ExpertFlow: 面向混合专家模型的低延迟异步推理

ExpertFlow: Enabling Low-Latency Asynchronous Inference for Mixture of Expert Models

(申请清华大学工学硕士学位论文)

培 养 单 位 ： 计算机科学与技术系

学　　科 ： 计算机科学与技术

研　究　生 ： 欧 俊 杰

指 导 教 师 ： 翟 季 冬　副教授

二〇二三年六月

ExpertFlow: Enabling Low-Latency Asynchronous Inference for Mixture of Expert Models

Thesis submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Master of Science

in

Computer Science and Technology

by

Gabriele Oliaro

Thesis Supervisor: Professor Zhai Jidong

June, 2023

学位论文指导小组、公开评阅人和答辩委员会名单

指导小组名单

翟季冬	副教授	清华大学

公开评阅人名单

翟季冬	副教授	清华大学
薛巍	教授	清华大学
章明星	助理教授	清华大学

答辩委员会名单

主席	薛巍	教授	清华大学
委员	李涓子	教授	清华大学
	刘洋	教授	清华大学
	翟季冬	副教授	清华大学
	张松海	副教授	清华大学
	李鹏	副研究员	清华大学
秘书	戴音		清华大学

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）按照上级教育主管部门督导、抽查等要求，报送相应的学位论文。

本人保证遵守上述规定。

作者签名：_____ 导师签名：_____

日　　期：_____ 日　　期：_____

摘 要

随着深度学习模型规模的指数级增长，稀疏激活的混合专家（MoE）模型架构再次受到了迅速的关注。混合专家模型利用条件计算技术，可以在扩大规模规模的同时，使其训练所需计算量（FLOPs）以次线性的速度增长。随着 AI 计算进入到百亿亿次时代，MoE 层正在逐渐成为深度神经网络（DNNs）的重要组成部分，许多相关研究团队已经投入了大量的资源来建设高效 MoE 模型训练系统。近期，许多 MoE 模型被陆续发布，比如 Switch-C，这次世界上首个公开发布的万亿参数模型。尽管 MoE 模型的训练受到了广泛关注，但目前对其推理的研究较少。本研究提出了一个名为 ExpertFlow 的高效低延迟 MoE 推理系统。该框架支持多 GPU 和多节点的分布式推理，并且基于纯异步任务调用方式实现。实验表明，相比于英伟达提出的 FasterTransformer 框架，ExpertFlow 可以实现 1.95 倍的吞吐率，平均延迟可以降低 13%。为了推广 MoE 模型的使用，我们开源了 ExperFlow 系统的全部代码：https://github.com/flexflow/FlexFlow/tree/inference。

关键词：混合专家，Transformer，模型推理，分布式系统，异步任务

ABSTRACT

The exponential growth in the size of deep learning models has led to a burgeoning interest in the sparsely activated Mixture of Expert (MoE) model architecture. MoE uses conditional computation to scale the model size with a sub-linear growth in the corresponding number of computations (FLOPs) needed to train it. As AI computation enter the exascale computing era, the MoE layer is becoming a key component of deep neural networks (DNNs), and several research teams have dedicated significant resources to building efficient MoE training systems. Recently, many MoE models have been released, including Switch-C, which is the first open-source trillion-parameters model in the world. Although much attention has been given to MoE training, there has been less research on inference. In this thesis, we present ExpertFlow, a low-latency system for efficient inference of MoE models. The framework supports multi-GPU and multi-node distributed inference and is implemented by a fully asynchronous task scheduling manner. We compare ExpertFlow to NVIDIA's FasterTransformer framework in our experiments and find that it achieves up to 1.95x higher throughput and up to 13% lower latency on average. Our goal is to make the MoE architecture more accessible, and we have open-sourced all of our code at https://github.com/flexflow/FlexFlow/tree/inference.

Keywords: Mixture of Experts; Transformer; Inference; Distributed System; Asynchronous Tasks

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ACRONYMS

BSP	Bulk Synchronous Parallel
CI	Continuous Integration
CNN	Convolutional Neural Network
DNN	Deep Neural Network
FFN	Feed-Forward Network
FLOPs	Floating Point Operations Per Second
FT	FasterTransformer
GEMM	General Matrix Multiplication
GPT	Generative Pre-trained Transformer
LLM	Large Language Model
MoE	Mixture of Experts
NCCL	NVIDIA Collective Communications Library
NLP	Natural Language Processing
RNN	Recursive Neural Network

CHAPTER 1 INTRODUCTION

This chapter situates the project within the broader research landscape to enable readers to evaluate its significance and contributions relative to existing literature. We begin by highlighting the importance of Mixture of Experts (MoE) models in deep learning, motivating our efforts to increase the accessibility of this model to machine learning scientists and the wider public. Next, we provide an overview of the challenges associated with MoE inference and introduce existing work in this field. To guide readers through the following chapters, we include an outline of the thesis, serving as a roadmap for navigating the rest of the document.

1.1 Major breakthroughs of MoE models

Mixture of Experts models have emerged as a crucial component of the most extensive deep learning models. Although first introduced in 1991[1], the MoE architecture gained renewed attention in 2017 when Google[2] published a study demonstrating the advantages of incorporating one or more MoE layers into deep neural networks (DNNs) for tasks such as language modeling or machine translation. Since then, the high scalability potential of the model has made it a thriving area of research, and recent studies[3-6] have shown ways to overcome the challenges that had initially prevented the widespread adoption of MoE models. These challenges include implementation difficulty, training instabilities, and high communication costs due to load imbalances[6]. In recent years, the MoE architecture has facilitated the development of the largest DNN models ever created, such as the Switch Transformer[6], which boasts 1.6 trillion parameters, and BaGuaLu[7], which has the potential to create a 174-trillion-parameter AI model, despite not being trained to convergence.

Current industry trends demonstrate that there is a competition among industry leaders to create larger and larger models, driven by the realization that simply increasing the number of parameters, dataset size, and corresponding amount of computational resources utilized to train a model may be the winning strategy to improve the model's generalization power[8]. At the same time, hardware capabilities are struggling to keep pace with the ever-increasing computational costs, and many researchers predict the end of Moore's Law is imminent[9]. If these trends continue, it is highly likely that MoE layers will be-

come an irreplaceable component of the most extensive DNN models in the next few years, as it may not be feasible to train the latest models in a dense manner by updating every parameter for each input token in each iteration.

1.2 Overview of MoE inference

As the challenges of training MoEs are being addressed, the MoE architecture is becoming more popular, and pre-trained MoE model checkpoints are becoming available to the research community. For example, Google recently open-sourced the pre-trained checkpoint of Switch-C, the largest version of their SwitchTransformer model, and the first trillion-parameters pre-trained language model to be available on HuggingFace[10]. The next logical step to allow the general public to benefit from the latest MoE models is to build efficient serving systems to be able to serve predictions at low latency and high throughput. So far, the existing works on MoE inference have been limited. DeepSpeed[11] and Fairseq[12] are the two main publicly-available DNN systems supporting MoE inference. More details about these systems will be offered in Section 2.3.

Building a high-performance MoE inference system is challenging for a variety of reasons. First of all, inference requests have stringent latency requirements, so unlike in the training phase, we cannot focus solely on maximizing the throughput. Next, batching requests is complicated by the fact that requests are generally small, can have varying sequence lengths, and their arrival times is not known in advance. This, together with the fact that each request will utilize only a small fraction of the available experts makes it difficult to achieve high GPU utilization rates. Finally, MoE models tend to be larger than dense ones. This is to be expected, since one of the main selling points of MoE layers is that they allow researchers to trade computational costs for a larger number of parameters. The large size of the models means that in many cases we will need multiple GPUs to serve a MoE model, but current inference systems are built on the assumption that each model will fit on a single GPU. Building a model-parallel inference system for MoE, however, is challenging for many reasons, chiefly among all the communication overheads. In fact, current communication collectives or inter-GPU communication libraries such as NCCL[13] are not optimized for the communication patterns found in sparsely-activated models.

1.3 Key Contributions

The key contributions of the ExpertFlow project are the following:

- We propose a fully asynchronous distributed system for MoE inference, where every computation is split into fine-grained non-preemptible tasks that are launched automatically by the runtime as soon as their data dependencies become available.
- We integrate incremental decoding with speculative inference and dynamic batching to reduce the latency while maintaining the same quality of the output
- We design efficient kernels for the fused experts and attention operators to boost the performance by maximizing the level of parallelism. We implement these kernels in CUDA.

1.4 Thesis overview

The thesis is organized as follows. Chapter 2 offers additional background on the MoE architecture, the Transformer and GPT architectures, and existing inference systems for these models. Chapter 3 offers an high-level overview of ExpertFlow's design and implementation, introducing each of ExpertFlow's components, and explaining their purpose. In addition, the chapter discusses how we built ExpertFlow on top of the FlexFlow asynchronous distributed system, and how we leveraged the asynchronous nature to perform inference more efficiently than a bulk synchronous parallel (BSP) system would be able to support. Chapter 4 discusses the techniques that we used to be able to support auto-regressive transformer models. Chapter 5 describes the custom kernels we built for MoE models. Chapter 6 describes our evaluation effort, and includes all our performance measurements. Chapter 7 is dedicated to the conclusions, lesson learned from the project, limitations of ExpertFlow, and ideas for future work.

CHAPTER 2 BACKGROUND AND RELATED WORKS

In this chapter, we introduce the relevant background (Section 2.1) and related works (Section 2.2).

2.1 Background

Our inference system is designed primarily for serving GPT-based MoE models. In this section, we discuss the components of a typical GPT-MoE architecture, starting from the underlying transformer architecture (Section 2.1.1), the GPT architecture and how it differs from the original transfomer model (Section 2.1.2), the MoE layer (Section 2.1.3), and finally how GPT models can be *MoEfied* to effectively incorporate MoE layers (Section 2.1.4).

2.1.1 Transformer

The transformer architecture is a type of neural network architecture that was introduced in 2017 by Vaswani et al.[14] It was primarily designed for natural language processing (NLP) tasks such as language modeling, machine translation, or text summarization. Since then, transformers have become the most popular architecture in deep learning, and they have been adapted for media types other than text, for example images[15], audio[16], and video[17].

The key component in the transformer architecture is the self-attention module, which allows the model to take into account the context of a given token by weighing the relevance of different parts of the input sequence while processing it. During the training phase, when the context both before and after the current word is available, a transformer may use a bidirectional attention. During inference, on the other hand, the transfomer uses a causal attention mechanism to only consider the previous tokens as the context for the current word. Compared to previous NLP architectures such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs), the transformer architecture has the advantage that it does not require the sequential processing of input tokens. This opens up opportunities for parallel implementation, which translate to better performance.

The transformer architecture (see Figure 2.1) consists of an encoder and a decoder, each composed of multiple layers of identical sub-modules. The encoder takes an input

sequence and generates a hidden representation of the sequence, while the decoder takes the hidden representation and generates an output sequence.

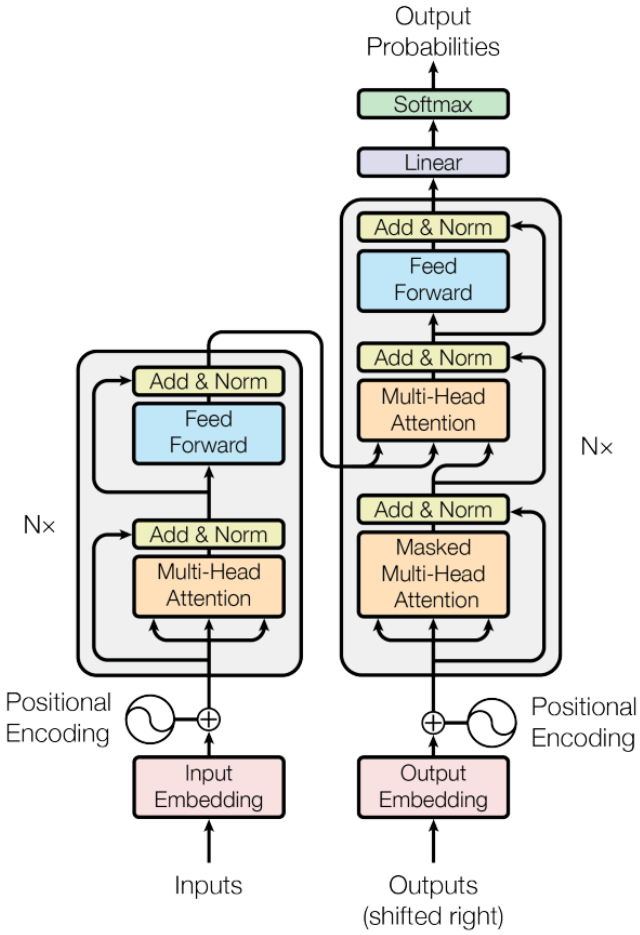


Figure 2.1 **The Transformer architecture**. The illustration[14] shows the original Transformer architecture, with both the encoder and the decoder modules, featuring the multihead-attention operator

Each layer in the transformer architecture contains two types of sub-modules: the multi-head attention module (see Figure 2.2) and the feedforward neural network. The encoder layer only has a single attention sub-module, whereas the decoder contains two, unless the model is a decoder-only model. The multi-head attention module computes a weighted average of the input sequence, where the weights are determined by the attention mechanism. The attention mechanism computes the similarity between each token in the input sequence and every other token, and uses these similarities to compute a set of attention weights. These weights are used to compute a weighted average of the input sequence, where the weights reflect the importance of each token in the sequence. The feedforward neural network applies a linear transformation to the input sequence, followed

by a non-linear activation function such as ReLU. Finally, a layer normalization is applied to the output of the attention, and a residual connection is added. The transformer architecture also includes positional encoding, which allows the model to distinguish between tokens based on their position in the sequence. The positional encoding is added to the input embeddings before they are processed by the decoder.

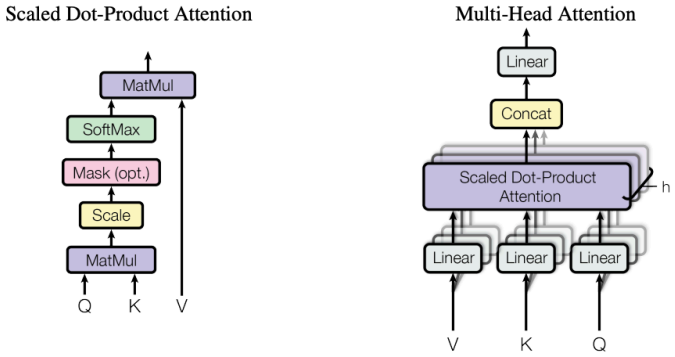


Figure 2.2 **The Multihead Attention operator** [14]

2.1.2 GPT models

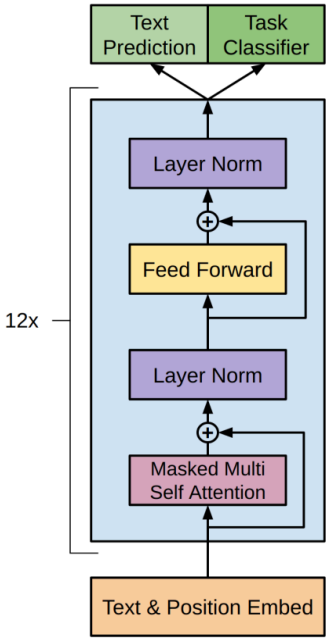


Figure 2.3 **The GPT architecture**. The illustration [18] shows the original GPT architecture. GPT-2 and GPT-3 use the same architecture, with very small modifications

A typical GPT architecture is shown in Figure 2.3. The GPT architecture[18-19] differs from a plain Transformer architecture in a few ways. Overall, the main difference is that GPT models are decoders-only, meaning that they only employ the decoder layers from

the transformer, and do not use the encoder layers. In addition to dropping the encoder layers, the GPT has the following characteristics:

- Unidirectional attention: The original Transformer architecture is bidirectional, meaning that it takes into account both past and future tokens when encoding a sequence of text. The GPT architecture, on the other hand, is unidirectional and only considers the past tokens when generating output.
- Causal Masking: The original Transformer architecture is only required to use masking to prevent the model from attending to future tokens during inference, whereas the GPT architecture uses masking to prevent the model from attending to any tokens beyond the current one during both training and inference.
- Positional Encoding: The GPT architecture uses a slightly different method of positional encoding compared to the original Transformer architecture. In GPT, the positional encoding is added directly to the input embeddings, whereas in the original Transformer architecture, the positional encoding is added to the output of the multi-head attention layer.
- Training tasks: The pre-training objectives used for the GPT architecture are different from the original Transformer. Specifically, GPT is pre-trained using a language modeling objective, where the model is trained to predict the next token in a sequence, while the original Transformer is pre-trained using a masked language modeling objective, where the model is trained to predict masked tokens in a sequence.

Overall, the GPT architecture is a modification of the Transformer architecture, tailored specifically for language generation tasks such as text completion and dialogue generation. The differences in architecture and pre-training objectives are designed to make the model better suited to these tasks, resulting in state-of-the-art performance on a variety of natural language generation benchmarks.

2.1.3 Mixture of Experts

A Mixture-of-Experts (MoE) layer, whose structure can be seen in Figure 2.4, combines a set of *experts* with a *gating network* (also called *routing network*). Each expert is usually a feed-forward neural network (FFN), and the gating network is a function routing each input token to a sparse subset of the experts. Through training of the gating network, different experts will thus end up specializing on different subsets of the input data. The dynamic nature of the MoE architecture comes from two main sources. First, tokens are

not distributed equally among experts (although MoE models tend to use loss terms to prevent excessive load imbalance) due to the use of the gating function. In Figure 2.5, for example, we can see how the popularity of various experts changes over the course of the training iterations. Second, in many MoE models, the *capacity factor* is adjusted dynamically during training[4,6,20], resulting in additional changes to the execution plan.

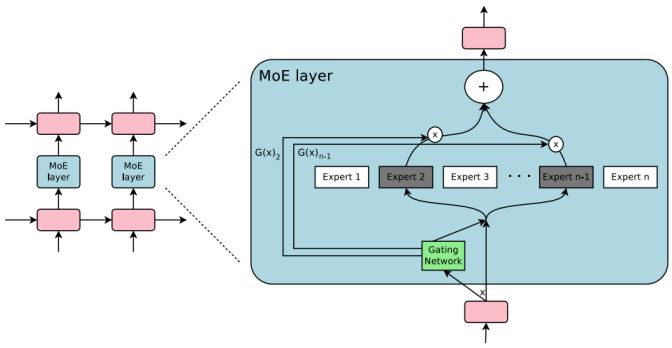
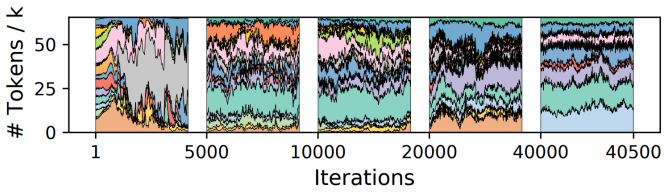
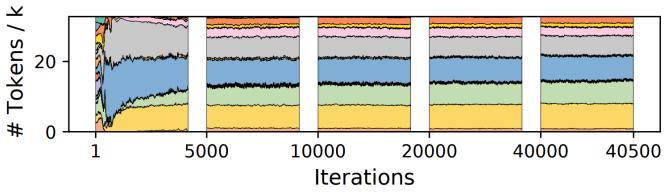


Figure 2.4 **The Mixture of Expers (MoE) architecture**. The illustration[2] shows a sparsely-gated MoE layer embedded within a recurrent language model.



(a) Layer 12 of MoE-BERT-Deep.



(b) Layer 8 of MoE-GPT.

Areas with different colors represent different experts.

Figure 2.5 **Load balance among experts across iterations**[5] Two models show different patterns of expert popularity over the course of training with the FasterMoE framework

2.1.3.1 Architecture

A Mixture-of-Experts (MoE) layer can be defined as follows[2]:

$$y = \sum_{i}^{n} G(x)_i E_i(x) \tag{2.1}$$

where E_i is the i-th expert and $G(x)$ is a gating network returning a sparse n-dimensional vector. There can be thousands of experts, but whenever $G(X)_i = 0$, we

don't have to compute $E_i(x)$, since it won't be a factor in the output.

An important parameter in MoE architectures is the *expert capacity*, or the batch size of each expert (the maximum number of tokens than can routed to an expert). A commonly-used formula[4] for the expert capacity is shown in Equation 2.2, where k is the parameter from the top-k gating function, T is the number of tokens per batch, E is the number of experts, and f is the *capacity factor*.

$$expert\_capacity = k \cdot f \cdot \frac{T}{E} \tag{2.2}$$

The expert capacity is usually tuned through the capacity factor, and allows us to trade between the likelihood of expert overflow (when the number of token dispatched to an expert exceeds the expert capacity) and greater computation and communication costs[6].

2.1.4 GPT-MoE

The GPT model can be easily *MoEfied* by substituting the feed forward network in the decoder with a MoE layer. For example, the sparse models[21] whose checkpoints are available in the Fairseq repository ① use 12 (15B-parameters model), 24 (52B- parameters and 207B-parameters models) or 32 (1.1T-parameters model) decoder layers, where every other layer contains a MoE component instead of the regular FFN. In Listing 2.1, we can see the 15B parameters model, where the six even-numbered decoder layers contain a FFN with two fully-connected layers, and the six odd-numbered decoder layers contain a MoE Layer instead.

Listing 2.1 Fairseq's 15B-parameters MoE model

```
TransformerLanguageModel(
  (decoder): TransformerDecoder(
    (dropout_module): FairseqDropout(p=0.1)
    (embed_tokens): Embedding(50264, 768, padding_idx=1)
    (embed_positions): SinusoidalPositionalEmbedding()
    (layers): ModuleList(
      (0): TransformerDecoderLayer(
        [checkpointed]
        (dropout_module): FairseqDropout(p=0.1)
        (self_attn): MultiheadAttention(
          (dropout_module): FairseqDropout(p=0.1)
          (k_proj): Linear(in_features=768, out_features=768, bias=True)
          (v_proj): Linear(in_features=768, out_features=768, bias=True)
          (q_proj): Linear(in_features=768, out_features=768, bias=True)
          (out_proj): Linear(in_features=768, out_features=768, bias=True)
        )
        (self_attn_layer_norm): LayerNorm((768,), eps=1e-05,
            elementwise_affine=True)
        (activation_dropout_module): FairseqDropout(p=0.0)
        (fc1): Linear(in_features=768, out_features=3072, bias=True)
```

① https://github.com/facebookresearch/fairseq/tree/main/examples/moe_lm

```
        (fc2): Linear(in_features=3072, out_features=768, bias=True)
        (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=
            True)
      )
      (1): TransformerDecoderLayer(
        [checkpointed]
        (dropout_module): FairseqDropout(p=0.1)
        (self_attn): MultiheadAttention(
          (dropout_module): FairseqDropout(p=0.1)
          (k_proj): Linear(in_features=768, out_features=768, bias=True)
          (v_proj): Linear(in_features=768, out_features=768, bias=True)
          (q_proj): Linear(in_features=768, out_features=768, bias=True)
          (out_proj): Linear(in_features=768, out_features=768, bias=True)
        )
        (self_attn_layer_norm): LayerNorm((768,), eps=1e-05,
            elementwise_affine=True)
        (moe_layer): MOELayer()
        (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=
            True)
      )

      ...


      (11): TransformerDecoderLayer(
        [checkpointed]
        (dropout_module): FairseqDropout(p=0.1)
        (self_attn): MultiheadAttention(
          (dropout_module): FairseqDropout(p=0.1)
          (k_proj): Linear(in_features=768, out_features=768, bias=True)
          (v_proj): Linear(in_features=768, out_features=768, bias=True)
          (q_proj): Linear(in_features=768, out_features=768, bias=True)
          (out_proj): Linear(in_features=768, out_features=768, bias=True)
        )
        (self_attn_layer_norm): LayerNorm((768,), eps=1e-05,
            elementwise_affine=True)
        (moe_layer): MOELayer()
        (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=
            True)
      )
    )
    (layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    (output_projection): Linear(in_features=768, out_features=50264, bias=
        False)
  )
)
```

2.2 Related Works

In this section, we introduce existing serving systems for Transformers/GPT (Section 2.2.1) and GPT-MoE (Section 2.3) models.

2.2.1 Transformer serving systems

There is a large number of systems that can serve transformers. In this section, we focus on the systems that are specifically optimized for transformers. These systems are also the ones that have more similar components to ExpertFlow.

2.2.1.1 TurboTransformers

TurboTransformer[22] is an inference system for transformers from Tencent. The system consists of two components (see Figure 2.6): a serving framework, and a computational runtime.

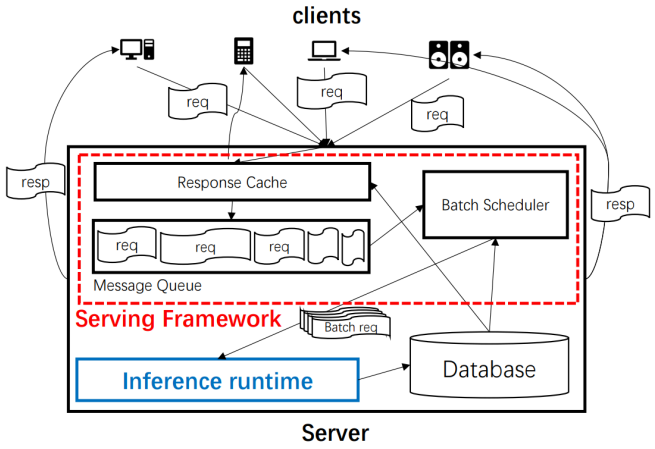


Figure 2.6 **The TurboTransformers system** A diagram[5] showing the components of the TurboTransformers system

The serving framework receives requests from the user via a gRPC/HTTP endpoint, and uses a dynamic programming algorithm to decide how to batch together the requests that arrive over a specific interval of time. The batch assignment considers the length of each request and attempts to minimize the amount of padding needed while also maximizing the performance gains that come from batching together as many requests as possible.

The computational runtime takes the DNN model as input, represents it as a computational graph, and fuses all the non-GEMM kernels together. The fused computations are implemented with custom CUDA kernels that come with the TurboTransformers framework. In particular, the framework focuses on optimizing the performance of reduction kernels, such as those needed in the Softmax and LayerNorm operators. In addition to the kernel optimizations, the computational runtime also includes a custom memory manager to minimize the GPU memory overhead by reusing as much allocated memory as possible to store intermediate results. This is made possible thanks to a memory allocator that

takes into consideration the topology of the computational graph to decide when to assign different tensors to the same region in memory.

The main limitations of the TurboTransformers system are the lack of support for the MoE architecture, and more importantly, the lack of support for multi-GPU and multi-node machines.

2.2.1.2 FasterTransformer and Triton

FasterTransformer[23] (FT) is an open-source inference system by NVIDIA supporting multi-GPU execution. As the name suggests, the inference system is specialized for Transformers, and it supports a wide variety of models, including BERT[24], XLNet[25], GPT[18-19,26], OPT[27], GPT-MoE, BLOOM[28], GPT-J[29], Longformer[30], T5[31], Swin Transformer[32], and ViT[15]. The FasterTransformer's backend is implemented using C++, CUDA and cuBLAS, and several front-ends are available, including C++, TensorFlow and PyTorch. The repository contains an example MoE implementation ①, which we use in our evaluation to compare the performance of ExpertFlow with FT.

Unlike TurboTransformers, FasterTransformer does not come with a serving framework, but NVIDIA released Triton[33], an open-source inference system that can be coupled with FasterTransformer via the Triton FasterTransformer Backend[34].

2.2.1.3 Orca

Orca[35] is a distributed inference system for transformers, employing optimizations such as dynamic batching and iteration-level scheduling to boost the performance of variable-length requests whose arrival times are not known in advance. Unfortunately, the Orca code is closed-source, and the authors are not planning to make it publicly available as it is being used in production by for-profit company FriendliAI, so we cannot compare the performance with our system.

2.2.2 Speculative decoding systems

Two speculative decoding systems[36-37] have been introduced in recent months. The main difference between these systems and ExpertFlow is that the former ones only support the use of a single *speculator* (i.e. a single small model), which may not align well with the larger target model because of the gap in the number of parameters between the

① https://github.com/NVIDIA/FasterTransformer/blob/main/docs/gpt_guide.md#gpt-with-moe

smaller and larger model. On the other hand, ExpertFlow supports the use of a combination of boost-tuned smaller models, which greatly improve the quality of the speculated tokens.

2.3 MoE Serving Systems

There are several distributed inference systems that are capable of serving MoE models. For example, FasterTransformer mentioned above, together with Fairseq[12] (with additional support for the Tutel[4] optimizations) and Alpa[38]. These systems, however, were designed primarily for training and do not have as many optimizations as DeepSpeed-MoE[11] for the MoE inference scenario.

2.3.1 DeepSpeed-MoE

DeepSpeed-MoE[11] is currently the most well-optimized distributed inference framework with best performance for serving MoE models. The optimizations behind DeepSpeed-MoE's performance can be divided in three categories: the parallelization plan (see Figure 2.7), the hierarchical all-to-all communications (see Figure 2.8), and the use of an efficient kernel for the MoE layers.

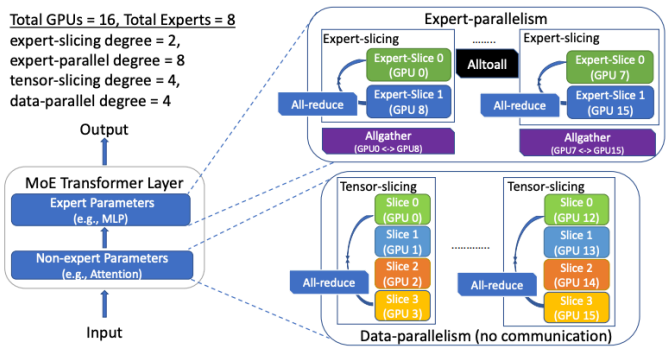


Figure 2.7 **The DeepSpeed-MoE parallelization mechanism[11]**

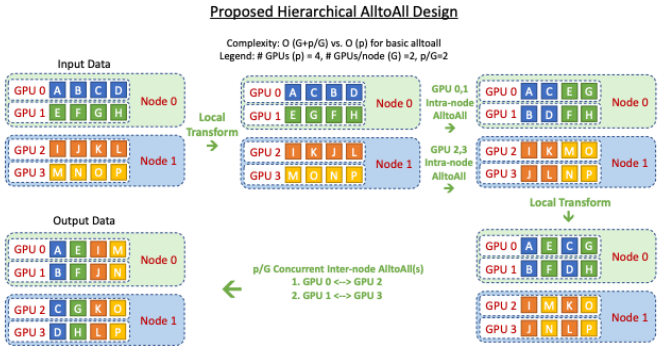


Figure 2.8 **The DeepSpeed-MoE Hierarchical all-to-all design**[11]

DeepSpeed-MoE's parallelization plan consists of a combination of data, model and expert parallelism. Figure 2.7 shows how a GPT-MoE model will be parallelized on a machine with 16 GPUs. In each MoE layer, experts are divided in 8 blocks (`expert parallel degree = 8`), and the tensor containing the parameters for each expert block is partitioned across 2 GPUs (`expert slicing degree = 2`). The tensors containing the parameters from the non-MoE layers are split across 4 GPUs (`tensor slicing degree = 4`), and they are replicated 4 times (`data parallel degree = 4`).

CHAPTER 3 DESIGN AND IMPLEMENTATION

This chapter introduces the overall design of the ExpertFlow system, and includes a high-level description of all of its components. We begin with an overview (Section 3.1), followed by a discussion of ExpertFlow's fully asynchronous runtime (Section 3.2). Next, we discuss how we parallelize the computations across GPUs (Section 3.3). Later, we illustrate ExpertFlow's speculative inference component (Section 3.4). Finally, we discuss the implementation of ExpertFlow, and the relation between ExpertFlow, FlexFlow[39], and Legion[40] (Section 3.5).

3.1 Overview

The ExpertFlow system is pictured in Figure 3.1. To better illustrate the interplay among the various components of the system, we added numbers 1-7 on the figure below. The numbers show the path that a request takes from the user through the system. We discuss the element corresponding to each number below.

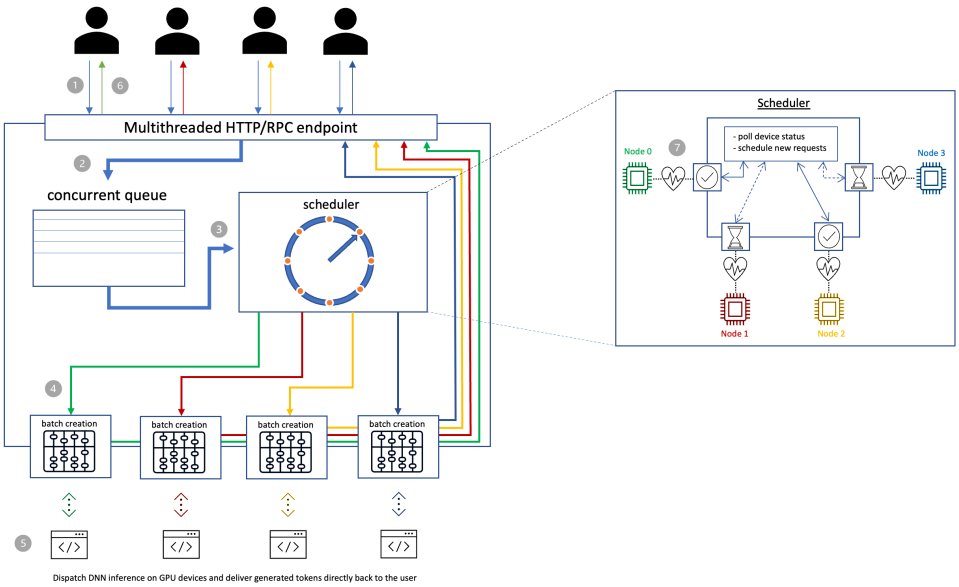


Figure 3.1 **The ExpertFlow system**

The ExpertFlow frontend consists of an interactive interface, through which the user can submit a request (1) by typing a prompt in a text box and pressing the submit button. The request is transmitted to the ExpertFlow server via HTTP or RPC, and received by a

multithreaded server that is able to handle multiple requests from multiple users simultaneously. As they are received, the requests are stored in a high-speed lock-free concurrent queue (2), where they wait for their turn to be processed. The following steps in each request's journey through the ExpertFlow system are determined by the scheduler (3). When ExpertFlow is running, the scheduler polls each GPU node on the cluster to determine the current load, and dispatches requests to the first available node in round-robin fashion. After a request is assigned to a GPU, it will be routed to the corresponding Batch Manager (4), which is responsible for copying the request's tokens onto the model's input tensor, and updating the tensor metadata. Once the batch is ready, the runtime uses the FlexFlow[39,41] API to launch the asynchronous tasks (5) that allow us to execute the DNN model. Once each iteration completes, the generated tokens are immediately returned to the original users (6). Note that while in the illustration above, for the sake of simplicity, each batch contains requests from a distinct user, batches generally contain requests from many different users at a time. The requests do not need to have the same length, nor to be at the same generation phase (e.g. at a given iteration, in a batch, a request may be in the process of generating token #5, while another request may be generating token #9). After each iteration has completed, if more requests are present in the queue, the scheduler (7) will work together with the Batch Manager to add to the batch as many such requests as can fit while retaining the generated tokens from existing requests that have not yet completed. After rearranging, the updated batch is again sent to the corresponding node for the next iteration.

3.2 Asynchronous Tasks-Based Runtime

The ExpertFlow runtime is fully asynchronous. Each computation is split into small, non-preemptible tasks, which constitute the unit of parallelization. When a tasks is launched, it returns a *future*, and the runtime will automatically schedule the task as soon as its data dependencies become available. The main advantage of using an asynchronous system is that it allows us to reduce the GPU idle time in the presence of computations that take different amount of times on different GPUs. This is often the case when working with MoE models, as in each MoE layer experts (either separately or in blocks) are often placed on different devices, and tokens are distributed unevenly across experts, leading to some devices having to do more work than others.

In the figures below, we illustrate the advantage of using an asynchronous system by

comparing the schedule of computations for a one-layer MoE model (parallelized with expert and data parallelism) in the synchronous and asynchronous scenarios. In both cases, each GPU processes a batch of requests and runs the multi-head attention, gating network, and top-k layers locally. Then, the experts layer is executed by dispatching tokens to the experts, which are sharded across GPUs. The order in which the experts computations from each batch (represented in Figure 3.2 and Figure 3.3 with different colors) are scheduled does not matter, as there are no data dependencies.

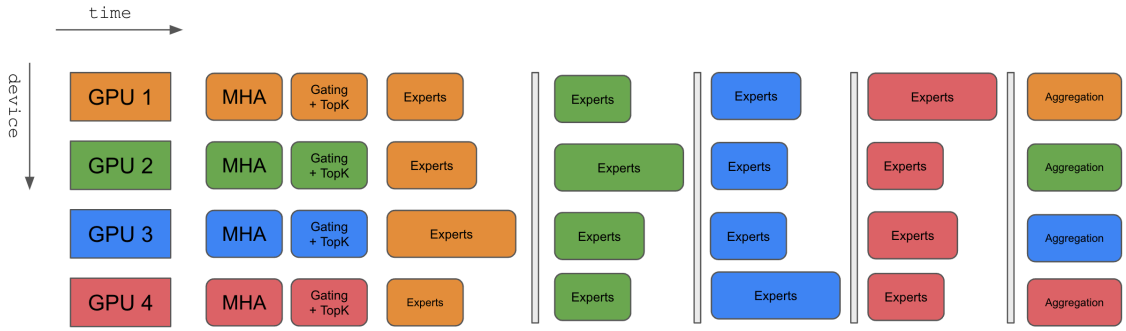


Figure 3.2 **Schedule of MoE computations in a synchronous scenario.** There is no data dependency among the expert computations of different data-parallel batches (represented here in different colors). However, we have to wait until all GPUs have finished their expert-layer computations from the previous batch before we can start working on the next, leading to idle time.

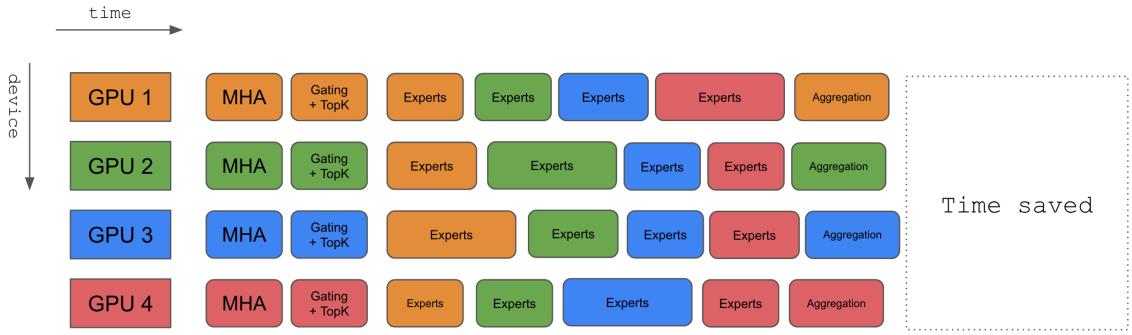


Figure 3.3 **Schedule of MoE computations in the asynchronous scenario.** The asynchronous runtime allows us to remove the synchronization barriers after each experts layer, allowing GPUs that finish their computations earlier to start working on the next tasks

In Figure 3.3, we can see that after removing the synchronization barriers, the GPU idle time is greatly reduced, leading to a smaller total execution time.

3.3 Parallelization Plan

ExpertFlow uses a combination of data, model and pipeline parallelism. A typical GPT-MoE model parallelized by ExpertFlow is shown in Figure 3.4. The parallelization

plan for a given machine is controlled by several parameters. In particular, the number of in-flight batches (P), the number of available devices (D), the number of experts (E_{tot}) and the desired size of each block of fused experts (E_{block}).

Data parallelism is enabled whenever the number of in-flight batches is larger than 1 and the number of available GPUs is also greater than 1 (Equation 3.1). In this setup, up to $min(P, D)$ inflight batches are independently handled by a separate device, which will take care of both handling the data and running the model. The inflight batch is assigned to the device by the scheduler.

$$\text{Data parallelism} \iff P > 1 \land D > 1 \tag{3.1}$$

Model parallelism is enabled in two different scenarios. First of all, if the size of the model is larger than the memory available on any given GPU, we will be required to split the model's weight tensors across multiple GPUs, in tensor parallelism fashion. In addition, a particular form of model parallelism called *expert parallelism* is available to the MoE layers in our models. Expert parallelism is activated whenever the condition in Equation 3.2 is met. Expert parallelism can be used for a similar reason as model parallelism: when the number of experts is significant (as is often the case), we may not be able to fit of all of them on a single device, so we can instead distributed them across the available nodes.

$$\text{Expert parallelism} \iff E_{tot} > E_{block} \land D > 1 \tag{3.2}$$

Expert parallelism differs, however, from tensor parallelism in that each expert weight is not (necessarily) partitioned. Experts are sorted in groups, with E_{block} experts in each block. The most naive sorting algorithm can simply assign experts to blocks using each expert's index, but more effective sorting will instead consider each expert's popularity (separating the most popular experts to avoid overwhelming a node), which can be estimated either offline or online. Unlike the training phase, the weights of the gating network, which effectively determine the distribution of the expert assignment function, do not change over time, so we can afford to use more static load balancing assignments. In addition to efficiently sorting the experts into blocks and placing the expert blocks on different devices, we also employ *experts fusion* to increase the device utilization, especially under the smaller batch sizes that are typical in the inference phase. Expert fusion consists of fusing the computations from the experts in a block into a single kernel. Finally, *pipeline parallelism* allows us to run multiple stages of a model in parallel. This form of parallelism can be deployed manually by dividing a model's layers in different stages and

placing them on different GPUs. More commonly, pipeline parallelism can be activated automatically as a consequence of the asynchronous execution of the model's inference tasks when Equation 3.3 holds, and especially when the number of CUDA streams is larger than 1.

$$\text{pipeline parallelism} \implies P > D \tag{3.3}$$

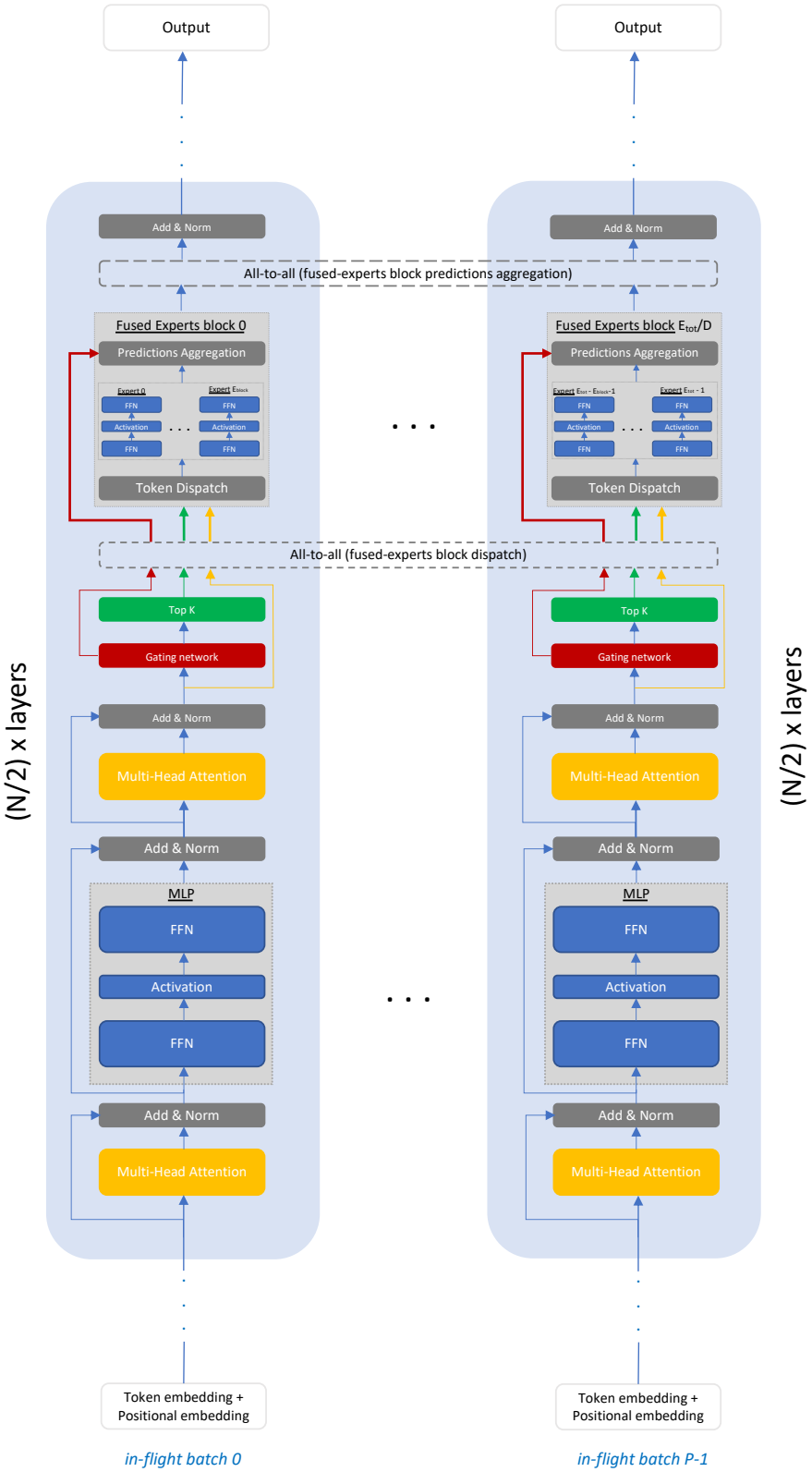


Figure 3.4 **A typical GPT-MoE model parallelized in ExpertFlow**

3.4 Speculative Inference

In order to further speed up the inference process when using large models, we employ the techniques of *speculative inference* and *token tree verification*[42]. These techniques can be used when we have access to one or multiple smaller versions of the target MoE model. The smaller model(s) will be less accurate than the bigger one, but they will be able to generate tokens much faster. We can combine the speed of the smaller model(s) and the accuracy of the bigger model by using the smaller model(s) in incremental decoding mode for a few steps, merging the candidate tokens in a tree, and verifying all the branches of the tree at once using the larger model. Whenever the bigger model detects an incorrect generated token, it issues a correction, discards the subsequent generated tokens, and hands back control to the smaller model(s). Figure 3.5 show a simplified illustration of our system, comparing it to a more traditional incremental decoding system.

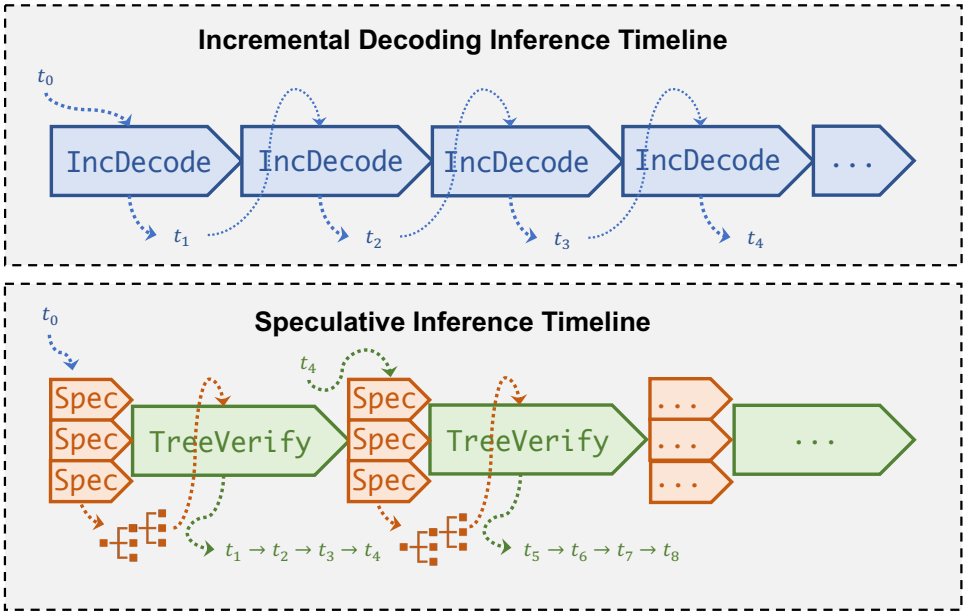


Figure 3.5 **Overview of Speculative Inference and Token Tree Verification in ExpertFlow**

Given a target model, at each step, the incremental decoding system runs the full model to generate the next token in the sequence. The generated token is then passed to the model again, and the process repeats until the full sequence has been generated. Hence, the total running time to generate N tokens will be equal to N times the latency of generating a single token. Our speculative inference system, on the other hand, first uses a set of smaller model(s), which we call *speculator(s)*, to generate a tree of potential tokens for the next spots in the sequence. Because we use speculators that are one or two orders of magnitude smaller than the target model, the time required to generate all the token tree

predictions using the speculators is generally negligible compared to the time required to generate even a single token using the much larger target model.

In the simplest case, we can use a single speculator, consisting of a smaller version of the larger MoE model. In this scenario, the speculated trees will form a single sequence that we can directly pass to the larger model for verification. In more complex cases, we can use a pool of collectively *boost-tuned* small models, where each small model can be either an already existing smaller version of the large MoE model, or it can be generated through distillation or quantization. After each small model has generated a distinct sequence of potential tokens, we merge the sequences into a single token tree, to remove duplicate tokens (i.e. identical tokens at the same position in different branches). After generating the tree of candidate tokens, we flatten the tree in depth-first search (DFS) order and perform token tree verification by running the larger target model on the resulting sequence. The full process of the token tree generation and verification is illustrated in Figure 3.6.

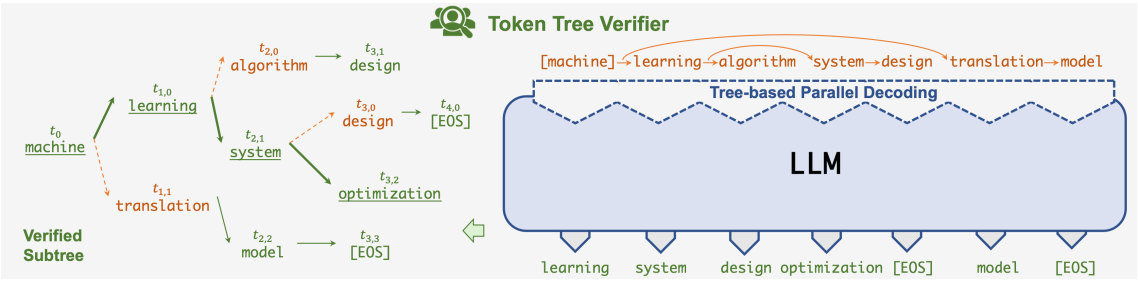


Figure 3.6 **Token Tree Generation and Token Tree Verification in ExpertFlow**

Because the number of tokens in the tree is small, the time required to verify all the tokens in the tree will be essentially equivalent to the time required to generate a single token. The total time saved by using speculative inference can then be approximated by the average token matching rate, or the number of tokens in the token tree that, on average, match the correct tokens as determined by larger model.

3.5 Implementation

We build ExpertFlow on top of FlexFlow[39,41], a distributed DNN framework originally designed for training. We extend FlexFlow to support inference. We reuse FlexFlow's abstractions for deep learning, in particular the `Layer`, `Tensor`, `ParallelTensor` and `Operator` abstractions. Thanks to these abstractions, we can build the MoE model using a similar API to PyTorch or Tensorflow, and use

FlexFlow's compiler to automatically translate it into a parallel computational graph, where each node is an asynchronous task, and each edge is a data dependency.

3.5.1 Legion

FlexFlow is built on top of Legion[40] (Section 3.5.1), a data-centric parallel programming framework. Legion is an asynchronous, task-based distributed execution engine. Tasks are the basic unit of the Legion parallel computation, their execution is non-preemptible, and they can call any C++ function, including those allocating or deallocating memory, but they cannot use packages other than Legion to implement parallelism or concurrency. To write a Legion program, the user implements each task as a C++ function (which may call other functions), and defines each task's inputs and outputs using Legion logical regions.

Internally, tasks are defined as operations that transform one or more logical regions, and the system automatically schedules and executes these tasks in parallel, based on their dependencies and resource availability.

In Legion, control flow is handled using a dynamic dependency graph, which represents the dependencies between different tasks and logical regions. As tasks complete, the system automatically updates the graph and schedules new tasks to execute, based on their dependencies. This approach provides a more flexible and expressive way to manage control flow, and can adapt to changes in the system at runtime.

3.5.2 Mapping

The system provides a programming model that is based on the concept of logical regions, which are abstract data structures that can be mapped to physical memory resources in a way that is transparent to the programmer. The Legion mapping interface is the mechanism through which the logical regions are mapped to physical resources. The interface provides a set of functions that allow the programmer to specify how the logical regions should be mapped, as well as to query the system for information about the mapping. The mapping interface also provides a mechanism for specifying constraints on the mapping. These constraints can be used to specify affinity between the logical region and the physical resources, as well as to specify other properties of the mapping, such as the layout of the physical memory.

Legion allows the programmer to register a custom mapper to make the mapping policy decisions. In ExpertFlow we largely reuse the FlexFlow custom mapper, with a few

modifications. In particular, we manually map the operators according to the parallelization plan discussed in Section 3.3.

3.6 Debugging and Profiling

One advantage of building our system on top of Legion is the ability to use all the Legion tools that are available for debugging and profiling. In particular, during the development of ExpertFlow we frequently used Legion Spy ① as well Legion Prof ② for profiling. For instance, a screenshot of the profiling results from our MoE inference application is shown below:

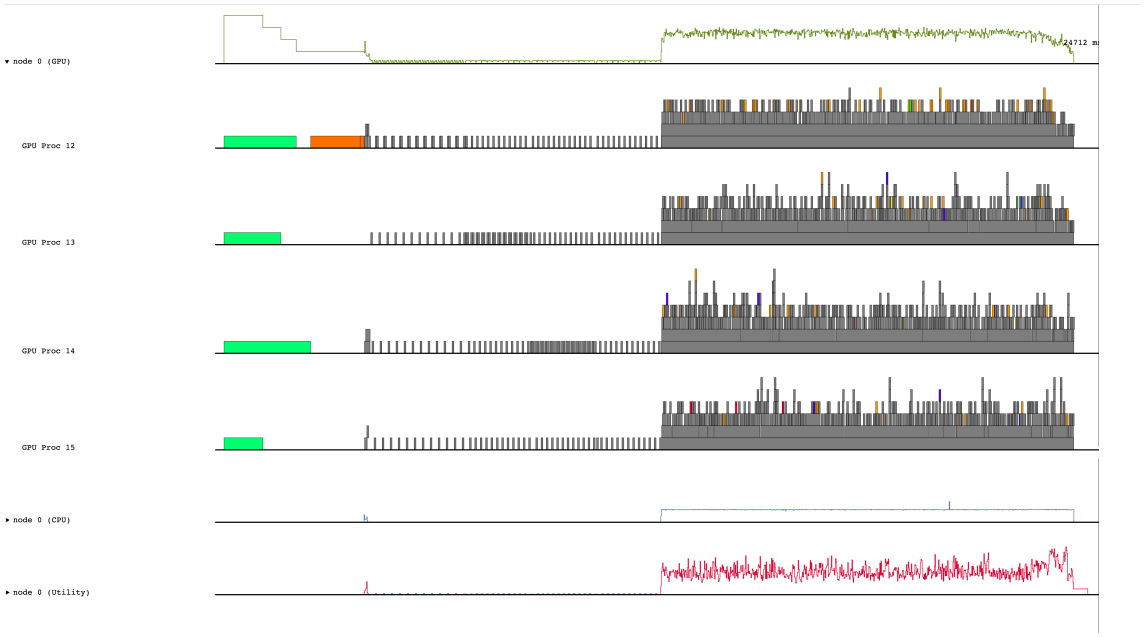


Figure 3.7 **Legion Prof** output for a ExpertFlow test run

From the figure, we can check the utilization of each CPU and GPU, the communication between nodes, and much more. Zooming in (see for example Figure 3.8) we can see each task individually, and get more fine-grained details, such as whether a CPU/GPU task is blocking.

① https://legion.stanford.edu/debugging/index.html#legion-spy
② https://legion.stanford.edu/profiling/#legion-prof

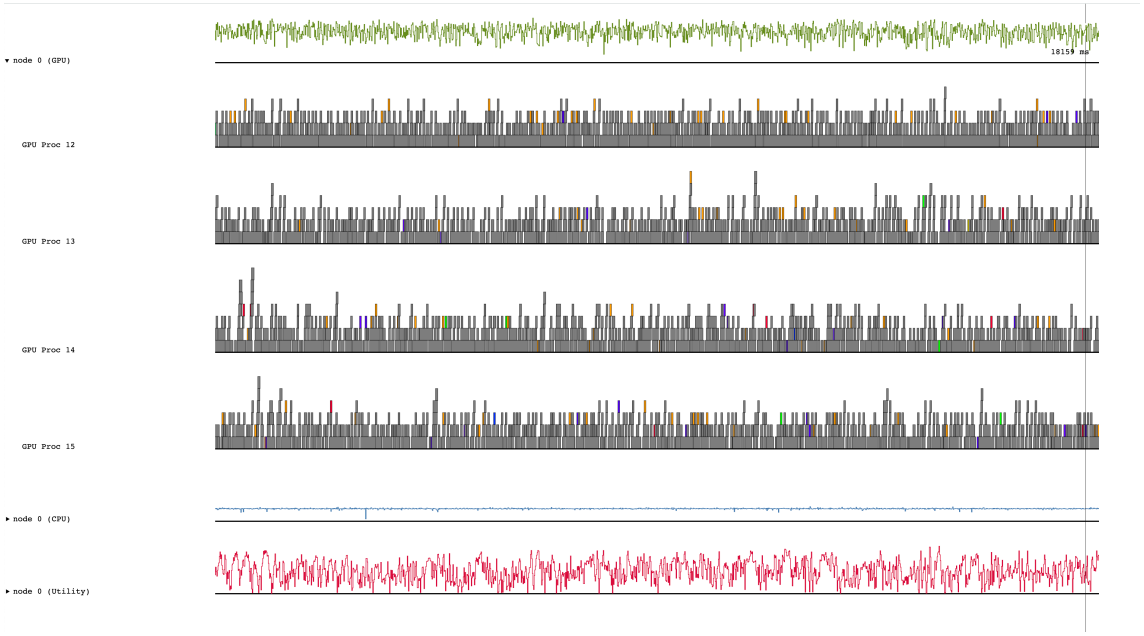


Figure 3.8 **Legion Prof** output for a ExpertFlow test run (zoomed in)

3.7 Conclusion

In this chapter, we have discussed the main components of ExpertFlow's design. In Section 3.1, we illustrated the path taken by each requests through our system. In Section 3.2, we discussed the asynchronous nature of the runtime we developed for ExpertFlow. In Section 3.3 we talked about how we utilize multiple GPUs (or multiple nodes) to parallelize the computations. Next, in Section 3.4 we delved into the details of our speculative inference approach. In addition to discussing the design, we offered more details on how we implemented ExpertFlow (Section 3.5), and we described the relation between ExpertFlow, FlexFlow, and Legion.

CHAPTER 4 AUTOREGRESSIVE TRANSFORMER INFERENCE

4.1 Autoregressive transformer inference

A key challenge preventing transformer inference systems from achieving low latency is the *autoregressive* nature of many generative tasks of interest, such as sentence completion, language modeling or machine translation. By ”autoregressive” we mean that in these tasks, each generated token depends on all those preceding it. This dependency is required, otherwise the model will simply output a sequence of unrelated words. As a consequence, we need to generate each token sequentially, and feed it back to the model to generate the following token in the following iteration. Some existing works have tried to push the performance by performing inference in a non-autoregressive fashion, by relaxing the assumption of conditional dependence of tokens on the preceding ones, but these techniques have so far been unable to match the quality of the output of their autoregressive counterparts, without increasing the computational costs.

Unlike these previous approaches, in ExpertFlow we employ three key components to support conditionally-dependent token generation while at the same time significantly reducing the overhead of serial generation. The three components are orthogonal and can be independently applied to improve the performance. This chapter focuses on these three optimizations, which are discussed in the sections below.

4.2 Dynamic Batching

In both training and inference, batching multiple input sequences together is essential to maximize the device utilization, thus optimizing the throughput. Batching, however, can also be a source of inefficiency. First of all, if the requests in a batch do not all have the same length, we have to pad them to the maximum sequence length, resulting in wasted computations. In the inference case, additional challenges are that we may not be able to fill an entire batch with requests, as requests are not all available in advance, and we don’t know when they will arrive. This requires more padding, and more wasted computations. Finally, and most importantly, because of the autoregressive nature of many transformers models, inference requires one iteration for each token to be generated. Since the requests

in a single batch may have different final lengths, and we need to run the model once for each token to be generated, we will need to run the model on the batch a number of times equal to the maximum number of tokens to be generated, across all requests in the batch. This will require a large amount of computations, since the inference stage for all requests with a lower number of tokens to be generated will have already completed. In addition, the requests that have already completed will have to wait for the straggler request to complete before the result can be returned to the client, resulting in a large latency overhead.

In ExpertFlow, we use a dynamic batching design to reap the benefits of batching (increased device utilization), while keeping the overhead caused by the disuniform sequence lengths and arrival times to a minimum. We do that through the following steps:

1. Merge the sequence and batch dimensions together, effectively treating the tokens from the batch's request as a single sequence. This allows us to store all the tokens in contiguous memory, avoiding the need for padding each request to the maximum sequence length. To keep all tensors of the same size, we will still need to pad the flattened sequence to $batch\_size \cdot max\_seq\_len$, but leaving all the padding in contiguous memory after the tokens. As the padding is no longer fragmented between the requests, we can avoid unnecessary computations by sending the token count to the model's operator, so they can simply skip over the last $token\_embedding\_dim \cdot ((batch\_size \cdot max\_seq\_len) - token\_count)$ entries in the tensor.
2. We update the batch at each generative step, instead of waiting for all the requests to be completed.
3. We attach some metadata to each batch, to facilitate steps 1 and 2. This is especially important for operators such as the attention, which needs to know what sequence each tokens belongs to, and at what position it is located. We use the `BatchConfig` struct in Listing 4.1.

Listing 4.1 BatchConfig

```
class BatchConfig {
public:
  BatchConfig();
  bool register_new_request(size_t guid,
                            int initial_length,
                            int tokens_to_generate);
  void prepare_next_batch();
  int update_results(InferenceResult const &ir);
  void update_num_active_requests_tokens();
  int num_active_requests() const;
```

```
  int num_active_tokens() const;
  void print() const;
  static int const MAX_NUM_REQUESTS = MAX_REQUESTS;
  static int const MAX_NUM_TOKENS = InferenceResult::MAX_NUM_TOKENS;
  // static int const MAX_SEQUENCE_LENGTH = MAX_SEQ_LEN;
  //  These are set by update
  int num_tokens, num_requests;
  bool cached_results;
  int token_start_idx[MAX_NUM_REQUESTS]; // index of first token in a request
                                         // that should be processed in the
                                         // current batch/iteration
  int token_last_available_idx
      [MAX_NUM_REQUESTS]; // last valid token index in a request. This
          includes
                          // both the prompt and generated tokens
  int num_processing_tokens[MAX_NUM_REQUESTS]; // a request's number of tokens
                                               // being processed in the
                                                   current
                                               // batch/iteration
  size_t max_sequence_length[MAX_NUM_REQUESTS];

  struct token_idxs {
    size_t request_index;  // the index within the BatchConfig of the request
                           // that the token belongs to
    size_t token_position; // the index indicating the position of each token
                           // within its request
  };

  struct SampleIdxs {
    size_t num_samples;
    size_t guids[InferenceResult::MAX_NUM_TOKENS]; // the guid of the request
                                                   // each token belongs to
    token_idxs token_indexes[InferenceResult::MAX_NUM_TOKENS];
  };

  SampleIdxs token2ids;
  size_t request_guid[MAX_NUM_REQUESTS];
  bool request_completed[MAX_NUM_REQUESTS];
};
```

4.3 Incremental decoding

Similarly to other existing transformer systems[12,35], we employ incremental decoding to eliminate the redundant computations that would otherwise be needed when generating tokens in a autoregressive fashion. To understand why autoregressive decoding will engender redundant computations (in the absence of incremental decoding), we have to look at how the multi-head attention operator works (see Algorithm 4.1).

Algorithm 4.1 Multi-Head Self-Attention algorithm

Input: x: input sequence, W_{qkv}: Q/K/V projection weights, W_o: output projection weights, num_heads: number of attention heads, seq_len: maximum number of tokens in each request, $batch\_size$: number of requests in each batch, emb_dim: embedding dimension
Require: shape of $x = [emb\_dim, seq\_len, batch\_size]$
Require: shape of $W_{qkv} = [emb\_dim, Q\_proj + K\_proj + V\_proj, num\_heads]$
Require: shape of $W_o = [V\_proj \times num\_heads, emb\_dim]$

$QKV\_projs \leftarrow W_{qkv}^T x$
$Q \leftarrow QKV\_projs\,[:, : Q\_proj, :]$ ▷*Shape :* $(num\_heads, Q\_proj, seq\_len, batch\_size)$
$K \leftarrow QKV\_projs\,[:, Q\_proj : Q\_proj + K\_proj, :]$ ▷*Shape :* $(\dots, K\_proj, \dots)$
$V \leftarrow QKV\_projs\,[:, Q\_proj + K\_proj :, :]$ ▷*Shape :* $(\dots, V\_proj, \dots)$
$QK^T \leftarrow einsum("ijkl, ijmn \rightarrow klmni", Q, K)$ ▷*Shape :* $(seq\_len, batch\_size, seq\_len, batch\_size, num\_heads)$
$QK^T \leftarrow CausalMasking(QK^T)$ ▷*Set entries above diagonal to* $-\infty$
$attn \leftarrow softmax(\frac{QK^T}{\sqrt{K\_proj}})$
$attn \leftarrow einsum("ijklm, mnkl \rightarrow ijnm", attn, V)$ ▷*Shape :* $(seq\_len, batch\_size, V\_proj, num\_heads)$
$attn \leftarrow attn.reshape(seq\_len, batch\_size, V\_proj \times num\_heads)$
$output \leftarrow (attn \times W_o).transpose(-1, 0, 1)$ ▷*Shape :* $(emb\_dim, seq\_len, batch\_size)$

In the decoding case, given an input sequence x, we will first generate the Q_h, K_h and V_h projections by matrix multiplying x by the weight matrices W_h^Q, W_h^K and W_h^V, for each head h:

$$Q_h = (W_h^Q)^\top x \tag{4.1}$$

$$K_h = (W_h^K)^\top x \tag{4.2}$$

$$V_h = (W_h^V)^\top x \tag{4.3}$$

$$\tag{4.4}$$

Then, for each head, we compute the attention scores with the formula:

$$Attention(Q_h, K_h, V_h) = softmax\left(\frac{Q_h \times K_h^\top}{\sqrt{d_k}}\right) \times V_h \tag{4.5}$$

where d_k is the dimension of the Q_h and K_h arrays. The output of the multi-head attention layer is then obtained by concatenating the results of Equation 4.5 for all heads, and performing one more matrix multiplication to project the result in the output space:

$$Output = Concat(\{Attention(Q_h, K_h, V_h) \;\; \forall h \in [0, num\_heads - 1]\}) \times W_{out} \tag{4.6}$$

The length of the output tensor is equal to the length of each Q_h projection, which is in turn equal to the length of x. This can help us understand where the redundant computations come from. In fact, at the end of each iteration, we only save the output in the last token's slot, and discard all outputs in the preceding slots, as these will be unchanged from the

previous iterations.

Incremental decoding works by computing Q_h using only the last entry of x, so the output will also have only one entry, which is all that we need. In a model with only one attention layer, changing Equation 4.1 to Equation 4.7 below would be enough.

$$Q_h = (W_h^Q)^\top x[:-1] \tag{4.7}$$

This would compute the attention scores with respect to all preceding tokens (as we would still be using the full x array to compute K_h and V_h), but only output the result for the last token. We would then pass only one output to the following layers, thus saving a lot of unnecessary computations. However, in the scenario where the model contains multiple attention layers (as is usually the case), our optimization doesn't work, because after the first attention, all the following attention layers only receive one entry (the one corresponding to the last token), which is enough to compute Q_h, but not enough to compute K_h and V_h. Incremental decoding fixes this problem by letting each attention operator cache the Key (K_h) and Value (V_h) projections. This solution works because at each iteration, and in each attention layer, only the last entry of (K_h) and (V_h) is changed. Note that this is only the case after having processed the prompt, which will still require passing all the tokens to the model, so that we can initialize the Key and Value entries in the cache for all tokens in the prompt. For more details on how incremental decoding works, see Algorithm 4.2.

Algorithm 4.2 Multi-Head Self-Attention with Incremental Decoding

Input: x: input sequence, W_{qkv}: Q/K/V projection weights, W_o: output projection weights, K_cache & V_cache: the K/V projections for the previous tokens in all in-progress requests, num_heads: number of attention heads, $batch\_config$: the `BatchConfig` object with the batch's metadata
Require: shape of x = [emb_dim, batch_config.num_tokens]
Require: shape of W_{qkv} = [emb_dim, Q_proj + K_proj + V_proj, num_heads]
Require: shape of W_o = [V_proj ×num_heads, emb_dim]
Require: shape of K_cache = [K_proj, max_seq_len, num_heads, max_requests]
Require: shape of V_cache = [V_proj, max_seq_len, num_heads, max_requests]

$QKV\_projs \leftarrow W_{qkv}^T x$
Q ← QKV_projs [: , : Q_proj, :] ▷*Shape : (num_heads, Q_proj, batch_config.num_tokens)*
K ← QKV_projs [: , Q_proj : Q_proj + K_proj, :] ▷*Shape : (… , K_proj, …)*
V ← QKV_projs [: , Q_proj + K_proj : , :] ▷*Shape : (… , V_proj, …)*
processed_tokens ← 0
attn ← []
for $r \leftarrow 1$ **to** $batch\_config.num\_requests$ **do**
    $l_r \leftarrow$ current length of request r
    $n_r \leftarrow$ number of new tokens from request r in this batch
    Store new K projection in K_cache[: , $l_r : l_r + n_r$, : , r]
    Store new V projection in V_cache[: , $l_r : l_r + n_r$, : , r]
    $QK_r^T \leftarrow$ einsum("$ijk, jli \rightarrow kli$", Q [: , : , processed_tokens : processed_tokens + n_r], K_cache[: , : $l_r + n_r$, : , r]) ▷*Shape : (n_r, $l_r + n_r$, num_heads)*
    $QK_r^T \leftarrow CausalMasking(QK_r^T)$ ▷*Set entries above diagonal to $-\infty$*
    $attn_r \leftarrow softmax\left(\frac{QK_r^T}{\sqrt{K\_proj}}\right)$
    $attn_r \leftarrow$ einsum("$ijk, ljk \rightarrow ilk$", $attn_r$, V_cache[: , : $l_r + n_r$, : , r]) ▷*Shape : (n_r, V_proj, num_heads)*
    attn ← concatenate (attn, $attn_r$)
    processed_tokens ← $processed\_tokens + n_r$
end for
$attn \leftarrow attn.reshape(batch\_config.num\_tokens, V\_proj \times num\_heads)$
$output \leftarrow (attn \times W_o).transpose(-1, 0, 1)$ ▷*Shape : ($emb\_dim, batch\_config.num\_tokens$)*

Having formalized how incremental decoding works, we can now quantify the amount of redundant computations saved by this technique. To do so, we develop a simple model (see below) to estimate the total number of floating point operations (FLOPs) required by one iteration of the multi-head attention operator in one layer, both before (Algorithm 4.1) and after (Algorithm 4.2) adopting incremental decoding. We can then compute the difference between these two quantities, and we will now the number of FLOPs saved at each attention layer. The total number of FLOPs saved by the entire model will be much higher, as every layer (not just the attention layers) will be able to only process a single token per iteration (after the first iteration, where we need to pass the full prompt to the model). Computing the total number of FLOPs saved in the general case is not possible, since it depends on the exact structure of the model. However, if we are given a specific model, we can estimate the number of FLOPs required by each layer as a function of the

sequence length, and then compute the number of FLOPs saved, in the same way as we do below for the attention layer.

Let's now estimate the FLOPs required by Algorithm 4.1 and Algorithm 4.2. Before we begin, however, we need to make an assumption regarding the FLOPs required by matrix multiplication and the softmax operator. For the matrix multiplication, we assume that we will be using the naive matrix multiplication algorithm, which will require $2xyz$ FLOPs to multiply together matrix $A \in \mathbb{R}^{x \times y}$ and $B \in \mathbb{R}^{y \times z}$. For the softmax operator, we assume that, given the same matrix A that we just mentioned, the number of FLOPs to compute the result will be $3xy$.

For Algorithm 4.1, we will need the following number of FLOPs:

$$\text{TOTAL FLOPs} = 2h(d_k + d_k + d_v)d_m \cdot (l \cdot b) + \tag{4.8}$$

$$2hd_k(l \cdot b)^2 + \tag{4.9}$$

$$h \cdot (l \cdot b)^2 + \tag{4.10}$$

$$3h(l \cdot b)^2 + \tag{4.11}$$

$$2h(l \cdot b)^2 \cdot d_v + \tag{4.12}$$

$$2h(l \cdot b)d_v d_m \tag{4.13}$$

where we have used the following notation: h is the number of heads, d_k is the dimension of each Query and Key projection (which must be equal), d_v is the dimension of each Value projection, d_m is the hidden dimension of the model, l is the sequence length, and b is the batch size. When the key and value projections have the same dimensions (i.e. $d_k = d_v$), we can simplify the expression above as follows:

$$\text{TOTAL FLOPs} = 4h(l \cdot b)(2d_k d_m + (d_k + 1)(l \cdot b)) \tag{4.14}$$

Hence, for a single request of length l, the number of FLOPs will be:

$$\text{FLOPS (1 request)} = 4hl(l + d_k(2d_m + l)) \tag{4.15}$$

For Algorithm 4.2, we will need the following number of FLOPs:

$$\text{TOTAL FLOPs} = 2h(d_k + d_k + d_v)d_m \cdot \sum_r n_r + \tag{4.16}$$

$$2hd_k\left(\sum_r n_r \cdot \sum_r l_r\right) + \tag{4.17}$$

$$h \cdot \left(\sum_r n_r \cdot \sum_r l_r\right) + \tag{4.18}$$

$$3h\left(\sum_r n_r \cdot \sum_r l_r\right)+ \tag{4.19}$$

$$2h\left(\sum_r n_r \cdot \sum_r l_r\right)\cdot d_v+ \tag{4.20}$$

$$2h\left(\sum_r n_r\right)d_v d_m \tag{4.21}$$

where we have introduced the following additional notation: r is the request index, n_r is the number of new tokens from request r in the current iteration (n_r will be equal to the prompt length at iteration 0, and then $n_r = 1$ for all following iterations), and l_r is the current length of request r: this quantity will always equal n_r plus all the tokens previously generated for request r. Just like we did above, we can then simplify the formula above as follows:

$$\text{TOTAL FLOPs} = 2h\left(\sum_r n_r\right)\left(2d_m d_v + \left(2 + d_v\right)\left(\sum_r l_r\right) + d_k\left(2d_m + \left(\sum_r l_r\right)\right)\right) \tag{4.22}$$

which simplifies further when $d_v = d_k$:

$$\text{TOTAL FLOPs} = 4h\left(\sum_r n_r\right)\left(\left(\sum_r l_r\right) + d_k\left(2d_m + \left(\sum_r l_r\right)\right)\right) \tag{4.23}$$

For a single request, at the iteration where the total length is l and the number of new tokens is n, the formula above simplifies to:

$$\text{FLOPs (1 request)} = 4hn\left(l + d_k\left(2d_m + l\right)\right) \tag{4.24}$$

We can also rewrite Equation 4.24 in terms of only variable l, for the case where the number of new tokens is always 1 in the incremental phase:

$$\text{FLOPs (1 request)} = \begin{cases} 4h\left(l + d_k\left(2d_m + l\right)\right) & \text{incremental phase} \\ 4hl\left(l + d_k\left(2d_m + l\right)\right) & \text{prompt} \end{cases} \tag{4.25}$$

Given a single request with a prompt length of l_i and final length of l_f, we can now compute the number of FLOPs saved by incremental decoding with the formula below, where we let $\phi(l)$ be the number of FLOPs for a request of length l at the current iteration as computed by Equation 4.25, and $\psi(l)$ be the number of FLOPs for a request of length l at the current iteration as computed by Equation 4.15. In other words, $\phi(l)$ is the relevant equation when incremental decoding is active, and $\psi(l)$ is the equation to be used when we are not using incremental decoding. The number of FLOPs saved can then be computed

as follows:

$$\text{FLOPs saved} = \sum_{l=l_i}^{l_f} \phi(l) - \sum_{l=l_i}^{l_f} \psi(l) \tag{4.26}$$

which expands to:

$$\begin{aligned}\text{FLOPs saved} = 4hl_i\left(l_i + d_k\left(2d_m + l_i\right)\right) + \sum_{l=l_i+1}^{l_f}\left(4h\left(l + d_k\left(2d_m + l\right)\right)\right) - \\ + \sum_{l=l_i}^{l_f}\left(4hl(l + d_k(2d_m + l))\right)\end{aligned} \tag{4.27}$$

We can then solve with respect to l_i and l_f, and obtain:

$$\boxed{\begin{aligned}\text{FLOPs saved} = \frac{4}{3}h(l_f - l_i)(d_k\left(3d_m(l_f + l_i + 3) + l_f^2 + l_f(l_i + 3) + l_i^2 + 3l_i + 2\right) + \\ + l_f^2 + l_f(l_i + 3) + l_i^2 + 3l_i + 2)\end{aligned}} \tag{4.28}$$

We can see that Equation 4.28 always evaluates to a positive, non-zero number. To have a more concrete idea of the number of FLOPs saved, we can plug in some example values for l_i and l_f. For instance, assume that the prompt has a length of 32 tokens ($l_i = 32$), and we want to generate 96 additional tokens ($l_f = l_i + 96 = 128$). Plugging these values into Eq. 4.28 yields:

$$\text{FLOPs saved} = 128(21986 + d_k(21986 + 489d_m))h \tag{4.29}$$

Finally, it is worth mentioning that for batches with more than 1 request, Algorithm 4.2 will allow us to save a number of FLOPs that is **much larger** than the sum of the values obtained with formula 4.28 for each request. In fact, in the absence of incremental decoding, Algorithm 4.1 needs to pad all requests to the same maximum length, whereas Algorithm 4.2 does not require any padding, as it can integrate the dynamic batching optimizations discussed in Section 4.2.

4.4 Speculative decoding

While incremental decoding already allows us to greatly reduce the multi-head attention's overhead, the generation of the tokens is still sequential, resulting in high latency compared to running the same type of model in a non-autoregressive mode. To further improve the performance, we use speculative inference.

Speculative inference allow us to decrease the latency, as well as the GPU memory

accesses, which can often become a bottleneck. When it comes to the end-to-end latency, we have already discusses in Section 3.4 how the use of speculative inference can reduce the overhead by up to the average matching rate, or the average number of tokens tha the speculators can guess correctly.

In addition to the latency improvements, the reduction in the number of times we use the larger LLM model will reduce the GPU memory accesses needed to load the weights from CPU DRAM. This will result in further speedups, since memory accesses are often the bottleneck when it comes to GPU programs. This is especially the case when we offload some of the data and computations to CPU or flash storage.

4.5 Conclusion

In this chapter, we focused on the optimizations we employed to tackle the challenges that come with serving generative models in autoregressive mode. In Section 4.1 we first characterized the problem. In the following three sections, we discussed the three main optimizations in ExpertFlow: dynamic batching (Section 4.2), incremental decoding (4.3), and speculative decoding (4.4).

CHAPTER 5 KERNEL-LEVEL OPTIMIZATIONS

In this chapter we introduce the design and implementation of the two specialized GPU kernels in use by ExpertFlow. The first customized kernel is used to implement the fused-experts operator, which computes the predictions of a group of MoE experts assigned to the same device using a single kernel. The second customized kernel is used by the multi-head attention layers. While cuDNN already offers a heavily-optimized kernel for multi-head attention, it does not support caching the key and value projections, which is required for incremental decoding. In addition, it does not support the optimizations we require for speculative inference. In the following sections, we discuss the considerations that led us to the final version of each kernel, as well as the optimizations that we used.

5.1 Fused-Experts Operator

An essential factor to be able to run a Mixture-of-Experts model without incurring in great performance overhead is to use efficient GPU kernels. In particular, a MoE layer generally requires a fast implementation of a *gating network*, *top-k*, *token dispatch*, *expert prediction*, and *predictions aggregation* kernel. In this section, we illustrate the role of each kernel, and how we implemented each of them in our *fused-experts operator* in order to significantly optimize the performance when compared to the naive version.

5.1.1 MoE-layer kernels

Algorithm 5.1 Naive MoE-layer algorithm

Input: x: input sequence, W_i: weights of expert $i \in [0, E)$, $GatingNetwork$: the weights of the FFN that implements the gating network, k: how many experts to route each token to, E: total number of experts, C: expert capacity, seq_len: maximum number of tokens in each request, $batch\_size$: number of requests in each batch, emb_dim: embedding dimension
Require: shape of $x = [emb\_dim, seq\_len \cdot batch\_size]$
Require: shape of $W_i = [emb\_dim, emb\_dim]$
Require: $k \leq E$

1: num_tokens ← seq_len · batch_size
2: g ← Softmax(GatingNetwork(x)) ▷*gating network kernel*
3: top_values, top_indices ← TopK(g) ▷*top-k kernel*
4: z ← OneHotEncoding(top_indices, k, E, num_tokens, C) ▷*token dispatch kernel*
5: predictions ← x × z × $\left[W_0, \ \ldots, \ W_{E-1}\right]^\top$ ▷*expert prediction kernel*
6: output ← dot(top_values, predictions) ▷*predictions aggregation kernel*
7:
8: **procedure** OneHotEncoding(top_indices, k, E, num_tokens, C)
9: Allocate a size of $k \cdot E \cdot num\_tokens$ for tensor z
10: Allocate a size of E for boolean array $num\_assignments$
11: $z \leftarrow \{0\}$
12: $num\_assignments \leftarrow \{0\}$
13: **for** i ←0 **to** num_tokens **do**
14: **for** j ←0 **to** k **do**
15: $e \leftarrow top\_indices[j, i]$
16: **if** $num\_assignments[e] \leq C$ **then**
17: $num\_assignments \leftarrow num\_assignments + 1$
18: $z[j, e, i] = 1$
19: **end if**
20: **end for**
21: **end for**
22: return z
23: **end procedure**

Above, we present a naive algorithm (Algorithm 5.1) to implement a MoE layer. In this version, we focus on the vanilla MoE model where each expert consists of a single fully-connected layer, and the gating network is also made up of a single fully-connected layer, followed by a softmax and a top-k. More realistic MoE layers will use more advanced components to achieve better load balancing, and improve the model's generalization power; for instance, several models[1,3] add a stochastic components in the gating network. These additional components, however, are for the most part orthogonal to the issues we are discussing here, so we omit them for simplicity.

Each of lines 2-6 corresponds to one of the five typical MoE kernels introduced above. The algorithm should help demystifying what each such kernel does, and the dependencies between the kernels. We can implement Algorithm 5.1 in CUDA using a cuBLAS GEMM

operator for the matrix multiplication steps, cuDNN for the softmax, and a custom CUDA kernel for the remaining steps. We can further implement a single fused kernel using CUDA or CUTLASS. However, the sparsity introduced by the one-hot encoding, together with the sequential nature of the two for loops (which are difficult to parallelize due to the need to consider the expert capacity when assigning tokens to experts) within the OneHotEncoding procedure lead the naive algorithm to suffer from poor performance in practice.

5.1.2 ExpertFlow fast kernel implementation

In ExpertFlow, we speed up the implement the *token dispatch*, *expert prediction*, and *predictions aggregation* through the kernels shown below. In particular, Algorithm 5.2 allows us to parallelize the token dispatch operation, without exceeding the expert capacity. The kernel is carefully constructed so that each instruction is fully parallelizable. We take advantage of the Blelloch scan algorithm[43] to parallelize the *reduction* and *exclusive scan*, which would otherwise introduce a parallelization bottleneck. Each instruction in Algorithm 5.2 is implemented using the highly optimized NVIDIA Thrust library [44], which integrates seamlessly with the other kernels, which are written in CUDA. Thrust allows us to specify the execution policy for each instruction; by specifying the CUDA stream to use for each instruction, we can prevent any synchronization issue, while avoiding expensive operations such as `cudaDeviceSynchronize()`.

Algorithm 5.3 illustrates our implementation of the ComputeBatchedMatmulIndices kernel, which takes the results of the MoESortTokens kernel as input, and populates three arrays of pointers that will then be used by the BatchedMatmul step (Algorithm 5.4, line 9) to compute the expert predictions, and one more array to be used in the final aggregation phase. Compared to the original naive algorithm, our design allows us to save GPU memory by not having to store the large one-hot encoding tensor. In addition, we remove most sparsity from the computations, resulting in better opportunities for speedups, since it is easier to optimize dense kernels. Finally, we perform most of our operations by manipulating the expert assignment indices, instead of the actual token data. This is beneficial because each index only requires one integer to represent, as opposed to the many floats required to represent each token, so we can save a lot of space and time.

Algorithm 5.2 MoESortTokens kernel

Input: $top\_indices$: the indices of the k experts chosen by each token, $expert\_start\_idx$: the index of the first fused expert residing on the current device, $num\_experts\_per\_block$: number of fused experts residing on each device, C: expert capacity, seq_len: maximum number of tokens in each request, $batch\_size$: number of requests in each batch

Require: shape of $top\_indices = [k, seq\_len \cdot batch\_size]$

procedure MoESortTokens(top_indices, expert_start_idx, num_experts_per_block, C)
    num_indices ← length(num_indices)
    original_indices ← sequence [0, … , $num\_indices - 1$]
    StableSortByKey (original_indices, **key** = top_indices)
    StableSort (top_indices)
    lb_index ← LowerBound (top_indices, **value** = expert_start_idx)
    ub_index ← UpperBound (top_indices, **value** = expert_start_idx + num_experts_per_block)
    num_valid_assignments ← ub_index - lb_index
    if num_valid_assignments == 0 **then**
        Done
    end if
    nonzero_expert_labels, expert_start_indices ← Unique (top_indices[lb_index : ub_index])
    nonzero_expert_count ← length(nonzero_expert_labels)
    nonzero_expert_labels ← nonzero_expert_labels - expert_start_idx
    temp_sequence ← sequence [0, … , $nonzero\_expert\_count - 1$]
    exp_local_label_to_index ← Scatter (temp_sequence, **map** = nonzero_expert_labels)
    expert_start_indices ← Append (expert_start_indices, **value** = num_valid_assignments)
    expert_start_indices [1 :] ← expert_start_indices [1 :] - expert_start_indices [: -1]
    destination_start_indices ← expert_start_indices [: -1]
    destination_start_indices[destination_start_indices > C] ← C
    gemm_batch_count ← Reduce (destination_start_indices)
    ExclusiveScan (destination_start_indices)
end procedure

Algorithm 5.3 ComputeBatchedMatmulIndices kernel

Input: *sorted_indices*: the sorted indices of the k experts chosen by each token
Input: *original_indices* an array containing the reverse indices corresponding to the *sorted_indices*
Input: *exp_local_label_to_index*: a lookup table converting an expert label to its index
Input: *expert_start_indexes*: the indices of the first tokens (in order) assigned to each unique expert in the block
Input: *destination_start_indices*: the indices, for each expert, of the first slot in the index arrays belonging to each unique index
Input: *input*: the tensor containing the input tokens
Input: *weights*: tensor containing the weights for all the fused experts in the block
Input: *coefficients*: tensor containing the *top_values* from the TopK operator
Input: *output*: the output tensor
Input: *num_valid_assignments*: the number of tokens who are assigned to experts in the current block
Input: *lb_index*: index of first slot in the *sorted_indices* tensor to belong to an expert in the block
Input: *expert_start_idx*: the index of the first fused expert residing on the current device
Input: *data_dim*: embedding dimension of each token
Input: *out_dim*: hidden dimension of each output entry of the MoE layer
Input: C: expert capacity
Input: k: number of experts to which each token is assigned to

procedure ComputeBatchedMatmulIndices(sorted_indices, original_indices, exp_local_label_to_index, expert_start_indexes, destination_start_indices, input, weights, coefficients, output, num_valid_assignments, lb_index, experts_start_idx, data_dim, out_dim, C, k)
    Allocate a size of *num_valid_assignments* for each of pointer arrays *token_idx_array*, *weight_idx_array*, *coefficient_idx_array*, and *output_idx_array*
    token_idx_array ← {0}
    weight_idx_array ← {0}
    coefficient_idx_array ← {0}
    output_idx_array ← {0}
    parallelfor i ←0 **to** num_valid_assignments **do**
        global_expert_label ← sorted_indices[lb_index + i]
        local_expert_label ← global_expert_label - experts_start_idx
        expert_index ← exp_local_label_to_index[local_expert_label]
        within_expert_offset ← i - expert_start_indexes[expert_index]
        weight_params_count ← data_dim * out_dim
        if within_expert_offset < C **then**
            rev_idx ← original_indices[i + lb_index]
            token_idx ← (rev_idx / k)
            token_idx_array[destination_start_indices[expert_index] + within_expert_offset] ← &input[token_idx * data_dim]
            weight_idx_array[destination_start_indices[expert_index] + within_expert_offset] ← &weights[local_expert_label * weight_params_count]
            coefficient_idx_array[destination_start_indices[expert_index] + within_expert_offset] ← &coefficients[rev_idx]
            output_idx_array[destination_start_indices[expert_index] + within_expert_offset] ← &output[token_idx * out_dim]
        end if
    end parallelfor
    return *token_idx_array*, *weight_idx_array*, *coefficient_idx_array*, *output_idx_array*
end procedure

Algorithm 5.4 Fast algorithm for the MoE-layer

Input: x: input sequence, W_i: weights of expert $i \in [0, E)$, $GatingNetwork$: the weights of the FFN that implements the gating network, k: how many experts to route each token to, E: total number of experts, C: expert capacity, seq_len: maximum number of tokens in each request, $batch\_size$: number of requests in each batch, emb_dim: embedding dimension
Require: shape of $x = [emb\_dim, seq\_len \cdot batch\_size]$
Require: shape of $W_i = [emb\_dim, emb\_dim]$
Require: $k \leq E$

num_tokens ← seq_len · batch_size
g ← Softmax(GatingNetwork(x)) ▷*gating network kernel*
top_values, top_indices ← TopK(g) ▷*top-k kernel*
num_experts_per_block ← E / num_devices
sorted_indices, original_indices, exp_local_label_to_index, expert_start_indexes, destination_start_indices, num_valid_assignments, lb_index ← MoESortTokens(top_indices, expert_start_idx, num_experts_per_block, C)
experts_start_idx ← $device\_rank \cdot num\_experts\_per\_block$
weights ← $\left[W_0, \ldots, W_{E-1}\right]^\top$
token_idx_array, weight_idx_array, coefficient_idx_array, output_idx_array ← ComputeBatchedMatmulIndices(sorted_indices, original_indices, exp_local_label_to_index, expert_start_indexes, destination_start_indices, x, weights, coefficients, output, num_valid_assignments, lb_index, experts_start_idx, emb_dim, emb_dim, C, k)
predictions ← BatchedMatmul (**a=**token_idx_array, **b=**weight_idx_array, **coeff=**coefficient_idx_array)
AtomicElementwiseAdd (output_idx_array, predictions)

5.2 Incremental Multi-Head Attention Kernel

Our implementation of the kernel for the Multi-Head attention operator closely follows Algorithm 4.2, and is written in CUDA. We support incremental decoding by default, so we refer to the kernel as the *Incremental Multi-Head Attention Kernel*. We also make use of the cuBLAS and cuDNN libraries for operations such as matrix multiplications and activations, for which highly-optimized kernels available off-the-shelf are often faster than custom-designed ones. After designing algorithm 4.2, an important aspect we considered during the implementation phase was how to effectively store all the data in memory. We designed the data layout in a way that minimizes the number of data copies and data movements needed, and we structured each tensor in column-major order, as is customary in CUDA. The layouts of the tensors used in the incremental multi-head attention layer are shown below.

The input tensor (Figure 5.1) is shaped to have the data dimension (`data_dim`) as the leading dimension. `data_dim` is the size required to store a single token. The total capacity of the tensor is equal to the product of the batch size and the maximum sequence length. We set the batch size and sequence length for compatibility with other existing

operators in FlexFlow that are implemented using three-dimensional tensors. However, thanks to dynamic batching and incremental decoding, we are not constrained to have a maximum number of requests equal to `batch_size`, nor we need to pad each request to `max_sequence_length`. As a consequence, we can simply juxtapose the tokens from each request with no padding between them, and treat the tensor as a two-dimensional tensor where the second dimension is `num_active_tokens`, the number of tokens from each request at the current iteration. When `num_active_tokens` is less than `batch_size x max_sequence_length`, we leave the remaining empty space at the end. This padding can be safely ignored, and it is not included in any operation performed by the kernel, thus avoiding any unnecessary computation.

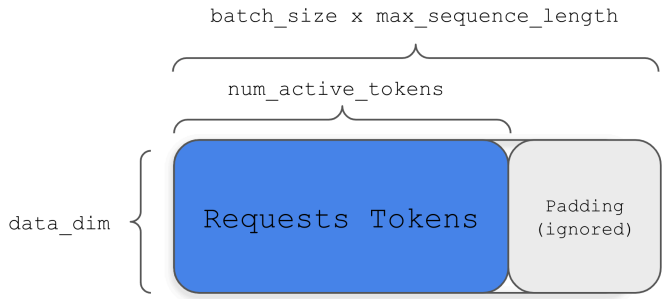


Figure 5.1 **The data layout of the input tensor in the Multi-Head Attention layer**

The second input to the attention kernel is the tensor containing the weights (Figure 5.2). The tensor contains a block of weights for each head, and each block, in turns, contains the the Query, Key, Value and Output projection weights. The first three are used to produce the Query, Key, and Values from the input tokens, whereas the last one is used to generate the final output given all the attention heads.

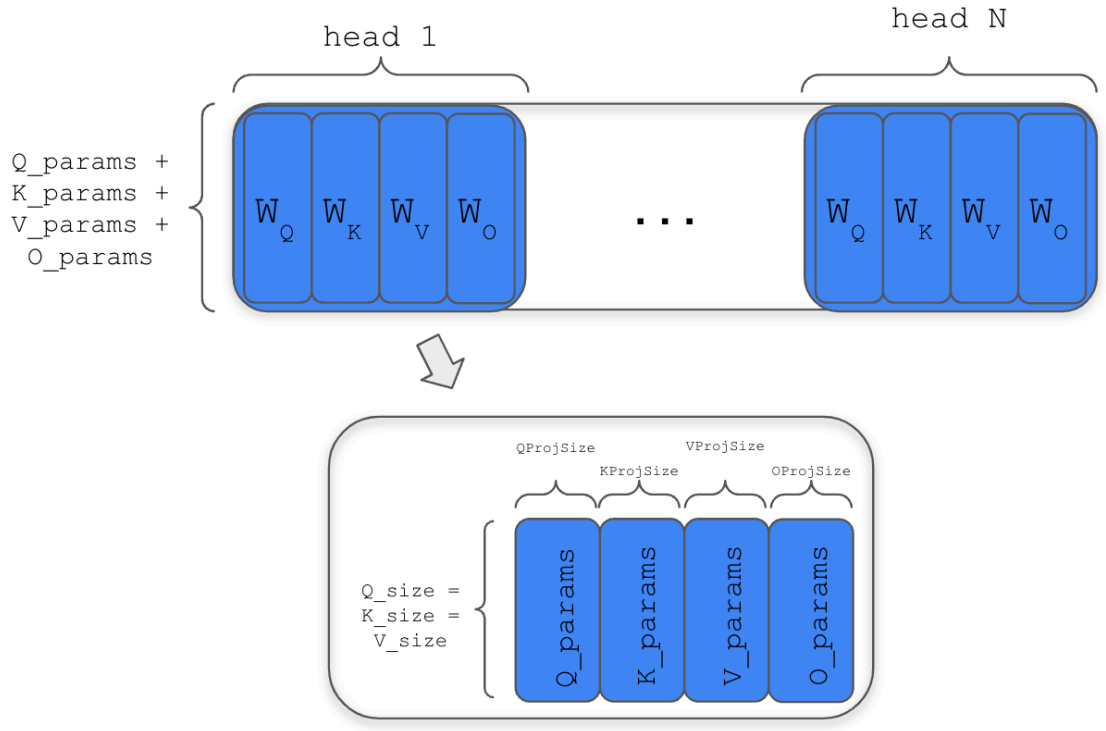


Figure 5.2 **The data layout of the weights tensor in the Multi-Head Attention layer**

As the first step of the attention kernel, we generate all the Query, Key, and Values from all the input tokens with a single `cublasGemmStridedBatchedEx`, where the stride is the size of each block of weights corresponding to a single head. We store the result in the Q/K/V projections tensor (Figure 5.3).

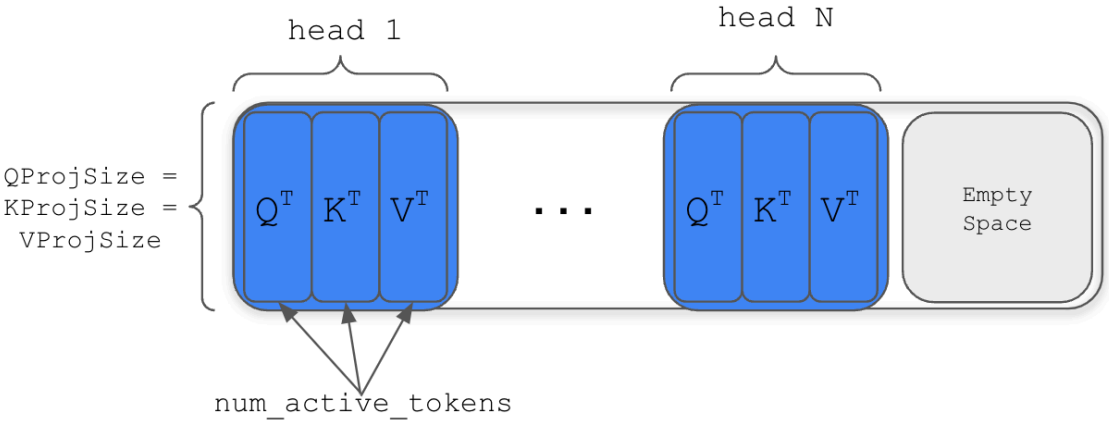


Figure 5.3 **The data layout of the Q/K/V projections tensor in the Multi-Head Attention layer**

Having computed the Q/K/V projections for the current input, we copy the Keys and Values into the cache, so that they can remain accessible across the next iterations. The

K-cache (Figure 5.4) and V-cache (Figure 5.5) share the same data layout: each tensor is divided into a block for each request, and each such block is further divided into a block for each head. Each head block contains enough room for the Keys or Values for the maximum possible number of tokens in a request.

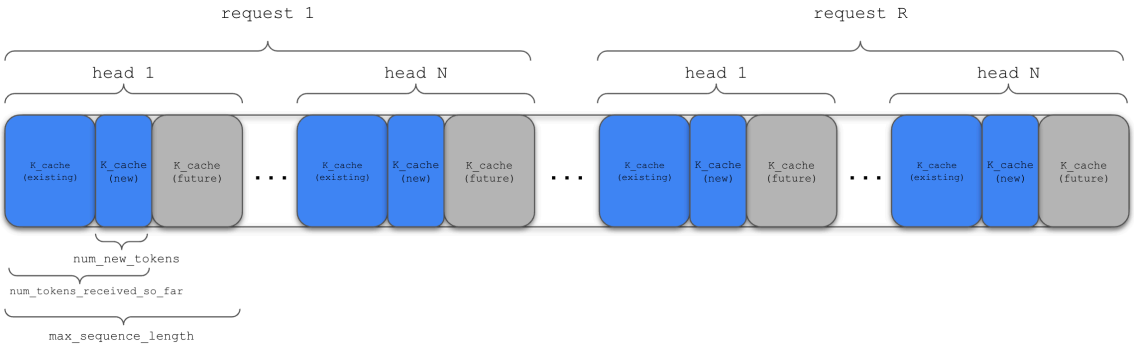


Figure 5.4 **The data layout of the K-cache tensor in the Multi-Head Attention layer**

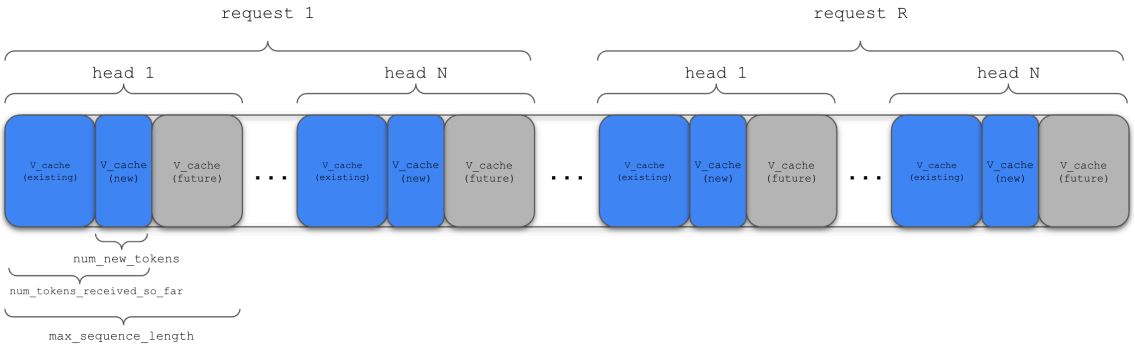


Figure 5.5 **The data layout of the V-cache tensor in the Multi-Head Attention layer**

After having stored the new Keys and Values in the cache, we proceed to compute the QK products (the product of the Query and Keys for each head), and store them in the QK-product tensor as shown in Figure 5.6. In incremental decoding mode, for each request, the Query only contains the projection obtained from the last generated token, whereas the Keys have one entry for each token in the sequence, starting from the beginning. For this reason, each head block in the QK-product tensor has a shape of `new_tokens` × `num_tokens_received_so_far`, where `new_tokens`= 1. The next step after generating the QK-products is to compute, for each head block, the softmax with respect to the `num_tokens_received_so_far`. Intuitively, the softmax will decide, for each token to be generated, how to weight every other token in the sequence as the token's context. Our operator implements a causal attention, or one when only the preceding tokens are used as a context for generating the next. To implement this, we set

all the entries above the diagonal in each head block in the QK-product tensor to $-\infty$. When taking the softmax, these entries will be transformed to 0, as desired.

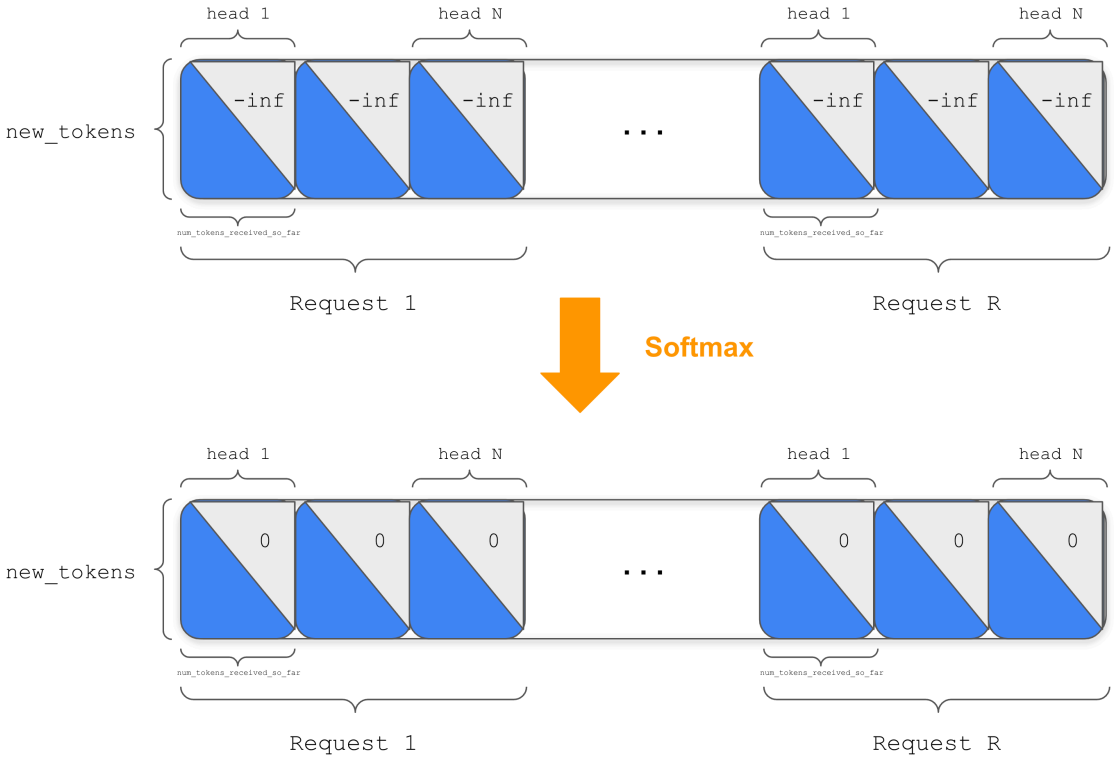


Figure 5.6 **The data layout of the QK-product tensor in the Multi-Head Attention layer**

Next, for each request, we compute the result (Figure 5.7) for each attention head by multiplying each head block from the QK-product tensor (after taking the softmax) by the corresponding head block in the V-cache tensor.

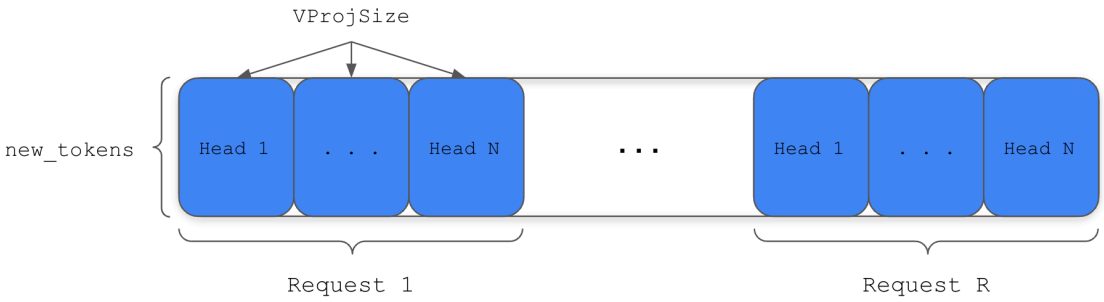


Figure 5.7 **The data layout of the tensor containing the results for each attention head in the Multi-Head Attention layer**

We can now compute the final output, combining the results from all heads, by matrix-multiplying the attention heads tensor (Figure 5.7) by the contiguous output projection weight tensor (Figure 5.8). The latter tensor is so called because it is obtained by combin-

ing all the output projection weight slices from the weight tensor (Figure 5.2), so that the slices are contiguous in memory.

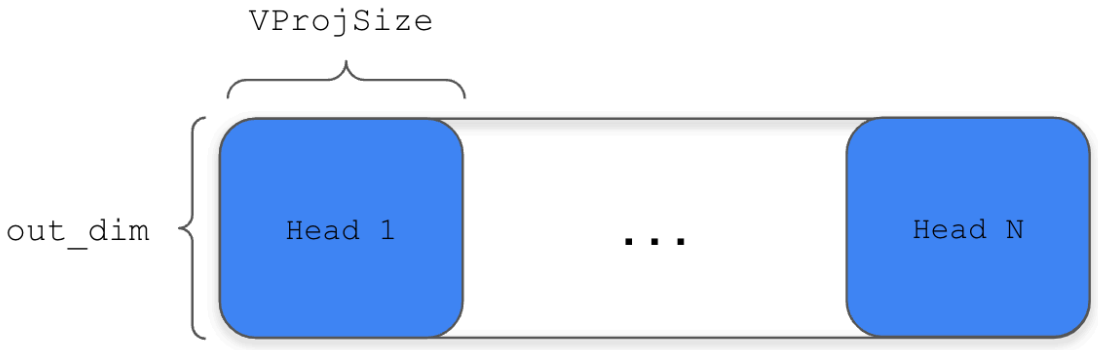


Figure 5.8 **The data layout of the output weights tensor in the Multi-Head Attention layer**

5.2.1 Optimizations for Speculative Decoding

Given the *Incremental Multi-Head Attention Kernel* that we described in the section above, we now discuss the changes and optimizations we employed to efficiently support speculative inference. Speculative inference has two stages: first, each speculator generates a sequence of tokens in autoregressive mode; next, we merge the sequences in a single tree and use the large language model to verify which tokens are correct. Each speculator simply runs the multi-head attention in incremental mode, so we can use the kernel from Section 5.2 with no additional change. For the verification stage, we describe below how we designed a variant of the *Incremental Multi-Head Attention Kernel* to efficiently support the token tree verification.

In the process of verifying multiple tokens in parallel for each request, we face a challenge: the sequences produced by the speculators may require storing conflicting Key and Value projections in the cache. The simplest solution to overcome this issue is to have separate Key/Value cache tensors for each speculator, as shown in the *Sequence-based Decoding* box in Figure 5.9. This approach allows us to verify each sequence in parallel, but it has a large memory overhead, and computations are often replicated, since the sequences produced by the speculators can often share a prefix.

An alternative approach is shown in the *Token-based Decoding* box in Figure 5.9. With this approach, we can have a single Key/Value cache tensor, greatly reducing the memory overhead. To avoid conflicts between different sequences, we process one token at a time, placing the result in the slot corresponding to the potential token's depth in the sequence, starting from the beginning of the prompt. By flattening the token tree in DFS

order, we can guarantee that, at the time we are processing each branch, the slots to the left of the first token in the branch will not have been overwritten with invalid values for the current branch. The main disadvantage of this approach is that it will be slower, as we won't be able to process multiple tokens in parallel.

Our optimized solution is shown in the *Tree-based Decoding* box in Figure 5.9. Our approach consists of batching together the tokens in the tree such that each token is the subsequent token's parent. This approach will allow us to maximize parallelism while avoiding conflicts.

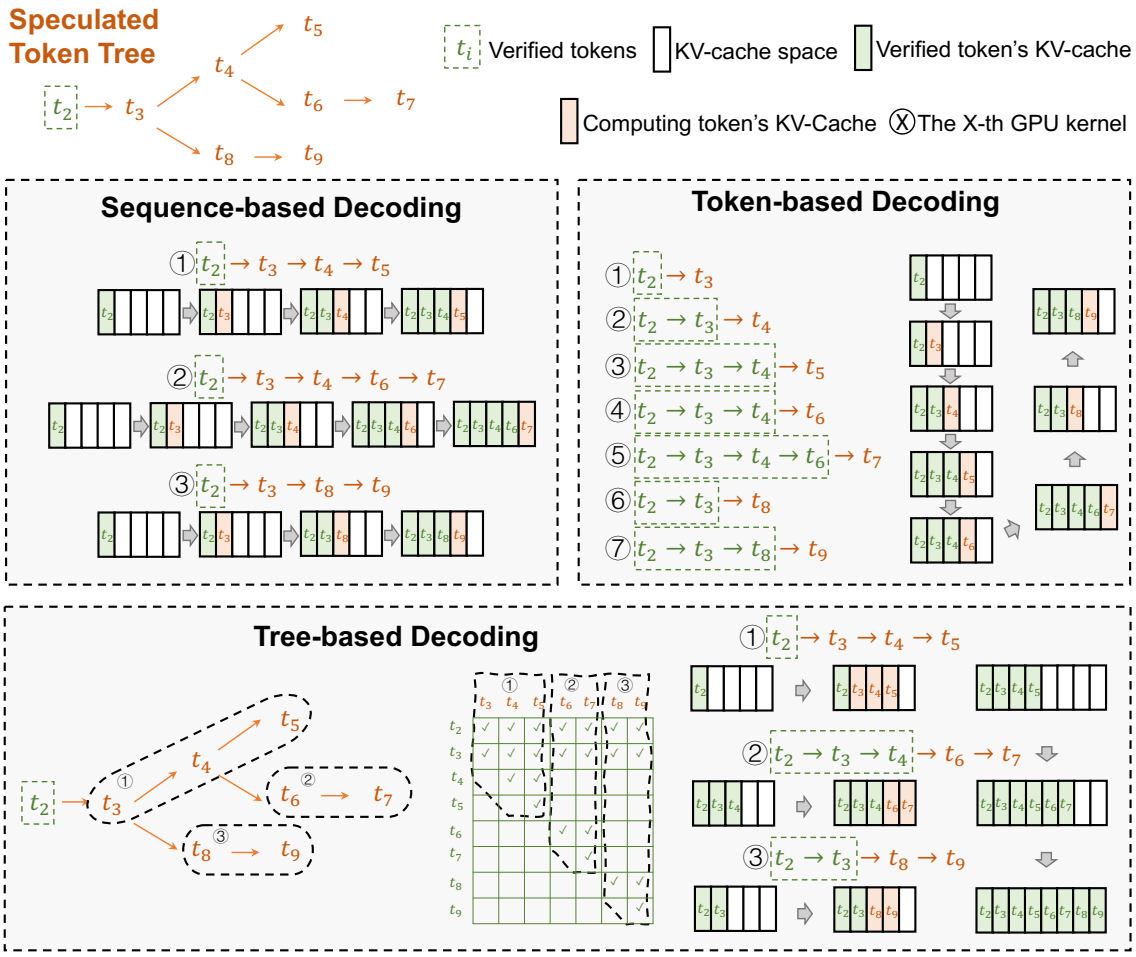


Figure 5.9 **Naive and optimized solutions to support speculative decoding in ExpertFlow's Multi-Head Attention Kernel**

5.3 Conclusion

In this chapter, we have discussed the ExpertFlow optimizations at the kernel level. In particular, our system uses two custom CUDA kernels: one to implement the fused-experts operator (Section 5.1) and one to implement the multi-head attention (Section

5.2), with a special variant to support token tree verification in the speculative inference scenario.

CHAPTER 6 EVALUATION

In this section, we describe our efforts to evaluate the correctness and the performance of ExpertFlow. Evaluating the correctness amounts to ensuring that our system always produces the expected results, so that users can be confident that using ExpertFlow instead of another existing system will not result in obtaining wrong or low quality output tokens. Evaluating the performance allows us to check that ExpertFlow is faster or at least comparable to existing solutions.

6.1 Evaluating correctness

We evaluate the correctness of ExpertFlow through a combination of unit tests and integration-tests.

6.1.1 Unit tests

ExpertFlow inherits from FlexFlow[39,41], the three levels of abstractions used to represent a DNN and execute its computations: *layers*, *operators*, and *asynchronous tasks*, which in turn execute one (or more) *kernel* on GPU. At each level of abstraction, we add multiple unit tests to verify the correctness under many different angles. For instance, at the layer level, we can most easily check the correctness of the structure of the model. At the operator level, we concentrate most tests that check the generated computational graph, the dimensions of the input and output tensors, and the parallelization strategies. At the task level, we can most easily check that the input and weights tensors are accessible, on the right device, and contain the correct data. Further, we can check the correctness of the GPU kernels.

We implement our unit tests, organized as described above, using different programming languages and frameworks. At the layer, and operator level, we added tests in the form of assertions, based on quantities computed in C++, often based on the Legion library. We also used the DOT language, together with GraphViz, to export and visualize graph-structured quantities, such as the computational graphs, or the data and temporal dependencies between different tensors, operators, and tasks. Finally, we verified the correctness of all the custom kernels by re-implementing them using C++ and LibTorch (the PyTorch C++ API), running the CUDA and C++ versions in parallel, and verifying that

the output tensors match. For the kernels that have a corresponding operator in PyTorch (e.g. the Linear layer, Softmax layer, LayerNorm, etc...), we also created a suite of *alignment tests*, where we compare the output tensors of our kernels against the results obtained with PyTorch for a range of different input tensors. We added the alignment tests to CI, to ensure that we do not accidentally introduce new bugs with new commits.

6.1.2 End-to-End tests

After having tested each component individually, we used end-to-end tests as an additional check on the correctness of our implementation. We achieved this by choosing three representative models for which the pre-trained checkpoints is available, loading the weights into ExpertFlow, executing the same batch of inference requests using the original code and the ExpertFlow code representing the same model, and ensuring that the outputs match. In addition to loading the weights, we also needed to ensure that all the random seeds matched, whenever there was an operator with a stochastic behavior. The models we chose for this test were:

1. LLAMA model[45] by Meta, a transformer model
2. Fairseq's 15B-parameters LM MoE model[46], also by Meta, a MoE model
3. GPT-MoE 中文 67 亿诗歌生成模型[47] by Modelscope, a MoE model

6.2 Evaluating performance

After evaluating the correctness of our framework, we ran several experiments to measure its performance. In the sections below, we discuss our experiment setup, and we present our results.

6.2.1 Setup

We run the experiments on a `g4dn.12xlarge` AWS machine with 48 vCPUs, 192 GiB of RAM memory, and 4 NVIDIA T4 GPUs (see Figure 6.1), each with 16 GB of memory (although the amount of usable memory is somewhat less). We initialize the instance with the Deep Learning AMI GPU PyTorch 1.13.0 (Ubuntu 20.04) 20221109. We run the FasterTransformer experiments within the `nvcr.io/nvidia/pytorch:22.07-py3` container, which comes with PyTorch built from source to add support for MPI, which is required by FasterTransformer to support the multi-GPU GPT program.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 515.65.01    Driver Version: 515.65.01    CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            On   | 00000000:00:1B.0 Off |                    0 |
| N/A   27C    P8    14W /  70W |      2MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla T4            On   | 00000000:00:1C.0 Off |                    0 |
| N/A   27C    P8    15W /  70W |      2MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  Tesla T4            On   | 00000000:00:1D.0 Off |                    0 |
| N/A   27C    P8    14W /  70W |      2MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  Tesla T4            On   | 00000000:00:1E.0 Off |                    0 |
| N/A   26C    P8    14W /  70W |      2MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figure 6.1 **The GPU setup**

6.2.2 GPT-MoE model

We ran our performance evaluation using the GPT-MoE model from Modelscope. We picked this model because it has full compatibility with FasterTransformer[23], as well as DeepSpeed[11], and the instructions for running it ① in FasterTransformer are available in the GPT guide from the FasterTransformer Github repository. We use the provided scripts ② for converting the Modelscope checkpoint, which was built based on the Megatron-LM[48] model specifications, to a format compatible with DeepSpeed and FasterTransformer. We then write our own custom scripts to convert and load the data into ExpertFlow.

The Modelscope MoE model is a fairly typical GPT-MoE model with 24 layers, half of which are regular FFN layers, and the other half are MoE. In particular, the odd-numbered layers are MoE. Regular FFN layers contain two dense layers, with an internal hidden

① https://github.com/NVIDIA/FasterTransformer/blob/main/docs/gpt_guide.md#gpt-with-moe

② https://github.com/NVIDIA/FasterTransformer/blob/main/examples/pytorch/gpt/utils/megatron_gpt_moe_ckpt_convert.py

dimension of size 4× as large as the hidden size. Each expert in the MoE layer also consists of two dense layers with an internal hidden dimension of size 4× as large as the hidden size. Additional settings are passed to the runtime via a human-readable `config.ini` file, whose contents are shown below (Listing 6.1):

Listing 6.1 config.ini

```
[gpt]
model_name = gpt
head_num = 16
size_per_head = 64
inter_size = 4096
num_layer = 24
max_pos_seq_len = 2048
vocab_size = 51200
has_adapters = False
adapter_inter_size = 0
layernorm_eps = 1e-05
start_id = 50256
end_id = 50256
weight_data_type = fp16
tensor_para_size = 4

[structure]
gpt_with_moe = 1
expert_num = 64
moe_layers = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
```

6.2.3 Experiments

6.2.3.1 Workload generation

The results of our experiments are shown below. We compare the performance of ExpertFlow to FasterTransformer, and report the throughput and latencies under different loads. We model the request arrivals using artificial traces where the arrival times are modeled by a Poisson process, and the request lengths follow a uniform distribution. This technique is similar to the approach taken in Orca[35]. We build a workload generator that takes as input the following parameters:

- the average requet arrival rate (λ)
- the total number of requests (N)
- the minimum (p_{min}) and maximum length (p_{max}) of the prompt
- the minimum (g_{min}) and maximum number (g_{max}) of tokens to generate

Given the values of the parameters above, our generator works as follows. For each request k, it generate the request's arrival time (t_k) by sampling from the Erlang distribution:

$$T = \{t_1, ..., T_N\} \sim Erlang(k, \lambda) \tag{6.1}$$

The PDF of the Erlang distribution is as follows:

$$p_{T_k} = P(T_k = t) = \frac{\lambda^k t^{k-1} e^{-\lambda t}}{(k-1)!} \tag{6.2}$$

From an implementation standpoint, we can simulate each request arrival time t_k, using just a uniform random number generator and the following formula:

$$t_k = -(1/k) \sum_{i=1}^{k} \log u_i \tag{6.3}$$

where u_i is the i-th random number generated from the uniform distribution

$$U \sim Uniform([0, 1]) \tag{6.4}$$

Next, the number of tokens in the prompt p_k and the number of tokens to generate (g_k) as output in each request are determined by sampling from the following uniform distributions:

$$p_k \sim Uniform([p_{min}, p_{max}]) \tag{6.5}$$

$$g_k \sim Uniform([g_{min}, g_{max}]) \tag{6.6}$$

Given the value of p_k, we generate the request's prompt by randomly sampling p_k values from the range of integers $[0, vocab\_size - 1]$. In summary, each request $k \in [0, N-1]$ is generated using the three parameters t_k (arrival time), p_k (# input tokens) and g_k (# output tokens).

6.2.3.2 End-to-End Results

Below, we provide the results for 4 different workloads, with $\lambda = 50, 100, 250, 500$ requests/second, $N = 2560$, $p_{min} = 8$, $p_{max} = 128$, $g_{min} = 1$, $g_{max} = 256 - p_{max}$. Table 6.1 shows the throughput (in terms of requests processed per second) for the arrival rates mentioned above. Table 6.2 shows the throughput (in terms of tokens processed per second) for the arrival rates mentioned above. We provide two different measurements for the throughput because the requests have different lengths, so it is not possible to simply calculate the tokens/s throughput from the request/s throughput. Next, Table 6.3 presents the statistics regarding the latency.

Table 6.1 Request serving throughput (requests/s)

Arrival rate (requests/s)	ExpertFlow throughput (requests/s)	FasterTransformer throughput (requests/s)
50	48.302	-
100	83.190	53.736
250	105.239	54.399
500	105.524	54.226

Table 6.2 Token generation throughput (token/s)

Arrival rate (requests/s)	ExpertFlow throughput (tokens/s)	FasterTransformer throughput (tokens/s)
50	3119.0	-
100	5373.75	3471.18
250	6806.7	3518.42
500	6815.65	3502.37

Table 6.3 Inference latency per request

Arrival rate (requests/s)	Inference System	**Average (ms)**	**Min (ms)**	**Max (ms)**
50	ExpertFlow	8,475.191	5,621.950	12,886.612
	FasterTransformer	-	-	-
100	ExpertFlow	10,217.809	5,674.259	17,971.726
	FasterTransformer	10,747.242	277.899	21,115.594
250	ExpertFlow	16,223.166	5,862.399	25,789.413
	FasterTransformer	18,646.640	224.331	36,858.986
500	ExpertFlow	18,541.337	6,141.7325	25,671.801
	FasterTransformer	21,297.635	275.063	42,134.64

Note that in the tables above, the results from FasterTransformer are missing for $\lambda = 50$ requests/s due to an issue with the framework, such that the execution of the MoE example from FasterTransformer hangs forever when the arrival rate is too low. We are currently working on fixing this bug.

Finally, we plot the latency as a function of the request length. We can see that curve from FasterTransformer is much more noisy than the one from ExpertFlow, and there seems to be a weaker correlation between between the latency and the request length. This is likely due to the padding required when batches of requests have different lengths.

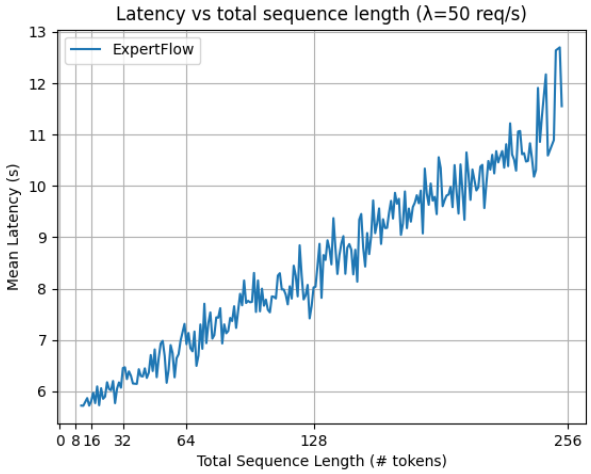


Figure 6.2 **The latency of serving a request as a function of the final request length** ($\lambda = 50$ requests/second)

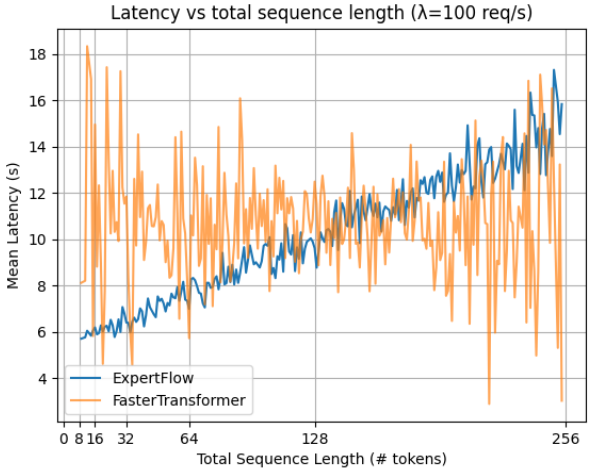


Figure 6.3 **The latency of serving a request as a function of the final request length** ($\lambda = 100$ requests/second)

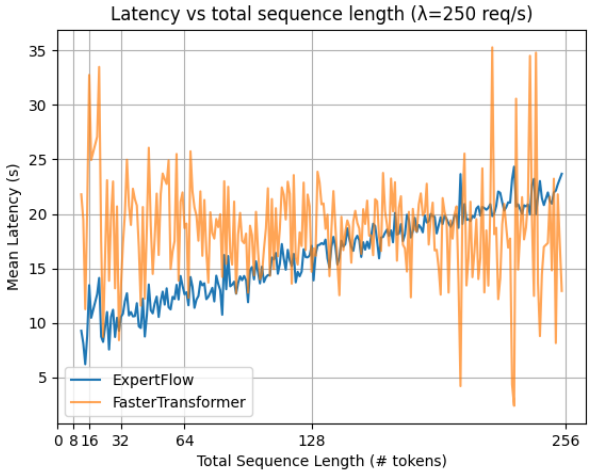


Figure 6.4 **The latency of serving a request as a function of the final request length** ($\lambda = 250$ requests/second)

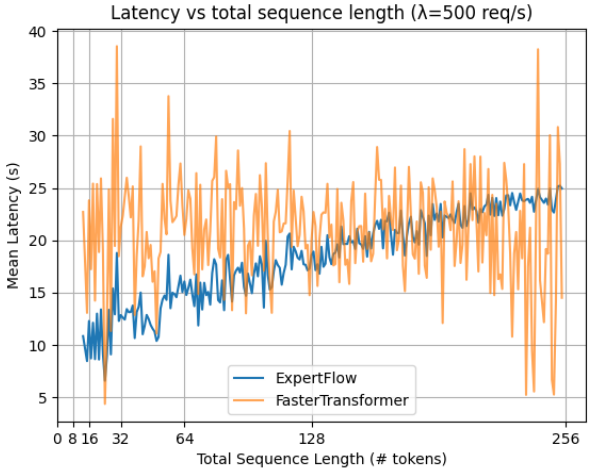


Figure 6.5 **The latency of serving a request as a function of the final request length** ($\lambda = 500$ requests/second)

6.2.3.3 Speculative Inference Evaluation

We evaluate the performance of speculative inference by measuring the speedup compared to incremental decoding. We use 5 publicly-available prompt datasets, and first run the LLAMA-7B model in incremental decoding mode, measuring the execution time. Next, we repeat the experiment and use the LLAMA-7B model for token-tree verification, and use a set of speculators derived from LLAMA-160M. We observe speedups between 1.91x and 2.75x, as shown in Figure 6.6.

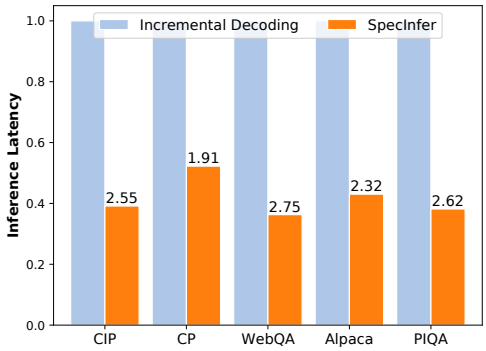


Figure 6.6 **End-to-end inference latency speedup of speculative inference** when compared to incremental decoding, using five public prompt datasets. We use LLAMA-7B as the LLM and all SSMs are derived from LLAMA-160M

6.2.3.4 Future work

Additional experiments could be useful to better understand the performance of ExpertFlow, the bottlenecks, and potential avenues for further improvement. Due to time constraints, we did not include these experiments in the current version of this document. Hence, we leave them to future work. Possible directions for additional evaluation efforts could be: an ablation study to evaluate the contribution of each of the system components towards the end-to-end performance; comparing our throughput and latency to DeepSpeed-MoE; running experiments under different sets of configurations.

6.3 Conclusion

In this chapter, we described our efforts to evaluate the correctness and performance of ExpertFlow. In Section 6.1 we illustrated how we used unit and integration tests to ensure that our system can produce correct results, which align with other inference systems. In Section 6.2, we explained the setup for our experiments, and provided the results to measure the performance of ExpertFlow when compared to the baseline.

CHAPTER 7 CONCLUSION

7.1 Summary

In this thesis, we presented ExpertFlow, an asynchronous multi-GPU serving system for Mixture of Experts (MoE) models. We began by introducing the MoE model, motivating the need for building a fast inference system, and referencing the existing works in the area. In the following chapters, we described the ExpertFlow system in detail, starting from the overall design, the parallelization plan, the runtime, and the underlying asynchronous distributed task-based platform (Legion). Next, we delved into the three components that allow ExpertFlow to obtain a competitive performance when serving MoE-based transformer models in autoregressive fashion: dynamic batching, incremental decoding, and speculative inference. Next, we introduced our optimizations at the kernel level, with a particular focus on the fused experts operator, and the multi-head attention operator. Finally, we evaluated ExpertFlow's performance and compared it to FasterTransformer, observing a speedup of up to 1.95x in the overall throughput and up to 13% in the average latency per request.

7.2 Limitations and Future Work

The system described in this thesis is still under active development as part of a research project that is expected to last for several more months to a year. The current implementation is the first prototype, or the Minimum Viable Product (MVP), or a general-purpose asynchronous inference system that will serve as a backbone for research at CMU. One limitation, of the current system, for example, is the lack of support for mixed precision, which we are currently working on implementing, in order to reduce the memory overhead. We are also actively working on adding support for offloading-based generative LLM inference. We thus expect ExpertFlow to evolve in several different directions in the future. In particular, we expect to continue optimizing the performance of the MoE kernel and operator, and we will further develop the speculative inference component.

REFERENCES

[1] Jacobs R A, Jordan M I, Nowlan S J, et al. Adaptive mixtures of local experts[J/OL]. Neural Computation, 1991, 3(1): 79-87. DOI: 10.1162/neco.1991.3.1.79.

[2] Shazeer N, Mirhoseini A, Maziarz K, et al. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer[M/OL]. arXiv, 2017. https://arxiv.org/abs/1701.06538. DOI: 10.48550/ARXIV.1701.06538.

[3] Lepikhin D, Lee H, Xu Y, et al. Gshard: Scaling giant models with conditional computation and automatic sharding[J/OL]. CoRR, 2020, abs/2006.16668. https://arxiv.org/abs/2006.16668.

[4] Hwang C, Cui W, Xiong Y, et al. Tutel: Adaptive mixture-of-experts at scale[J/OL]. CoRR, 2022, abs/2206.03382. https://arxiv.org/pdf/2206.03382.pdf.

[5] He J, Zhai J, Antunes T, et al. Fastermoe: Modeling and optimizing training of large-scale dynamic pre-trained models[C/OL]//PPoPP '22: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA: Association for Computing Machinery, 2022: 120–134. https://doi.org/10.1145/3503221.3508418.

[6] Fedus W, Zoph B, Shazeer N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity[J/OL]. Journal of Machine Learning Research, 2022, 23(120): 1-39. http://jmlr.org/papers/v23/21-0998.html.

[7] Ma Z, He J, Qiu J, et al. Bagualu: Targeting brain scale pretrained models with over 37 million cores[C/OL]//PPoPP '22: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA: Association for Computing Machinery, 2022: 192–204. https://doi.org/10.1145/3503221.3508417.

[8] Sutton R. The Bitter Lesson[EB/OL]. 2019[2022-06-16]. http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

[9] Rotman D. We're not prepared for the end of moore's law[J/OL]. MIT Technology Review, 2021. https://www.technologyreview.com/2020/02/24/905789/were-not-prepared-for-the-end-of-moores-law/.

[10] Switch transformers c - 2048 experts (1.6t parameters for 3.1 tb)[M/OL]. HuggingFace. https://huggingface.co/google/switch-c-2048.

[11] Rajbhandari S, Li C, Yao Z, et al. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale[C/OL]//Chaudhuri K, Jegelka S, Song L, et al. Proceedings of Machine Learning Research: volume 162 Proceedings of the 39th International Conference on Machine Learning. PMLR, 2022: 18332-18346. https://proceedings.mlr.press/v162/rajbhandari22a.html.

[12] Ott M, Edunov S, Baevski A, et al. fairseq: A fast, extensible toolkit for sequence modeling [C]//Proceedings of NAACL-HLT 2019: Demonstrations. 2019.

[13] Nvidia collective communication library (nccl)[M]. NVIDIA.

[14] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C/OL]//Guyon I, Luxburg U V, Bengio S, et al. Advances in Neural Information Processing Systems: volume 30. Curran Associates, Inc., 2017. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[15] Dosovitskiy A, Beyer L, Kolesnikov A, et al. An image is worth 16x16 words: Transformers for image recognition at scale[A]. 2020.

[16] Baevski A, Zhou Y, Mohamed A, et al. wav2vec 2.0: A framework for self-supervised learning of speech representations[J]. Advances in neural information processing systems, 2020, 33: 12449-12460.

[17] Arnab A, Dehghani M, Heigold G, et al. Vivit: A video vision transformer[C]//Proceedings of the IEEE/CVF international conference on computer vision. 2021: 6836-6846.

[18] Radford A, Narasimhan K, Salimans T, et al. Improving language understanding by generative pre-training[J]. Preprint, 2018.

[19] Radford A, Wu J, Child R, et al. Language models are unsupervised multitask learners[Z]. 2019.

[20] Kossmann F, Jia Z, Aiken A. Optimizing mixture of experts using dynamic recompilations [M/OL]. arXiv, 2022. https://arxiv.org/abs/2205.01848. DOI: 10.48550/ARXIV.2205.01848.

[21] Artetxe M, Bhosale S, Goyal N, et al. Efficient large scale language modeling with mixtures of experts[C/OL]//Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, 2022: 11699-11732. https://aclanthology.org/2022.emnlp-main.804.

[22] Fang J, Yu Y, Zhao C, et al. Turbotransformers: an efficient gpu serving system for transformer models[C]//Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2021: 389-402.

[23] NVIDIA. Fastertransformer[EB/OL]. 2023[2023-04-09]. https://github.com/NVIDIA/FasterTransformer.

[24] Devlin J, Chang M, Lee K, et al. BERT: pre-training of deep bidirectional transformers for language understanding[C/OL]//Burstein J, Doran C, Solorio T. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers). Association for Computational Linguistics, 2019: 4171-4186. https://doi.org/10.18653/v1/n19-1423.

[25] Yang Z, Dai Z, Yang Y, et al. Xlnet: Generalized autoregressive pretraining for language understanding[J]. Advances in neural information processing systems, 2019, 32.

[26] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners[C/OL]//Larochelle H, Ranzato M, Hadsell R, et al. Advances in Neural Information Processing Systems: volume 33. Curran Associates, Inc., 2020: 1877-1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[27] Zhang S, Roller S, Goyal N, et al. Opt: Open pre-trained transformer language models[A]. 2022.

[28] Scao T L, Fan A, Akiki C, et al. Bloom: A 176b-parameter open-access multilingual language model[A]. 2022.

[29] Wang B, Komatsuzaki A. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model [EB/OL]. 2021. https://github.com/kingoflolz/mesh-transformer-jax.

[30] Beltagy I, Peters M E, Cohan A. Longformer: The long-document transformer[A]. 2020.

[31] Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer[J/OL]. Journal of Machine Learning Research, 2020, 21(140): 1-67. http://jmlr.org/papers/v21/20-074.html.

[32] Liu Z, Lin Y, Cao Y, et al. Swin transformer: Hierarchical vision transformer using shifted windows[C]//Proceedings of the IEEE/CVF international conference on computer vision. 2021: 10012-10022.

[33] NVIDIA. The triton inference server[EB/OL]. https://github.com/triton-inference-server/server.

[34] NVIDIA. Triton fastertransformer backend[EB/OL]. https://github.com/triton-inference-server/fastertransformer_backend.

[35] Yu G I, Jeong J S, Kim G W, et al. Orca: A distributed serving system for Transformer-Based generative models[C/OL]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Carlsbad, CA: USENIX Association, 2022: 521-538. https://www.usenix.org/conference/osdi22/presentation/yu.

[36] Kim S, Mangalam K, Malik J, et al. Big little transformer decoder[A]. 2023. arXiv: 2302.07863.

[37] Leviathan Y, Kalman M, Matias Y. Fast inference from transformers via speculative decoding [A]. 2022. arXiv: 2211.17192.

[38] Zheng L, Li Z, Zhang H, et al. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning[C/OL]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Carlsbad, CA: USENIX Association, 2022: 559-578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

[39] Jia Z, Zaharia M, Aiken A. Beyond data and model parallelism for deep neural networks. [C/OL]//Talwalkar A, Smith V, Zaharia M. Proceedings of Machine Learning and Systems: volume 1. 2019: 1-13. https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf.

[40] Bauer M, Treichler S, Slaughter E, et al. Legion: Expressing locality and independence with logical regions[C/OL]//SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. 2012: 1-11. DOI: 10.1109/SC.2012.71.

[41] Unger C, Jia Z, Wu W, et al. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization[C/OL]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Carlsbad, CA: USENIX Association, 2022: 267-284. https://www.usenix.org/conference/osdi22/presentation/unger.

[42] Miao X, Oliaro G, Zhang Z, et al. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification[A]. 2023. arXiv: 2305.09781.

[43] Blelloch G E. Prefix sums and their applications[M/OL]//Sythesis of parallel algorithms. Morgan Kaufmann Publishers Inc., 1990: 35-60. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.6430.

[44] NVIDIA. Nvidia/thrust: The c++ parallel algorithms library.[EB/OL]. https://github.com/NVIDIA/thrust.

[45] Touvron H, Lavril T, Izacard G, et al. Llama: Open and efficient foundation language models [A]. 2023.

[46] Artetxe M, Bhosale S, Goyal N, et al. Efficient large scale language modeling with mixtures of experts[C/OL]//Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, 2022: 11699-11732. https://aclanthology.org/2022.emnlp-main.804.

[47] GPT-MoE 中文 67 亿诗歌生成模型[EB/OL]. https://www.modelscope.cn/models/PAI/nlp_gpt3_text-generation_0.35B_MoE-64/summary.

[48] Shoeybi M, Patwary M, Puri R, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism[A]. 2019.

ACKNOWLEDGEMENTS

First and foremost, I am deeply grateful to Prof. Jidong Zhai for his support and guidance throughout the entirety of this thesis project. From the proposal stage to the research process, final presentation, and revisions, Prof. Zhai's expertise and mentorship have been invaluable. Additionally, I am grateful for his feedback on my status reports and for his assistance in navigating the challenges posed by the border closure and my arrival at Tsinghua University, including helping me schedule meetings at times that worked in my timezone, and putting me in contact with fellow students that helped me understand and fill all required forms in time, despite knowing very little Chinese. I would also like to extend my heartfelt appreciation to Prof. Zhihao Jia from Carnegie Mellon University for co-advising the ExpertFlow project. His regular meetings, invaluable advice, and provision of necessary resources, including computing facilities for developing and testing the ExpertFlow prototype, have been immensely beneficial to my research progress. I am indebted to my Tsinghua classmates for their friendship and camaraderie, making me feel at home away from home. From online orientation events to working together on projects, meeting up in different cities like Paris and Milan, and finally being reunited on the beautiful Tsinghua campus in Beijing, these cherished memories have enriched my MS program experience. I am also grateful to my Tsinghua lab mates for their mentorship and assistance with my project, as well as helping me navigate campus life and having a unforgettable experience here at Tsinghua University. My heartfelt thanks also go to my CMU lab mates Daiyaan Arfeen, Xinhao Cheng, Zeyu Wang, Rae Wong, Zhihao Zhang, who have made significant contributions to the project, as well as to graduate students Reyna Abhyankar (UCSD) and Colin Unger (Stanford), who were involved extensively in the project. I would also like to recognize Xupeng Miao for helping me translate the title and abstract into Chinese. Finally, I am deeply grateful to my family, my girlfriend, and my friends for their unwavering love, support, and encouragement.

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

RESUME

个人简历

Gabriele Oliaro was born on July 31, 1998 in Turin, Piedmont, Italy. He attended high school in Milan, Italy and graduated in 2017. He began his bachelor's study in the School of Engineering and Applied Sciences, Harvard University, USA in August 2017, majoring in electrical engineering, and earned a Bachelor of Science degree in May 2021. He began his master's study in the Department of Computer Science, Tsinghua University, China in September 2021, and is expected to earn a Master of Science degree in Computer Science in June 2023.

在学期间完成的相关学术成果

学术论文

[1] Langlet J., Ben-Basat R., Oliaro G., Mitzenmacher M., Yu M., Antichi G. 2023. Direct Telemetry Access. In Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM '23). https://doi.org/10.48550/arXiv.2202.02270

[2] Langlet J., Ben-Basat R., Ramanathan S., Oliaro G., Mitzenmacher M., Yu M., Antichi G. 2021. Zero-CPU Collection with Direct Telemetry Access. ACM Workshop on Hot Topics in Networks 2021 (HotNets '21). https://doi.org/10.1145/3484266.3487366

[3] Oliaro G. Pitcher: A Probabilistic In-band Telemetry Checker. Bachelor Thesis. Harvard University, 2021.

预印本

[1] Miao X., Oliaro G., Zhang Z., Cheng X., Wang Z., Wong R., Chen Z., Arfeen D., Abhyankar R., Jia Z. 2023. SpecInfer: Accelerating Generative LLM Serving with Speculative Inference and Token Tree Verification. ArXiv PrePrint. https://doi.org/10.48550/arXiv.2305.09781

COMMENTS FROM THESIS SUPERVISOR

混合专家模型是当前自然语言处理领域一个重要模型。本文围绕混合专家模型的推理任务优化开展相关工作，主要的贡献包括：

1. 设计了低延迟的 MOE 推理系统，ExpertFlow，用于提供高效的 MOE 模型推理服务。该框架支持多 GPU 以及多节点的推理，并且实现异步模型。
2. 在 Nvdia 平台对 ExpertFlow 系统进行性能评价，结果表明该系统和 Faster-Transformer 相比，吞吐量高达 1.95 倍，平均延迟降低 13%。作者开源了相关代码。

RESOLUTION OF THESIS DEFENSE COMMITTEE

This thesis tackles the problem of performing efficient inference on large Mixture of Experts (MoE) models, specifically GPT models. MoE models are critical for scaling deep learning models to massive sizes but inference remains challenging.

The main contributions made by this thesis include: The thesis designs and implements ExpertFlow, a high-performance MOE inference framework with multi-GPU parallelism. Results show that ExpertFlow achieves up to 1.95x higher throughput and 13% lower latency than NVIDIA's FasterTransformer. ExpertFlow is open sourced, which is a solid implementation that can make MoE models more accessible.

The thesis shows that the author has solid basic theoretical and professional knowledge and demonstrates the author's ability of independent research. The thesis is well structured and organized. During the thesis defense, the author presented his work clearly and answered questions correctly. The defense committee has approved Gabriele Oliaro's thesis defense and nominated him to be awarded the degree of Master of Science.