number of total masked token predictions remains the same during pretraining. We note that with larger batch sizes, the wall clock pretraining runtime for our method is still smaller than the baselines. We also show downstream finetuning on our method pretrained

on the same number of tokens but fewer number of masked token predictions, denoted as Ours-150k. We verify that our proposed method can be integrated into a pretrained Transformer with some continuous pretraining. But we note that the amount of reduction in log perplexity for our method (-0.114) during pre-training is much larger than Longformer (-0.017) and Big Bird (-0.025) from 50K steps to 250K steps. The continuous pretraining for these baselines might have saturated since only the self-attention is approximated while our method might require more pretraining to adjust the parameters for a more aggressive approximation. So, we run a larger scale pretraining for our method. For the larger scale pretraining, we pretrain our method for 250K steps with batch size 512 and masking ratio 7.5%, denoted with Ours* in Table 6.2.

For encoder-decoder architecture of our method, we perform continuous pretraining from the public checkpoints of T5 for 250K steps with batch size 256 using the masked span prediction. Since each masked span (consists of multiple tokens) is replaced by a single special token, when using masking ratio of 15%, the number of special tokens in a sequence is not too large, and so we keep the masking ratio of 15% unchanged. To compute the relative positional bias for self-attention in T5, the position indices of queries and keys are needed (to calculate the position distance between each query and key). After applying our method, the input to the T5 block is the compressed sequence (some tokens are compressed while some tokens remain uncompressed). For each uncompressed token, the position index remains unchanged as its true position in the original sequence. For each new token that represents a compressed segment, the position index is the floored average of position indices of tokens in the segment. In this way, we can minimize the amount of modification needed to the internals of the T5 block, and the relative attention bias for the compressed sequence is an approximation of the attention bias for the original sequence.

Percentile	RoBERTa			T5				
	HotpotQA	QuALITY	WikiHop	WikiHop	HotpotQA	Qasper	QuALITY	ContractNLI
75th	1535	7603	2204	2399 / 6	1692 / 6	7029 / 29	7747 / 17	2991 / 4
95th	1928	8495	3861	4206 / 9	2129 / 10	10920 / 71	8603 / 28	5061 / 4

Percentile	T5							
	NarrativeQA	CNN/Dailymail	MediaSum	Arxiv	SummScreenFD	GovReport	QMSum	MultiNews
75th	90482 / 10	1242 / 87	2621 / 29	13477 / 364	12119 / 188	13304 / 811	19988 / 110	3032 / 379
95th	260533 / 18	1946 / 130	5061 / 64	26024 / 759	16722 / 330	23795 / 983	31749 / 162	6676 / 468

Table 6.9: Length statistics of each dataset. The values are the percentiles of number of tokens for the specific tokenizers. For T5 tokenizer, the left value of is for sequence lengths of encoder input, and the right value is for sequence lengths of decoder input.

LM	Encoder-Only			Encoder-Decoder				
Task	HotpotQA	QuALITY	WikiHop	WikiHop	HotpotQA	CNN/Dailymail	MediaSum	Qasper
Optimizer	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam
Weight Decay	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
LR Decay	Linear	Linear	Linear	Linear	Linear	Linear	Linear	Linear
Precision	FP16	FP16	FP16	BF16	BF16	BF16	BF16	BF16
Batch Size	32	16	32	32	32	32	32	{16, 32}
Learning Rate	{3e-5, 5e-5}	{3e-5, 5e-5}	{3e-5, 5e-5}	{1e-4, 3e-4}	{1e-4, 3e-4}	{1e-4, 3e-4}	{1e-4, 3e-4}	{1e-4, 3e-4}
Epochs	10	{10, 20}	10	10	10	10	10	{10, 20}
Warmup Steps	1000	200	1000	1000	1000	1000	1000	200
Max Output Length	-	-	-	32	40	256	256	128

LM	Encoder-Decoder							
Task	QuALITY	ContractNLI	NarrativeQA	Arxiv	SummScreenFD	GovReport	QMSum	MultiNews
Optimizer	Adam	Adam	Adam	Adam	Adam	Adam	Adam	Adam
Weight Decay	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
LR Decay	Linear	Linear	Linear	Linear	Linear	Linear	Linear	Linear
Precision	BF16	BF16	BF16	BF16	BF16	BF16	BF16	BF16
Batch Size	{16, 32}	{16, 32}	32	32	{16, 32}	{16, 32}	{16, 32}	32
Learning Rate	{1e-4, 3e-4}	{1e-4, 3e-4}	{1e-4, 3e-4}	{1e-4, 3e-4}	{1e-4, 3e-4}	{1e-4, 3e-4}	{1e-4, 3e-4}	{1e-4, 3e-4}
Epochs	{10, 20}	{10, 20}	5	{10, 20}	{10, 20}	{10, 20}	{10, 20}	10
Warmup Steps	200	1000	1000	1000	200	1000	100	1000
Max Output Length	90	4	47	512	512	512	310	512

Table 6.10: Hyperparameters for all experiments.

6.6 Limitations

Our method assumes that in many tasks, a subset of tokens are disproportionately responsible for the model prediction, the remaining non-VIP-tokens may play a role but are less critical. Our method excels specifically on such tasks by selectively locating relevant information in the sequence for the given VIP-tokens. As the experiments show, this choice is effective in many cases but this behavior is not universal. Occasionally, an embedding is pre-computed which must then serve multiple tasks concurrently, e.g., *both* text retrieval and natural language inference. In this case, if we do not know the tasks beforehand, VIP-token selection cannot be meaningfully performed. Further, VIP-token selection requires some understanding of the tasks. However, we believe that a reasonable selection can be made with some generic knowledge for most tasks or use cases. For example, for question answering tasks, we just use questions as VIP-tokens. For classification tasks (for example, Long Range Arane benchmark shown in Table 6.8), we simply use CLS token as the VIP-token since only CLS token is used for final prediction. For masked language modeling, we use the masked tokens as the VIP-tokens since only these masked tokens are used for final prediction. For question answering, the question tokens (and candidate tokens for multi-choice QA) are used as the VIP-tokens since they **(1)** are important to the specific task goals and **(2)** easily pre-identifiable by the user. In the worst case, if there is no obvious token to be selected, we can prepend some learnable "latent" tokens or certain user commands (such as "summarize" for summarization tasks as we used in our experiments) and use them as VIP-tokens (in fact, CLS tokens can be thought of as these learnable tokens).

To reduce the complexity of our implementation, the method is currently setup for the encoder module of the Transformer that assumes full access to the entire sequence. The proposed compression might be extended to approximate the computation in the decoder, but it needs more implementation work, so we leave it as future work. Consequently, the current implementation is less useful for decoder-only models. Having said that, there are still clear opportunities for decoder-only models, which are left as our future work. We briefly describe three possible options

to do so. **(1)** We can use the input tokens of the decoder as VIP-tokens to compress the representations of context sequence generated by the encoder before Cross Attention computation to reduce the cost of Cross Attention. **(2)** Auto-regressive decoding operates using Causal Attention at each step. This Causal Attention operation requires memory and computation that is linear in the length of the prefix. We can keep the same Causal Attention VIP-token (the representation of the token currently being generated) and apply our method to compress the representations of the previously generated tokens. This reduces the linear complexity of the Causal Attention operation to sublinear. This is useful for reducing the cost of inference. For training, we can break the sequence into two segments: prefix segment and decoding segment. Then, we can use the proposed compression in the prefix segment and vanilla computation in decoding segment. To prevent look ahead to the future tokens, we might only use the first token in the decoding segment as VIP-token. **(3)** In many cases, the current large language models (LLMs) are not used directly to generate an ultra long text. Rather, the requirements for processing ultra longer context manifests due to the need to incorporate user input and previous LLM responses (like ChatGPT) or to incorporate search results (such as New Bing). In these cases, there is a prefix context, and LLMs will generate text based on the user prompt and prefix context. We note that our method can indeed be applied to compress the prefix context, and the user prompt and currently generated tokens will be the VIP-tokens.

6.7 Summary

The goal of this chapter was to develop an algorithm which enables Transformer models to process ultra long sequences efficiently. We extended the idea described in Chapter 5 for approximating self-attention to reduce the overall cost of both MHA and FFN. Specifically, we developed a VIP-token centric sequence compression method to compress/decompress the input/output sequences of Transformer layers thereby reducing the complexity dependency on the sequence length N without sacrificing the model accuracy. Our empirical evaluation shows that our method

can be directly incorporated into existing pretrained models with some additional training. Also, it often has much higher efficiency compared to baselines with the same sequence length while offering better or competitive model accuracy. A preliminary version of this chapter was published as (Zeng et al., 2023b) and the codebase is available at `https://github.com/mlpen/vcc`.

7 LOW PRECISION INTEGER COMPUTATION FOR GENERAL MATRIX MULTIPLY

In the previous chapters, we discussed efficiency strategies for different modules of a Transformer model. In this chapter, we will focus on the efficiency of the fundamental operator of Transformer models: GEneral Matrix Multiply (GEMM), upon which all compute-heavy operations are built. Calculating the product of two matrices using GEMM is one of the most widely used operations in modern machine learning. Given matrices **A** and **B** of size $N \times D$ and $H \times D$ respectively, the output of a GEMM is calculated as

$$\mathbf{C} = \mathbf{A}\mathbf{B}^{\top}$$

Choosing the appropriate numerical precision or data type (FP32, FP16, or BF16) for GEMM is often important, and hinges on several factors including the specific application, characteristics of the data, model architecture, as well as numerical behavior such as convergence. This choice affects compute and memory efficiency most directly, since a disproportionately large chunk of the compute footprint of a model involves the GEMM operator. A good example is the large improvement in latency and memory achieved via low bit-width GEMM, and made possible due to extensive ongoing work on quantization (to low bit-width data types) and low-precision training (Banner et al., 2019; Nagel et al., 2019; Kim et al., 2021; Dettmers et al., 2022; Li and Gu, 2023; Xiao et al., 2023; Dettmers et al., 2022; Liu et al., 2023b,a; Lin et al., 2022; Li and Gu, 2023; Yuan et al., 2021; Ding et al., 2022; Li et al., 2023a; Wang et al., 2018b; Wu et al., 2018; Zhu et al., 2020; Wortsman et al., 2023). Integer quantization is being actively pursued for inference efficiency, and the use of *low bit-width* integers is universal to deliver the efficiency gains. However, this strategy often incurs large rounding errors when representing all matrix entries as low bit-width integers, and explains the drop in performance and thereby, a need for error correction techniques (Frantar et al., 2023; Xiao et al., 2023; Chee

et al., 2023; Adepu et al., 2024). So how much of the performance degradation is due to (a) rounding to integers versus (b) restricting to low bit-width integers? To answer this question, it appears worthwhile to check whether integer GEMMs will achieve parity without sophisticated techniques (for the inference stage, and more aspirationally, for training) for popular models if we do *not* restrict to low bit-width integers.

In this chapter, we discuss our work on low precision integer computation for Transformer models. The starting point of our work is to first experimentally verify that the aforementioned hypothesis – that integer GEMM may work – is true (see §7.1). But by itself, this finding offers no value proposition for efficiency. Nonetheless, this experiment is useful for the following reason. For a particular class of models (e.g., Transformers), we can easily contrast the corresponding input matrices **A** and **B** between (a) integer GEMM and (b) low bit-width integer GEMM and probe if any meaningful structure can be exploited. While there *is* a clear difference in the *outputs* of (a) integer GEMM versus (b) low bit-width integer GEMM, we find that *a large majority* of entries of **A** and **B** can be represented using low bit-width integers – and the difference in the outputs can be attributed to a few *heavy hitter* entries in **A** and **B**, that cannot be represented using low bit-width integers. Other works have also run into this issue of "outliers" and use high precision (Dettmers et al., 2022) or a separate quantization for these entries (Yuan et al., 2021; Xiao et al., 2023). Based on the observation that we can represent a large integer by a series of smaller integers, our algorithm, Integer Matrix Unpack (IM-Unpack), enables unpacking any integer into a series of low-bit integers. The benefit is that the calculation can be carried out entirely using low bit-width integer arithmetic and thus unifies calculations needed for heavy hitters and the other entries (already amenable to low-bit integer arithmetic). Specifically, IM-Unpack unpacks an integer matrix such that all values of the unpacked matrices always stay within the representable range of low bit-width integers (bit-width can be chosen arbitrarily, as low as two). We obtain the exact result of the original integer GEMM using purely low bit-width integer GEMMs. Since the bit-width of integer arithmetic

is independent of the actual range of the original matrices, the construction will simplify the hardware/compiler support by only needing support for *one* specific bit-width. The structure/contributions of this chapter is shown in Figure 7.1.

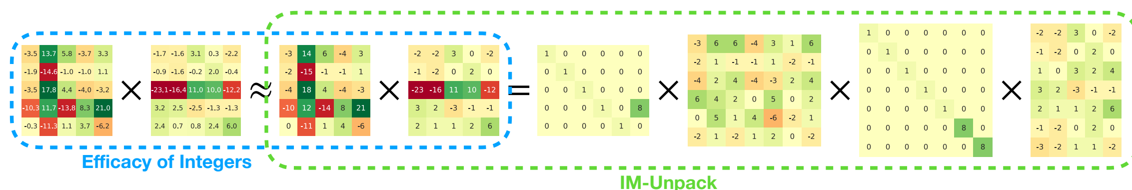

Figure 7.1: Overall Illustration. We verify the **Efficacy of Integers** (**Contribution 1**) in §7.1, but note that the integer matrices contain heavy hitters (§7.2). Then, we describe our proposed algorithm, **IM-Unpack** (**Contribution 2**), to resolve these heavy hitters in §7.3.

Notations. We first define notations for all relevant GEMMs. For the linear layer, let the input activation and parameter matrix be $\mathbf{X}$ and $\mathbf{W}$. Let the query, key, value matrices needed in self-attention be $\mathbf{Q}, \mathbf{K}, \mathbf{V}$. Below, we itemize all GEMMs used in a Transformer model discussed in §2.2:

$$\mathbf{Y} = \mathbf{X}\mathbf{W}^\top \quad \mathbf{P} = \mathbf{Q}\mathbf{K}^\top \quad \mathbf{O} = \mathbf{M}\mathbf{V}$$

where $\mathbf{M}$ is the attention score between $\mathbf{Q}$ and $\mathbf{K}$ defined as $\mathbf{M} = \text{softmax}(\mathbf{P})$ (omitting scaling factors). Now, given the gradient for $\mathbf{Y}, \mathbf{P}, \mathbf{O}$ denoted as $\nabla_\mathbf{Y}, \nabla_\mathbf{P}, \nabla_\mathbf{O}$, the other gradients are calculated via GEMMs as well:

$$\begin{aligned} \nabla_\mathbf{X} &= \nabla_\mathbf{Y}\mathbf{W} \quad & \nabla_\mathbf{Q} &= \nabla_\mathbf{P}\mathbf{K} \quad & \nabla_\mathbf{M} &= \nabla_\mathbf{O}\mathbf{V}^\top \\ \nabla_\mathbf{W} &= \nabla_\mathbf{Y}^\top\mathbf{X} \quad & \nabla_\mathbf{K} &= \nabla_\mathbf{P}^\top\mathbf{Q} \quad & \nabla_\mathbf{V} &= \mathbf{M}^\top\nabla_\mathbf{O} \end{aligned}$$

These notations will help refer to each type of GEMM later.

7.1 Round to Nearest: What do we lose?

Let us start by using the simplest Rounding To Nearest (RTN) to map FP to integers, and check the extent to which integer GEMMs work satisfactorily for both training

and inference, if we do *not* restrict to low bit-width integers. Specifically, for matrix $\mathbf{A}$, **all** entries of $\mathbf{A}$ are quantized via

$$\mathbf{A}_q = \text{round}(0.5\beta/\alpha_p(\mathbf{A})\mathbf{A}) \tag{7.1}$$

where $\alpha_p(\mathbf{A})$ gives the p-th percentile based on the magnitude of entries in $\mathbf{A}$, i.e., p% of entries in $\mathbf{A}$ fall in the interval $[-\alpha_p(\mathbf{A}), \alpha_p(\mathbf{A})]$. We only need $\alpha_p(\mathbf{A})$ as a meaningful estimate of the approximate range of values, and so we set $p = 95\%$ for all experiments except a few cases noted explicitly. The hyperparameter β is the number of distinct integers that we want to use to encode values that are within $[-\alpha_p(\mathbf{A}), \alpha_p(\mathbf{A})]$. Then, after quantization, the GEMM for the original matrices can be approximated (because we incur a rounding error) in the quantized domain using integer GEMMs. The approximated GEMM is computed using the quantized $\mathbf{A}$ and $\mathbf{B}$:

$$\mathbf{C} \approx \frac{\alpha_p(\mathbf{A})\alpha_p(\mathbf{B})}{(0.5\beta)^2}\mathbf{A}_q\mathbf{B}_q^\top \tag{7.2}$$

The scaling factor in (7.2) is used to undo the scaling in (7.1). Here, $\mathbf{A}_q\mathbf{B}_q^\top$ is an integer GEMM, as desired. For notational simplicity, if clear from context, we will drop the q subscript from $\mathbf{A}$ and $\mathbf{B}$.

Why Use Percentiles? We need a way of mapping the actual range of values in a floating point matrix to an integer range. In this process, we should ensure that most values fall within the desired range and fill up the representable range as much as possible, so we need a statistic to estimate the range of values in a FP matrix. We compared percentile and standard deviation and inspected different parameter matrices $\mathbf{W}$ and the corresponding inputs $\mathbf{X}$ in the LLaMA-7B model (Touvron et al., 2023). The outlier problem in $\mathbf{W}$ is moderate: we can see in Table 7.1, that both standard deviation and percentile estimates are stable across columns. On the other hand, the outliers in $\mathbf{X}$ are a bit more problematic and include a few entries that are much larger than the non-outliers. The estimation of standard deviation varies much more as shown in Table 7.1: even removing an extremely small subset of the largest outliers can sizably alter the estimates. In contrast, the percentile is

more stable. As a result, we choose percentile as the estimation of value range.

Number of Largest Outliers Removed		0	10	10^2	10^3
W	Standard Deviation	0.0082	0.0082	0.0082	0.0082
	95-Percentile	0.0177	0.0177	0.0177	0.0177
X	Standard Deviation	0.0330	0.0327	0.0320	0.0214
	95-Percentile	0.0280	0.0280	0.0280	0.0278

Table 7.1: Standard deviation vs percentile when removing largest outliers. **X** has 2.25×10^7 entries and **W** has 1.68×10^7 entries.

In §7.1.1–§7.1.2, we evaluate the efficacy of integer GEMMs as a replacement to FP GEMMs by evaluating how well simple RTN works for inference and training (error analysis of RTN is discussed in §7.1.3). We do not strictly constrain the quantized integers to be within the representable range of certain bit-widths (the maximal value of quantized integers can be very large, up to maximum of INT32), so we use INT without specifying bit-width to denote the data type used in RTN in these subsections. The use of integers with specific bit-widths will be discussed later in §7.2–§7.3.

7.1.1 Efficacy of Integers: Inference

A majority of the literature on quantized low precision calculations focuses on inference efficiency (Frantar et al., 2023; Chee et al., 2023; Liu et al., 2023a; Yuan et al., 2021; Frantar et al., 2023; Chee et al., 2023; Liu et al., 2023a; Yuan et al., 2021; Lin et al., 2022; Li and Gu, 2023; Yuan et al., 2021; Ding et al., 2022; Li et al., 2023a). Here, given a trained model, quantization seeks to reduce the precision of parameters and input activations to low precision. This allows faster low precision arithmetic for compute efficiency while maintaining model performance. So, we first evaluate how well RTN preserves model performance compared to baselines in this inference regime using downstream tasks: ARC (Clark et al., 2018), BoolQ (Clark et al., 2019), HellaSwag (Zellers et al., 2019a), PIQA (Bisk et al., 2020), and WinoGrande (Sakaguchi et al., 2021). Most quantization schemes for LLMs focus on quantizing GEMMs in the Linear layers, while quantization methods for Vision

Transformers are more ambitious and quantize *all* GEMMs in a Transformer. We follow this convention for baselines, but present all variants for RTN.

Quantize Parameters Only. One direction of quantization research focuses on quantizing the parameters for better storage and memory usage. We also evaluate how well RTN works for storage and memory efficiency. After quantization, the quantized $\mathbf{W}_{\mathrm{q}}$ usually contains a few hundreds of distinct integers. Simply representing $\mathbf{W}_{\mathrm{q}}$ in plain integer format would not be efficient and usually requires larger than 8 bits per value for memory. By inspecting the value distribution of $\mathbf{W}_{\mathrm{q}}$, we found that some values occur much more frequently than others, which creates a clear opportunity for compression. We simply apply Huffman Encoding (HE), which was also used in (Han et al., 2016) to compress models for memory efficiency, to use shorter encoding for more frequent values. As shown in Table 7.2, with RTN and HE, we are able to significantly reduce the average bits per value with small or no performance degradation and this leads to significantly better efficiency compared to baselines (Frantar et al., 2023; Chee et al., 2023; Liu et al., 2023a; Yuan et al., 2021) for both Transformer based LLMs and Vision Transformers.

Quantize GEMMs in Linear layers. It is common Xiao et al. (2023); Liu et al. (2023a) to try and quantize the weight and input activation of *linear layers* to low precision for compute efficiency. We summarize our comparisons in Table 7.3. Here, we compare RTN to (Xiao et al., 2023; Dettmers et al., 2022; Liu et al., 2023b,a). As shown in Table 7.3, a simple RTN works remarkably well compared to other baselines. We use INT as a data type for RTN here; in §7.3, we show that we can compute integer GEMMs of any bit-widths using arbitrarily low bit-width GEMMs.

Quantize all GEMMs. A more ambitious goal is to quantize *every* GEMM in a Transformer model for higher efficiency. The comparison results with (Lin et al., 2022; Li and Gu, 2023; Yuan et al., 2021; Ding et al., 2022; Li et al., 2023a) are summarized in Table 7.4. We can draw a similar conclusion that a simple RTN offers strong performance.

More Empirical Results on LLM Quantization. To evaluate how well RTN works

	Method	β	$\overline{\text{Bits}}$	ARC-c	ARC-e	BoolQ	HS	PIQA	WG
LLaMA-7B	Full-Precision	-	16	43.1	76.3	77.8	57.2	78.0	68.8
	GPTQ	-	4	37.4	72.7	73.3	54.9	77.9	67.9
	LLM-FP4	-	4	40.4	74.9	74.2	55.8	77.8	69.9
	QuIP	-	2	22.3	42.8	50.3	34.0	61.8	52.6
	RTN+HE	5	2.5	39.3	72.8	69.9	53.4	74.9	66.4
		7	2.9	42.6	73.9	72.3	55.9	77.0	67.4
		11	3.5	43.9	76.1	77.3	56.3	77.3	69.3
		15	4.0	43.0	75.7	77.5	57.0	78.0	69.2
		31	5.0	42.7	76.1	76.1	57.3	77.3	69.3

	Method	β	$\overline{\text{Bits}}$	Tiny	Small	Base	Large	Huge
ViT	Full-Precision	-	32	75.5	81.4	85.1	85.8	87.6
	PTQ4ViT	-	3	18.3	36.2	21.4	81.3	78.9
	RTN+HE	3	1.8	0.5	8.3	63.6	81.9	83.3
		5	2.4	38.2	69.0	81.1	84.9	86.7
		7	2.9	63.6	76.7	83.6	85.4	87.2
		15	4.0	73.4	80.5	84.8	85.7	87.6

Table 7.2: Inference: Comparison on LLaMA-7B zero-shot performance and ViT ImageNet classification when only quantize parameters. HS: HellaSwag, WG: WinoGrande.

on the inference of different models and different model sizes, we also run experiments on LLaMA-13B (Touvron et al., 2023), Mistral-7B (Jiang et al., 2023), and Phi-2 (Javaheripi et al., 2023). The results are summarized in Table 7.5, Table 7.6, and Table 7.7. To minimize code change, we only evaluate the quantization of linear layers, as in other quantization works (Xiao et al., 2023; Dettmers et al., 2022; Liu et al., 2023b,a), for Mistral-7B and and Phi-2.

7.1.2 Efficacy of Integers: Training

The transition from FP32 to FP16 and BF16 for GEMMs has doubled the compute efficiency of modern deep learning models. However, far fewer efforts have focused on low precision *training* (relative to inference) and this usually requires more sophisticated modifications (Wang et al., 2018b; Wu et al., 2018; Zhu et al., 2020; Wortsman et al., 2023). In this subsection, we evaluate how well quantizing *all* GEMMs (both forward pass and backward pass) using RTN works for training

	Method	β	Type	ARC-c	ARC-e	BoolQ	HS	PIQA	WG
LLaMA-7B	Full-Precision	-	BF16	43.1	76.3	77.8	57.2	78.0	68.8
	LLM.int8()	-	INT8‡	43.8	75.5	77.8	57.4	77.6	68.7
	SmoothQuant	-	INT8	37.4	74.4	74.0	55.0	77.5	69.6
	LLM-QAT	-	INT4	30.2	50.3	63.5	55.6	64.3	52.9
	LLM-FP4	-	FP4	33.6	65.9	64.2	47.8	73.5	63.7
	RTN	5	INT	39.3	72.8	69.9	53.4	74.9	66.4
		7	INT	42.6	73.9	72.3	55.9	77.0	67.4
		11	INT	43.9	76.1	77.3	56.3	77.3	69.3
		15	INT	43.0	75.7	77.5	57.0	78.0	69.2
		31	INT	42.7	76.1	76.1	57.3	77.3	69.3

	Method	β	Type	Tiny	Small	Base	Large	Huge
ViT	Full-Precision	-	FP32	75.5	81.4	85.1	85.8	87.6
	RTN	5	INT	3.9	36.9	78.7	83.6	85.3
		7	INT	41.0	70.9	82.8	84.9	86.7
		15	INT	71.4	79.8	84.6	85.6	87.5

Table 7.3: Inference: Comparison on LLaMA-7B zero-shot performance and ViT ImageNet classification when using quantized computations in all linear layers. The super-script ‡ indicates that LLM.int8() uses mixed-precision (INT8+FP16) to process outliers using FP16.

	Method	β	Type	ARC-c	ARC-e	BoolQ	HS	PIQA	WG
LLaMA-7B	Full-Precision	-	BF16	43.1	76.3	77.8	57.2	78.0	68.8
	RTN	5	INT	23.5	34.3	54.8	32.5	57.6	49.7
		7	INT	34.2	64.0	64.6	50.1	70.3	61.2
		11	INT	41.6	72.4	68.7	55.1	75.4	65.1
		15	INT	44.0	75.0	74.6	56.4	77.0	66.3
		31	INT	43.4	75.8	76.8	57.5	77.4	68.4

	Method	β	Type	Tiny	Small	Base	Large	Huge
ViT	Full-Precision	-	FP32	75.5	81.4	85.1	85.8	87.6
	FQ-ViT	-	INT8	-	-	83.3	85.0	-
	I-ViT	-	INT8	-	81.3	84.8	-	-
	PTQ4ViT	-	INT6*	66.7	78.3	82.9	84.9	86.6
	APQ-ViT	-	INT4	17.6	48.0	41.4	-	-
	RepQ-ViT	-	INT4	-	65.1	68.5	-	-
	RTN	5	INT	3.5	28.5	76.9	83.2	84.9
		7	INT	39.0	69.9	82.1	84.7	86.5
		15	INT	71.1	79.8	84.5	85.6	87.5

Table 7.4: Inference: Comparison on LLaMA-7B and ViT when quantize computation in all GEMMs. *: PTQ4ViT uses a twin uniform quantization so GEMMs cannot be performed on INT6 directly and requires some modifications.

Method	β	Type	ARC-c	ARC-e	BoolQ	HS	PIQA	WG
Full-Precision	-	BF16	48.0	79.5	80.6	60.0	79.2	72.1
SmoothQuant	-	INT8	45.5	76.3	76.5	58.0	78.0	72.1
	-	INT4	25.1	49.9	57.6	56.0	61.3	52.6
LLM-FP4	-	FP4	39.9	71.7	71.9	53.3	74.8	66.7
RTN	5	INT	37.6	70.0	69.1	51.9	72.4	64.6
	7	INT	44.1	76.1	73.5	57.3	76.7	67.6
	15	INT	46.9	78.8	79.4	59.0	78.2	72.5
	31	INT	48.0	79.7	80.2	59.9	78.0	71.3

Table 7.5: Inference: Comparison on LLaMA-13B when we quantize computation in all linear layers.

Method	β	Type	ARC-c	ARC-e	BoolQ	HellaSwag	PIQA	WinoGrande
Full-Precision	-	BF16	48.0	79.5	80.6	60.0	79.2	72.1
RTN	5	INT	25.1	44.4	54.8	37.7	57.9	52.0
	7	INT	38.0	66.9	70.1	53.3	72.5	64.2
	15	INT	45.9	77.6	80.0	59.5	77.5	71.5
	31	INT	47.9	79.3	80.0	60.5	78.6	70.9

Table 7.6: Inference: LLaMA-13B when we quantize computation in all GEMMs.

	Method	β	ARC-c	ARC-e	BoolQ	HellaSwag	PIQA	WinoGrande
Mistral-7B	Full-Precision	-	50.3	80.9	83.6	61.3	80.7	73.8
	RTN	5	38.1	70.5	69.9	53.9	73.3	61.4
		7	44.9	75.0	76.0	58.7	77.8	68.6
		15	48.8	79.7	80.3	60.8	79.6	73.2
		31	50.3	80.1	83.5	61.5	80.7	74.4
Phi-2	Full-Precision	-	20.6	26.1	41.3	25.8	54.3	49.3
	RTN	5	22.1	26.7	41.5	25.6	52.3	48.1
		7	21.3	25.8	40.9	25.8	53.9	49.5
		15	21.3	27.3	45.4	25.8	53.4	48.8
		31	21.0	25.8	40.8	25.7	53.0	50.7

Table 7.7: Inference: Mistral-7B and Phi-2 when we quantize computation in all linear layers.

Transformer models. To ensure that the updates can be properly accumulated for the parameters, we use FP32 for storing the parameters and use the quantized version for GEMMs. To limit the amount of compute but still gather useful empirical feedback, we evaluate RTN on RoBERTa (Liu et al., 2019a) pretraining using masked language modeling (Devlin et al., 2019) on the English Wikipedia corpus (Wikimedia, 2019), ImageNet classification (Russakovsky et al., 2015) using ViT (Dosovitskiy et al., 2021), and T5-Large (Raffel et al., 2020) finetuning. All hyperparameters (including random seed) are the same for full-precision and RTN quantized training.

RoBERTa. As shown in Figure 7.2, when $p = 95\%$, for both Small and Base models, the RTN quantized training gives an almost identical log perplexity (loss) curve as FP32 training for $\beta \in \{15, 31, 255\}$. For larger β, the curve is even closer to the FP32 training curve. We see that $\beta = 31$ already gives a remarkably good result. Surprisingly, despite a marginally higher training log perplexity when using RTN, the validation log perplexity of RTN ($\beta = 31$ and $\beta = 255$) is marginally lower than FP32 and BF16, see Table 7.8.

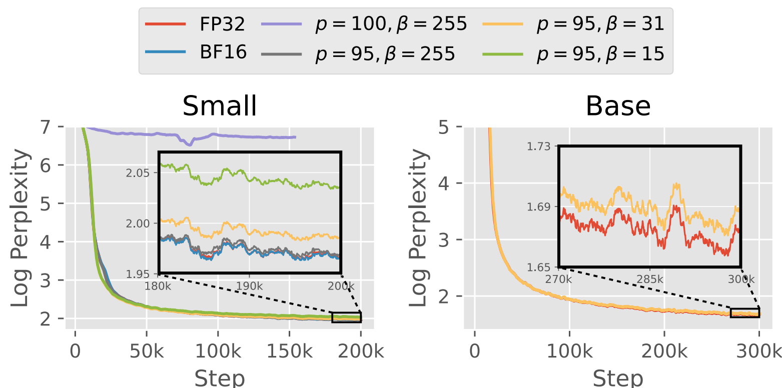

Figure 7.2: Training: Comparison of RoBERTa loss curves.

ViT. For ViT, compared to RoBERTa pretraining, we found that it may be necessary to allow the gradients $\nabla_\mathbf{Y}, \nabla_\mathbf{P}, \nabla_\mathbf{O}$ of the model to have higher bit-widths. As shown in Figure 7.3, when β is the same ($\beta = 31$ and $\beta = 127$ for the set $\{\mathbf{X}, \mathbf{W}, \mathbf{Q}, \mathbf{K}, \mathbf{M}, \mathbf{V}\}$ and $\{\nabla_\mathbf{Y}, \nabla_\mathbf{P}, \nabla_\mathbf{O}\}$, we see divergence in the middle of training. Alternatively, when

Size	FP32	BF16	$\beta = 255$	$\beta = 31$	$\beta = 15$
Small	1.869	1.868	1.823	1.840	1.891
Base	1.611	-	-	1.601	-

Table 7.8: Training: Validation log perplexity of RoBERTa.

using a larger β for only the set $\{\nabla_{\mathbf{Y}}, \nabla_{\mathbf{P}}, \nabla_{\mathbf{O}}\}$, the loss curve of RTN quantized training is almost identical to FP32 training. Surprisingly, we observed similar results as RoBERTa training: despite marginally higher training loss when using RTN, the validation top-1 accuracy of RTN is higher than FP32 as shown in Figure 7.3 and Table 7.9.

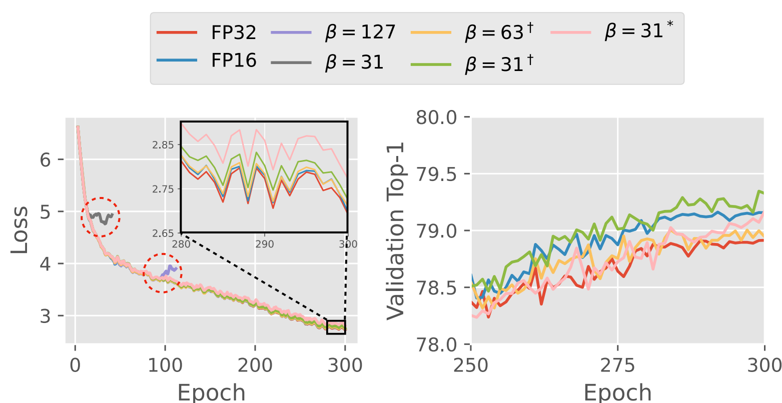

Figure 7.3: Training: Comparison of ViT-Small. † and *: we set $\beta = 16383$ and $\beta = 1023$, respectively, for the set $\{\nabla_{\mathbf{Y}}, \nabla_{\mathbf{P}}, \nabla_{\mathbf{O}}\}$.

FP32	FP16	$\beta = 63^{\dagger}$	$\beta = 31^{\dagger}$	$\beta = 31^{*}$
78.91	79.16	78.94	79.33	79.17

Table 7.9: Training: Validation top-1 accuracy of ViT-Small.

Larger Models. To understand of how well RTN works in the context of training for larger models without using too much compute, we finetune a T5-Large model on the first 50K instance of XSum summarization dataset (Narayan et al., 2018) using BF16 and RTN, and show the results in Figure 7.4. The validation metrics are shown in Table 7.10. We can draw a similar conclusion that RTN quantized training gives similar results as BF16 training for T5-Large finetuning.

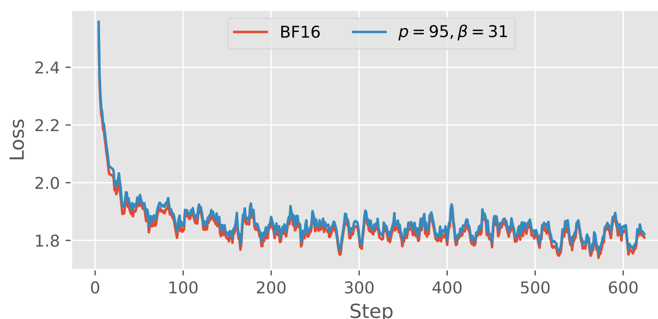

Figure 7.4: Loss curves of T5-Large finetuning on 1/4 of XSum dataset for 1 epoch.

Method	β	Type	Loss	Rouge1	Rouge2	Rougel	Rougelsum
Full-Precision	-	BF16	1.65	36.12	13.00	29.21	29.20
RTN	31	INT	1.66	36.03	13.83	29.03	29.04

Table 7.10: Validation metrics of T5-Large finetuning on 1/4 of XSum dataset for 1 epoch.

7.1.3 Analyzing the Error of Rounding to Nearest (RTN)

We can also perform a simple technical analysis for RTN. Since the scaling in (7.1) and (7.2) will not affect the analysis, we can assume, without loss of generality, that the scaling is simply set to 1. We are interested in the error

$$\mathbf{E} = \mathbf{A}\mathbf{B}^\top - \mathbf{A}_q\mathbf{B}_q^\mathbf{T}$$

We can examine a specific entry in the product $\mathbf{C}$. Let $\mathbf{u}$ and $\mathbf{v}$ be the rows of $\mathbf{A}$ and $\mathbf{B}$, respectively, whose inner product $\sum_i^d \mathbf{u}[i]\mathbf{v}[i]$ is a specific entry in $\mathbf{C}$. In Step 1, each $\mathbf{u}[i]$ and $\mathbf{v}[i]$ is rounded to the nearest integer, $\hat{\mathbf{u}}[i]$ and $\hat{\mathbf{v}}[i]$, resulting in a deviation of at most ± 0.5 from the original entries.

Let the error incurred in this i-th term due to rounding be denoted as

$$\epsilon[i] = \hat{\mathbf{u}}[i]\hat{\mathbf{v}}[i] - \mathbf{u}[i]\mathbf{v}[i]$$

After some calculations, we see that the product $\hat{\mathbf{u}}[i]\hat{\mathbf{v}}[i]$ of the rounded values will

deviate by at most $\pm 0.5(|\mathbf{u}[i]| + |\mathbf{v}[i]| + 0.5)$ from the product $\mathbf{u}[i]\mathbf{v}[i]$ of the original values. Let $\mathbf{c}[i] = 0.5(|\mathbf{u}[i]| + |\mathbf{v}[i]| + 0.5)$, and let c be a vector whose i-th entry is $\mathbf{c}[i]$. Then, it follows that $-\mathbf{c}[i] \leqslant \epsilon[i] \leqslant \mathbf{c}[i]$. We can now examine the total error in the inner product $\sum_i \mathbf{u}[i]\mathbf{v}[i]$.

When $\epsilon[i] > 0$, this term over-contributes, and when $\epsilon[i] < 0$, it under-contributes. Assuming that these two scenarios are equally likely, the cumulative error in $\sum_i \mathbf{u}[i]\mathbf{v}[i]$ can be represented as a sum involving a Rademacher-distributed variable X (half-half chance of being $+1$ or -1), modulated by $\epsilon[i]$ as coefficients. Our interest is in the error $\sum_i \mathbf{x}[i]\epsilon[i]$, where $\mathbf{x}[i]$ is a set of random variables following a Rademacher distribution (Hitczenko and Kwapień, 1994). We want to check whether the probability of a bad event, where this sum (error) exceeds a suitably high threshold, decays quickly as the threshold increases. This is related to Tomaszewski's conjecture (Keller and Klein, 2022), and in particular, we now know that

$$\mathcal{P}(\sum_i \mathbf{x}[i]\epsilon[i] > t \, \|\epsilon\|_2) < \exp(-t^2/2)$$

and

$$\mathcal{P}(|\sum_i \mathbf{x}[i]\epsilon[i]| > t \, \|\epsilon\|_2) < 2\exp(-t^2/2)$$

implying an exponential decay in the probability of the error exceeding a threshold as desired. By incorporating the maximal possible error, denoted as $\mathbf{c}[i]$, we obtain:

$$\mathcal{P}(|\sum_i \mathbf{x}[i]\epsilon[i]| > t \, \|\mathbf{c}\|_2) < 2\exp(-t^2/2)$$

Since $-\mathbf{c}[i] \leqslant \epsilon[i] \leqslant \mathbf{c}[i]$, we can also use Hoeffding's inequality, which is less tight but has a similar form of dependency.

However, the concentration bounds do not adequately explain the strong empirical behavior. To assess impact on the impact on training, if desired, we can use results such as the one in (De Sa et al., 2018) for convergence analysis but with some modifications to the optimization. For example, if we use LP-SVRG in (De Sa

et al., 2018) and just use stochastic rounding instead of deterministic RTN (fixed point arithmetic in (De Sa et al., 2018) is the same as the integer arithmetic with scaling that we use in (7.2)), then under the assumption that the objective function is μ-strongly convex with respect to parameters (and an additional L-Lipschitz requirement), with an additional cost of variance reduction, we can get a linear convergence rate.

7.2 What happens with Low Bit-Width?

Converting floating point to integers alone will *not* provide efficiency benefits. Rather, we want to use a representation that can be efficiently computed (and why low bit-width integers are common in integer quantization). Notice that as a direct consequence of RTN, by (7.1), 95% of values can be represented using β distinct numbers, which requires only $\log_2(\beta + 1)$ bits. For example, if $\beta = 15$, then we can represent these 95% of values with 4-bit signed integers, which is already low bit-width. So, is there still a problem?

It turns out that the challenge involves dealing with the *remaining* 5% of entries. To get a sense of how large these values are, we calculate the ratio $\alpha_{100}(\cdot)/\alpha_{95}(\cdot)$ between the maximum and 95th-percentile of the magnitude of each matrix in GEMMs when performing (a) inference (forward pass) of LLaMA-7B and ViT-Large and (b) training (forward pass and backward pass) of RoBERTa-Small at different training phases. We can check the ratios in Table 7.11 and Table 7.12, respectively. We see extremely large values across both training and inference and across the entire duration of training, so simply increasing the representation bit width of low precision integers by a few more bits will *not* be sufficient to represent these heavy hitters.

We performed experiments studying different ways of handling these heavy hitters when quantizing all GEMMs (linear layers and self-attention computation) in Transformer models. Unless β is inordinately large (based on Table 7.11 and Table 7.12, more than 10^5 times larger than our choice of β for $p = 95\%$), simply ensuring

Model	**X**	**W**	**Q**	**K**	**M**	**V**
LLaMA-7B	141312.0	47.8	8.4	8.1	4448.0	36.2
ViT-Large	284402.4	34.8	4.3	4.3	120.0	8.9

Table 7.11: Maximal ratios between the maximum and 95-percentile of magnitudes of each matrix involved in GEMMs.

Progress	**X**	**W**	∇_Y	**Q**	**K**	∇_P	**M**	**V**	∇_O
1/3	28.7	7.1	292.5	3.7	3.0	309365.2	3924.6	3.1	25.8
2/3	25.7	13.8	235.4	4.2	2.7	283742.8	2283.3	3.3	32.4
3/3	22.0	16.0	290.3	4.0	3.0	218376.0	2018.6	3.4	28.9

Table 7.12: Maximal ratios between the maximum and 95-percentile of magnitudes of each matrix involved in GEMMs during the training of RoBERTa-Small.

that the heavy hitters lie within the representable range of β for $\beta = 255$ or $\beta = 127$ results in a huge performance drop as shown in Table 7.13. On the other hand, clipping the extreme heavy hitters (at the 99.5-percentile) also fails as shown in Table 7.13. Our observations for training are similar – we can see the loss curves for $p = 100\%$, $\beta = 255$ and $p = 95\%$, $\beta = 31$ in Figure 7.2.

	p	β	Clip	ARC-c	ARC-e	BoolQ	HS	PIQA	WG
LLaMA-7B	Full-Precision			43.1	76.3	77.8	57.2	78.0	68.8
	100	255	No	35.8	66.2	57.8	47.4	71.3	63.9
	99.5	∞	Yes	21.4	25.5	60.2	25.8	53.5	49.9
	95	31	No	43.4	75.8	76.8	57.5	77.4	68.4

	p	β	Clip	Tiny	Small	Base	Large	Huge
ViT	Full-Precision			75.5	81.4	85.1	85.8	87.6
	100	127	No	53.9	69.1	72.0	81.6	83.6
	99.5	∞	Yes	11.3	24.1	9.0	15.8	0.6
	95	15	No	71.1	79.8	84.5	85.6	87.5

Table 7.13: Catastrophic performance degradation when restricting outliers to a representable range of quantized domain or clipping the outliers on zero-shot inference of LLaMA-7B and ImageNet classification of quantized ViT models. $p = 100$ means we keep outliers within representable range of β distinct integers. $\beta = \infty$ means that we do not quantize the values. Clip means we clip the values that are larger than p-percentile.

As briefly mentioned earlier, some ideas have been proposed to process these so-called outliers. The approach in (Dettmers et al., 2022) exploits the location structure of where these outliers occur and moves the columns or rows of each matrix (with these outliers) into a different matrix, then GEMM is performed using FP16. The authors in (Xiao et al., 2023) propose to smooth the outliers in activation and mitigate the quantization difficulty via a transformation. This strategy requires specialized GEMM hardware support for different precisions and may even lower the performance as shown in our baseline comparisons in §7.1.1 and §7.1.2.

Goals. We desire an approach that does not alter the results of integer GEMMs; in other words, all results in §7.1.1 and §7.1.2 must remain exactly the same, yet we should not need calculations using different precisions. This may appear unrealistic but our simple procedure, IM-Unpack, allows representing heavy hitters using low bit-width integers. Calculations are carried out using low bit-width integer arithmetic. Specifically, IM-Unpack unpacks a matrix containing heavy hitters into a *larger* unpacked matrix (we study how large the expansion will be in §7.3.2) whose values are all representable by low bit-width integers. IM-Unpack obtains the exact output of the original GEMM using purely low bit-width integer GEMMs on these unpacked matrices.

7.3 IM-Unpack: Integer Matrix Unpacking

Our approach starts with a simple observation that, for example, a 32-bit integer v can be represented as

$$v = v_0 + 128v_1 + 128^2v_2 + 128^3v_3 + 128^4v_4$$

where v_i are 8-bit integers. Multiplication/addition of two 32-bit integers can be performed on these decomposed 8-bit integers followed by some post-processing steps (scaling via bit shifting and accumulation). This unpacking does enable performing high bit-width arithmetic using lower bit-widths, but it achieves this at the cost of requiring more operations. For example, one 32-bit addition now becomes

five 8-bit additions with some follow up processing, and one 32-bit multiplication becomes twenty five 8-bit multiplications (distributive law).

Remark 7.1. *The reason why this unpacking is still useful is because the additional work depends on the number and spatial distribution of the heavy hitters/outliers. We harvest gains because outliers account for a very small portion of the matrices that appear in practice in training/inference stages of Transformer models. Exploitation of the sparsity of outliers in quantization can also be found in Dettmers et al. (2022); Xiao et al. (2023).*

Let b be the target bit-width of low bit-width integers and $s = 2^{b-1}$ be the representable range of bit-width b: b-bit integers can represent a set $\{-s+1, \cdots, 0, \cdots, s-1\}$. We refer to any integers *inside* of this set as In-Bound (IB) values and any integers *outside* of this set as Out-of-Bound (OB) values, which will be used in later discussion to refer to the values that need to be unpacked. We will first show how to unpack a vector to multiple low bit-width vectors. Then, we will discuss how to unpack a matrix using different strategies to achieve better results in different cases in §7.3.1. Lastly, we will evaluate how well IM-Unpack works in §7.3.2, and provide an end to end quantization baseline comparison and discuss model speedup in §7.3.3 when employing our IM-Unpack as supplement to §7.1.

Unpacking an integer vector. Let $\mathbf{v}$ be an integer vector and define a function:

$$m(\mathbf{v}, s, i) = \text{floor}(\mathbf{v}/s^i) \text{ mod } s \tag{7.3}$$

such that for all i, all entries of $m(\mathbf{v}, s, i)$ are bounded (IB), i.e., lie in the interval $[-s+1, s-1]$. When s is clear from the context, we shorten the LHS of (7.3) to just $m(\mathbf{v}, i)$. Then,

$$\mathbf{v} = \sum_{i=0}^{\infty} s^i m(\mathbf{v}, i) \tag{7.4}$$

Note that $\mathbf{v}/s^i$ decreases to 0 exponentially fast, so we are able to unpack a vector with just a few low bit-width vectors.

7.3.1 Variants of Matrix Unpacking

In this subsection, we discuss different strategies of matrix unpacking for different structure-types of matrices. First, we discuss the case where **A** is the matrix containing OB values to be unpacked and **B** is a matrix whose values are all IB. Next, we discuss how unpacking works when both **A** and **B** contains OB values.

Unpacking row vectors. We start with the simplest means of unpacking a matrix: unpacking the row vectors. Given a matrix **A**, if one row of **A** contains OB values, we can unpack the row to multiple rows whose entries are all bounded. The exact procedure is described in Algorithm 5 and illustrated in Figure 7.5. In Figure 7.5, when the second row in **A** contains OB values, we can unpack it to two row vectors (the second and fifth row) and the post-processing step takes the form of applying $\mathbf{\Pi}_A$ to the unpacked matrix $\mathbf{A}_u$.

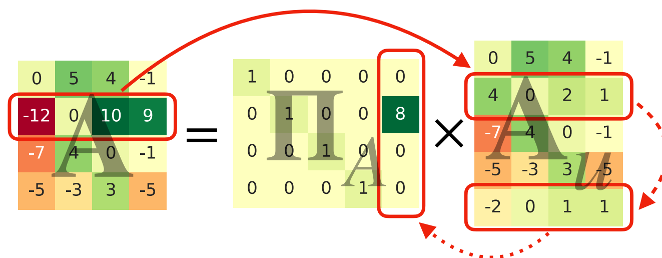

Figure 7.5: Illustration of unpacking row vectors. The solid, dashed, and dotted arrows correspond to lines 5, 4, and 6 in Algorithm 5

Reconstructing A. **A** can be reconstructed using the unpacked matrix $\mathbf{A}_u$ whose entries are IB and a sparse matrix $\mathbf{\Pi}$ whose column contains *only one* non-zero:

$$\mathbf{A}_u, \mathbf{\Pi}_A = \text{UnpackRow}(\mathbf{A}, b)$$
$$\mathbf{A} = \mathbf{\Pi}_A \mathbf{A}_u$$

Here, applying $\mathbf{\Pi}_A$ to $\mathbf{A}_u$ can be efficiently computed easily (for example, via `torch.index_add`).

Algorithm 5 UnpackRow$(\mathbf{A}, b)$

1: Let $\mathbf{\Pi} \leftarrow \mathbf{I}$ and $s \leftarrow 2^{b-1}$ and $i \leftarrow 0$
2: **while** $\mathbf{A}[i,:]$ exists **do**
3:   **if** $\mathbf{A}[i,:]$ contains OB entries **then**
4:     Append floor$(\mathbf{A}[i,:]/s)$ as a new row to $\mathbf{A}$
5:     $\mathbf{A}[i,:] \leftarrow \mathbf{A}[i,:]$ mod s
6:     Append $s\mathbf{\Pi}[:,i]$ as a new column to $\mathbf{\Pi}$
7:   **end if**
8:   $i \leftarrow i+1$
9: **end while**
10: **return** $\mathbf{A}, \mathbf{\Pi}$

Are we done? If we do not care about maximizing efficiency, then the above scheme already provides a way to perform high bit-width GEMM using low bit-width GEMM. However, this might not be the optimal unpacking strategy for some matrices. For example, consider the left matrix shown in Figure 7.6. Since every row of this matrix contains OB values, every row needs to be unpacked, resulting in a much larger matrix. In this case, it might be better to try and unpack the column vectors. Let us apply a similar idea of unpacking row vectors to unpack column vectors of $\mathbf{A}$:

$$\begin{aligned} \mathbf{A} &= \mathbf{A}'_u \mathbf{\Pi}'_A \\ \mathbf{A}\mathbf{B}^\top &= \mathbf{A}'_u \mathbf{\Pi}'_A \mathbf{B}^\top \end{aligned} \tag{7.5}$$

While unpacking column vectors is reasonable, the sparse matrix $\mathbf{\Pi}'_A$ creates an issue when performing a GEMM of two lower bit-width matrices: $\mathbf{\Pi}'_A$ has to be applied to $\mathbf{A}'_u$ or $\mathbf{B}^\top$ before GEMM, but the result/output may contain OB entries after the application, disabling low bit-width integer GEMM. This problem is similar to per-channel quantization. It is not simple to handle and becomes more involved when $\mathbf{B}$ also need to be unpacked.

Unpacking column vectors. Alternatively, let us look at how $\mathbf{A}\mathbf{B}^\top$ is computed via

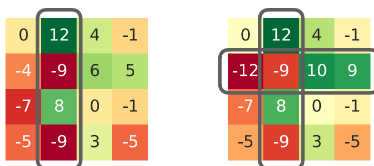

Figure 7.6: Left: Failure case for unpacking rows. Right: Failure case for unpacking rows or columns alone.

outer product of column vectors:

$$\mathbf{C} = \mathbf{A}\mathbf{B}^\top = \sum_{i=1}^{D} \mathbf{A}[:,i]\mathbf{B}[:,i]^\top$$

Let us look at the i-th outer product. Let us try unpacking $\mathbf{A}[:,i]$ using (7.4), then we have

$$\mathbf{A}[:,i]\mathbf{B}[:,i]^\top = \sum_{j=0}^{\infty} s^j m(\mathbf{A}[:,i],j)\mathbf{B}[:,i]^\top$$

Suppose that $m(\mathbf{A}[:,i],j) = 0$ for $j \geqslant K$, then we can unpack one outer product to K outer products. This is equivalent to appending $m(\mathbf{A}[:,i],j)$ for $0 \leqslant j < K$ to the columns of $\mathbf{A}$, appending K identical $\mathbf{B}[:,i]$ to the columns of $\mathbf{B}$, and maintaining a diagonal matrix to keep track of the scaling factor s^j. The exact procedure is described in Algorithm 6, and Figure 7.7 shows a visualization of unpacking columns. Using column unpacking, we have

$$\begin{aligned} \mathbf{A}_u, \mathbf{B}_e, \mathbf{S}_u &= \text{UnpackColumn}(\mathbf{A}, \mathbf{B}, \mathbf{I}, b) \\ \mathbf{A}\mathbf{B}^\top &= \mathbf{A}_u\mathbf{S}_u\mathbf{B}_e^\top \end{aligned}$$

Naively, this still suffers from the same problem as discussed in (7.5) in that there is a diagonal scaling matrix between two low bit-width matrices making low bit-width GEMMs difficult. However, since $\mathbf{S}_u$ is a diagonal matrix whose diagonal entries consist of a few distinct factors in $\{1, s, s^2, ...\}$, we can easily compute one GEMM for

each distinct diagonal entry as shown in Algorithm 7.

$$\mathbf{A}\mathbf{B}^\top = \text{ScaledMatMul}(\mathbf{A}_u, \mathbf{B}_e, \mathbf{S}_u)$$

Further, since s is a power of 2, the scaling can be efficiently implemented via bit shifting.

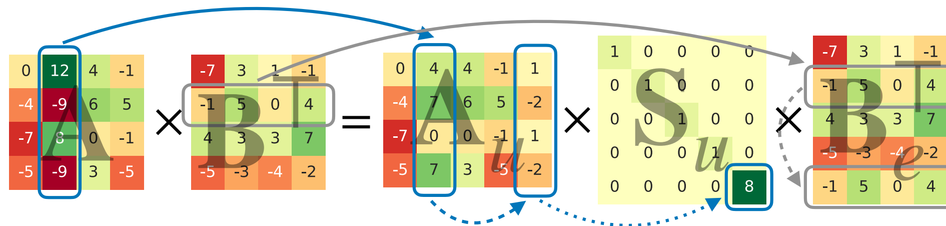

Figure 7.7: Illustration of unpacking column vectors. The blue solid, dashed, and dotted arrows correspond to lines 5, 4, and 7 in Algorithm 5, and the gray dashed arrow corresponds to line 6 in Algorithm 5.

Algorithm 6 UnpackColumn($\mathbf{A}, \mathbf{B}, \mathbf{S}, b$)

1: Let $s \leftarrow 2^{b-1}$ and $i \leftarrow 0$
2: **while** $\mathbf{A}[:, i]$ exists **do**
3:     **if** $\mathbf{A}[:, i]$ contains OB entries **then**
4:         Append floor($\mathbf{A}[:, i]/s$) as a new column to $\mathbf{A}$
5:         $\mathbf{A}[:, i] \leftarrow \mathbf{A}[:, i] \bmod s$
6:         Append $\mathbf{B}[:, i]$ as a new column to $\mathbf{B}$
7:         Append $s\mathbf{S}[i, i]$ as a new diagonal entry to $\mathbf{S}$
8:     **end if**
9:     $i \leftarrow i + 1$
10: **end while**
11: **return** $\mathbf{A}, \mathbf{B}, \mathbf{S}$

Are we done yet? Unpacking columns is efficient for the left matrix shown in Figure 7.6. However, neither unpacking rows nor unpacking columns will be efficient for unpacking the right matrix shown in Figure 7.6. All rows and columns contains OB values. Unpacking rows or columns alone will not be ideal. For the right matrix in

Algorithm 7 ScaledMatMul($\mathbf{A}, \mathbf{B}, \mathbf{S}$)

1: Let $\mathbf{C} \leftarrow 0$
2: **for all** distinct diagonal entry s^i in $\mathbf{S}$ **do**
3: Let $\mathcal{J}$ be the index set where $\mathbf{S}[j, j] = s^i$ for $j \in \mathcal{J}$
4: $\mathbf{C} \leftarrow \mathbf{C} + s^i \mathbf{A}[:, \mathcal{J}] \mathbf{B}[:, \mathcal{J}]^\top$
5: **end for**
6: **return C**

Figure 7.6, a better strategy is to unpack the second row and the second column simultaneously.

Unpacking both rows and columns simultaneously. Our final strategy combines row and column unpacking together and selectively performs row unpack or column unpack based on the number of OB values that can be eliminated. The procedure is described in Algorithm 8, and we provide an illustration of unpacking both dimensions in Figure 7.8. With this procedure, we can obtain the output of high bit-width GEMM using low bit-width as:

$$\begin{aligned} \mathbf{A}_u, \mathbf{B}_e, \mathbf{S}_u, \mathbf{\Pi}_A &= \text{UnpackBoth}(\mathbf{A}, \mathbf{B}, \mathbf{I}, b) \\ \mathbf{A}\mathbf{B}^\top &= \mathbf{\Pi}_A \mathbf{A}_u \mathbf{S}_u \mathbf{B}_e^\top \end{aligned} \tag{7.6}$$

Here, $\mathbf{A}_u \mathbf{S}_u \mathbf{B}^\top$ can be calculated via Algorithm 7, and applying $\mathbf{\Pi}_A$ can be carried out efficiently as discussed.

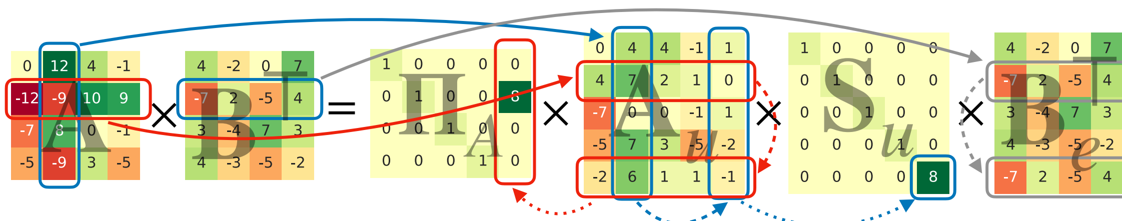

Figure 7.8: Illustration of unpacking both rows and columns based on the OOB counts. The red solid, dashed, and dotted arrows correspond to lines 8, 7, and 9 in Algorithm 8. The blue solid, dashed, and dotted arrows correspond to lines 12, 11, and 14 in Algorithm 8, and the gray dashed arrow corresponds to line 13 in Algorithm 8.

Algorithm 8 UnpackBoth($\mathbf{A}, \mathbf{B}, \mathbf{S}, b$)

1: Let $s \leftarrow 2^{b-1}$ and
2: **while** True **do**
3: Let $(c_0, i), (c_1, j)$ be the tuples of top OB count in row/column vectors and corresponding index
4: **if** $c_0 = 0$ and $c_1 = 0$ **then**
5: **break**
6: **else if** $c_0 \geqslant c_1$ **then**
7: Append floor$(\mathbf{A}[i, :]/s)$ as a new row to $\mathbf{A}$
8: $\mathbf{A}[i, :] \leftarrow \mathbf{A}[i, :] \bmod s$
9: Append $s\mathbf{\Pi}[:, i]$ as a new column to $\mathbf{\Pi}$
10: **else**
11: Append floor$(\mathbf{A}[:, j]/s)$ as a new column to $\mathbf{A}$
12: $\mathbf{A}[:, j] \leftarrow \mathbf{A}[:, j] \bmod s$
13: Append $\mathbf{B}[:, j]$ as a new column to $\mathbf{B}$
14: Append $s\mathbf{S}[j, j]$ as a new diagonal entry to $\mathbf{S}$
15: **end if**
16: **end while**
17: **return** $\mathbf{A}, \mathbf{B}, \mathbf{S}, \mathbf{\Pi}$

Combining everything. Since we have different strategies for unpacking, let us first define a unified interface in Algorithm 9. One can verify that for any strategy s_A:

$$\begin{aligned} \mathbf{A}_u, \mathbf{B}_e, \mathbf{S}_u, \mathbf{\Pi}_A &= \text{Unpack}(\mathbf{A}, \mathbf{B}, \mathbf{I}, b, s_A) \\ \mathbf{A}\mathbf{B}^\top &= \mathbf{\Pi}_A \mathbf{A}_u \mathbf{S}_u \mathbf{B}_e^\top \end{aligned} \tag{7.7}$$

In the previous discussion, $\mathbf{B}$ was assumed to have all IB values. When $\mathbf{B}$ contains OB values, we note that $\mathbf{B}$ can be unpacked in a similar manner, and the choice of unpacking strategies for $\mathbf{B}$ is independent of the unpacking strategy for $\mathbf{A}$. For example, $\mathbf{A}$ can be unpacked row-wise, while $\mathbf{B}$ is unpacked column-wise. By taking the unpacked $\mathbf{A}_u, \mathbf{B}_e, \mathbf{S}_u, \mathbf{\Pi}_A$ from (7.7), we can further unpack $\mathbf{B}$ using strategy s_B:

$$\begin{aligned} \mathbf{B}_{eu}, \mathbf{A}_{ue}, \mathbf{S}_{uu}, \mathbf{\Pi}_B &= \text{Unpack}(\mathbf{B}_e, \mathbf{A}_u, \mathbf{S}_u, b, s_B) \\ \mathbf{A}\mathbf{B}^\top &= \mathbf{\Pi}_A \mathbf{A}_{ue} \mathbf{S}_{uu} \mathbf{B}_{eu}^\top \mathbf{\Pi}_B^\top \end{aligned}$$

Here, values in both $\mathbf{A}_{ue}$ and $\mathbf{B}_{eu}$ are IB, and the result can be obtained similar to discussion in (7.6).

As we noted during the discussion, $\mathbf{\Pi}_A, \mathbf{\Pi}_B, \mathbf{S}_{uu}$ are special sparse matrices. $\mathbf{\Pi}_A$ and $\mathbf{\Pi}_B$ are sparse matrices whose column contains only one non-zero, and $\mathbf{S}_{uu}$ are diagonal matrices whose diagonal entries consist of a few distinct factors. As a result, $\mathbf{A}_{ue}\mathbf{S}_{uu}\mathbf{B}_{eu}^\top$ is computed via Algorithm 7 using GEMMs on $\mathbf{A}_{ue}$ and $\mathbf{B}_{eu}$, and applying $\mathbf{\Pi}_A$ and $\mathbf{\Pi}_B$ is possible simply via `torch.index_add`.

Algorithm 9 Unpack($\mathbf{A}, \mathbf{B}, \mathbf{S}, b$, strategy)

1: **if** strategy is UnpackRow **then**
2: $\quad \mathbf{A}_u, \mathbf{\Pi}_A \leftarrow$ UnpackRow($\mathbf{A}, b$)
3: $\quad \mathbf{S}_u, \mathbf{B}_e \leftarrow \mathbf{S}, \mathbf{B}$
4: **else if** strategy is UnpackColumn **then**
5: $\quad \mathbf{A}_u, \mathbf{B}_e, \mathbf{S}_u \leftarrow$ UnpackColumn($\mathbf{A}, \mathbf{B}, \mathbf{S}, b$)
6: $\quad \mathbf{\Pi}_A \leftarrow \mathbf{I}$
7: **else**
8: $\quad \mathbf{A}_u, \mathbf{B}_e, \mathbf{S}_u, \mathbf{\Pi}_A \leftarrow$ UnpackBoth($\mathbf{A}, \mathbf{B}, \mathbf{S}, b$)
9: **end if**
10: **return** $\mathbf{A}_u, \mathbf{B}_e, \mathbf{S}_u, \mathbf{\Pi}_A$

Summary. We introduced three strategies to unpack a matrix to low bit-width integer matrices for different structures of OB values in a matrix. While these strategies work for arbitrary matrices, we can clearly see that these unpacking strategies are most efficient when the OB values concentrate in a few columns and rows. Luckily, the matrices of interest in Transformer models indeed have this property, which is studied and exploited in several works (Dettmers et al., 2022; Xiao et al., 2023).

7.3.2 Evaluating Unpacking Overhead

The idea of IM-Unpack is to use more low bit-width arithmetic operations to compute a high bit-width operation. As we see in the description of IM-Unpack algorithm, the number of row and column vectors will increase, so the unpacked matrices $\mathbf{A}_{ue}$ and $\mathbf{B}_{eu}$ are a larger in terms of size compared to $\mathbf{A}$ and $\mathbf{B}$, which

obviously increases the computational cost of low bit-width GEMMs. In this subsection, we evaluate how much this cost will increase. For two matrices $\mathbf{A}$ and $\mathbf{B}$, the complexity of a GEMM is $\mathcal{O}(NDH)$. Similarly, let N', D' be the size of $\mathbf{A}_{ue}$ and H' be the number of rows of $\mathbf{B}_{eu}$. The cost of $\mathbf{A}_{ue}\mathbf{S}_{uu}\mathbf{B}_{eu}^{\top}$ is $\mathcal{O}(N'D'H')$, we can directly measure the unpack ratio

$$r = \frac{N'D'H'}{NDH}$$

to understand by how much the cost for low bit-width GEMMs increases. We use LLaMA-7B to study the unpack ratio r when using different unpacking strategies (Table 7.14). Note that since unpacking both requires keeping track of the OB count in each row and column vector which is not as fast as the other two strategies, we only use it for unpacking parameters $\mathbf{W}$ for inference since it can be performed once when loading the model. The Mix in Table 7.14 means that for each GEMM, we compare different strategies and choose the optimal strategy that results in the smallest unpack ratio. We note that the unpack ratios of computing $\mathbf{Y}$ and $\mathbf{P}$ are quite reasonable, but the ratios of computing $\mathbf{O}$ is larger. This is expected since the large outliers of the self-attention matrix $\mathbf{M}$ mainly concentrate in the diagonal (Beltagy et al., 2020). Similarly, we also evaluate the unpack ratios of ViT-Large, which are shown in Table 7.15. The overall results are similar to what was observed in unpack ratios of LLaMA-7B (Table 7.14).

We also study the unpack ratios of each type of quantized GEMMs at different training phases, and show the results of Mix strategy in Table 7.16. The ratios stay relatively unchanged as training progresses. Also, we can observe similarly high unpack ratio when computing $\mathbf{O}$ and $\nabla_{\mathbf{V}}$ since these GEMMs involve the self-attention matrix $\mathbf{M}$. Lastly, we verify that we can unpack matrices to arbitrarily low integer matrices (Table 7.17). The 2-bit setting is the lowest bit width that can be used for symmetric signed integers $\{-1, 0, 1\}$.

Can We Use as Low as 1-bit Encoding? As discussed, we use 2 bits to encode $\{-1, 0, +1\}$ for symmetric encoding, so the 2 bit setting is the lowest bit width that

β					5			15			31		
Integer Bits b					3	4	5	4	5	6	5	6	7
Linear (**Y**)	**X**	Row	**W**	Row	2.67	1.93	1.57	2.47	2.02	1.73	2.12	2.00	1.74
		Row		Col	10.76	2.35	1.61	9.91	5.36	1.84	8.44	5.62	1.86
		Row		Both	5.46	1.95	1.57	5.15	2.20	1.73	4.71	2.24	1.75
		Col		Row	3.80	1.32	1.06	3.98	1.64	1.16	3.93	1.68	1.17
		Col		Col	15.40	1.62	1.09	16.00	4.01	1.25	15.69	4.33	1.27
		Col		Both	5.21	1.34	1.06	6.04	1.76	1.16	5.98	1.82	1.17
	Mix				2.6	1.27	1.06	2.44	1.4	1.15	2.1	1.42	1.16
AS (**P**)	**Q**	Row	**K**	Row	1.97	1.60	1.0	2.00	1.87	1.15	2.00	1.87	1.18
		Row		Col	3.22	1.64	1.0	5.35	2.07	1.17	5.36	2.09	1.20
		Col		Row	1.81	1.04	1.0	2.91	1.14	1.01	2.91	1.15	1.01
		Col		Col	3.36	1.08	1.0	8.66	1.32	1.03	8.67	1.35	1.03
	Mix				1.72	1.03	1.0	1.95	1.13	1.01	1.95	1.14	1.01
AO (**O**)	**M**	Row	**V**	Row	6.02	4.18	3.27	4.72	3.65	3.02	3.93	3.24	2.81
		Row		Col	15.10	4.53	3.35	18.21	4.64	3.16	15.07	4.21	2.95
		Col		Row	16.29	8.14	5.12	11.28	6.98	4.91	8.41	5.84	4.42
		Col		Col	42.21	8.76	5.21	43.57	9.11	5.09	32.31	7.74	4.61
	Mix				5.98	4.11	3.16	4.7	3.62	2.97	3.92	3.22	2.77

Table 7.14: Averaged unpack ratios of each type of GEMMs in LLaMA-7B: linear layers (computing $\mathbf{Y}$), attention score (computing $\mathbf{P}$), and attention output (computing $\mathbf{O}$) when using different unpack strategies and integer bit-width b under quantization β settings. AS: Attention Score, AO: Attention Output.

our method supports. However, with small adjustments, it is possible in principle to derive a 1-bit encoding scheme for this 2-bit encoding. Given $\mathbf{A}$ and $\mathbf{B}$ matrices whose values are $\{-1, 0, +1\}$, we can easily decompose

$$\mathbf{A} = \mathbf{A}_p - \mathbf{A}_n$$
$$\mathbf{B} = \mathbf{B}_p - \mathbf{B}_n$$

where $\mathbf{A}_p, \mathbf{A}_n, \mathbf{B}_p, \mathbf{B}_n$ consist of values $\{0, 1\}$. Then,

$$\mathbf{A}\mathbf{B}^\top = \mathbf{A}_p\mathbf{B}_p^\top - \mathbf{A}_n\mathbf{B}_p^\top - \mathbf{A}_n\mathbf{B}_p^\top + \mathbf{A}_n\mathbf{B}_n^\top$$

becomes four 1-bit GEMMs, so 1-bit encoding is feasible in this sense.

One benefit of 1 bit quantization is that multiplication in GEMM becomes logic AND and accumulation becomes counting the number of 1's. Notice that similar benefits

		β			5			7			15		
		Integer Bits b			3	4	5	3	4	5	4	5	6
Linear (Y)	**X**	Row	**W**	Row	2.90	2.00	1.55	3.01	2.38	1.59	2.63	2.22	1.54
		Row		Col	10.97	2.32	1.56	12.34	4.12	1.65	10.46	4.31	1.62
		Row		Both	6.24	2.08	1.55	6.84	2.82	1.60	6.22	2.76	1.56
		Col		Row	2.33	1.39	1.20	3.38	1.51	1.26	3.06	1.46	1.25
		Col		Col	8.89	1.64	1.22	13.97	2.63	1.32	12.22	2.81	1.32
		Col		Both	4.99	1.44	1.20	7.99	1.76	1.27	7.59	1.78	1.26
		Mix			2.60	1.27	1.06	2.44	1.40	1.15	2.10	1.42	1.16
AS (P)	**Q**	Row	**K**	Row	1.84	1.07	1.00	2.01	1.35	1.00	1.99	1.40	1.00
		Row		Col	3.06	1.07	1.00	6.38	1.39	1.00	6.34	1.46	1.00
		Col		Row	1.34	1.01	1.00	2.50	1.04	1.00	2.49	1.05	1.00
		Col		Col	2.39	1.01	1.00	8.25	1.08	1.00	8.24	1.10	1.00
		Mix			1.33	1.01	1.00	1.91	1.04	1.00	1.90	1.04	1.00
AO (O)	**M**	Row	**V**	Row	2.84	2.07	1.65	3.07	2.24	1.80	2.56	2.11	1.78
		Row		Col	5.78	2.12	1.65	11.12	2.47	1.81	9.22	2.35	1.79
		Col		Row	3.98	2.26	1.64	4.69	2.57	1.81	3.58	2.33	1.77
		Col		Col	8.42	2.29	1.64	16.97	2.83	1.81	12.92	2.60	1.77
		Mix			2.25	1.61	1.32	2.55	1.77	1.42	2.22	1.70	1.42

Table 7.15: Averaged unpack ratios of each type of GEMMs in ViT-Large: linear layers (computing **Y**), attention score (computing **P**), and attention output (computing **O**) when using different unpack strategies and integer bit length b under quantization β settings. AS: Attention Score, AO: Attention Output.

	Progress	1/3			2/3			3/3		
	Integer Bits b	5	6	7	5	6	7	5	6	7
Linear	$\mathbf{Y}$	2.00	1.31	1.08	2.00	1.32	1.07	2.00	1.32	1.05
	$\nabla_{\mathbf{X}}$	1.50	1.31	1.15	1.50	1.30	1.16	1.50	1.30	1.15
	$\nabla_{\mathbf{W}}$	1.98	1.25	1.04	1.98	1.25	1.03	1.98	1.25	1.03
AS	$\mathbf{P}$	1.66	1.04	1.00	1.42	1.05	1.0	1.40	1.04	1.00
	$\nabla_{\mathbf{Q}}$	2.22	1.90	1.71	2.22	1.91	1.7	2.24	1.92	1.71
	$\nabla_{\mathbf{K}}$	1.79	1.06	1.00	1.49	1.07	1.0	1.45	1.07	1.00
AO	$\mathbf{O}$	3.11	2.71	2.30	3.10	2.68	2.24	3.10	2.62	2.22
	$\nabla_{\mathbf{M}}$	1.21	1.10	1.04	1.21	1.10	1.04	1.21	1.10	1.04
	$\nabla_{\mathbf{V}}$	2.88	2.52	2.12	2.87	2.48	2.10	2.86	2.41	2.09

Table 7.16: Averaged unpack ratios of each type of quantized GEMMs in both forward and backward of a RoBERTa-Small when using different integer bit length b at different training phrases of the $\beta = 31$ experiment in Figure 7.2. The optimal strategy (Mix as in Table 7.14) for each GEMM is used.

				2	3	4	5	6	7
Integer Bits b				2	3	4	5	6	7
X	Row	**W**	Row	7.24	3.80	2.63	2.22	1.54	1.43
	Row		Col	194.89	27.52	10.46	4.31	1.62	1.43
	Row		Both	85.92	13.80	6.22	2.76	1.56	1.43
	Col		Row	19.27	4.85	3.06	1.46	1.25	1.12
	Col		Col	526.31	35.86	12.22	2.81	1.32	1.13
	Col		Both	27.06	13.31	7.59	1.78	1.26	1.13
	Both		Row	7.62	3.39	2.58	1.64	1.42	1.33
	Both		Col	206.32	24.45	10.27	3.15	1.49	1.34
	Both		Both	79.19	11.16	6.09	2.01	1.43	1.34
Mix				6.29	2.98	2.24	1.40	1.23	1.11

Table 7.17: Averaged ratios of quantized GEMMs ($\beta = 15$) in linear layers on ViT-Large when using different strategies and a range of integer bit-widths b to the lowest bit-width possible.

can be exploited in the 2-bit encoding for $\{-1, 0, +1\}$. The mapping between 2-bit binary representation and decimal numbers is $00_b = 0, 01_b = 1, 10_b = 0, 11_b = -1$. We call the left bit a sign bit and right bit a value bit. We note that the multiplication becomes logical XOR in the sign bit and logical AND in the value bit, and accumulation becomes counting the number of 1's in value bit and then subtracting the number of 11_b. We have not performed experiments evaluating this approach yet.

7.3.3 End-to-End Quantization Baseline Comparison

An end-to-end baseline comparison requires combining information from Table 7.2, Table 7.3, and Table 7.4 with Table 7.14 and Table 7.15. As we change the bit-width of integers that we will use, the unpack ratio r can be determined from Table 7.14 and 7.15. Since after unpacking, the calculation is similar or the same as standard GEMMs (except that the input sizes are different), we could first estimate the runtime of GEMMs, and discuss the runtime overhead of unpacking later.

Note that INT2, INT3, INT5, INT6, INT7 GEMM implementations are not yet publicly available. Although INT4 and INT8 are now available in NVIDIA's CUTLASS, their integration (e.g., in PyTorch) is ongoing (more discussion below). As a result, we can only estimate the runtime of these GEMMs based on publicly available information. According to NVIDIA (a), INT4 offers a 4× performance bump compared

Method	β	Type	r	bits × r/16	ARC-c	ARC-e	BoolQ	HS	PIQA	WG
Full-Precision	-	BF16	1	1	43.1	76.3	77.8	57.2	78.0	68.8
LLM.int8()	-	INT8+FP16	1	> 0.5	43.8	75.5	77.8	57.4	77.6	68.7
SmoothQuant	-	INT8	1	0.5	37.4	74.4	74.0	55.0	77.5	69.6
LLM-QAT	-	INT4	1	0.25	30.2	50.3	63.5	55.6	64.3	52.9
LLM-FP4	-	FP4	1	0.25	33.6	65.9	64.2	47.8	73.5	63.7
RTN+IMUnpack	5	INT4	1.27	0.318	39.3	72.8	69.9	53.4	74.9	66.4
	5	INT5	1.06	0.331	39.3	72.8	69.9	53.4	74.9	66.4
	7	INT4	1.41	0.352	42.6	73.9	72.3	55.9	77.0	67.4
	7	INT5	1.14	0.356	42.6	73.9	72.3	55.9	77.0	67.4
	11	INT5	1.27	0.397	43.9	76.1	77.3	56.3	77.3	69.3
	11	INT6	1.07	0.401	43.9	76.1	77.3	56.3	77.3	69.3
	15	INT5	1.40	0.438	43.0	75.7	77.5	57.0	78.0	69.2
	15	INT6	1.15	0.431	43.0	75.7	77.5	57.0	78.0	69.2
	31	INT6	1.42	0.532	42.7	76.1	76.1	57.3	77.3	69.3
	31	INT7	1.16	0.507	42.7	76.1	76.1	57.3	77.3	69.3

Table 7.18: Inference: end to end baseline comparison combining information from Table 7.3 and Table 7.14 for LLaMA-7B.

to FP16, and INT8 is 2× performance compared to FP16. We measured the runtime of INT4 and INT8 GEMMs in CUTLASS standalone, and the performance is indeed 4× and 2×, respectively, compared to FP16. We can reasonably assume that the performance of x-bit INT GEMMs can approach $\frac{16}{x}$ performance compared to FP16, and the runtime approaches $\frac{x}{16}$ of the runtime of FP16. Further, the runtime of GEMMs is linearly proportional to $n \times d \times h$ for matrices of size $n \times d$ and $h \times d$. When comparing runtime of GEMMs for different sizes, the ratio $r = \frac{n \times d \times h}{n' \times d' \times h'}$ can give an accurate estimate of the runtime comparison when these GEMMs finally become available (we verified this linear relationship in FP16 GEMMs for a contiguous range of matrix sizes, not just a power of 2). As a result, we can use $\frac{xr}{16}$ as a proxy for the runtime of our GEMMs. In Table 7.18, we provided $\frac{xr}{16}$ values (bits × r/16 column) for comparing across different baselines and the corresponding model accuracy. We see that our method has better model accuracy with faster runtimes under the proposed proxy when not accounting for the runtime overhead of quantization.

The only remaining task is to check the runtime overhead of quantization and unpacking. Since the goal is to speed up GEMMs, the quantization overhead must be small or else, the overhead will eliminate the gain of faster GEMMs. As a

Shape $n \times d \times h$	GEMM FP32	GEMM FP16	Clone FP32	UnpackCol FP32						UnpackRow FP32					
Bit Width	-	-	-	2		4		8		2		4		8	
Unpack Ratio	-	-	-	1.21	1.44	1.21	1.44	1.21	1.44	1.21	1.44	1.21	1.44	1.21	1.44
$2^{13} \times 2^{13} \times 2^{13}$	63.4	5.0	0.8	1.0	0.9	1.0	1.0	1.1	1.1	1.1	1.1	1.0	1.0	1.0	1.0
$2^{13} \times 2^{13} \times 2^{14}$	116.9	10.4	1.2	1.4	1.3	1.4	1.4	1.5	1.5	1.5	1.4	1.4	1.4	1.5	1.5
$2^{13} \times 2^{13} \times 2^{15}$	252.7	25.1	2.0	2.2	2.2	2.2	2.2	2.4	2.5	2.5	2.2	2.3	2.3	2.4	2.4
$2^{14} \times 2^{13} \times 2^{13}$	126.3	10.5	1.2	1.4	1.4	1.4	1.4	1.5	1.5	1.5	1.4	1.4	1.4	1.5	1.5
$2^{14} \times 2^{13} \times 2^{14}$	253.8	20.9	1.6	1.8	1.8	1.8	1.8	2.0	2.0	2.0	1.8	1.8	1.9	1.9	2.0
$2^{14} \times 2^{13} \times 2^{15}$	510.8	60.1	2.4	2.6	2.6	2.6	2.7	2.9	3.0	3.0	2.6	2.7	2.7	2.9	2.9
$2^{15} \times 2^{13} \times 2^{13}$	255.5	20.9	2.0	2.2	2.2	2.2	2.2	2.4	2.5	2.4	2.2	2.3	2.3	2.4	2.5
$2^{15} \times 2^{13} \times 2^{14}$	514.3	60.0	2.4	2.6	2.6	2.6	2.7	2.9	2.9	3.0	2.6	2.7	2.7	2.9	2.9
$2^{15} \times 2^{13} \times 2^{15}$	957.7	121.5	3.2	3.4	3.4	3.5	3.5	3.8	3.9	3.9	3.5	3.6	3.6	3.8	3.9

Table 7.19: Runtime overhead of UnpackColumn and UnpackRow compared to GEMM and Clone for different bit-widths, unpack ratios r, and input matrix shapes.

result, the code needs to be implemented as custom CUDA kernels to minimize the overhead. We implemented UnpackRow and UnpackCol cuda kernels for FP32 matrices A and B. The supported unpack bit widths are 2, 4, 8 by packing the bit representation of 16 INT2 or 8 INT4 or 4 INT8 into an INT32. For other bit-widths, an efficient implementation would need additional hardware support. We expect the runtime of these kernels would be halved if the input date type is FP16.

We compare these kernels to the `tensor.clone()` operation. The clone operation is the simplest pytorch operation that copies the values of a memory chunk (tensor) to another memory chunk (new tensor). The scaling and rounding operations in common quantization methods share a similar runtime as clone operation. We profiled UnpackRow and UnpackCol kernels (scaling and rounding are also computed within the kernel) for matrices of different sizes, bit-widths, and unpack ratio r. The runtime overhead of quantization and unpacking together is only between $1\times$ to $1.5\times$ of the runtime of clone operation applied to the matrices of the same size. The runtime ratio approaches 1 as the matrix size increases! We note that the kernels are still not fully optimized, and therefore, further code optimization might further lower the runtime. The overhead is very small relative to clone operation, so any other quantization scheme is unlikely to be much faster than our method in terms of overhead. For estimating the overhead compared to GEMMs, we also

Method	β	Type	r	bits $\times$ r/16	Speedup	ARC-c	ARC-e	BoolQ	HS	PIQA	WG
Full-Precision	-	BF16	1	1	-	43.1	76.3	77.8	57.2	78.0	68.8
RTN+IMUnpack	5	INT4	1.27	0.318	82%	39.3	72.8	69.9	53.4	74.9	66.4
	7	INT4	1.41	0.352	74%	42.6	73.9	72.3	55.9	77.0	67.4
	11	INT5	1.27	0.397	64%	43.9	76.1	77.3	56.3	77.3	69.3
	15	INT6	1.15	0.431	57%	43.0	75.7	77.5	57.0	78.0	69.2
	31	INT7	1.16	0.507	44%	42.7	76.1	76.1	57.3	77.3	69.3

Table 7.20: Inference: estimated speedup of the entire model for LLaMA-7B.

provide runtime for FP32 GEMM and FP16 GEMM for these matrix sizes alongside the runtimes for clone, UnpackRow, and UnpackCol as shown in Table 7.19.

Estimated Overall Speedup of Entire Models. We have carefully calculated the speedup of the entire model runtime. We measure the breakdowns of different operators in LLaMA-7B and LLaMA-70B for an input of shape [16, 512] where 16 is the batch size and 512 is the sequence length. The overall model runtime of LLaMA-7B and LLaMA-70B is 1.84s and 18.92s, respectively. The overall runtime of GEMMs for Linear layers accounts for 77.32% and 88.96% of the overall model runtime, respectively. We exclude GEMMs in attention computation for simplicity which are accounted for in the much smaller 22.68% and 11.04% of the model's runtime (this percentage includes GEMMs in attention computation, activation functions, layer norm, residual connection, etc.). Let p be the percentage of GEMMs spent on Linear layers and h be the runtime ratio of the runtime overhead of IMUnpack compared to the runtime of GEMMs, if we use $\frac{x}{16}$ as runtime of INT-x GEMM compared to FP16 GEMMs, we estimate the efficiency gain of using INT-x for model computation. The speedup of the overall model runtime can be estimated via $\frac{1}{(\frac{xr}{16}+h)p+(1-p)}$ using Amdahl's law. We know from Table 7.19 that $h < 0.1$, so we can populate the speedup of the overall model runtime in Table 7.20 using the worst possible overhead $h = 0.1$. Since hardware and software support for different bit-widths is not publicly available, this estimate is based on calculations using actual measurements and profiling.

7.4 Limitations

To simplify the presentation, we used the simplest RTN quantization, which might not deliver the optimal performance. More sophisticated techniques are likely to further improve the results. For example, we may be able to remove the demands of large β for the set $\{\nabla_{\mathbf{Y}}, \nabla_{\mathbf{P}}, \nabla_{\mathbf{O}}\}$ for ViT training. The current unpacking strategies cannot handle the self-attention matrix $\mathbf{M}$ efficiently since the outliers mainly concentrate on the diagonal region rather than rows or columns; this needs further investigation.

7.5 Summary

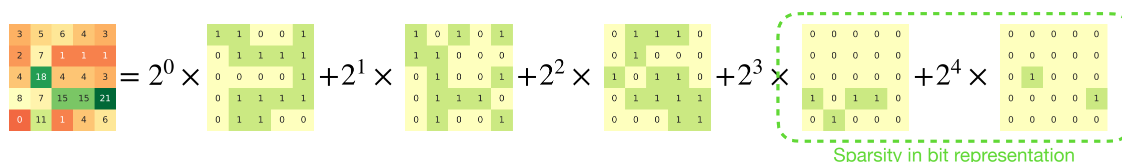

Figure 7.9: Illustration of the bit representation of a matrix. Heavy-hitters have higher order non-zero bits. When a matrix contains heavy-hitters, its bit representation has a sparsity structure in the higher order bits as illustrated.

In this chapter, we verified the efficacy of integer GEMMs for both training and inference for Transformer-based models in language modeling and vision. A simple RTN quantization strategy works well compared to baselines. But in this setting, the presence of large outliers/heavy-hitters makes it difficult to make use of efficient low bit-width integer GEMMs since these outliers are much larger than the representable range of low bit-width integers. We take a "multi-resolution" view (Chapter 5) in how we extract a spectrum of bit-width tradeoffs. This is loosely similar to sparsity but here, instead of making a zero versus non-zero distinction between the entries, our heavy-hitters (which need higher bit-width representations) are analogous to "non-sparse" entries (as illustrated in Figure 7.9). To handle high bit-width heavy-hitters, we developed an algorithm to unpack integer matrices that contains arbitrarily large values to slightly larger matrices with the property that all values

lie within the representable range of low bit-width integers and a procedure to obtain the GEMM output of original matrices using only low bit-width integer GEMMs on the unpacked matrices followed by some scaling (using bit shifting) and accumulation. Our algorithm can simplify the design of hardware and improve the power efficiency by only supporting low bit-width integer GEMMs for both training and inference. A preliminary version of this chapter was published as (Zeng et al., 2024) and the codebase is available at `https://github.com/vsingh-group/im-unpack`.

8 CONCLUSIONS

In this thesis, we described how we improve the efficiency of Transformer models operating at different abstractions of the computations needed. We show that via locality sensitive hashing inspired memory lookup based algorithms, multi-resolution approaches, and low precision integer computation, we are able to significant accelerate the computation while preserving the quality of Transformer models. Direct applications of these ideas usually are sub-optimal and likely to incur specific challenges. In each chapter, we addressed these challenges when designing algorithms using ideas appropriate to the chosen abstraction of Transformer computation.

1. In Chapter 3, we leveraged the idea of locality sensitive hashing based importance sampling to develop a low variance approximation of softmax in estimating self-attention mechanism computation. And then, we noted the problem faced by this sampling algorithm when running on a GPU due to the discrepancy between target distribution and proposal distribution. To address this difficulty, we proposed a modification of the sampling procedure and the target distribution to achieve a functionally similar mechanism compared to self-attention. This modification allows a simplified algorithm and makes it GPU and backpropagation friendly, which eventually leads to a novel memory lookup based algorithm for approximating the self-attention mechanism.

2. In Chapter 4, we extended the idea discussed in Chapter 3 as a replacement to compute-intensive matrix multiplication based feedforward network and evaluated the feasibility of deploying it on CPUs. We noted the challenges of applying the idea in Chapter 3 directly to feedforward network approximation, such as the need for rehashing and handling a large number of hashes. Then, we described a new type of algorithm that directly learns the hash functions and hash tables. Similar to the formulation in Chapter 3, we developed a feedforward network whose operations purely consists of memory lookup and

showed that this lookup based module is competitive to vanilla feed-forward network, while requiring much reduced compute resources.

3. In Chapter 5, inspired by the visualization of self-attention, we revisit the classical but underexplored Multi-Resolution Analysis (MRA) for approximating self-attention. We find it is sufficient to reconstruct the self-attention matrices by keeping only a small number of wavelet coefficients. However, computing and determining nonzero coefficients efficiently is nontrivial and require accessing the underlying matrices that need to be approximated. We investigate the specific modifications and approximations needed to make MRA operate efficiently for approximating self-attention, and proposed a simple MRA approximation scheme that is efficient yet effective in terms of approximation quality and various applications.

4. In Chapter 6, taking inspiration from the analysis of MRA approximation for self-attention in Chapter 5, we discussed a sequence compression idea to extend MRA approximation to the computation of the entire Transformer block. This further boosts Transformer models' efficiency on processing sequences whose lengths are much longer than applications explored in Chapter 5. To avoid sacrificing model performance, we analyze the incurred approximation error for the important tokens and use the idea discussed in Chapter 5 to derive a procedure to minimize the error while preserving the efficiency. Further, based on the characteristic of MRA approximations, we designed a data structure that allows us to further improve the efficiency by eliminating unnecessary and redundant computation when applying MRA on sequence compression.

5. In Chapter 7, we explored possible efficiency opportunities of the most basic operation that Transformer models and most deep learning models are built upon, GEneral Matrix Multiply (GEMM), when using low precision integer arithmetic. We noted that while simply rounding scaled floating point matrices to integer matrices to enable integer arithmetic is very effective in both

training and inference of various Transformer models, the existence of outliers is the main roadblock in achieving efficient low precision arithmetic. Then, to removing this hurdle, we developed an algorithm which unpacks these outliers to smaller values that can be processed using efficient low precision arithmetic while ensuring the arithmetic results remain unchanged.

8.1 Future Directions

Transformer models have been successfully applied to a wide range of applications and domains. However, their usage can be costly as discussed in §1.2. The development of efficient algorithms to accelerate Transformer computations or to minimize the resources needed is crucial. This will play an important role in applying Transformers to both existing and future applications, and will benefit from the contributions presented in this thesis. We will outline several near to medium-term research directions that we plan to explore.

8.1.1 Efficient and Effective Sequence Modeling Mechanisms

The Multi-Head Attention (MHA) mechanism within the Transformer architecture offers a strong capability for sequence modeling, but the quadratic cost of self-attention limits the scalability of Transformer models. This quadratic cost has initiated active research efforts trying to address this challenge. Two popular directions are sparse attention and low rank linear attention. In Chapter 3 and 5, we discussed two approaches to make these ideas efficient while preserving their efficacy. These works attempt to mimic the behavior of self-attention and allow efficient computation. There exists a distinct line of attack that does not seek to replicate the behavior of self-attention, but tries to design new sequence modeling mechanisms inspired by the MHA mechanism. Some existing reports have demonstrated impressive results and suggested that we may even be able to outperform Transformer models in certain settings. Examples include RWKV (Peng et al., 2023), Structured State Space Model (Gu et al., 2022), and Mamba (Gu

and Dao, 2023). This departure from approximating self-attention is inspiring, and could potentially lead to next generation of model architectures.

Given our successful attempts at efficient self-attention, it could be beneficial and interesting to continue exploring this direction for efficient and effective sequence modeling mechanisms that are inspired by but might be slightly different from self-attention. Success here could contribute to the development of next generation model architectures that are more efficient and effective than the now-dominant Transformer architecture.

8.1.2 Memory Lookup based Transformers

The common compute hardware for Transformer models and other deep learning models is the GPU because of its massive compute capacity. The use of other compute resources such as CPU and memory/storage are less explored. In Chapter 3 and 4, we developed two approaches to replace self-attention and feedforward network using memory lookup based operations. We use multiple large hash tables to approximate each components of a Transformer model, which is similar to creating a Rainbow Table (a hash table used in cryptography (Oechslin, 2003)) to save computation by replacing certain compute-heavy operations with a memory lookup. Since these methods are designed to require minimal compute resource and instead relies on memory accesses, we know that these methods are memory bounded in terms of memory bandwidth for fast access to multiple hash tables and memory size for storage of large hash tables. While a GPU has high memory bandwidth, its memory size is usually quite limited compared to a CPU or hard drive.

A CPU generally has low compute capacity but high memory bandwidth if the large cache size of a CPU is leveraged properly. Compared to the 2TB/sec memory bandwidth of A100 80GB (NVIDIA, a), a CPU has 1TB/sec, 1TB/sec, 400GB/sec, and 100GB/sec bandwidth for the L1, L2, L3, and DRAM, respectively (Intel, 2016). The sizes of these memory are 32KB, 256KB, 8MB or more, and 4GB to 1TB for L1, L2, L3, and DRAM respectively (Intel, 2016). These bandwidths are

not much worse than A100. Therefore, our hypothesis is that with optimized implementations for memory reuse and cache hit rate, the runtime of memory lookup based Transformer on a CPU can be brought within only a few factors of runtime of a standard Transformer on a GPU. In our preliminary attempt in Chapter 4, the CPU implementation is less than 4 times slower than the GPU implementation. When coupled with the almost unlimited storage in an array of hard drives for extremely large hash tables, we might be able to develop an algorithm that utilizes this memory hierarchy to allow us to run Transformer using simply memory access with very small amount of compute and without the support of GPUs.

8.1.3 YOSO for Spiking Neural Network

Spiking Neural Networks (SNNs) (Maass, 1997) are highly power-efficient when deployed on hardware specifically designed to support their spiking mechanisms. These mechanisms feature sparse additions and time-decaying signals, without involving any multiplication. The SNN community is trying to develop spiking based Transformer. For example, SpikFormer (Zhou et al., 2023) designed a spiking based efficient self-attention. Similarly, in Chapter 3, we developed the YOSO formulation for efficient self-attention, which has a similar characteristic in that its operations are mostly sparse addition. An attention mechanism accepts three inputs: queries, keys, and values. In YOSO, we first construct a hash table using keys and values by sparsely adding each value to a specific table entry determined by the corresponding key, then queries are used to gather outputs using sparse addition as well from the hash table as the output of our attention.

The main distinction between YOSO and SNN based Transformer models is that time decaying signals are not part of the YOSO formulation. However, we believe that the time decaying feature can be very useful. When using YOSO in an autoregressive setting, the hash table can be viewed as the recurrence state of the past history. Many works on efficient self-attentions or similar variants have shown that it is necessary to "forget" the distant history to introduce locality as well as compensate for the finite sized recurrence state and introduced a variety of "forget-

ting" mechanisms (Gu et al., 2022; Peng et al., 2021; Qin et al., 2022). Interestingly, we conjecture that the time decaying feature of SNN might provide an effortless approach to introducing "forgetting" in our YOSO formulation. Specifically, in autoregressive setting, the model receives tokens in a sequential order. Given the first token, the recurrent state (hash table) is computed by adding this token to the hash table. Then, the signals in the recurrent state (hash table) will be gradually decaying (forgetting) while waiting for the second token. When the second token arrives, it is added to the decayed recurrent state forming a new recurrent state. This process continues until all tokens in the sequence are received and processed by the model. In this way, the signal from distant history will decay to zero while recent history will retain its signal in the recurrent state. With the time decay signal ability provided by the hardware, if we organize the tokens in a timely manner to add into the recurrent state, we can achieve the behavior described above effortlessly.

If this attempt is successful, this result opens up possibility of using power efficient hardware for efficient Transformer designs. Coupled with a similar formulation in Chapter 4 for the feedforward network, we might be able to bring the success of Transformer to SNN while benefiting from the superior power efficiency of SNN.

8.1.4 Low Precision Integer Computing

Switching from 32-bit floating point to 16-bit floating point has doubled the compute speed for the same hardware. With saturating hardware improvement, moving to lower precision computation is a common trend in the deep learning community. New compute hardware has started to support lower precision computation such as 4-bit and 8-bit integers, and 4-bit and 8-bit floating point computations to keep up with the compute requirement of modern Transformer models. In Chapter 7, we demonstrated a simple rounding with some algorithmic innovations to handle outliers allows effortless training and inference using low precision integer computations.

However, there are still a number of remaining challenges as we discussed in Chapter 7, such as the requirement for high precision for some back-propagation

calculations in certain settings and the inability to handle self-attention computation efficiently. These limitations require further study. Further, while there are some experiments testing the scalability of the idea discussed in Chapter 7, the results are limited and further evaluation at a much larger scale is critical to ensure the scalability of this work.

Additionally, this work opens up new efficiency opportunities for low precision integer training. When using floating point numbers, small values will be represented precisely, which might not be necessary for deep learning applications. In contrast, small values in fixed point numbers, such as integers, will simply be rounded to zero, resulting in abundant sparsity structures within the models, such as sparse activations, sparse gradients, and sparse parameters (Frankle and Carbin, 2019; Li et al., 2023b; Lin et al., 2018), which can be exploited for compute, memory, or communication efficiency. Success in this direction could greatly accelerate the computation when the compute hardware remains unchanged.

REFERENCES

Adepu, Harshavardhan, Zhanpeng Zeng, Li Zhang, and Vikas Singh. 2024. Framequant: Flexible low-bit quantization for transformers. In *International conference on machine learning*.

AMD. Amd zen deep neural network (zendnn). *AMD*.

Andoni, Alexandr, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. In *Advances in neural information processing systems*.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. In *Advances in neural information processing systems: Deep learning symposium*.

Banner, Ron, Yury Nahshan, and Daniel Soudry. 2019. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in neural information processing systems*.

Beltagy, Iz, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

Bengio, Yoshua. 2013. Estimating or propagating gradients through stochastic neurons. *arXiv preprint arXiv:1305.2982*.

Bhat, Ashok. 2021. Machine learning on - arm servers an update. linaro virtual connect 2021.

Bisk, Yonatan, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the aaai conference on artificial intelligence*.

Brooks, Tim, Bill Peebles, Connor Holmes, Will DePue, Yufei Guo, Li Jing, David Schnurr, Joe Taylor, Troy Luhman, Eric Luhman, Clarence Ng, Ricky Wang, and Aditya Ramesh. 2024. Video generation models as world simulators. *OpenAI*.

Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in neural information processing systems*.

Bulatov, Aydar, Yuri Kuratov, and Mikhail Burtsev. 2022. Recurrent memory transformer. In *Advances in neural information processing systems*.

Burd, Thomas, Wilson Li, James Pistole, Srividhya Venkataraman, Michael McCabe, Timothy Johnson, James Vinh, Thomas Yiu, Mark Wasio, Hon-Hin Wong, Daryl Lieu, Jonathan White, Benjamin Munger, Joshua Lindner, Javin Olson, Steven Bakke, Jeshuah Sniderman, Carson Henrion, Russell Schreiber, Eric Busta, Brett Johnson, Tim Jackson, Aron Miller, Ryan Miller, Matthew Pickett, Aaron Horiuchi, Josef Dvorak, Sabeesh Balagangadharan, Sajeesh Ammikkallingal, and Pankaj Kumar. 2022. Zen3: The amd 2nd-generation 7nm x86-64 microprocessor core. In *Ieee international solid-state circuits conference*, vol. 65, 1–3.

Candès, Emmanuel J, Xiaodong Li, Yi Ma, and John Wright. 2011. Robust principal component analysis? *Journal of the ACM (JACM)* 58(3):1–37.

Carion, Nicolas, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. End-to-end object detection with transformers. In *European conference on computer vision*.

Cauchy, Augustin, et al. 1847. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris* 25(1847):536–538.

Charikar, Moses. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings on 34th annual ACM symposium on theory of computing, may 19-21, 2002, montréal, québec, canada*, ed. John H. Reif, 380–388. ACM.

Charikar, Moses, and Paris Siminelakis. 2017. Hashing-based-estimators for kernel density in high dimensions. In *Annual ieee symposium on foundations of computer science*, 1032–1043.

Chee, Jerry, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. 2023. QuIP: 2-bit quantization of large language models with guarantees. In *Advances in neural information processing systems*.

Chen, Beidi, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. 2021a. Scatterbrain: Unifying sparse and low-rank attention. In *Advances in neural information processing systems*.

Chen, Beidi, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. 2021b. {MONGOOSE}: A learnable {lsh} framework for efficient neural network training. In *International conference on learning representations*.

Chen, Beidi, Tharun Medini, James Farwell, sameh gobriel, Charlie Tai, and Anshumali Shrivastava. 2020. Slide : In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. In *Proceedings of machine learning and systems*, ed. I. Dhillon, D. Papailiopoulos, and V. Sze, vol. 2, 291–306.

Chen, Mingda, Zewei Chu, Sam Wiseman, and Kevin Gimpel. 2022. SummScreen: A dataset for abstractive screenplay summarization. In *Annual meeting of the association for computational linguistics*.

Chen, Shi, and Qi Zhao. 2019. Shallowing deep networks: Layer-wise pruning based on feature representations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Chen, Zihan, Hongbo Zhang, Xiaoji Zhang, and Leqi Zhao. 2018. Quora question pairs.

Cheng, Yu, Felix X Yu, Rogerio S Feris, Sanjiv Kumar, Alok Choudhary, and Shi-Fu Chang. 2015. An exploration of parameter redundancy in deep networks with circulant projections. In *The ieee/cvf international conference on computer vision*.

Child, Rewon, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*.

Cho, Kyunghyun, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, eighth workshop on syntax, semantics and structure in statistical translation*, ed. Dekai Wu, Marine Carpuat, Xavier Carreras, and Eva Maria Vecchi, 103–111. Doha, Qatar: Association for Computational Linguistics.

Choi, Sanghyuk Roy, and Minhyeok Lee. 2023. Transformer architecture and attention mechanisms in genome data analysis: a comprehensive review. *Biology* 12(7):1033.

Choromanski, Krzysztof Marcin, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. 2021. Rethinking attention with performers. In *International conference on learning representations*.

Chui, Charles. 1992. *An introduction to wavelets*, vol. 1.

Clark, Christopher, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*.

Clark, Peter, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

Cohan, Arman, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. 2018. A discourse-aware attention model for abstractive summarization of long documents. *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Coifman, Ronald R, and Mauro Maggioni. 2006. Diffusion wavelets. *Applied and computational harmonic analysis* 21(1):53–94.

Dao, Tri, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in neural information processing systems*.

d'Ascoli, Stéphane, Hugo Touvron, Matthew Leavitt, Ari Morcos, Giulio Biroli, and Levent Sagun. 2021. Convit: Improving vision transformers with soft convolutional inductive biases. In *International conference on machine learning*.

Dasigi, Pradeep, Kyle Lo, Iz Beltagy, Arman Cohan, Noah A. Smith, and Matt Gardner. 2021. A dataset of information-seeking questions and answers anchored in research papers. In *Conference of the north american chapter of the association for computational linguistics: Human language technologies*.

Daubechies, I. 1992. *Ten lectures on wavelets*. CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics.

De Sa, Christopher, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R Aberger, Kunle Olukotun, and Christopher Ré. 2018. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*.

Dehghani, Mostafa, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Peter Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, Rodolphe Jenatton, Lucas Beyer, Michael Tschannen, Anurag Arnab, Xiao Wang, Carlos Riquelme Ruiz, Matthias Minderer, Joan Puigcerver, Utku Evci, Manoj Kumar, Sjoerd Van Steenkiste, Gamaleldin Fathy Elsayed, Aravindh Mahendran, Fisher Yu, Avital Oliver, Fantine Huot, Jasmijn Bastings, Mark

Collier, Alexey A. Gritsenko, Vighnesh Birodkar, Cristina Nader Vasconcelos, Yi Tay, Thomas Mensink, Alexander Kolesnikov, Filip Pavetic, Dustin Tran, Thomas Kipf, Mario Lucic, Xiaohua Zhai, Daniel Keysers, Jeremiah J. Harmsen, and Neil Houlsby. 2023. Scaling vision transformers to 22 billion parameters. In *International conference on machine learning*.

Dettmers, Tim, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. GPT3.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in neural information processing systems*.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the north american chapter of the association for computational linguistics*.

Ding, Yifu, Haotong Qin, Qinghua Yan, Zhenhua Chai, Junjie Liu, Xiaolin Wei, and Xianglong Liu. 2022. Towards accurate post-training quantization for vision transformer. In *Acm international conference on multimedia*, 5380–5388.

Dolan, William B., and Chris Brockett. 2005. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the third international workshop on paraphrasing (IWP2005)*.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. In *International conference on learning representations*.

Edwards, Tim. 1991. Discrete wavelet transforms: Theory and implementation. *Universidad de* 1991:28–35.

Elman, Jeffrey L. 1990. Finding structure in time. *Cognitive science* 14(2):179–211.

Fabbri, Alexander, Irene Li, Tianwei She, Suyi Li, and Dragomir Radev. 2019. Multi-news: A large-scale multi-document summarization dataset and abstractive hierarchical model. In *Annual meeting of the association for computational linguistics*.

Fedus, William, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*.

Frankle, Jonathan, and Michael Carbin. 2019. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International conference on learning representations*.

Frantar, Elias, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. OPTQ: Accurate quantization for generative pre-trained transformers. In *International conference on learning representations*.

Gao, Leo, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Gavish, Matan, Boaz Nadler, and Ronald R Coifman. 2010. Multiscale wavelets on trees, graphs and high dimensional data: Theory and applications to semi supervised learning. In *The international conference on machine learning*.

Georganas, Evangelos, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of high-performance deep learning convolutions on simd architectures. In *Proceedings of the international conference for high performance computing, networking, storage, and analysis*. SC '18, IEEE Press.

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

Gu, Albert, and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*.

Gu, Albert, Karan Goel, and Christopher Ré. 2022. Efficiently modeling long sequences with structured state spaces. In *The international conference on learning representations*.

Guo, Mandy, Joshua Ainslie, David Uthus, Santiago Ontanon, Jianmo Ni, Yun-Hsuan Sung, and Yinfei Yang. 2022. LongT5: Efficient text-to-text transformer for long sequences. In *Conference of the north american chapter of the association for computational linguistics*.

Hackbusch, Wolfgang. 1985. *Multi-grid methods and applications*, vol. 4. Springer Science & Business Media.

Hammond, David K, Pierre Vandergheynst, and Rémi Gribonval. 2011. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis* 30(2):129–150.

Han, Song, Huizi Mao, and William J. Dally. 2016. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *International conference on learning representations*.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the ieee the ieee/cvf conference on computer vision and pattern recognition*.

Heideman, Michael, Don Johnson, and Charles Burrus. 1985. Gauss and the history of the fast fourier transform. *Archive for History of Exact Sciences* 34:265–277.

Hendrycks, Dan, and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Hitczenko, Paweł, and Stanisław Kwapień. 1994. On the rademacher series. In *Probability in banach spaces, 9*, ed. Jørgen Hoffmann-Jørgensen, James Kuelbs, and Michael B. Marcus, 31–36. Boston, MA: Birkhäuser Boston.

Horowitz, Mark. 2014. 1.1 computing's energy problem (and what we can do about it). In *Ieee international solid-state circuits conference digest of technical papers*, 10–14.

Hu, Shenglong. 2015. Relations of the nuclear norm of a tensor and its matrix flattenings. *Linear Algebra and its Applications* 478:188–199.

Huang, Luyang, Shuyang Cao, Nikolaus Parulian, Heng Ji, and Lu Wang. 2021. Efficient attentions for long document summarization. In *Conference of the north american chapter of the association for computational linguistics: Human language technologies*.

Intel. Intrinsics for intel advanced matrix extensions (intel(r) amx) instructions. intel c++ compiler classic developer guide and reference.

———. 2016. Memory performance in a nutshell.

Ioffe, Sergey, and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*.

Ithapu, Vamsi K, Risi Kondor, Sterling C Johnson, and Vikas Singh. 2017. The incremental multiresolution matrix factorization algorithm. In *The ieee/cvf conference on computer vision and pattern recognition*.

Jang, Eric, Shixiang Gu, and Ben Poole. 2017. Categorical reparameterization with gumbel-softmax. In *International conference on learning representations*.

Javaheripi, Mojan, Sébastien Bubeck, Marah Abdin, Jyoti Aneja, Sebastien Bubeck, Caio César Teodoro Mendes, Weizhu Chen, Allie Del Giorno, Ronen Eldan, Sivakanth Gopi, et al. 2023. Phi-2: The surprising power of small language models. *Microsoft Research Blog*.

Jiang, Albert Q., Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel,

Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Jiang, Albert Q., Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Jouppi, Norman P., Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *2021 acm/ieee 48th annual international symposium on computer architecture (isca)*, 1–14.

Jouppi, Norman P., Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay

Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*.

Kahn, H. 1950. Random sampling (monte carlo) techniques in neutron attenuation problems–i. *Nucleonics* 6(5):27; passim.

Kaplan, Jared, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Katharopoulos, Angelos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. 2020. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International conference on machine learning*.

Keller, Nathan, and Ohad Klein. 2022. Proof of tomaszewski's conjecture on randomly signed sums. *Advances in Mathematics* 407:108558.

Kiefer, Jack, and Jacob Wolfowitz. 1952. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics* 462–466.

Kim, Sehoon, Amir Gholami, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. I-bert: Integer-only bert quantization. In *International conference on machine learning*.

Kingma, Diederik P., and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International conference on learning representations*.

Kirillov, Alexander, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. 2023. Segment anything. In *The ieee/cvf international conference on computer vision*.

Kitaev, Nikita, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. In *International conference on learning representations (iclr)*.

Klimt, Bryan, and Yiming Yang. 2004. The enron corpus: A new dataset for email classification research. In *Proceedings of the 15th european conference on machine learning*, 217–226. ECML'04, Berlin, Heidelberg: Springer-Verlag.

Knight, Will. 2023. Openai's ceo says the age of giant ai models is already over. *Wired*.

Kočiský, Tomáš, Jonathan Schwarz, Phil Blunsom, Chris Dyer, Karl Moritz Hermann, Gábor Melis, and Edward Grefenstette. 2018. The NarrativeQA reading comprehension challenge. *Transactions of the Association for Computational Linguistics*.

Koehn, Philipp. 2005. Europarl: A parallel corpus for statistical machine translation. In *Proceedings of machine translation summit x: Papers*, 79–86. Phuket, Thailand.

Kondor, Risi, Nedelina Teneva, and Vikas Garg. 2014. Multiresolution matrix factorization. In *International conference on machine learning*.

Koreeda, Yuta, and Christopher Manning. 2021. ContractNLI: A dataset for document-level natural language inference for contracts. In *Conference on empirical methods in natural language processing*.

Kovaleva, Olga, Alexey Romanov, Anna Rogers, and Anna Rumshisky. 2019. Revealing the dark secrets of bert. In *Conference on empirical methods in natural language processing*.

Krizhevsky, Alex, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

Lan, Zhenzhong, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. Albert: A lite bert for self-supervised learning of language representations. In *International conference on learning representations*.

Latif, Siddique, Aun Zaidi, Heriberto Cuayahuitl, Fahad Shamshad, Moazzam Shoukat, and Junaid Qadir. 2023. Transformers in speech processing: A survey. *arXiv preprint arXiv:2303.11607*.

Le, Quoc, Tamas Sarlos, and Alex Smola. 2013. Fastfood - approximating kernel expansions in loglinear time. In *30th international conference on machine learning*.

Le Scao, Teven, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2023. Bloom: A 176b-parameter open-access multilingual language model.

LeCun, Yann. 2023. *twitter:1706545305762582580*.

Lee, Ann B, and Boaz Nadler. 2007. Treelets: A tool for dimensionality reduction and multi-scale analysis of unstructured data. In *International conference on artificial intelligence and statistics*.

Lee-Thorp, James, Joshua Ainslie, Ilya Eckstein, and Santiago Ontanon. 2022. Fnet: Mixing tokens with fourier transforms. In *Conference of the north american chapter of the association for computational linguistics: Human language technologies*, ed. Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz.

Leswing, Kif. 2023a. Meet the $10,000 nvidia chip powering the race for a.i. *cnbc*.

———. 2023b. Nvidia's top a.i. chips are selling for more than $40,000 on ebay. *cnbc*.

Levy, Omer, Yoav Goldberg, and Ido Dagan. 2015. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*.

Li, Zhikai, and Qingyi Gu. 2023. I-vit: Integer-only quantization for efficient vision transformer inference. In *The ieee/cvf international conference on computer vision*.

Li, Zhikai, Junrui Xiao, Lianwei Yang, and Qingyi Gu. 2023a. Repq-vit: Scale reparameterization for post-training quantization of vision transformers. In *The ieee/cvf international conference on computer vision*.

Li, Zonglin, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J. Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, and Sanjiv Kumar. 2023b. The lazy neuron phenomenon: On emergence of activation sparsity in transformers. In *International conference on learning representations*.

Lichtenau, Cedric, Alper Buyuktosunoglu, Ramon Bertran, Peter Figuli, Christian Jacobi, Nikolaos Papandreou, Haris Pozidis, Anthony Saporito, Andrew Sica, and Elpida Tzortzatos. 2022. Ai accelerator on ibm telum processor: Industrial product. In *Proceedings of the 49th annual international symposium on computer architecture*, 1012–1028. ISCA '22, New York, NY, USA: Association for Computing Machinery.

Lin, Yang, Tianyu Zhang, Peiqin Sun, Zheng Li, and Shuchang Zhou. 2022. Fq-vit: Post-training quantization for fully quantized vision transformer. In *International joint conference on artificial intelligence*, 1173–1179.

Lin, Yujun, Song Han, Huizi Mao, Yu Wang, and Bill Dally. 2018. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International conference on learning representations*.

Linsley, Drew, Junkyung Kim, Vijay Veerabadran, Charles Windolf, and Thomas Serre. 2018. Learning long-range spatial dependencies with horizontal gated recurrent units. In *Advances in neural information processing systems*.

Liu, Hao, Matei Zaharia, and Pieter Abbeel. 2024. Harnessing overlap in blockwise transformers for near-infinite context. In *The twelfth international conference on learning representations*.

Liu, Shih-yang, Zechun Liu, Xijie Huang, Pingcheng Dong, and Kwang-Ting Cheng. 2023a. LLM-FP4: 4-bit floating-point quantized transformers. In *Conference on empirical methods in natural language processing*.

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

Liu, Yizhi, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019b. Optimizing cnn model inference on cpus. In *Proceedings of the 2019 usenix conference on usenix annual technical conference*, 1025–1040. USENIX Association.

Liu, Zechun, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. 2023b. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*.

Lu, Jiachen, Jinghan Yao, Junge Zhang, Xiatian Zhu, Hang Xu, Weiguo Gao, Chunjing Xu, Tao Xiang, and Li Zhang. 2021. Soft: Softmax-free transformer with linear complexity. In *Advances in neural information processing systems*.

Maas, Andrew L., Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Annual meeting of the association for computational linguistics: Human language technologies*.

Maass, Wolfgang. 1997. Networks of spiking neurons: the third generation of neural network models. *Neural networks* 10(9):1659–1671.

Maddison, Chris J, Daniel Tarlow, and Tom Minka. 2014. A* sampling. In *Advances in neural information processing systems*.

Mallat, Stéphane. 1999. *A wavelet tour of signal processing*. Elsevier.

Micikevicius, Paulius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed precision training. In *International conference on learning representations*.

Mittal, Sparsh, Poonam Rajput, and Sreenivas Subramoney. 2022. A survey of deep learning on cpus: Opportunities and co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems* 33(10):5095–5115.

Moczulski, Marcin, Misha Denil, Jeremy Appleyard, and Nando de Freitas. 2016. ACDC: A structured efficient linear layer. In *International conference on learning representations*, ed. Yoshua Bengio and Yann LeCun.

Nagel, Markus, Mart van Baalen, Tijmen Blankevoort, and Max Welling. 2019. Data-free quantization through weight equalization and bias correction. In *The ieee/cvf international conference on computer vision*.

Nangia, Nikita, and Samuel Bowman. 2018. ListOps: A diagnostic dataset for latent tree learning. In *Conference of the north american chapter of the association for computational linguistics: Student research workshop*.

Narayan, Shashi, Shay B. Cohen, and Mirella Lapata. 2018. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. In *Conference on empirical methods in natural language processing*.

Nassif, Nevine, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire rapids: The next-generation intel xeon scalable processor. In *Ieee international solid-state circuits conference*, vol. 65, 44–46.

Neyshabur, Behnam, and Nathan Srebro. 2015. On symmetric and asymmetric lshs for inner product search. In *International conference on machine learning*.

Nori, Anant V., Rahul Bera, Shankar Balachandran, Joydeep Rakshit, Om J. Omer, Avishaii Abuhatzera, Belliappa Kuttanna, and Sreenivas Subramoney. 2021. Reduct: Keep it close, keep it cool! : Efficient scaling of dnn inference on multi-core

cpus with near-cache compute. In *2021 acm/ieee 48th annual international symposium on computer architecture (isca)*, 167–180.

NVIDIA. a. Nvidia a100. *NVIDIA*.

———. b. Nvidia b100. *NVIDIA*.

———. c. Nvidia h100. *NVIDIA*.

———. d. Nvidia v100. *NVIDIA*.

Oechslin, Philippe. 2003. Making a faster cryptanalytic time-memory trade-off. In *The 23rd annual international cryptology conference*, 617–630. Lecture Notes in Computer Science. 2729.

OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang,

Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel

Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Owen, Art B. 2013. *Monte carlo theory, methods and examples*. https://artowen.su.domains/mc/.

Pang, Richard Yuanzhe, Alicia Parrish, Nitish Joshi, Nikita Nangia, Jason Phang, Angelica Chen, Vishakh Padmakumar, Johnny Ma, Jana Thompson, He He, and Samuel Bowman. 2022. QuALITY: Question answering with long input texts, yes! In *Conference of the north american chapter of the association for computational linguistics: Human language technologies*.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*.

Patel, Dylan, and Afzal Ahmad. 2023. The inference cost of search disruption – large language model cost analysis. *semianalysis*.

Peebles, William, and Saining Xie. 2023. Scalable diffusion models with transformers. In *The ieee/cvf international conference on computer vision*.

Peng, Bo, Eric Alcaide, Quentin G. Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, G Kranthikiran, Xuming He, Haowen Hou, Przemyslaw Kazienko, Jan Kocoń, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Xiangru Tang, Bolun Wang, Johan Sokrates Wind, Stansilaw Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Jian Zhu, and

Rui Zhu. 2023. Rwkv: Reinventing rnns for the transformer era. In *Conference on empirical methods in natural language processing*.

Peng, Hao, Nikolaos Pappas, Dani Yogatama, Roy Schwartz, Noah Smith, and Lingpeng Kong. 2021. Random feature attention. In *International conference on learning representations*.

Peng, Xi, Canyi Lu, Zhang Yi, and Huajin Tang. 2018. Connections between nuclear-norm and frobenius-norm-based representations. *IEEE Transactions on Neural Networks and Learning Systems* 29(1):218–224.

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical recipes: the art of scientific computing, 3rd edition*. Cambridge University Press.

Qin, Zhen, Weixuan Sun, Hui Deng, Dongxu Li, Yunshen Wei, Baohong Lv, Junjie Yan, Lingpeng Kong, and Yiran Zhong. 2022. cosformer: Rethinking softmax in attention. In *International conference on learning representations*.

Radev, Dragomir R., Pradeep Muthukrishnan, Vahed Qazvinian, and Amjad Abu-Jbara. 2013. The ACL anthology network corpus. *Language Resources and Evaluation* 47(4):919–944.

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

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

Raffel, Colin, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*.

Rajpurkar, Pranav, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. In *Conference on empirical methods in natural language processing*.

Rotem, Efraim, Adi Yoaz, Lihu Rappoport, Stephen J. Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, Monica Gupta, and Ahmad Yasin. 2022. Intel alder lake cpu architectures. *IEEE Micro* 42(3):13–19.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323(6088):533–536.

Russakovsky, Olga, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. In *International journal of computer vision (ijcv)*.

Saad, Yousef. 2003. *Iterative methods for sparse linear systems*. 2nd ed. Society for Industrial and Applied Mathematics.

Sakaguchi, Keisuke, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM* 64(9):99–106.

Saxton, David, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. 2019. Analysing mathematical reasoning abilities of neural models. In *International conference on learning representations*.

See, Abigail, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Annual meeting of the association for computational linguistics*.

Shaham, Uri, Elad Segal, Maor Ivgi, Avia Efrat, Ori Yoran, Adi Haviv, Ankit Gupta, Wenhan Xiong, Mor Geva, Jonathan Berant, and Omer Levy. 2022. SCROLLS: Standardized comparison over long language sequences. In *Emnlp*.

Shanks, J.L. 1969. Computation of the fast walsh-fourier transform. *IEEE Transactions on Computers* C-18(5):457–459.

Shrivastava, Anshumali, and Ping Li. 2014. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). In *Advances in neural information processing systems*.

Simic, Slavko. 2009. Jensen's inequality and new entropy bounds. *Applied Mathematics Letters* 22(8):1262–1265.

Simonyan, Karen, and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *International conference on learning representations*.

Socher, Richard, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Conference on empirical methods in natural language processing*.

Spring, Ryan, and Anshumali Shrivastava. 2017. A new unbiased and efficient class of lsh-based samplers and estimators for partition function computation in log-linear models. *arXiv preprint arXiv:1703.05160*.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*.

Suno-AI. 2023. *Suno-ai*. https://suno.com.

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *The ieee/cvf conference on computer vision and pattern recognition*.

Tang, Duyu, Bing Qin, and Ting Liu. 2015. Document modeling with gated recurrent neural network for sentiment classification. In *Conference on empirical methods in natural language processing*.

Tay, Yi, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. 2021. Long range arena : A benchmark for efficient transformers. In *International conference on learning representations*.

Team, Gemini, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul R. Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Jack Krawczyk, Cosmo Du, Ed Chi, Heng-Tze Cheng, Eric Ni, Purvi Shah, Patrick Kane, Betty Chan, Manaal Faruqui, Aliaksei Severyn, Hanzhao Lin, YaGuang Li, Yong Cheng, Abe Ittycheriah, Mahdis Mahdieh, Mia Chen, Pei Sun, Dustin Tran, Sumit Bagri, Balaji Lakshminarayanan, Jeremiah Liu, Andras Orban, Fabian Güra, Hao Zhou, Xinying Song, Aurelien Boffy, Harish Ganapathy, Steven Zheng, HyunJeong Choe, Ágoston Weisz, Tao Zhu, Yifeng Lu, Siddharth Gopal, Jarrod Kahn, Maciej Kula, Jeff Pitman, Rushin Shah, Emanuel Taropa, Majd Al Merey, Martin Baeuml, Zhifeng Chen, Laurent El Shafey, Yujing Zhang, Olcan Sercinoglu, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, Alexandre Frechette, Charlotte Smith, Laura Culp, Lev Proleev, Yi Luan, Xi Chen, James Lottes, Nathan Schucher, Federico Lebron, Alban Rrustemi, Natalie Clay, Phil Crone, Tomas Kocisky, Jeffrey Zhao, Bartek Perz, Dian Yu, Heidi Howard, Adam Bloniarz, Jack W. Rae, Han Lu, Laurent Sifre, Marcello Maggioni, Fred Alcober, Dan Garrette, Megan Barnes, Shantanu Thakoor, Jacob Austin,

Gabriel Barth-Maron, William Wong, Rishabh Joshi, Rahma Chaabouni, Deeni Fatiha, Arun Ahuja, Gaurav Singh Tomar, Evan Senter, Martin Chadwick, Ilya Kornakov, Nithya Attaluri, Iñaki Iturrate, Ruibo Liu, Yunxuan Li, Sarah Cogan, Jeremy Chen, Chao Jia, Chenjie Gu, Qiao Zhang, Jordan Grimstad, Ale Jakse Hartman, Xavier Garcia, Thanumalayan Sankaranarayana Pillai, Jacob Devlin, Michael Laskin, Diego de Las Casas, Dasha Valter, Connie Tao, Lorenzo Blanco, Adrià Puigdomènech Badia, David Reitter, Mianna Chen, Jenny Brennan, Clara Rivera, Sergey Brin, Shariq Iqbal, Gabriela Surita, Jane Labanowski, Abhi Rao, Stephanie Winkler, Emilio Parisotto, Yiming Gu, Kate Olszewska, Ravi Addanki, Antoine Miech, Annie Louis, Denis Teplyashin, Geoff Brown, Elliot Catt, Jan Balaguer, Jackie Xiang, Pidong Wang, Zoe Ashwood, Anton Briukhov, Albert Webson, Sanjay Ganapathy, Smit Sanghavi, Ajay Kannan, Ming-Wei Chang, Axel Stjerngren, Josip Djolonga, Yuting Sun, Ankur Bapna, Matthew Aitchison, Pedram Pejman, Henryk Michalewski, Tianhe Yu, Cindy Wang, Juliette Love, Junwhan Ahn, Dawn Bloxwich, Kehang Han, Peter Humphreys, Thibault Sellam, James Bradbury, Varun Godbole, Sina Samangooei, Bogdan Damoc, Alex Kaskasoli, Sébastien M. R. Arnold, Vijay Vasudevan, Shubham Agrawal, Jason Riesa, Dmitry Lepikhin, Richard Tanburn, Srivatsan Srinivasan, Hyeontaek Lim, Sarah Hodkinson, Pranav Shyam, Johan Ferret, Steven Hand, Ankush Garg, Tom Le Paine, Jian Li, Yujia Li, Minh Giang, Alexander Neitz, Zaheer Abbas, Sarah York, Machel Reid, Elizabeth Cole, Aakanksha Chowdhery, Dipanjan Das, Dominika Rogozińska, Vitaliy Nikolaev, Pablo Sprechmann, Zachary Nado, Lukas Zilka, Flavien Prost, Luheng He, Marianne Monteiro, Gaurav Mishra, Chris Welty, Josh Newlan, Dawei Jia, Miltiadis Allamanis, Clara Huiyi Hu, Raoul de Liedekerke, Justin Gilmer, Carl Saroufim, Shruti Rijhwani, Shaobo Hou, Disha Shrivastava, Anirudh Baddepudi, Alex Goldin, Adnan Ozturel, Albin Cassirer, Yunhan Xu, Daniel Sohn, Devendra Sachan, Reinald Kim Amplayo, Craig Swanson, Dessie Petrova, Shashi Narayan, Arthur Guez, Siddhartha Brahma, Jessica Landon, Miteyan Patel, Ruizhe Zhao, Kevin Villela, Luyu Wang, Wenhao Jia, Matthew Rahtz, Mai Giménez, Legg Yeung, James Keeling, Petko Georgiev, Diana Mincu, Boxi Wu, Salem Haykal, Rachel Saputro, Kiran Vodrahalli, James Qin, Zeynep Cankara, Abhanshu Sharma,

Nick Fernando, Will Hawkins, Behnam Neyshabur, Solomon Kim, Adrian Hutter, Priyanka Agrawal, Alex Castro-Ros, George van den Driessche, Tao Wang, Fan Yang, Shuo yiin Chang, Paul Komarek, Ross McIlroy, Mario Lučić, Guodong Zhang, Wael Farhan, Michael Sharman, Paul Natsev, Paul Michel, Yamini Bansal, Siyuan Qiao, Kris Cao, Siamak Shakeri, Christina Butterfield, Justin Chung, Paul Kishan Rubenstein, Shivani Agrawal, Arthur Mensch, Kedar Soparkar, Karel Lenc, Timothy Chung, Aedan Pope, Loren Maggiore, Jackie Kay, Priya Jhakra, Shibo Wang, Joshua Maynez, Mary Phuong, Taylor Tobin, Andrea Tacchetti, Maja Trebacz, Kevin Robinson, Yash Katariya, Sebastian Riedel, Paige Bailey, Kefan Xiao, Nimesh Ghelani, Lora Aroyo, Ambrose Slone, Neil Houlsby, Xuehan Xiong, Zhen Yang, Elena Gribovskaya, Jonas Adler, Mateo Wirth, Lisa Lee, Music Li, Thais Kagohara, Jay Pavagadhi, Sophie Bridgers, Anna Bortsova, Sanjay Ghemawat, Zafarali Ahmed, Tianqi Liu, Richard Powell, Vijay Bolina, Mariko Iinuma, Polina Zablotskaia, James Besley, Da-Woon Chung, Timothy Dozat, Ramona Comanescu, Xiance Si, Jeremy Greer, Guolong Su, Martin Polacek, Raphaël Lopez Kaufman, Simon Tokumine, Hexiang Hu, Elena Buchatskaya, Yingjie Miao, Mohamed Elhawaty, Aditya Siddhant, Nenad Tomasev, Jinwei Xing, Christina Greer, Helen Miller, Shereen Ashraf, Aurko Roy, Zizhao Zhang, Ada Ma, Angelos Filos, Milos Besta, Rory Blevins, Ted Klimenko, Chih-Kuan Yeh, Soravit Changpinyo, Jiaqi Mu, Oscar Chang, Mantas Pajarskas, Carrie Muir, Vered Cohen, Charline Le Lan, Krishna Haridasan, Amit Marathe, Steven Hansen, Sholto Douglas, Rajkumar Samuel, Mingqiu Wang, Sophia Austin, Chang Lan, Jiepu Jiang, Justin Chiu, Jaime Alonso Lorenzo, Lars Lowe Sjösund, Sébastien Cevey, Zach Gleicher, Thi Avrahami, Anudhyan Boral, Hansa Srinivasan, Vittorio Selo, Rhys May, Konstantinos Aisopos, Léonard Hussenot, Livio Baldini Soares, Kate Baumli, Michael B. Chang, Adrià Recasens, Ben Caine, Alexander Pritzel, Filip Pavetic, Fabio Pardo, Anita Gergely, Justin Frye, Vinay Ramasesh, Dan Horgan, Kartikeya Badola, Nora Kassner, Subhrajit Roy, Ethan Dyer, Víctor Campos Campos, Alex Tomala, Yunhao Tang, Dalia El Badawy, Elspeth White, Basil Mustafa, Oran Lang, Abhishek Jindal, Sharad Vikram, Zhitao Gong, Sergi Caelles, Ross Hemsley, Gregory Thornton, Fangxiaoyu Feng, Wojciech Stokowiec, Ce Zheng, Phoebe Thacker, Çağlar Ünlü,

Zhishuai Zhang, Mohammad Saleh, James Svensson, Max Bileschi, Piyush Patil, Ankesh Anand, Roman Ring, Katerina Tsihlas, Arpi Vezer, Marco Selvi, Toby Shevlane, Mikel Rodriguez, Tom Kwiatkowski, Samira Daruki, Keran Rong, Allan Dafoe, Nicholas FitzGerald, Keren Gu-Lemberg, Mina Khan, Lisa Anne Hendricks, Marie Pellat, Vladimir Feinberg, James Cobon-Kerr, Tara Sainath, Maribeth Rauh, Sayed Hadi Hashemi, Richard Ives, Yana Hasson, Eric Noland, Yuan Cao, Nathan Byrd, Le Hou, Qingze Wang, Thibault Sottiaux, Michela Paganini, Jean-Baptiste Lespiau, Alexandre Moufarek, Samer Hassan, Kaushik Shivakumar, Joost van Amersfoort, Amol Mandhane, Pratik Joshi, Anirudh Goyal, Matthew Tung, Andrew Brock, Hannah Sheahan, Vedant Misra, Cheng Li, Nemanja Rakićević, Mostafa Dehghani, Fangyu Liu, Sid Mittal, Junhyuk Oh, Seb Noury, Eren Sezener, Fantine Huot, Matthew Lamm, Nicola De Cao, Charlie Chen, Sidharth Mudgal, Romina Stella, Kevin Brooks, Gautam Vasudevan, Chenxi Liu, Mainak Chain, Nivedita Melinkeri, Aaron Cohen, Venus Wang, Kristie Seymore, Sergey Zubkov, Rahul Goel, Summer Yue, Sai Krishnakumaran, Brian Albert, Nate Hurley, Motoki Sano, Anhad Mohananey, Jonah Joughin, Egor Filonov, Tomasz Kępa, Yomna Eldawy, Jiawern Lim, Rahul Rishi, Shirin Badiezadegan, Taylor Bos, Jerry Chang, Sanil Jain, Sri Gayatri Sundara Padmanabhan, Subha Puttagunta, Kalpesh Krishna, Leslie Baker, Norbert Kalb, Vamsi Bedapudi, Adam Kurzrok, Shuntong Lei, Anthony Yu, Oren Litvin, Xiang Zhou, Zhichun Wu, Sam Sobell, Andrea Siciliano, Alan Papir, Robby Neale, Jonas Bragagnolo, Tej Toor, Tina Chen, Valentin Anklin, Feiran Wang, Richie Feng, Milad Gholami, Kevin Ling, Lijuan Liu, Jules Walter, Hamid Moghaddam, Arun Kishore, Jakub Adamek, Tyler Mercado, Jonathan Mallinson, Siddhinita Wandekar, Stephen Cagle, Eran Ofek, Guillermo Garrido, Clemens Lombriser, Maksim Mukha, Botu Sun, Hafeezul Rahman Mohammad, Josip Matak, Yadi Qian, Vikas Peswani, Pawel Janus, Quan Yuan, Leif Schelin, Oana David, Ankur Garg, Yifan He, Oleksii Duzhyi, Anton Älgmyr, Timothée Lottaz, Qi Li, Vikas Yadav, Luyao Xu, Alex Chinien, Rakesh Shivanna, Aleksandr Chuklin, Josie Li, Carrie Spadine, Travis Wolfe, Kareem Mohamed, Subhabrata Das, Zihang Dai, Kyle He, Daniel von Dincklage, Shyam Upadhyay, Akanksha Maurya, Luyan Chi, Sebastian Krause, Khalid Salama, Pam G Rabinovitch, Pavan

Kumar Reddy M, Aarush Selvan, Mikhail Dektiarev, Golnaz Ghiasi, Erdem Guven, Himanshu Gupta, Boyi Liu, Deepak Sharma, Idan Heimlich Shtacher, Shachi Paul, Oscar Akerlund, François-Xavier Aubet, Terry Huang, Chen Zhu, Eric Zhu, Elico Teixeira, Matthew Fritze, Francesco Bertolini, Liana-Eleonora Marinescu, Martin Bölle, Dominik Paulus, Khyatti Gupta, Tejasi Latkar, Max Chang, Jason Sanders, Roopa Wilson, Xuewei Wu, Yi-Xuan Tan, Lam Nguyen Thiet, Tulsee Doshi, Sid Lall, Swaroop Mishra, Wanming Chen, Thang Luong, Seth Benjamin, Jasmine Lee, Ewa Andrejczuk, Dominik Rabiej, Vipul Ranjan, Krzysztof Styrc, Pengcheng Yin, Jon Simon, Malcolm Rose Harriott, Mudit Bansal, Alexei Robsky, Geoff Bacon, David Greene, Daniil Mirylenka, Chen Zhou, Obaid Sarvana, Abhimanyu Goyal, Samuel Andermatt, Patrick Siegler, Ben Horn, Assaf Israel, Francesco Pongetti, Chih-Wei "Louis" Chen, Marco Selvatici, Pedro Silva, Kathie Wang, Jackson Tolins, Kelvin Guu, Roey Yogev, Xiaochen Cai, Alessandro Agostini, Maulik Shah, Hung Nguyen, Noah Ó Donnaile, Sébastien Pereira, Linda Friso, Adam Stambler, Adam Kurzrok, Chenkai Kuang, Yan Romanikhin, Mark Geller, ZJ Yan, Kane Jang, Cheng-Chun Lee, Wojciech Fica, Eric Malmi, Qijun Tan, Dan Banica, Daniel Balle, Ryan Pham, Yanping Huang, Diana Avram, Hongzhi Shi, Jasjot Singh, Chris Hidey, Niharika Ahuja, Pranab Saxena, Dan Dooley, Srividya Pranavi Potharaju, Eileen O'Neill, Anand Gokulchandran, Ryan Foley, Kai Zhao, Mike Dusenberry, Yuan Liu, Pulkit Mehta, Ragha Kotikalapudi, Chalence Safranek-Shrader, Andrew Goodman, Joshua Kessinger, Eran Globen, Prateek Kolhar, Chris Gorgolewski, Ali Ibrahim, Yang Song, Ali Eichenbaum, Thomas Brovelli, Sahitya Potluri, Preethi Lahoti, Cip Baetu, Ali Ghorbani, Charles Chen, Andy Crawford, Shalini Pal, Mukund Sridhar, Petru Gurita, Asier Mujika, Igor Petrovski, Pierre-Louis Cedoz, Chenmei Li, Shiyuan Chen, Niccolò Dal Santo, Siddharth Goyal, Jitesh Punjabi, Karthik Kappaganthu, Chester Kwak, Pallavi LV, Sarmishta Velury, Himadri Choudhury, Jamie Hall, Premal Shah, Ricardo Figueira, Matt Thomas, Minjie Lu, Ting Zhou, Chintu Kumar, Thomas Jurdi, Sharat Chikkerur, Yenai Ma, Adams Yu, Soo Kwak, Victor Ähdel, Sujeevan Rajayogam, Travis Choma, Fei Liu, Aditya Barua, Colin Ji, Ji Ho Park, Vincent Hellendoorn, Alex Bailey, Taylan Bilal, Huanjie Zhou, Mehrdad Khatir, Charles Sutton, Wojciech Rzadkowski, Fiona Macintosh, Konstantin Sha-

gin, Paul Medina, Chen Liang, Jinjing Zhou, Pararth Shah, Yingying Bi, Attila Dankovics, Shipra Banga, Sabine Lehmann, Marissa Bredesen, Zifan Lin, John Eric Hoffmann, Jonathan Lai, Raynald Chung, Kai Yang, Nihal Balani, Arthur Bražinskas, Andrei Sozanschi, Matthew Hayes, Héctor Fernández Alcalde, Peter Makarov, Will Chen, Antonio Stella, Liselotte Snijders, Michael Mandl, Ante Kärrman, Paweł Nowak, Xinyi Wu, Alex Dyck, Krishnan Vaidyanathan, Raghavender R, Jessica Mallet, Mitch Rudominer, Eric Johnston, Sushil Mittal, Akhil Udathu, Janara Christensen, Vishal Verma, Zach Irving, Andreas Santucci, Gamaleldin Elsayed, Elnaz Davoodi, Marin Georgiev, Ian Tenney, Nan Hua, Geoffrey Cideron, Edouard Leurent, Mahmoud Alnahlawi, Ionut Georgescu, Nan Wei, Ivy Zheng, Dylan Scandinaro, Heinrich Jiang, Jasper Snoek, Mukund Sundararajan, Xuezhi Wang, Zack Ontiveros, Itay Karo, Jeremy Cole, Vinu Rajashekhar, Lara Tumeh, Eyal Ben-David, Rishub Jain, Jonathan Uesato, Romina Datta, Oskar Bunyan, Shimu Wu, John Zhang, Piotr Stanczyk, Ye Zhang, David Steiner, Subhajit Naskar, Michael Azzam, Matthew Johnson, Adam Paszke, Chung-Cheng Chiu, Jaume Sanchez Elias, Afroz Mohiuddin, Faizan Muhammad, Jin Miao, Andrew Lee, Nino Vieillard, Jane Park, Jiageng Zhang, Jeff Stanway, Drew Garmon, Abhijit Karmarkar, Zhe Dong, Jong Lee, Aviral Kumar, Luowei Zhou, Jonathan Evens, William Isaac, Geoffrey Irving, Edward Loper, Michael Fink, Isha Arkatkar, Nanxin Chen, Izhak Shafran, Ivan Petrychenko, Zhe Chen, Johnson Jia, Anselm Levskaya, Zhenkai Zhu, Peter Grabowski, Yu Mao, Alberto Magni, Kaisheng Yao, Javier Snaider, Norman Casagrande, Evan Palmer, Paul Suganthan, Alfonso Castaño, Irene Giannoumis, Wooyeol Kim, Mikołaj Rybiński, Ashwin Sreevatsa, Jennifer Prendki, David Soergel, Adrian Goedeckemeyer, Willi Gierke, Mohsen Jafari, Meenu Gaba, Jeremy Wiesner, Diana Gage Wright, Yawen Wei, Harsha Vashisht, Yana Kulizhskaya, Jay Hoover, Maigo Le, Lu Li, Chimezie Iwuanyanwu, Lu Liu, Kevin Ramirez, Andrey Khorlin, Albert Cui, Tian LIN, Marcus Wu, Ricardo Aguilar, Keith Pallo, Abhishek Chakladar, Ginger Perng, Elena Allica Abellan, Mingyang Zhang, Ishita Dasgupta, Nate Kushman, Ivo Penchev, Alena Repina, Xihui Wu, Tom van der Weide, Priya Ponnapalli, Caroline Kaplan, Jiri Simsa, Shuangfeng Li, Olivier Dousse, Fan Yang, Jeff Piper, Nathan Ie, Rama Pasumarthi, Nathan Lintz, Anitha Vijayakumar, Daniel

Andor, Pedro Valenzuela, Minnie Lui, Cosmin Paduraru, Daiyi Peng, Katherine Lee, Shuyuan Zhang, Somer Greene, Duc Dung Nguyen, Paula Kurylowicz, Cassidy Hardin, Lucas Dixon, Lili Janzer, Kiam Choo, Ziqiang Feng, Biao Zhang, Achintya Singhal, Dayou Du, Dan McKinnon, Natasha Antropova, Tolga Bolukbasi, Orgad Keller, David Reid, Daniel Finchelstein, Maria Abi Raad, Remi Crocker, Peter Hawkins, Robert Dadashi, Colin Gaffney, Ken Franko, Anna Bulanova, Rémi Leblond, Shirley Chung, Harry Askham, Luis C. Cobo, Kelvin Xu, Felix Fischer, Jun Xu, Christina Sorokin, Chris Alberti, Chu-Cheng Lin, Colin Evans, Alek Dimitriev, Hannah Forbes, Dylan Banarse, Zora Tung, Mark Omernick, Colton Bishop, Rachel Sterneck, Rohan Jain, Jiawei Xia, Ehsan Amid, Francesco Piccinno, Xingyu Wang, Praseem Banzal, Daniel J. Mankowitz, Alex Polozov, Victoria Krakovna, Sasha Brown, MohammadHossein Bateni, Dennis Duan, Vlad Firoiu, Meghana Thotakuri, Tom Natan, Matthieu Geist, Ser tan Girgin, Hui Li, Jiayu Ye, Ofir Roval, Reiko Tojo, Michael Kwong, James Lee-Thorp, Christopher Yew, Danila Sinopalnikov, Sabela Ramos, John Mellor, Abhishek Sharma, Kathy Wu, David Miller, Nicolas Sonnerat, Denis Vnukov, Rory Greig, Jennifer Beattie, Emily Caveness, Libin Bai, Julian Eisenschlos, Alex Korchemniy, Tomy Tsai, Mimi Jasarevic, Weize Kong, Phuong Dao, Zeyu Zheng, Frederick Liu, Fan Yang, Rui Zhu, Tian Huey Teh, Jason Sanmiya, Evgeny Gladchenko, Nejc Trdin, Daniel Toyama, Evan Rosen, Sasan Tavakkol, Linting Xue, Chen Elkind, Oliver Woodman, John Carpenter, George Papamakarios, Rupert Kemp, Sushant Kafle, Tanya Grunina, Rishika Sinha, Alice Talbert, Diane Wu, Denese Owusu-Afriyie, Cosmo Du, Chloe Thornton, Jordi Pont-Tuset, Pradyumna Narayana, Jing Li, Saaber Fatehi, John Wieting, Omar Ajmeri, Benigno Uria, Yeongil Ko, Laura Knight, Amélie Héliou, Ning Niu, Shane Gu, Chenxi Pang, Yeqing Li, Nir Levine, Ariel Stolovich, Rebeca Santamaria-Fernandez, Sonam Goenka, Wenny Yustalim, Robin Strudel, Ali Elqursh, Charlie Deck, Hyo Lee, Zonglin Li, Kyle Levin, Raphael Hoffmann, Dan Holtmann-Rice, Olivier Bachem, Sho Arora, Christy Koh, Soheil Hassas Yeganeh, Siim Põder, Mukarram Tariq, Yanhua Sun, Lucian Ionita, Mojtaba Seyedhosseini, Pouya Tafti, Zhiyu Liu, Anmol Gulati, Jasmine Liu, Xinyu Ye, Bart Chrzaszcz, Lily Wang, Nikhil Sethi, Tianrun Li, Ben Brown, Shreya Singh, Wei Fan, Aaron Parisi, Joe

Stanton, Vinod Koverkathu, Christopher A. Choquette-Choo, Yunjie Li, TJ Lu, Abe Ittycheriah, Prakash Shroff, Mani Varadarajan, Sanaz Bahargam, Rob Willoughby, David Gaddy, Guillaume Desjardins, Marco Cornero, Brona Robenek, Bhavishya Mittal, Ben Albrecht, Ashish Shenoy, Fedor Moiseev, Henrik Jacobsson, Alireza Ghaffarkhah, Morgane Rivière, Alanna Walton, Clément Crepy, Alicia Parrish, Zongwei Zhou, Clement Farabet, Carey Radebaugh, Praveen Srinivasan, Claudia van der Salm, Andreas Fidjeland, Salvatore Scellato, Eri Latorre-Chimoto, Hanna Klimczak-Plucińska, David Bridson, Dario de Cesare, Tom Hudson, Piermaria Mendolicchio, Lexi Walker, Alex Morris, Matthew Mauger, Alexey Guseynov, Alison Reid, Seth Odoom, Lucia Loher, Victor Cotruta, Madhavi Yenugula, Dominik Grewe, Anastasia Petrushkina, Tom Duerig, Antonio Sanchez, Steve Yadlowsky, Amy Shen, Amir Globerson, Lynette Webb, Sahil Dua, Dong Li, Surya Bhupatiraju, Dan Hurt, Haroon Qureshi, Ananth Agarwal, Tomer Shani, Matan Eyal, Anuj Khare, Shreyas Rammohan Belle, Lei Wang, Chetan Tekur, Mihir Sanjay Kale, Jinliang Wei, Ruoxin Sang, Brennan Saeta, Tyler Liechty, Yi Sun, Yao Zhao, Stephan Lee, Pandu Nayak, Doug Fritz, Manish Reddy Vuyyuru, John Aslanides, Nidhi Vyas, Martin Wicke, Xiao Ma, Evgenii Eltyshev, Nina Martin, Hardie Cate, James Manyika, Keyvan Amiri, Yelin Kim, Xi Xiong, Kai Kang, Florian Luisier, Nilesh Tripuraneni, David Madras, Mandy Guo, Austin Waters, Oliver Wang, Joshua Ainslie, Jason Baldridge, Han Zhang, Garima Pruthi, Jakob Bauer, Feng Yang, Riham Mansour, Jason Gelman, Yang Xu, George Polovets, Ji Liu, Honglong Cai, Warren Chen, XiangHai Sheng, Emily Xue, Sherjil Ozair, Christof Angermueller, Xiaowei Li, Anoop Sinha, Weiren Wang, Julia Wiesinger, Emmanouil Koukoumidis, Yuan Tian, Anand Iyer, Madhu Gurumurthy, Mark Goldenson, Parashar Shah, MK Blake, Hongkun Yu, Anthony Urbanowicz, Jennimaria Palomaki, Chrisantha Fernando, Ken Durden, Harsh Mehta, Nikola Momchev, Elahe Rahimtoroghi, Maria Georgaki, Amit Raul, Sebastian Ruder, Morgan Redshaw, Jinhyuk Lee, Denny Zhou, Komal Jalan, Dinghua Li, Blake Hechtman, Parker Schuh, Milad Nasr, Kieran Milan, Vladimir Mikulik, Juliana Franco, Tim Green, Nam Nguyen, Joe Kelley, Aroma Mahendru, Andrea Hu, Joshua Howland, Ben Vargas, Jeffrey Hui, Kshitij Bansal, Vikram Rao, Rakesh Ghiya, Emma Wang, Ke Ye, Jean Michel

Sarr, Melanie Moranski Preston, Madeleine Elish, Steve Li, Aakash Kaku, Jigar Gupta, Ice Pasupat, Da-Cheng Juan, Milan Someswar, Tejvi M., Xinyun Chen, Aida Amini, Alex Fabrikant, Eric Chu, Xuanyi Dong, Amruta Muthal, Senaka Buthpitiya, Sarthak Jauhari, Nan Hua, Urvashi Khandelwal, Ayal Hitron, Jie Ren, Larissa Rinaldi, Shahar Drath, Avigail Dabush, Nan-Jiang Jiang, Harshal Godhia, Uli Sachs, Anthony Chen, Yicheng Fan, Hagai Taitelbaum, Hila Noga, Zhuyun Dai, James Wang, Chen Liang, Jenny Hamer, Chun-Sung Ferng, Chenel Elkind, Aviel Atias, Paulina Lee, Vít Listík, Mathias Carlen, Jan van de Kerkhof, Marcin Pikus, Krunoslav Zaher, Paul Müller, Sasha Zykova, Richard Stefanec, Vitaly Gatsko, Christoph Hirnschall, Ashwin Sethi, Xingyu Federico Xu, Chetan Ahuja, Beth Tsai, Anca Stefanoiu, Bo Feng, Keshav Dhandhania, Manish Katyal, Akshay Gupta, Atharva Parulekar, Divya Pitta, Jing Zhao, Vivaan Bhatia, Yashodha Bhavnani, Omar Alhadlaq, Xiaolin Li, Peter Danenberg, Dennis Tu, Alex Pine, Vera Filippova, Abhipso Ghosh, Ben Limonchik, Bhargava Urala, Chaitanya Krishna Lanka, Derik Clive, Yi Sun, Edward Li, Hao Wu, Kevin Hongtongsak, Ianna Li, Kalind Thakkar, Kuanysh Omarov, Kushal Majmundar, Michael Alverson, Michael Kucharski, Mohak Patel, Mudit Jain, Maksim Zabelin, Paolo Pelagatti, Rohan Kohli, Saurabh Kumar, Joseph Kim, Swetha Sankar, Vineet Shah, Lakshmi Ramachandruni, Xiangkai Zeng, Ben Bariach, Laura Weidinger, Amar Subramanya, Sissie Hsiao, Demis Hassabis, Koray Kavukcuoglu, Adam Sadovsky, Quoc Le, Trevor Strohman, Yonghui Wu, Slav Petrov, Jeffrey Dean, and Oriol Vinyals. 2024. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Touvron, Hugo, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov,

Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Trinh, Trieu H, and Quoc V Le. 2018. A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847*.

Vanian, Jonathan, and Kif Leswing. 2023. Chatgpt and generative ai are booming, but the costs can be extraordinary. *CNBC*.

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

Wang, Alex, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018a. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *International conference on learning representations*.

Wang, Naigang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018b. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems*.

Wang, Sinong, Belinda Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*.

Wei, Jason, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent abilities of large language models. *Transactions on Machine Learning Research*. Survey Certification.

Welbl, Johannes, Pontus Stenetorp, and Sebastian Riedel. 2018. Constructing datasets for multi-hop reading comprehension across documents. *Transactions of the Association for Computational Linguistics*.

Wikimedia. 2019. *Wikimedia downloads*. https://dumps.wikimedia.org.

Williams, Adina, Nikita Nangia, and Samuel Bowman. 2018. A broad-coverage challenge corpus for sentence understanding through inference. In *Conference of the north american chapter of the association for computational linguistics: Human language technologies*.

Wortsman, Mitchell, Tim Dettmers, Luke Zettlemoyer, Ari S. Morcos, Ali Farhadi, and Ludwig Schmidt. 2023. Stable and low-precision training for large-scale vision-language models. In *Advances in neural information processing systems*.

Wu, Shuang, Guoqi Li, Feng Chen, and Luping Shi. 2018. Training and inference with integers in deep neural networks. In *International conference on learning representations*.

Wu, Yuhuai, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. 2022. Memorizing transformers. In *International conference on learning representations*.

Wuu, John, Rahul Agarwal, Michael Ciraula, Carl Dietz, Brett Johnson, Dave Johnson, Russell Schreiber, Raja Swaminathan, Will Walker, and Samuel Naffziger. 2022. 3d v-cache: the implementation of a hybrid-bonded 64mb stacked cache for a 7nm x86-64 cpu. In *Ieee international solid-state circuits conference*, vol. 65, 428–429.

Xiao, Guangxuan, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *International conference on machine learning*.

Xiong, Ruibin, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. 2020. On layer nor-

malization in the transformer architecture. In *International conference on machine learning*.

Xiong, Yunyang, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. 2021. Nyströmformer: A nyström-based algorithm for approximating self-attention. In *Proceedings of the aaai conference on artificial intelligence*.

Xu, Yufei, Qiming Zhang, Jing Zhang, and Dacheng Tao. 2021. Vitae: Vision transformer advanced by exploring intrinsic inductive bias. In *Advances in neural information processing systems*.

Yang, Zhilin, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*.

Yang, Zhilin, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on empirical methods in natural language processing*.

Yang, Zichao, Marcin Moczulski, Misha Denil, Nando De Freitas, Le Song, and Ziyu Wang. 2015. Deep fried convnets. In *The ieee/cvf international conference on computer vision*.

Yuan, Zhihang, Chenhao Xue, Yiqi Chen, Qiang Wu, and Guangyu Sun. 2021. Ptq4vit: Post-training quantization for vision transformers with twin uniform quantization. In *European conference on computer vision*.

Zaheer, Manzil, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big bird: Transformers for longer sequences. In *Advances in neural information processing systems*.

Zellers, Rowan, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019a. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*.

Zellers, Rowan, Ari Holtzman, Hannah Rashkin, Yonatan Bisk, Ali Farhadi, Franziska Roesner, and Yejin Choi. 2019b. Defending against neural fake news. In *Advances in neural information processing systems*.

Zeng, Zhanpeng, Michael Davies, Pranav Pulijala, Karthikeyan Sankaralingam, and Vikas Singh. 2023a. LookupFFN: Making transformers compute-lite for CPU inference. In *International conference on machine learning*.

Zeng, Zhanpeng, Cole Hawkins, Mingyi Hong, Aston Zhang, Nikolaos Pappas, Vikas Singh, and Shuai Zheng. 2023b. VCC: Scaling transformers to 128k tokens or more by prioritizing important tokens. In *Advances in neural information processing systems*.

Zeng, Zhanpeng, Sourav Pal, Jeffery Kline, Glenn M Fung, and Vikas Singh. 2022. Multi resolution analysis (MRA) for approximate self-attention. In *International conference on machine learning*.

Zeng, Zhanpeng, Karthikeyan Sankaralingam, and Vikas Singh. 2024. Im-unpack: Training and inference with arbitrarily low precision integers. In *International conference on machine learning*.

Zeng, Zhanpeng, Yunyang Xiong, Sathya Ravi, Shailesh Acharya, Glenn M Fung, and Vikas Singh. 2021. You only sample (almost) once: Linear cost self-attention via bernoulli sampling. In *International conference on machine learning*.

Zhai, Xiaohua, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. 2022. Scaling vision transformers. In *The ieee/cvf conference on computer vision and pattern recognition*.

Zhang, Chengliang, Minchen Yu, Wei Wang, and Feng Yan. 2019. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving.

In *Proceedings of the 2019 usenix conference on usenix annual technical conference*, 1049–1062. USENIX Association.

Zhang, Xiangyu, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *The ieee/cvf conference on computer vision and pattern recognition*.

Zhong, Ming, Da Yin, Tao Yu, Ahmad Zaidi, Mutethia Mutuma, Rahul Jha, Ahmed Hassan Awadallah, Asli Celikyilmaz, Yang Liu, Xipeng Qiu, and Dragomir Radev. 2021. QMSum: A new benchmark for query-based multi-domain meeting summarization. In *Conference of the north american chapter of the association for computational linguistics: Human language technologies*.

Zhou, Zhaokun, Yuesheng Zhu, Chao He, Yaowei Wang, Shuicheng YAN, Yonghong Tian, and Li Yuan. 2023. Spikformer: When spiking neural network meets transformer. In *International conference on learning representations*.

Zhu, Chenguang, Yang Liu, Jie Mei, and Michael Zeng. 2021. Mediasum: A large-scale media interview dataset for dialogue summarization. In *Conference of the north american chapter of the association for computational linguistics: Human language technologies*.

Zhu, Feng, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. 2020. Towards unified int8 training for convolutional neural network. In *The ieee/cvf conference on computer vision and pattern recognition*.

Zhu, Yukun, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *The ieee/cvf international conference on computer vision*.

Zhu, Zhenhai, and Radu Soricut. 2021. H-transformer-1D: Fast one-dimensional hierarchical attention for sequences. In *Annual meeting of the association for computational linguistics*.