A Hybrid Online and Offline Requests Inference Serving System for LLM in Single GPU Environment

YUCHEN SHEN

M.Phil (Eng)

Supervisor: Dong Yuan
Associate Supervisor: Huaming Chen

A thesis submitted in fulfilment of
the requirements for the degree of
Master of Philosophy

School of Electrical and Computer Engineering
Faculty of Engineering
The University of Sydney
Australia

7 March 2025

Abstract

The application scope of Large Language Models (LLMs) has significantly expanded, with these models adapting well to different tasks through a minimal addition of extra parameters. However, due to limited GPU memory, which cannot accommodate all model parameters, running multitasking LLM inference on consumer-grade personal computers presents challenges. Current methods attempt to address this by offloading unused parameters to main memory and prefetching them when needed. Unfortunately, these methods exhibit noticeable latency issues due to the overhead associated with data transfer.

We propose FixGen, a single GPU LLM online-offline mixed inference serving system that supports multi-task inference. First, we develop FixPool to optimize memory management by centralizing storage in a fixed memory space and optimizing parameter storage procedures, thereby reducing PCIe resource consumption and improving efficiency. Secondly, we use a router to select appropriate operators to compute requests in the prefill and decode stages within the same batch to reduce the response latency for sudden online requests.

FixGen reduces the online request-response latency between 1.06 and 1.84 times that of the conventional method while maintaining the throughput to offline requests on a single GPU from OPT-6.7b to OPT-30b.

In conclusion, our study highlights that employing FixGen has improved resource utilization and multitasking inference efficiency in single GPU systems. By unifying parameter management and batch strategy, FixGen enables the deployment of powerful Large Language Models (LLMs) on local consumer-grade AI PCs, making more potential applications feasible. Thus, FixGen offers a promising direction for further research and development in the field of deep learning.

Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Acknowledgements

First and foremost, I wish to express my deepest gratitude to all those who have supported and assisted me throughout my research and the writing of this thesis.

I am deeply grateful to my supervisor, Prof. Dong Yuan, for his invaluable guidance in the selection of the research topic, the development of research methodologies, and the meticulous revisions of this thesis. His patient instruction during my initial unfamiliarity with academic writing, along with his insightful suggestions, has been instrumental in my significant progress. Beyond academics, his support in my personal life, coupled with his tolerance and kindness, has deeply influenced me and instilled in me an appreciation for the allure of both scholarship and life.

I also sincerely thank the honorable members of the review committee for their valuable feedback and suggestions that have greatly improved the quality of this thesis.

My heartfelt thanks go to Yuning Zhang, Nan Yang for their substantial assistance and support during the experimental phase. Despite working in different fields, he offered numerous constructive suggestions and provided encouragement during challenging times, allowing me to proceed smoothly with my writing.

I would like to extend special thanks to my girlfriend, Chunyu Wang, for her assistance in refining the language, charts, and illustrations.

Lastly, I am profoundly grateful to my family and friends for their unwavering support and encouragement throughout my research journey. Without their selfless support, the accomplishments reflected in this thesis would not have been possible. Once again, I extend my sincere appreciation to everyone who has contributed and supported me along the way.

List of Publication

Papers that are under review

Yuchen Shen, Yuning Zhang and Dong Yuan (2023) " FixGen: A Hybrid Online and Offline Request LLM Inference Service System for Single GPU Environment" Under review at the MLsys 2025.

Contents

Abstract ii

Statement of Originality iii

Acknowledgements iv

List of Publication v

Contents vi

List of Figures x

Chapter 1 Introduction 1

1.1 Motivation 1

1.2 Challenge 3

1.3 Research Objectives 5

1.4 Contribution 6

1.5 Thesis Overview 7

Chapter 2 Literature Review and Preliminaries 9

2.1 Overview of Developments for LLMs 9

2.2 Foundations of Efficient Inference 11

2.2.1 Parallelism and Resource Partitioning 12

2.2.2 Hardware Accelerator 14

2.3 Architecture of LLMs 16

2.3.1 Basic Transformer 16

2.3.2 The structure of LLMs 19

2.3.3 Fine-tune of LLM 21

2.3.3.1 General Fine-tuning 21

2.3.3.2 Adapter Fine-tuning Structure 22
2.4 Optimization for LLM inference 24
2.4.1 Kernel Level Optimization 25
2.4.1.1 Common Kernel 25
2.4.1.2 Kernel Fusion 26
2.4.1.3 Customized Kernel 27
2.4.2 Memomry Management for LLM inference 28
2.4.2.1 General Memory Management 30
2.4.2.2 Resource Limited Memory Management 31
2.4.2.3 KV Cache Management 31
2.4.3 Batch Strategy for LLM inference 33
2.4.3.1 General Batch Strategy for LLM 33
2.4.3.2 Limited Resource Batch Strategy 35

Chapter 3 System Overview **36**
3.1 FixGen Architecture 36
3.1.1 Memory Management: FixPool 38
3.1.2 Inference Process and Batch Strategy 38
3.2 Fixgen Inference Workflow 39

Chapter 4 Memory Management: FixPool **42**
4.1 Memory Structure 42
4.1.1 Parameter Distribution in Memory Layer 44
4.1.2 Working Space 47
4.2 Storage Strategy 48
4.2.1 Hyperparameter Policy 49
4.2.2 Task-specific Parameter Store 50
4.3 KV Cache Update 51
4.3.1 Update Process 51
4.3.2 Update Kernel Design 52

Chapter 5 Inference Process and Batch Strategy **56**

5.1 Batching Stategy 57
5.1.1 Online Request Strategy 57
5.1.2 Offline Request Strategy 59
5.2 Router 60
5.2.1 Router Process 62
5.2.2 Attention Kernel 65
5.2.2.1 General Attention 65
5.2.2.2 Hybrid Computation 67
5.3 Post-Inference 73
5.3.1 Update Process 73
5.3.2 Memory Reorganization 75

Chapter 6 Experiments and Evaluation **78**
6.1 Setup 78
6.1.1 Implementation 79
6.1.2 Hardware 81
6.1.3 Model 82
6.1.4 Baseline 83
6.1.5 Metrics 85
6.2 End-to-End Results on Synthetic Workloads 85
6.2.1 End-to-End Results on Primary Hardware Set 87
6.2.2 End-to-End Latency on A6000 set 89
6.3 Prefetch Latency 91
6.4 Multi-tasks Request Response Latency 91
6.5 Additional Experiment 92
6.5.1 Runtime Breakdown 92
6.5.2 Ablation Study 93

Chapter 7 Conclusion **94**
7.1 Future Outlook 95

Bibliography **97**

1 Terminology List 105

List of Figures

1.1 The process of generating new tokens of all requests in batch while receiving a sudden online request. The conventional approach take T1 to T3 while FixGen can only take T1 to T2. 4

1.2 The structure of thesis. 8

2.1 Transformer Architecture [Vas+17] 18

2.2 (left) Self-attention. (right) Multi-Head Attention consists of several attention layers running in parallel. [Vas+17] 18

2.3 Architecture of Adapters [Hu+23] 23

2.4 Architecture of Paged KV cache[Kwo+23] 32

3.1 The Overview of the FixGen workflow: ❶ The user's online request is first stored temporarily in the request pool. ❷ The batch strategy prepares the batch and allocates the memory for the KV cache needed for the batch in advance. ❸ Assign memory to each request and prepare necessary parameters. ❹ The model computes the next token.❺ Preload parameter to the workspace. ❻ Access parameter for computation. ❼ Post-processing saves the newly generated token in the request. ❽ Return the completed request ❾. Recalculate the space required for the next round of computation and adjust the batch according to the required space. 37

4.1 The Layer structure design of Fixpool and the parameter distribution for memory layers. 43

4.2 The process of storing the KV cache. 52

5.1 Comparison between general attention and hybrid computing. 62

6.1 The average response latency (Second) for one sudden online request that can be achieved by a different system with different batch sizes from OPT-6.7b to OPT-30b. The prefill length=512 and decode length=32. 86

6.2 Throughput comparison under different memory constraints for OPT models from 6.7B to 30B with `batch size`=32, `prefill length`=512, `decode length`=32. DeepSpeed Inference cannot set setting the parameter offloading ratio. Accelerate and DeepSpeed Inference are out of memory for OPT-13B and OPT-30B. 88

6.3 Throughput comparison under different memory constraints for OPT models from 6.7B to 30B with batch size=16, prefill length=512, decode length=32. DeepSpeed Inference cannot set setting the parameter offloading ratio. Accelerate and DeepSpeed Inference are out of memory for OPT-13B and OPT-30B. 89

6.4 The average response latency (Second) for one sudden online request that can be achieved by a different system with different batch sizes from OPT-6.7b to OPT-30b in A6000. The prefill length=512 and decode length=32. 90

CHAPTER 1

Introduction

1.1 Motivation

Recently, transformer-based Large Language Models (LLMs) have shown a notable improvement in a wide range of areas [Vas+17], with their size expanding to better accommodate enhanced performance [Bro+20; Dev+19; Ope+24; Tou+23a; Zha+22a]. For example, GPT-3[Rad+19] and GPT-4 [Ope+24] have demonstrated that through a few-shot learning, they can efficiently adapt to a variety of tasks, while the BERT pretraining paradigm [Dev+19] and fine-tuning have laid the foundation for task-specific optimizations.

A core advantage of LLMs lies in their powerful general language representation capabilities acquired through pretraining, enabling them to adapt to a wide range of downstream tasks. When fine-tuned for specific tasks, these models are further optimized using task-specific data, resulting in enhanced performance. This "pretraining-finetuning" paradigm has become the dominant model development approach, allowing LLMs to achieve state-of-the-art performance in various fields, including machine translation, text generation, and question-answering systems.

In addition, as the scale of models continues to grow, researchers are exploring ways to optimize training efficiency, reduce inference costs, and improve adaptability to sparse data. Models such as the GPT [Ope+24] series and the LLaMA [Tou+23b] series exemplify this trend, showing not only the potential of scaling, but also pushing the performance limits through architectural improvements and algorithmic optimizations.

However, to load these models and their task-specific parameters for various applications, large memory resources are typically required [Zho+22; She+23b; Du+22]. This high memory demand not only significantly increases the resource consumption of LLMs during both training and inference, but also greatly limits their practical deployment. Despite the powerful capabilities and precise task execution offered by LLMs, the resource bottlenecks they face, especially in terms of memory and computational requirements, remain a major challenge during real-world deployment.

This memory consumption can be alleviated in high-performance computing centers and cloud platforms, as these environments can provide sufficient computational and storage resources. However, most personal computers, especially consumer-grade Artificial Intelligence Personal Computers (AIPC) and edge devices, often struggle to meet the memory requirements needed to run LLMs. For these devices, running LLMs demands significant memory, storage, and computational resources, which in many cases leads to performance degradation or even complete failure. As a result, many potential applications that could benefit from LLMs, such as intelligent assistants on low-power devices, real-time translation systems, personalized recommendations, and medical decision support, are excluded from these environments.

Due to demands for privacy protection, maintaining stable connections and responsiveness, and supporting offline usage, running LLMs locally has become an urgent need [Zha+24]. In such scenarios, users often require the ability to run multiple applications on the same device, each of which may depend on LLMs to efficiently handle different tasks. These tasks can vary significantly in type and requirements, resulting in the model needing to balance the computational and resource demands of multiple applications simultaneously during runtime.

For example, in the financial sector, a trader may require an LLM to simultaneously perform tasks such as financial analysis [YTT23], stock trading [Li+24a]. These tasks typically have different demands in terms of request volume, request frequency, and response latency, requiring the model to handle varying workloads efficiently while meeting the specific needs of each task.

These tasks can typically be categorized into two types:

1) Sudden online tasks that typically involve a lower number of concurrent requests but have high request frequencies and require rapid responses. For example, applications such as intelligent assistants or real-time question-answering systems are highly sensitive to latency, requiring the system to deliver accurate responses within extremely short timeframes. In addition, the request timing for these tasks is unpredictable, with requests potentially arriving at any moment. This requires that the system has sufficient elasticity to handle sudden surges in request flow without compromising service quality or user experience.

2) Offline tasks that have high volume, such as text translation. Throughput is often used as a performance metric for such inference tasks since they do not require a quick response. Individual users often have only one computer equipped with a commodity GPU as their computing device. This necessitates the adoption of reasonable methods to effectively manage computing loads and memory usage while quickly responding to online requests and maintaining high throughput for offline requests, presenting challenges to current technology.

Two main approaches dominate the current landscape of LLM inference systems: one emphasizes efficient inference on servers with abundant computational resources, while the other targets optimization for specific tasks on edge devices. The environment of consumer-grade personal computers, which involves a small user base but requires the ability to process various types of tasks, has been neglected. Nevertheless, achieving optimization in this context encounters a number of challenges.

1.2 Challenge

This multitask, high-concurrency operating environment requires the model to possess not only strong computational power on the local device, but also efficiently manage and schedule memory and computational resources. This ensures a balanced allocation of resources between different applications, preventing resource contention and maintaining optimal performance for each task. The deployment of LLMs on personal computers presents two primary challenges: memory management and batch processing.

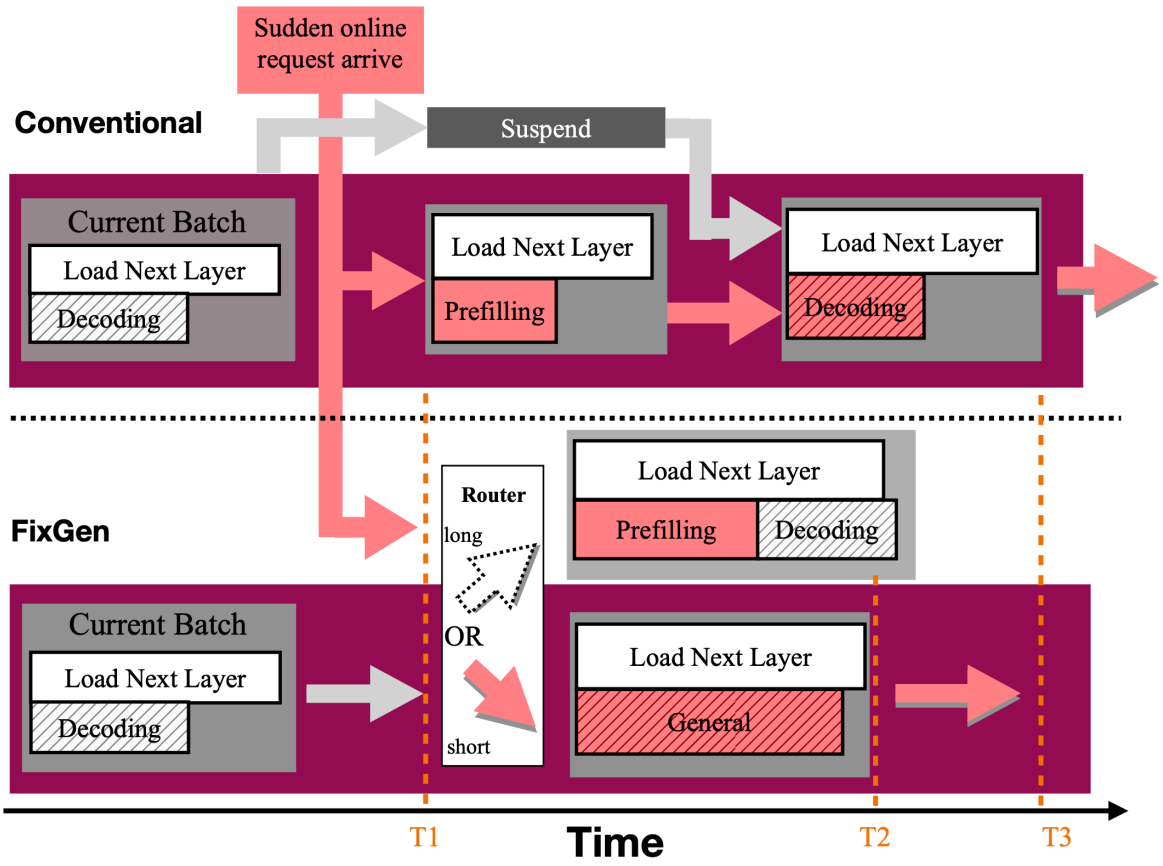

FIGURE 1.1: The process of generating new tokens of all requests in batch while receiving a sudden online request. The conventional approach take T1 to T3 while FixGen can only take T1 to T2.

1) Offloading techniques are used in resource-limited LLMs systems, where parameters are stored in CPU memory and on disk and are prefetched to GPU memory only when needed. Maximizing GPU utilization can reduce prefetch latency and enhance efficiency. Existing inference systems in resource-limited environments, such as FlexGen [She+23a], prioritize high throughput on a single GPU, allocating as much GPU memory as possible to accommodate more requests, suitable for offline computation of requests. The use of online requests introduces problems such as sudden online requests joining an ongoing system needing to allocate space based on their length and the maximum volume of text they can generate, which can lead to memory shortages. The frequent memory allocation and release operations that come with it can affect the efficiency of the system.

2) Conventional reasoning systems cannot handle unexpected tasks in real time while inference, which brings high latency for online requests. The reason is that LLM inference systems typically divide the inference process for requests into two stages: prefilling and decoding. If not prefilled, sudden online requests cannot be added directly to batches in the decoding stage. This prevents these requests from being immediately incorporated into the running batch and requires them to wait until the current batch is completed before joining the next batch's prefilling stage, significantly increasing their response latency. A conventional solution shown in Fig. 1.1 is to pause the current batch, wait for the sudden online requests to complete the prefilling stage, and then integrate them back into the current batch to continue operations. However, this approach affects overall throughput and latency.

1.3 Research Objectives

Current mainstream solutions usually focus on offline-only or online-only environments without considering memory management across multiple tasks. Currently, there are some swap-in and swap-out solutions that enable LLMs to run on personal computers. However, these approaches often fail to consider the actual user experience, as users rarely perform a single task on a computer.

Our research will focus on:

1. Addressing the batch processing challenge in a hybrid offline-online request environment. The goal is to maximize system responsiveness and minimize latency for online tasks while maintaining high throughput, ensuring that user-facing online tasks are processed promptly even during periods of high demand.

2. Memory management in a consumer-grade PC environment with optimized resource allocation for multi-task scenarios. Swap-in and swap-out operations are a critical bottleneck for large model inference in resource-constrained settings. Our research aims to minimize the overall latency of these operations by strategically utilizing memory space and I/O resources, thereby reducing unnecessary swap activities.

1.4 Contribution

To address the challenges mentioned above, we have introduced FixGen, an LLM inference system that allocates **Fix**ed memory for LLM to **Gen**erate texts for both online and offline requests in single GPU environments as shown in Fig. 1.1. This framework is meticulously crafted with the principal aim of minimizing online request response latency and maximizing the memory ultilization to improve the efficiency of LLM inference on single GPU environment.

It utilizes a memory management system specifically designed for LLM offloading scenarios to use memory and I/O resources efficiently. In order to quickly allocate memory for bursty online requests, we allocate all memory in advance, and will try to keep the same type of parameter contiguous in storage space in order to cope with prefetch operations. In addition, FixGen uses prefilling and decoding operators sequentially or general operators to infer requests in both phases in one batch to quickly respond to online requests with a high throughput of offline requests.

In summary, our contributions are as follows.

We proposed the FixGen inference service system, a low-latency, high-throughput system designed to handle both online and offline requests on a single GPU simultaneously. FixGen can dynamically adjust the offloading ratio to maximize the utilization of GPU memory and switch between different attention operators according to the specific needs of the batch for efficient inference.

We proposed FixPool, a low-latency memory management system designed to precisely control memory distribution and offloading in a single GPU environment. During the setup phase of the entire system, Fixpool pre-allocates the memory required for the inference process of each layer of the LLM. This memory is divided into fixed-size blocks, which are used to store the weights of the computational layers used for inference and the KV cache corresponding to runtime requests.

We propose a universal batching strategy for mixed offline and online request environments, which is shown in Fig. 1.1 as well. This method ensures that all online requests are prioritized for computation and aims to increase the batch size as much as possible, considering the type of task at hand to fully utilize the GPU's memory capacity. In order to reduce the response latency of online requests, we reason about requests in the same batch that are in the decoding and prefilling phases in a mixed way. To this end, we modify the flash attention operator to be compatible with general attention. Additionally, we employ a router to select the appropriate attention kernel based on the length of the request, thereby enhancing the computational efficiency during the mixed prefilling and decoding phases. In this process, careful consideration is also given to the efficient utilization and cleanup of memory.

Finally, our last contribution is the application of our method across models of varying sizes. We conducted extensive experiments on two different GPUs to verify the effectiveness of our FixGen. The results of these experiments were highly encouraging, demonstrating FixGen's exceptional ability to enhance response speed for requests. The experimental results show that we have reduced the online request-response latency by from 1.06 to 1.84 times compared to the current state-of-the-art methods while maintaining the throughput for offline requests.

1.5 Thesis Overview

Figure 1.2 shows the architecture of this thesis. In this thesis, we have seven chapters to conclude my work. In chapter 1, we have presented our motivation and show our solution and contribution based on motivation. In chapter 2, we review the relevant optimization methods and some existing solutions for LLM inference on a commercial-grade PC, especially in a single GPU environment. The overview of our proposed FixGen system is introduced in chapter 3. Subsequently, we will detail the design of FixGen with a focus on two aspects: memory management and batch processing strategies in next two chapters. In chapter 4, the design of memory management in FixGen, including the architecture of Fixpool and its management, as well as the allocation of memory resources, will be described. In chapter 5, we will explain the batch processing workflow within FixGen, such as how to efficiently mix

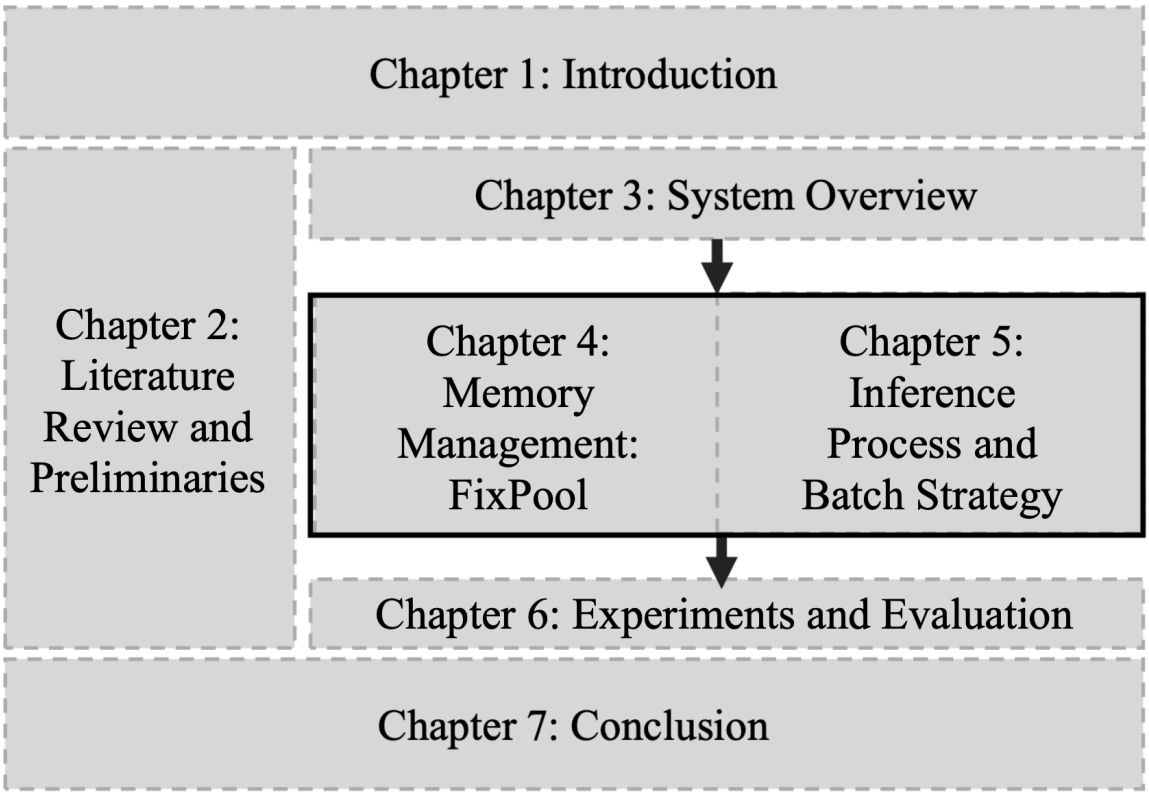

FIGURE 1.2: The structure of thesis.

online and offline tasks, and how to maximize the use of the memory system discussed in the previous chapter during the batch processing sequence. Relevant experiments and their results with respect to FixGen will be presented in the chapter 6. These results demonstrate the efficiency of FixGen's operation. Finally, in chapter 7, we will summarize the contributions of this work, the advantages and limitations of FixGen, as well as its potential future applications and areas for improvement.

CHAPTER 2

Literature Review and Preliminaries

In this chapter, we will present a review of the literature on LLM and its inference, inference in a resource-limited environment, and batching strategy. The inference performance of LLMs is highly dependent on their architecture. We first introduce the recent development and challenge of LLM in section 2.1. Then in section 2.2, we introduce the general concepts of parallel computing and the hardware that supports parallel computing, which are essential to understand the inference of LLMs. In Section 2.3, we will explore the structure of LLMs, following the evolution from the original transformer model to the current state of LLMs. We will also discuss how fine-tuning can impact the architecture of these LLMs. In the final section, 2.4, we will list some LLM inference service systems and discuss optimizations in LLM inference from three perspectives: kernel optimization, memory management, and batch processing. This section aims to provide a comprehensive understanding of the strategies used to enhance the efficiency and effectiveness of large-model inference processes.

2.1 Overview of Developments for LLMs

In recent years, artificial intelligence has made significant advancements, and using powerful AI tools to enhance efficiency has become commonplace. These tools include ChatGPT [Ope+24], Copilot [Git21], Midjourney [Bor23], and Adobe Firefly [Ang24]. In the field of natural language processing (NLP), the focus has shifted towards developing Artificial General Intelligence (AGI). Some of the current language models are now capable of handling multiple tasks, achieving performance in various domains that is comparable to, or even surpasses, that of humans, thereby opening up countless potential applications. However,

despite their impressive capabilities, LLMs face significant challenges, particularly in terms of memory requirements and computational resources. These models require extensive memory and processing power, making them difficult to deploy on personal computers or resource-constrained environments.

The tremendous success of NLP is not only due to the scale of training data or innovations in training methods, but also significantly attributed to the introduction of transformer models [Vas+17]. These models provide exceptional global contextual understanding capabilities at the cost of substantial computational power. To achieve more precise results, the size of the model weights has been growing exponentially. Containing billions of parameters, these LLMs possess profound learning abilities. They can update and memorize the knowledge and skills required for specific tasks, which demands high computational resources due to their massive parameter count. Due to their size, these models are commonly referred to as LLMs.

Due to the powerful capabilities of LLMs, they can be applied directly to different tasks, and some techniques allow enhancing their performance on specific downstream tasks by training only a subset of additional parameters without altering the core weights of the model itself [Hu+23; Tou+23a]. For users who need to deploy LLMs locally, using a robust LLM to handle multiple tasks is a reasonable demand. Depending on the characteristics of the specific tasks, their workload and the required response times may vary. Therefore, conventional LLMs inference systems are not easily adaptable to such scenarios where multiple types of task are present. Traditional batch processing typically focuses on environments with only one type of task, necessitating further research into how batch processing strategies can be optimized for environments where multiple task types coexist simultaneously to achieve the most satisfactory results.

Since the resources required for LLM inference are significantly less than those needed for training, which involves multiple iterations, gradient logging, and repeated parameter adjustments to achieve better results, running LLMs locally is feasible. Inference only involves the model's forward propagation to generate the next predicted text. Local inference offers several benefits that cloud service providers may not be able to match: it maximizes user privacy, grants users greater control, allows for offline usage, and provides exclusive access

to hardware resources. Consequently, despite many service providers offering cloud-based services for LLMs, there is still a demand for local inference capabilities. To meet this demand, some companies have launched AI personal computers (AIPCs), designed as the next generation of PCs equipped with dedicated AI processing units. However, commercial-grade hardware still struggles to meet the extensive memory requirements of LLMs. Some systems utilize quantization methods to compress LLMs, trading off some accuracy to reduce resource consumption. Meanwhile, other systems employ offloading techniques that utilize CPU, RAM, and other resources to assist in LLM inference, thereby reducing the dependency on GPUs.

In summary, running LLMs locally has many potential demands and is a direction worthy of research. In this field, in addition to reducing the memory consumption of LLM, the specific needs of tasks must also be considered. Local environments can be complex due to the diversity of users, and LLM inference systems need to allocate computational resources efficiently to meet the requirements of different requests in scenarios with multiple task types.

2.2 Foundations of Efficient Inference

The advanced performance of LLMs comes at a substantial computational cost. In practical applications, the inference process, which involves making predictions based on new input data, is particularly critical because it directly impacts the model's response time and operational costs. Consequently, accelerating model inference through parallel computing at both the hardware and software levels is crucial. Therefore, before introducing methods for efficient inference of LLMs, this section will first provide the necessary background knowledge on parallel computing models. Efficient inference relies heavily on the deep optimization of both hardware and software, making these factors crucial to improve the efficiency of model inference. Therefore, in the literature review, we will specifically address this aspect to lay the groundwork for subsequent discussions on efficient inference methods.

2.2.1 Parallelism and Resource Partitioning

LLMs typically handle parameters and inputs in the form of matrices. Since there is generally no dependency between the numbers within the matrix, parallel computation can significantly optimize deep learning, improving both training and inference speeds. The matrix representation of input data and model parameters offers four intuitive methods for parallel processing: **data parallelism**, **grouped parameter parallelism**, **model parallelism**, and **pipeline parallelism**. These parallel methods partition the resources required by the model, reducing the computational resources needed for each part. For machines with limited resources, similar partitioning strategies are also essential to ensure efficient resource utilization and inference performance.

Data parallelism [Li+20; Ren+21; Sti19] is a form of parallel computing in which a data set is split into smaller sections, each piece being processed simultaneously across multiple processing units. This approach is particularly effective when the same computation needs to be applied to large volumes of data, allowing for significant reductions in processing time. Unlike traditional CPUs, which are optimized for serial processing and general-purpose tasks, model processors, such as GPUs, are designed to support a large number of parallel computing operations. These processors excel at handling tasks that involve repetitive computations over large datasets, such as those encountered in machine learning and data analysis. The ability to perform many calculations concurrently makes data parallelism a powerful technique for speeding up training and inference in LLMs, particularly in fields like deep learning and high-performance computing.

Grouped Parameter Parallelism[Raj+20] represents a specialized form of data parallelism that partitions optimizer states, parameter gradients, and model parameters across different processors to conserve GPU memory when training LLMs. The primary advantage of grouped parameter parallelism is its ability to significantly reduce memory consumption while increasing overall training throughput by expanding the number of data parallel paths.

Furthermore, the parallelism technique that combines "intra-group parameter slicing and inter-group data parallelism" allows for a more efficient allocation of communication bandwidth both within and between machines, thereby further improving training performance.

Model parallelism [Li+23; Nar+21a] is a parallel computing strategy that involves dividing a LLM into multiple smaller segments, where each segment is processed independently on separate processing units. This approach is particularly useful when the model is too large to fit into the memory of a single processing unit or when different parts of the model require distinct computational resources. Model parallelism allows for the efficient distribution of computational workloads across multiple processors, enabling the training and inference of models that exceed the memory or processing power of a single machine.

Pipeline Parallelism [Nar+19; Gua+23; Nar+21b] assigns different layers of a neural network to separate processors, allowing point-to-point communication of data dependencies between upstream and downstream executors. Efficient pipeline parallel scheduling strategies based on this approach support high-performance scheduling algorithms such as 1F1B[Gua+23]. Furthermore, by overlapping communication with computation, these strategies effectively conceal communication latency, thus enhancing overall training efficiency.

This technique is commonly used in deep learning, where large neural networks, especially those with millions or billions of parameters, can be partitioned and distributed across multiple GPUs or other accelerators. Using model parallelism, more complex models can be trained, and computationally intensive tasks that would otherwise be impractical on a single processor can be tackled. Within resource-limited settings, it is advisable to adopt computational partitioning techniques that facilitate linear execution, effectively exchanging temporal resources for spatial ones to ensure the feasibility of inference tasks. Nonetheless, the precise methodologies for such partitioning remain a subject worthy of rigorous research and academic deliberation.

2.2.2 Hardware Accelerator

Common model accelerators differ significantly in design and functionality from traditional CPUs which focus on low-latency complex computation, mainly because they are specifically engineered to support large-scale parallel processing and complex data operations. These accelerators optimize certain types of computation, such as those used in deep learning and machine learning algorithms, thus significantly enhancing computational speed and efficiency. For tasks that involve handling large datasets and complex models, this type of hardware is particularly crucial. In the realm of model computation, especially in artificial intelligence, commonly used specialized hardware includes Graphics Processing Units (GPUs), which focus on parallel processing, Neural Processing Units (NPUs) focus on processing nerual network operation with energy saving, and Tensor Processing Units (TPUs) [Jou+17] developed by Google which focus on server side computation. GPUs are utilized as accelerators in this work, and we will analyze each type of accelerator and provide an explanation for their selection later.

GPUs: Originally designed for rendering graphics, GPUs are highly effective in handling parallel computational tasks required for deep learning due to their architecture, which includes thousands of cores. This makes them particularly well-suited for matrix and vector operations. Nvidia's Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model that provides developers with interfaces to harness the computational power of GPUs for tasks such as matrix multiplication. The use of GPUs and CUDA has become the de facto standard for accelerating deep learning tasks. This widespread adoption is in part due to the relatively robust and open community that CUDA developers have established. Not only do popular deep learning frameworks such as PyTorch and TensorFlow support CUDA, which provides developers with straightforward interfaces for constructing, training, and inferring models, but CUDA also offers official deep learning acceleration libraries such as cuDNN [Che+14], cuBLAS [KKK22], and CUTLASS [Tha+23]. In particular, CUTLASS is fully open source, facilitating its use in both academic research and commercial applications. On the other hand, GPUs are widely available and supported by a comprehensive supply chain. As the field of deep learning continues to evolve, GPUs and

CUDA are expected to remain integral components of the deep learning ecosystem, driving future breakthroughs in the field.

NPUs: NPUs are microprocessors specifically engineered to perform the complex computations required by deep learning and machine learning algorithms. These units are often embedded in endpoint devices such as mobile devices, smart home devices, and autonomous vehicles, enabling rapid and efficient data processing and inference capabilities. NPUs provide hardware-level acceleration, particularly optimizing operations like matrix multiplication, and manage power consumption more effectively than traditional GPUs when executing specific types of machine learning algorithm. Often designed as part of a system-on-chip (SoC), NPUs are integrated with other processing units, such as CPUs and GPUs, enhancing the rate and efficiency of data processing while reducing communication latency. In recent years, AI PCs have often come equipped with specialized hardware to enhance their abilities in handling AI-related tasks.

TPUs: TPU is a specialized Application-Specific Integrated Circuit (ASIC) developed by Google, designed to accelerate tensor computations in machine learning applications. The primary objective of TPU is to enhance the efficiency and speed of deep learning computations, particularly within large-scale data center environments, where they support the training and inference of complex machine learning models. TPUs feature a highly parallel architecture that enables them to process large volumes of data simultaneously, significantly boosting processing speed and efficiency. Offering performance that surpasses conventional CPUs and GPUs, TPUs excel particularly in training and inference tasks for deep learning models, achieving faster data processing speeds. Compared to other types of processors, TPUs offer a more favorable ratio of energy consumption to computational output, which is especially crucial for running LLMs. Unlike NPUs, TPUs are generally used as infrastructure for cloud platforms rather than being deployed on edge devices. It is not commercial grade hardware. TPUs are engineered to handle large-scale tensor operations with high throughput, which makes it crucial to structure deep learning models and training processes to leverage these capabilities. This often involves batching multiple samples and optimizing data pipelines to ensure that TPUs continually receive data, preventing idle times, and maximizing their

utilization. Alpha Go [Sil+16] is an example of using TPU to rapidly train the model to play chess.

This paper will primarily focus on model acceleration technologies related to GPUs and CUDA. Compared to other mainstream accelerators, CUDA not only supports intuitive programming but also allows development using C/C++ languages. Furthermore, through the NVIDIA ecosystem, it provides efficient development tools and libraries, lowering the development barrier. CUDA is supported by a wide range of hardware, including all NVIDIA GPUs, including consumer-grade. These GPUs, based on various architectures such as Fermi, Kepler, Maxwell, Pascal, Volta, Turing, Ampere, and Ada Lovelace, can all be programmed using CUDA. CUDA is well-suited for large-scale parallel computing tasks, particularly in the field of deep learning. Although other accelerators may have advantages in specific areas, such as custom requirements or low-power inference, CUDA excels in terms of development convenience and versatility.

2.3 Architecture of LLMs

LLMs are voluminous models based on deep learning algorithms that can understand and generate human language. LLMs are trained on vast amounts of textual data to learn the structure and semantics of language, enabling them to perform various language tasks. The architecture of LLMs is largely similar, typically involving transformers and Multi-Layer Perceptrons (MLP) for inference. This section will discuss the structure of LLMs. Since LLMs generally consist of transformers, we will first introduce the structure and principles of transformers. Next, we will discuss the common architecture of LLMs and the well-known examples of such models. In the latter half of this section, we will explore the fine-tune of LLM and how it affects the structure of LLM.

2.3.1 Basic Transformer

Google introduced a new deep learning model specifically designed for NLP in 2017, known as the Transformer [Vas+17]. Compared to models based on Convolutional Neural Networks

(CNNs), the Transformer demonstrated superior performance in terms of convergence accuracy in predictions. It showed a strong ability to understand the global context that enables the LLM to achieve remarkable precision.

Fig. 2.1 illustrates the transformer structure. A transformer is made up of two main components: the decoder in the right part and the encoder in the left part. In the encoder, N identically structured layers are stacked, each consisting of a multi-head attention layer and a fully connected layer. Additionally, each sub-layer is surrounded by residual connections and normalization layers. The decoder also features a similar design, consisting of N layers that are structurally identical. However, each layer in the decoder has three main sub-layers, arranged in sequence: a masked multi-head attention layer, a multi-head attention layer, and a fully connected layer. Similarly to the encoder, these three sub-layers in the decoder are also surrounded by residual connections and normalization layers. The transformer model requires substantial computational power but delivers powerful performance.

The self-attention mechanism endows the transformer with a powerful ability to understand the global context. This mechanism allows the model to attend to all positions within the sequence simultaneously when processing sequential data, which is computationally expensive, especially with longer sequences. Fig. 2.2 shows the structure of self-attention. In the sub-attention module of a transformer, there are initially three linear layers. Each input element passes through these three linear layers and is transformed into three distinct vectors: Query (Q), Key (K) and Value (V). The model computes the dot products of the Q vectors with all K vectors to measure their similarity. These dot product scores are then scaled down by dividing by the square root of the dimension of the K vectors to prevent overly large scores. Following this, a Softmax function is applied to transform these scores into weights. These weights reflect the importance of each element in the input relative to the current element, effectively determining how much attention each element should receive. Finally, these weights are used to perform a weighted sum of the V vectors, generating the final output vector. This output contains information from the entire input sequence, effectively aggregating the most relevant details determined by the attention mechanism to produce the context-rich representation needed for subsequent processing steps in the model.

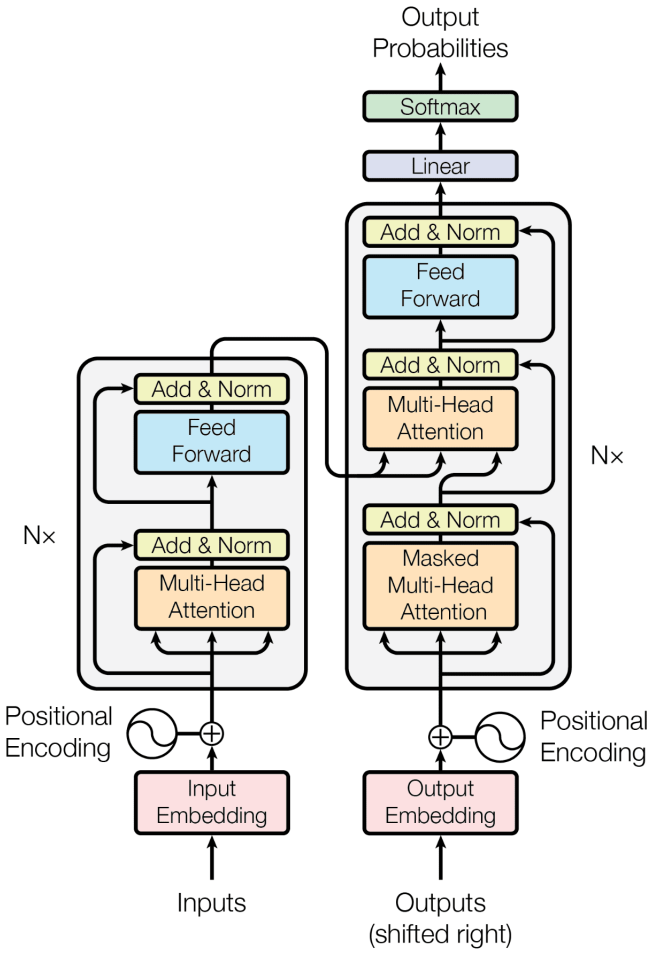

FIGURE 2.1: Transformer Architecture [Vas+17]

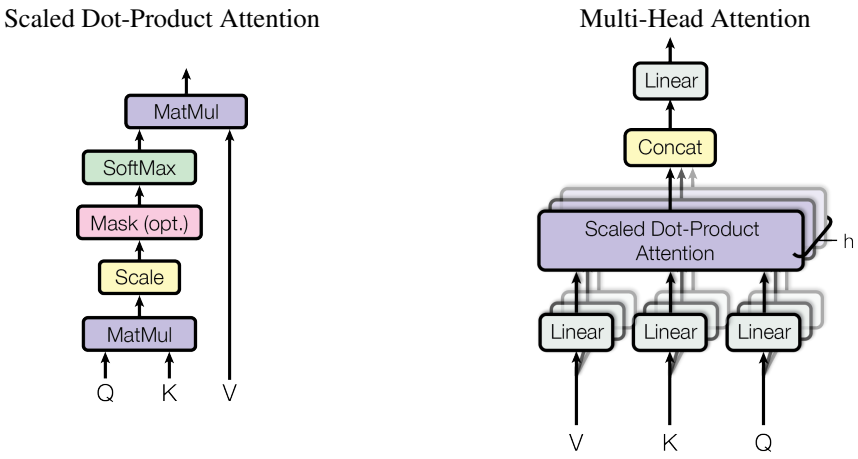

FIGURE 2.2: (left) Self-attention. (right) Multi-Head Attention consists of several attention layers running in parallel. [Vas+17]

In the transformer architecture, multi-head attention is employed, which is an extension of self-attention that allows the model to learn information in parallel across different representational subspaces. This allows the model to diversely interpret information from different representational spaces at different positions, enhancing the ability to capture various aspects of the input. In multi-head attention, each input token generates multiple sets of Q, K, and V tuples. For each set of QKV, the model performs self-attention calculations in parallel. After completing the attention calculations for all heads, the output vectors produced by each head are concatenated into a long vector. This long vector then passes through another linear transformation layer, which integrates the diverse information from the multiple heads to generate the final output vector. This step is crucial in multi-head self-attention because it allows the model to utilize information from multiple representational subspaces, thereby achieving a more comprehensive understanding of the input data.

2.3.2 The structure of LLMs

The success of the Transformer has led to the development of a series of LLMs based on its architecture, such as GPT [Ope+24], BERT [Dev+19], Llama [Tou+23a] and OPT [Zha+22a]. These models have set new performance benchmarks across multiple natural language processing tasks. The powerful capabilities of these LLMs have not only improved applications such as machine translation, text generation, and semantic understanding, but also propelled artificial intelligence technology into broader application areas, sparking a new wave of technological transformation.

Google proposed BERT [Dev+19] in 2018. BERT utilizes a bidirectional transformer encoder structure, employing a pre-training approach to learn a universal representation of language that enhances its understanding of language context. Its pre-training tasks include: 1) Randomly masking words in the input and then predicting these words. 2) Predicting whether one sentence follows another in a text. Through a strategy of pretraining followed by fine-tuning, BERT achieves unprecedented performance across various NLP tasks, particularly in the area of semantic understanding.

OPEN AI has focused on harnessing the capabilities of the transformer decoder, leading to the development of the GPT model series [RN18]. GPT operates with a unidirectional Transformer to pre-train a language model, emphasizing the generation of text. The GPT series has continuously improved model performance by progressively increasing both model size and data scale. During its pre-training phase, GPT employs unsupervised learning by predicting the subsequent words following a given text. For specific tasks, GPT requires additional training to fine-tune the model, adapting it to particular application scenarios. The GPT model excels in natural language generation and is capable of producing coherent, fluent, and contextually relevant text. As the GPT model has evolved, subsequent versions, such as GPT-2, GPT-3, GPT-4 [Rad+19; Ope+24] have expanded the original design primarily through increases in model size and the extension of training data. These upgraded models have further enhanced the quality and accuracy of the generated text through larger parameter counts and more sophisticated training strategies.

Some institutions have also provided the research community with models that offer efficient transparency and greater openness. OPT [Zha+22a] is an open source project published by Meta in 2022, with the aim of providing an open source and cost-effective LLM. It use the tranformer decoder struture as well. A notable feature of the OPT model is its fully open-source strategy. Meta has not only made the model's weights and training code public but also disclosed the processes for handling and selecting training data, which represents a significant step in terms of the model's transparency and reproducibility. This project offers multiple versions ranging from 125M to 175B parameters, covering needs from small to very large scales. OPT was designed with cost-efficiency in mind, making it suitable for resource-constrained environments. Its purpose is to maintain performance comparable to that of GPT-3 while reducing the costs of training and deployment.

In 2023, Meta released the LLaMA model [Tou+23b], which provides several different size versions ranging from 7B to 65B parameters, focusing on efficiency and accessibility under various computing resources. The aim is to maintain high performance on smaller model scales by optimizing the transformer decoder architecture and the training process. LLaMA utilized a large amount of training data, but Meta emphasized particularly the processing

and selection of data to enhance the quality and efficiency of the model. LLaMA offers extensive access to its parameters and configurations, enabling the research community to further explore and develop these models.

2.3.3 Fine-tune of LLM

For specific downstream tasks, LLMs often require fine-tuning to achieve optimal performance. Fine-tuning methods vary in their approach, with some techniques embedding additional layers or modules into the LLM, thereby modifying its architecture to better suit the task at hand. These methods enable the model to adapt to specific domains or problem types while leveraging the pretrained knowledge embedded in the model. Common fine-tuning strategies include the following approaches:

2.3.3.1 General Fine-tuning

Full Model Fine-tuning [Ope+24] refers to the process of adjusting the weights of all layers in an LLM to optimize its performance for a specific task or domain. This method allows the model to learn more task-specific patterns and nuances, leading to significant improvements in accuracy, robustness, and relevance. However, this fine-tuning approach often requires substantial computational resources, including both processing power and memory, and may result in longer training times. Despite these resource demands, the increased performance justifies its use in high-stakes applications where precision and adaptability are critical.

Partial Layer Fine-tuning [Liu+23] involves adjusting only the top layers or a subset of layers throughout the model, rather than fine-tuning the entire network. This method capitalizes on the fact that the lower layers of a model tend to learn general, task-agnostic features, such as basic patterns and representations, which are useful across various domains. In contrast, the higher layers are responsible for capturing more specific, task-oriented features that are critical for performance in specialized applications. By fine-tuning only the top layers, this approach reduces computational costs and training time, while maintaining a high degree of task-specific adaptability. Additionally, it allows the model to retain the generalization

ability provided by the lower layers, making it an efficient strategy for transfer learning in scenarios where labeled data for the target task is limited.

Prompt Tuning [Liu+22] involves training a set of task-specific prompt parameters while keeping the underlying pretrained model fixed. This technique leverages the model's preexisting knowledge and capabilities by optimizing a set of carefully crafted input prompts to guide the model's response toward a desired output. The core idea is to frame the input data in a manner that aligns with the task-specific requirements, allowing the model to generate more relevant and accurate predictions without the need for extensive retraining. Prompt tuning offers a more efficient alternative to full-model fine-tuning, as it significantly reduces computational demands by focusing only on the modification of input representations. Furthermore, it is particularly useful in scenarios where domain-specific data is sparse or when adapting a model to a new task with minimal resource expenditure.

Adapter Tuning [Hu+22; Hu+23] involves inserting a set of lightweight, trainable adapters into the pre-trained model, with the parameter updates being confined solely to these adapters. This approach enables efficient fine-tuning for a specific task by adjusting only a small subset of parameters, rather than retraining the entire model. Adapters function as modular components that adapt the behavior of the model to the task at hand, while the weights of the base model remain fixed, preserving the generalizability of the model in all tasks. Adapter tuning is particularly advantageous in resource-constrained environments, as it reduces both computational costs and memory requirements, making it a scalable solution for multi-task learning scenarios. Additionally, this approach facilitates faster adaptation to new tasks with minimal data, offering a promising strategy for efficiently deploying LLMs in diverse real-world applications.

2.3.3.2 Adapter Fine-tuning Structure

Adapter tuning requires the embedding of additional adapter layers in the LLM, altering its structure. Each adapter typically consists of one or a few linear layers. Depending on the type of adapter, these layers are embedded at different points within the model. Common types of

adapters include prefix tuning, Low-Rank Adaptation (LoRA), series adapter, and parallel adapter. Fig. 2.3 shows how different adapters affect the LLM structure.

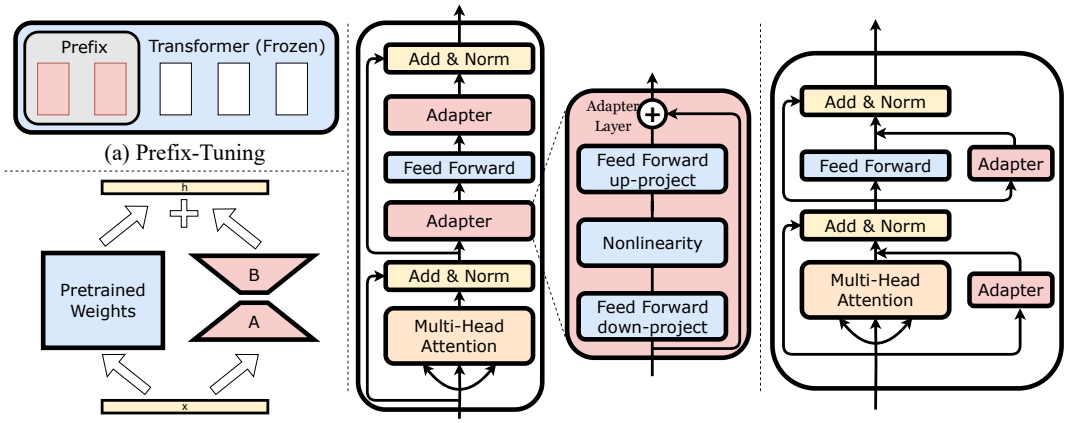

FIGURE 2.3: Architecture of Adapters [Hu+23]

Prefix-Tuning [LL21] is a lightweight model adjustment method that adds a series of fixed, trainable vectors to the model's input prefix. This approach allows the model to adjust to new contexts or tasks by altering the initial conditions of the model's processing sequence, effectively steering its behavior without requiring fine-tuning of the entire model. By leveraging these trainable prefixes, Prefix-Tuning achieves task-specific adaptation with significantly reduced computational and storage costs compared to full-model fine-tuning, making it an efficient choice for resource-constrained environments.

LoRA [Hu+22] modifies the existing weights of a model by introducing low-rank matrices into the model's weight matrix. Specifically, this technique achieves its goal by decomposing the weight matrix into the product of two smaller matrices, effectively reducing the number of trainable parameters. Using this decomposition, LoRA enables efficient fine-tuning of LLMs, significantly lowering computational and storage requirements while maintaining performance. This approach allows the model to adapt rapidly to new tasks with minimal resource overhead, which makes it particularly advantageous in resource-constrained or rapid prototyping scenarios. Additionally, the low-rank adaptation preserves the pretrained knowledge of the original model, ensuring that fine-tuning introduces only task-specific adjustments without compromising generalization.

Series adapter [Hou+19] involves inserting adapter modules sequentially between each layer of the model. These adapters are typically compact neural networks, such as one or two feed-forward layers, designed to adjust the flow of information through the layers. By serving as task-specific components, the adapters enable the model to specialize for particular tasks without modifying the original model parameters. This modularity ensures efficient parameter sharing and fine-tuning across tasks, reducing the need for retraining the entire model. Additionally, series adapters can be customized to capture task-specific nuances while leveraging the general knowledge embedded in the pre-trained model, thus improving both task adaptation and computational efficiency.

Parallel Adapter [He+21] shares similarities with the Series Adapter, but differs in its architectural placement of adapter modules. Instead of inserting the adapters sequentially between layers, they are placed in parallel with each layer of the model. These parallel adapters process the input data alongside the original layer and merge their output with the layer's output to collectively determine the input for the next layer. This parallel arrangement facilitates the integration of task-specific features while preserving the operational flow of the original model. By minimizing disruption to the model's architecture, parallel adapters offer a flexible and efficient mechanism to augment the model's capabilities, enabling adaptation to new tasks with minimal retraining or performance compromise.

2.4 Optimization for LLM inference

LLM inference techniques aim to enhance the performance of models during actual deployment through various optimization methods. These techniques focus on reducing latency, minimizing memory usage, and lowering energy consumption, among other aspects. In this chapter, we will explore several strategies to optimize the inference performance of LLMs. We begin with kernel-level optimizations, which involve fine-tuning the low-level computational operations to enhance efficiency. Next, we discuss memory management for LLM systems, focusing on techniques to optimize memory allocation and usage to handle extensive data and model parameters efficiently. Finally, we will cover system-level optimizations for batch

processing, which aim to improve throughput by effectively managing the processing of multiple data instances simultaneously.

2.4.1 Kernel Level Optimization

In deep learning, a "kernel" refers to a small parallel program that is executed on a GPU, and it is considered the smallest unit of computation in deep learning frameworks. Kernels are designed to perform specific computational tasks, such as matrix multiplications, convolutions, or activation functions, which are fundamental operations in neural network models. By leveraging the GPU's highly parallel architecture, which consists of thousands of cores capable of executing tasks simultaneously, kernels can significantly accelerate the computations involved in training and inference. This parallelism enables deep learning models to handle vast amounts of data and complex computations in a fraction of the latency that it would take on a traditional CPU. The efficient execution of these kernels is crucial for optimizing both training times and the deployment of models, making GPU-accelerated kernels a key component in the performance of modern deep learning systems.

2.4.1.1 Common Kernel

Common kernel programming techniques include CUDA, Triton, and OpenCL. For NVIDIA graphics cards, CUDA is often considered the optimal choice. Since CUDA is directly developed and supported by NVIDIA, it typically has the first access to the latest features and optimizations of NVIDIA GPUs. Additionally, most mainstream deep learning frameworks, such as TensorFlow and PyTorch, provide support for CUDA, allowing developers to efficiently harness the computational power of NVIDIA GPUs for accelerated model training and inference.

In LLM inference, the foundational kernels are matrix multiplication, matrix addition, and normalization kernels. These kernels alone are sufficient for performing inference computations in LLMs. Given their ubiquity in deep learning, NVIDIA has developed specialized libraries such as cuBLAS and cuDNN to optimize these kernels through hardware parallel

processing capabilities. However, because these kernels are relatively simple, they must be repeatedly called in each layer of the model, each call introducing memory reads and writes, as well as setup and initialization costs. As LLMs have consistent layer structures, a common optimization strategy is to combine multiple basic operations into a single operation. This technique, known as kernel fusion, reduces execution time and resource consumption by minimizing the overhead associated with multiple individual operations.

2.4.1.2 Kernel Fusion

Some systems have implemented operator fusion specifically for general models, enhancing efficiency and performance. TVM [Che+18] is a deep learning compiler that optimizes the execution of the model by combining common kernels during the compilation process. Apollo [Zha+22b] classifies kernels and performs fusion according to these classifications. By grouping similar operations, Apollo can reduce computational complexity and improve model execution efficiency. In CUTLASS [Tha+23], the grouped GEMM (General Matrix Multiply) functionality allows multiple independent matrix multiplication operations to be performed within the same kernels.

Some work targets kernel fusion specifically for LLM inference. In FlashAttention [Dao+24], a fused multihead attention kernel is designed for direct use in LLM inference, encapsulating the entire attention mechanism within a single operator to enhance inference efficiency. A key innovation is the introduction of blockwise computation for the softmax operation. In the blocked softmax operator, each thread block computes the softmax of a sub-block of input data and records its maximum value. After computing the softmax for each sub-block, the results are merged into the overall softmax using the maximum values from the sub-blocks. Mathematically, this approach is equivalent to safe softmax, ensuring numerical stability while using parallelism for improved computational efficiency. In practical use, the blocked softmax operator may introduce errors as a result of the increase in the number of computations. These errors typically arise from numerical precision limitations during intermediate steps, such as summing exponentials.

TurboTransformer [Fan+21] is an efficient inference framework designed to optimize the performance of transformer encoders. This is achieved by decomposing all kernels within the encoder into a directed acyclic graph (DAG) representation, which reveals the dependency relationships among operators. This DAG representation not only provides a precise depiction of the computational flow but also serves as a structured foundation for further optimization. Building on this, TurboTransformer combines multiple adjacent operators into a more efficient composite operator, reducing both data transfer and computation overhead. This optimization strategy significantly improves the operational efficiency of transformer encoders and reduces inference latency, outperforming traditional methods that execute operators independently.

2.4.1.3 Customized Kernel

To enhance functionality, some systems have designed custom kernels. The authors of ByteTransformer [Zha+23] identified that in traditional LLM inference, requests within a batch need to be padded to the same length, with padding zeros that carry no semantic value, leading to computational waste. In response, they developed a custom kernel that supports concatenation of request attention, tailoring kernels specifically for updating request information. This innovation addresses the inefficiency by directly processing varying-length requests without the need for padding, optimizing both processing time and resource use.

In the field of multitask LLM inference, the authors of Punica [Che+24] observed that when a system handles multiple downstream tasks simultaneously, multiple adapters cannot be executed in parallel. To address this, they introduced an operator based on grouped GEMM, specifically designed for parallel computation of multiple adapters. Two different kernels are used for the shrink operator and the expand operator to achieve the best efficiency. Similarly, in s-lora [She+23b], a design similar to punica kernels is proposed, but extends the concept with SGMV (Shape-agnostic GEMM for Multiple Vectors), which is not constrained by matrix shapes. This further enhances the practicality of parallel computation for adapters, improving efficiency and adaptability in processing multiple tasks simultaneously.

Even with identical computations, using different kernels for various shapes of input matrices can optimize the use of a GPU's parallel capabilities. In LLM inference, since each inference

step typically generates only one new token, multiple inference steps are required to complete a task. Starting from the second round of inference, the input matrix is adjusted to include only the most recently generated token, effectively reducing the computational load. The initial round of inference is known as the prefilling stage, while the subsequent round is referred to as the decoding stage. These stages employ different attention kernels due to the different lengths of input QKV to maximize the utilization of the GPU's capabilities. In the prefilling phase, the shape of the Q, K, and V matrices is (n, h, hd), where n denotes the length of the request, h is the number of heads, and hd is the dimension of each head. During the decoding phase, the length of Q is reduced to (1, h, hd), while the shapes of K and V remain at (n, h, hd). Consequently, for the matrix multiplication involved in the softmax(q*k)*v calculation, different strategies must be employed to allocate tasks to each GPU thread. This ensures that the workload across all threads is balanced and that more threads are utilized in parallel to maximize GPU performance.

2.4.2 Memomry Management for LLM inference

As LLMs become increasingly prevalent, specialized systems dedicated to LLM inference have emerged. These systems are specifically designed to improve the efficiency of the LLM inference process, memory management being one of the most critical components. The key elements of an LLM inference service system often include sophisticated resource allocation strategies that optimize memory usage. By focusing on effective memory management, these systems not only speed up the inference process but also reduce operational costs while ensuring high performance. As technology advances and the demand for LLM applications grows, these systems are expected to continuously evolve, become more stable, and provide reliable services to an expanding user base.

The memory consumption required by LLMs is substantial. We analyze the memory consumption of LLMs with a focus on the pressure exerted on memory allocation. During LLM inference, the most memory-consuming portions include model parameters, the KV cache, and parameters for downstream tasks. The model parameter refers to the weights within the model's backbone, as previously mentioned. These parameters are fundamental to the model

and are trained to capture general features and relationships from large data sets. However, the "parameter for downstream task" refers to the weights required for adapters specifically designed for particular downstream tasks. KV cache is a unique feature of memory consumption in LLM inference, stemming from a widely used method to accelerate LLM inference. In LLMs, KV cache refers to the storage of precomputed key value pairs that are derived from the model's attention mechanisms.

Since the model typically generates only one new token per inference step, multiple rounds of inference are required to complete a full request. By caching these key-value pairs, the model can quickly retrieve previous information in each round of inference, thereby speeding up the inference process and enhancing efficiency. This approach significantly reduces the need for repetitive computation and conserves computational resources. However, it also means that a substantial amount of memory is needed to continuously store these key-value pairs, particularly when handling complex or lengthy dialogues, which places a more significant strain on memory resources.

The parameters of the model backbone are associated with its configuration and structure. In models with the OPT architecture [Zha+22a], OPT-6.7b will use approximately 12GB, and OPT-30b backbone parameters consume over 55 GB. In OPT-30b, 1000 tokens require approximately 1.28GB to store KV. As the number of requests increases and model generation progresses, the number of tokens can accumulate rapidly. When handling multiple concurrent requests or generating large volumes of text, memory consumption can increase significantly. When performing various downstream tasks using the Parameter Efficiency Transformers (PET) model, each adapter occupies a small amount of space, but the application can involve many adapters, which can collectively consume a significant amount of memory. The memory capacity of commercial GPUs typically ranges from 6GB to 48GB [NVI24], which can be quite limiting for LLM memory demands. It will trigger Out-Of-Memory (OOM) errors.

Popular deep learning frameworks such as PyTorch [Ans+24], TensorFlow [Mar+15], and Keras [Cho15] come with built-in memory management designs. When using these frameworks, users can focus solely on the architecture and construction of the network without having to handle the complexities of memory management. This embedded functionality

simplifies the development process, allowing practitioners to concentrate on optimizing model performance and experimenting with different configurations rather than managing low-level details such as memory allocation and optimization.

Generic LLM inference systems, such as ORCA [Yu+22], TurboTransformers [Fan+21], Hugging Face Accelerate [Gug+22], ByteTransformer [Zha+23], use the memory management interface provided by PyTorch [Ans+24] or Cuda, directly allocating all the space required in the GPU for computations. They have only considered environments with abundant resources, but consumer-grade AI PC lack the computational resources necessary to utilize such systems. Although frameworks like PyTorch, TensorFlow, and Keras offer general memory management techniques to accommodate a wide range of users, more specific memory management methods may be more advantageous for LLM inference.

2.4.2.1 General Memory Management

Due to the vast size of LLMs, their inference is often distributed across multiple accelerators or machines. The concepts of data parallelism and model parallelism discussed earlier are directly relevant to this practice. Most systems designed for LLM inference need to consider how to distribute the model's weights and computations across multiple devices to optimize performance. For example, tools such as DeepSpeed [Ami+22], Hugging Face Accelerate [Gug+22], and s-lora [She+23b] pay attention to handle these challenges.

To address the LLM memory consumption problem, numerous approaches have been proposed to enhance memory efficiency during LLM inference. Quantization [Ega+24; Li+24b; Liu+24b; She+23a] has emerged as a particularly effective strategy in this domain. Recent studies demonstrate that model parameters can be quantized to 2-4 bits with tolerable levels of error [Liu+24b; Ega+24], significantly reducing memory usage while maintaining acceptable performance. Despite these advancements, such optimizations often fall short in resource-constrained environments where memory and computational resources are severely limited.

2.4.2.2 Resource Limited Memory Management

The dominant strategy for LLM inference systems for resource-limited environments is offloading [Ami+22; Ren+21; EM23; Gug+22; Zha+24]. Based on this, Flexgen [She+23a] allocates more resources to support larger batches by storing fewer model parameters on the GPU, thus improving overall throughput. This system only explores the maximum possible throughput of the model, defaulting to all requests having the same length, meanwhile, in a realistic environment where the difference in the size and frequency of each request can be significant. However, its memory management system allocates the same amount of memory for key-value pairs (KV) for each request, which only leads to resource wastage.

Instead of maximizing batch size, HeteGen [Zha+24] focuses on balancing the distribution of workload between devices and device communication to achieve lower latency. This approach effectively supports high-latency inference scenarios when operating with a batch size of one. However, this approach faces significant limitations in maintaining high throughput, especially when dealing with a large number of offline tasks, due to the limited amount of work that can be shared by the CPU in the presence of a high computational volume.

2.4.2.3 KV Cache Management

Caching [Liu+24a; Gar+21] is a commonly used optimization method in machine learning. The memory required for the KV cache is directly affected by the batch size and sequence length and will grow with the length of the request during operation. Traditional inference systems pre-allocate a maximum memory length for each request to store the KV cache, leading to wasted internal space fragmentation [Pop+22].

Due to the varying request lengths and unpredictable output sizes in LLM inference, memory wastage can occur. Kwon et al. have proposed the structure of a Page KV cache to reduce memory wastage in the PagedAttention [Kwo+23]. The structure of the Paged KV cache is shown in Fig. 2.4. Such a design still has the following drawbacks in single GPU systems: KVs must be stored separately by tiers for loading, creating many discontinuous memory

fragments in the paged KV cache and preventing read-out bursts from Direct Memory Access (DMA) [NVI15], which negatively affects the efficiency of prefetching.

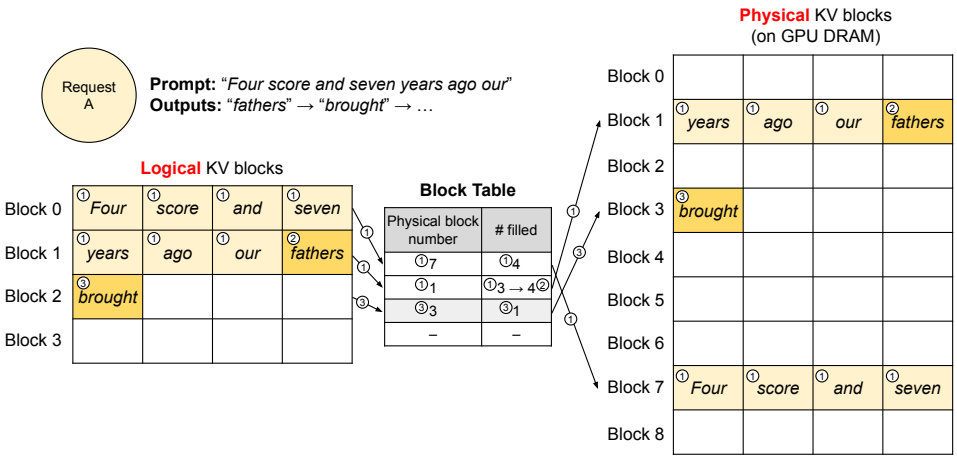

FIGURE 2.4: Architecture of Paged KV cache[Kwo+23]

PETs are a widely used method that allows the pre-trained model to adapt to downstream tasks by fine-tuning only a tiny subset of task-specific parameters [Din+23; Hu+23; Liu+22; BGR22]. The authors of S-LoRA realized that the KV cache has the same properties as PET parameters: the length is not fixed and varies with the selected request, and both have at least one identical dimension [She+23b]. Unified Paging is proposed in S-Lora, a scheme based on the paged KV cache, which limits the length of each page to 1 token to avoid waste and allows PETs parameters to share this space. This may result in more and smaller discontinuous fragments, the smallest of which is only one token long, affecting the offloading speed and making it difficult to offload KVs to CPU memory due to the significant discontinuity of the storage.

The KV caching system, by reserving the maximum length of memory for each request, leads to significant memory wastage. Although some existing solutions, such as PagedAttention, somewhat alleviate this issue, they also exacerbate memory fragmentation. For systems that require offloading, the memory management design should minimize memory wastage and store parameters in contiguous memory spaces as much as possible. This reduces the overhead and latency during prefetching, thereby enhancing system efficiency. The length and initiation time of online tasks are often unpredictable, which impacts the KV cache: 1)

In cases of limited memory resources, this unpredictability can frequently lead to shortages when allocating KV cache for new tasks. 2) Allocating memory for incoming online requests introduces additional latency, affecting the system's response speed and overall efficiency.

2.4.3 Batch Strategy for LLM inference

Exploring efficient batch processing strategies is one of the key factors in enhancing the performance of LLM inference. By adopting a batch processing approach, which primarily involves integrating individual inference requests into larger batches, several clear benefits emerge. Firstly, it reduces the overhead associated with resource allocation and scheduling between requests, as more tasks can be executed in parallel within the same timeframe. Secondly, batch operations can minimize idle time for GPUs, enhancing resource utilization efficiency. Lastly, by reducing the time required to process individual requests, the overall response time is optimized, which is particularly crucial for applications that need to handle large volumes of real-time data. An effective batching strategy should maximize resource utilization while ensuring that the quality of the model inference is maintained.

2.4.3.1 General Batch Strategy for LLM

Traditional attention kernels require that each request within a batch have the same length due to the limitation of the attention kernel, but in practical applications, the lengths of requests can vary. The standard approach is to pad shorter requests with zeros until they match the length of the longest request. While this padding is necessary for processing, it can lead to a wasteful use of computational resources as the operations performed on the padding do not contribute to the final output.

TurboTransformer [Fan+21] addresses this problem by optimizing the way requests are split, with the aim of minimizing waste while efficiently completing inference tasks. TurboTransformer improves its batching strategy by pre-recording the estimated latencies for potential batch combinations of requests. Then it employs dynamic programming techniques to determine the configuration that minimizes the total estimated latency when assembling requests

into different batches. The set of batches with the shortest projected latency is selected as the final solution.

ByteTransformer [Zha+23] addresses the issue of zero padding by concatenating requests end-to-end within the same batch. Using specialized kernel, it supports this configuration and handles different request lengths in parallel. These operators work by using additional parameters that include the length and starting position of each request, enabling the simultaneous computation of attention operations for requests of varying lengths within the same batch.

Although requests can be computed synchronously within the same batch, the completion time for these requests can vary significantly. Some requests may involve generating long articles, while others may require only a sentence or two, and the specific lengths are often unpredictable. Consequently, in a batch inference scenario, most requests might complete their inference quickly, but a few longer or more complex requests may continue to occupy resources. This disparity leads to resource inefficiency, as computational resources remain tied up, servicing a small number of requests while the majority have already finished, resulting in wasted capacity and potential delays.

Orcas [Yu+22] has identified inefficiencies in traditional inference processes and proposed a novel solution by shifting the granularity of resource management from the request level to the iteration level. In this approach, after each iteration of inference - where a new token or character is generated -Orcas updates the batch by releasing resources from requests that have been completed and reallocating those resources to other pending requests. This dynamic resource management strategy helps minimize idle time and resource wastage by ensuring that computational resources, such as memory and processing power, are continuously redistributed to active requests. By maintaining high resource utilization throughout the inference process, Orcas effectively enhances system throughput, reduces latency, and ensures that resources are always allocated to tasks that require them, improving the overall efficiency of the inference pipeline.

To more efficiently complete inference, systems such as TurboTransformers [Fan+21], ORCA [Yu+22], and ByteTransformer [Zha+23] process requests in the prefilling and decoding stages

separately, using operators optimized for each stage to achieve higher end-to-end speed and throughput. Kernels for mixing computation of the prefilling and decoding stages reduce the efficiency of LLM inference. For example, general attention from PyTorch [Ans+24] cannot efficiently utilize GPU. Hybrid computing by alternately using two operators specifically for prefilling and decoding to perform attention requires the invocation of more operators, thereby reducing efficiency.

2.4.3.2 Limited Resource Batch Strategy

However, there are particular batching strategies for systems with limited resources. FlexGen [She+23a] aims to process as many requests as possible, requiring frequent manual adjustments to prevent memory shortages. In contrast, HeteGen [Zha+24] is designed for the online environment. Processes only one request at a time. In some padding-free systems where memory concerns are addressed, they use a greedy algorithm to add requests to batches, reserve space for the KV cache of requests, and preemptively terminate requests when the reserved space is insufficient, such as vLLM [Kwo+23], S-lora [She+23b].

These systems are designed with the assumption that the workload consists solely of either offline or online tasks, lacking a batch processing strategy for a mix of online and offline tasks. In real-world scenarios, both offline and online tasks may need to be processed within the same system. In systems such as Flexgen [She+23a] and Deepspeed [Ami+22], online requests cannot be directly added to batches because new requests come into the pre-fill stage, and the batch in the current batch works in the decoding process. As a result, the system cannot provide quick and low-latency responses to online requests.

CHAPTER 3

System Overview

3.1 FixGen Architecture

To address the challenges outlined above, we propose an inference serving framework called FixGen. The primary objective of our design is to optimize the allocation and management of both memory and computational resources in a single GPU setup, ensuring maximum throughput while minimizing computational overhead. Using this streamlined approach, we aim to enhance the efficiency of the system, particularly in terms of resource utilization. Furthermore, our framework is designed to seamlessly integrate and optimize batch processing tasks, effectively balancing the demands of real-time (online) and non-real-time (offline) computations. This dual focus allows FixGen to handle a wide range of workloads with varying time sensitivities, enabling optimal performance across diverse applications without compromising responsiveness or throughput. Fig. 3.1 shows the overview of FixGen and its Fixpool memory system.

The system is divided into two main components, each of which addresses a different aspect of performance optimization. The first component focuses on memory design, where we have tailored memory usage specifically for single-card LLM inference. This optimization aims to maximize the efficiency of memory utilization, ensuring that available memory is used effectively without unnecessary waste. Additionally, we have incorporated advanced prefetching techniques to further enhance memory access performance, reducing the time spent waiting for data to be loaded and improving overall system throughput.

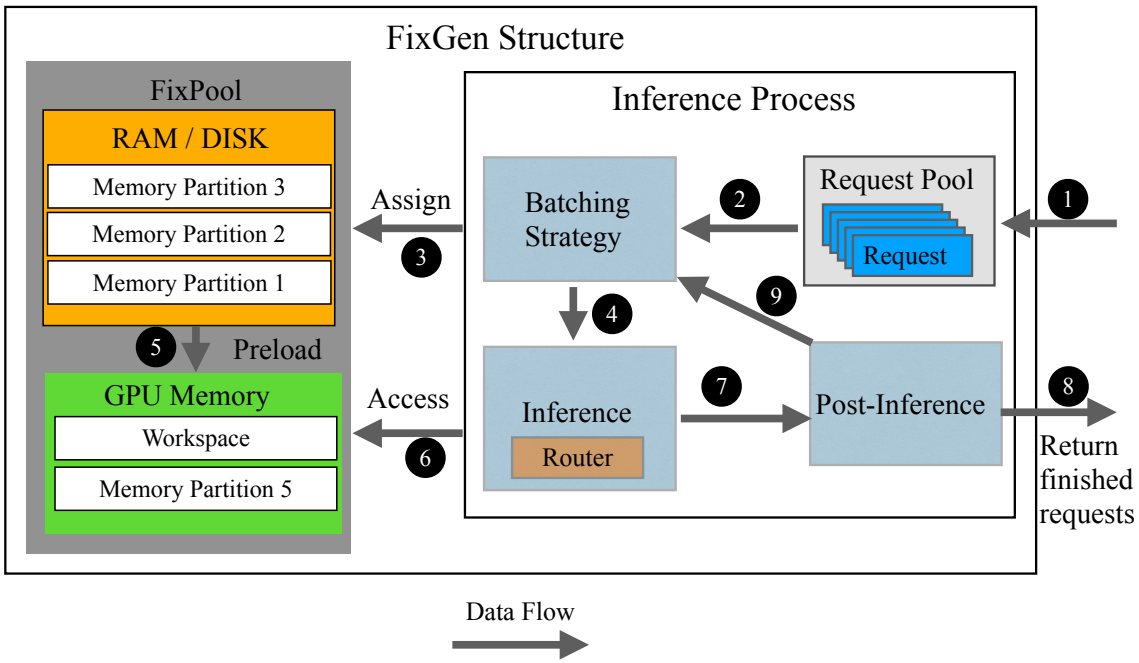

FIGURE 3.1: The Overview of the FixGen workflow: ❶ The user's online request is first stored temporarily in the request pool. ❷ The batch strategy prepares the batch and allocates the memory for the KV cache needed for the batch in advance. ❸ Assign memory to each request and prepare necessary parameters. ❹ The model computes the next token.❺ Preload parameter to the workspace. ❻ Access parameter for computation. ❼ Post-processing saves the newly generated token in the request. ❽ Return the completed request ❾. Recalculate the space required for the next round of computation and adjust the batch according to the required space.

The second component is dedicated to batch processing, where we have developed a specialized batching strategy and processing methods that are tailored to the unique characteristics of requests in PC environments. This batch processing approach is designed with an emphasis on minimizing latency, ensuring that the system can handle requests quickly and efficiently even under varying workloads. By optimizing the batch composition and processing pipeline, we aim to reduce overhead and maximize the responsiveness of the system, making it well-suited for both online real-time and non-real-time inference tasks.

3.1.1 Memory Management: FixPool

Figure. 3.1 (left) illustrates FixPool, the primary memory management module utilized by FixGen. In FixPool, the model's weights and KV cache are stored in segmented blocks within preallocated memory partitions across GPU memory, CPU memory, and disk. This hierarchical storage strategy carefully considers the performance characteristics of different storage devices with the aim of balancing storage efficiency and access speed.

Given the limited capacity of GPU memory, the weights and cache data required for computation are dynamically loaded from slower storage media (such as CPU memory or disk) to the GPU during runtime. This ensures efficient computation while optimizing memory usage and alleviating GPU memory constraints. By minimizing I/O wait times, this dynamic loading mechanism enhances overall performance.

The design and management of the memory structure are central to FixGen's efficient operation, which encompasses mechanisms such as memory partitioning, data prefetching, and dynamic migration. The details of these implementations will be thoroughly discussed in Chapter 4, enabling readers to better understand FixGen's innovative approaches to memory management.

3.1.2 Inference Process and Batch Strategy

The FixGen batch processing workflow is illustrated on the right side of Fig. 3.1. When the system receives user requests, they are collected into a request pool. The batch processing strategy module then determines which requests can be included in the next round of inference, based on the number and size of GPU memory blocks, to ensure efficient resource utilization.

Once a batch of requests is selected, they are passed to the inference module. The inference module runs the reasoning process of the LLM, generating the next token for each request. During this process, a specially designed KV cache update mechanism efficiently manages the context state of each request, ensuring continuity and accuracy of the inference.

After inference for the current token is completed, the generated results are passed to the post-inference module for detokenization. Requests that have completed token inference are marked as idle, and the memory blocks allocated to them are released. The batch processing strategy module dynamically reallocates these freed resources, prioritizing pending requests in the request pool, and incorporating them into the next round of inference.

This batch processing workflow in FixGen achieves smooth inference and optimized system throughput through efficient collaboration between resource allocation and dynamic loading. Further details of the batch processing mechanism and its design will be elaborated in Chapter 5.

3.2 Fixgen Inference Workflow

The FixGen workflow operates as follows:

❶ Request Pooling: When a user submits an online request, it is temporarily stored in a request pool. This allows multiple requests to be managed and processed in a coordinated manner, optimizing throughput.

❷ Batch Preparation and Memory Allocation: A batch strategy groups multiple requests together to maximize computational efficiency. During this step, the memory required for the Key-Value (KV) cache is pre-allocated based on the size and complexity of the batch, ensuring smooth processing without runtime memory bottlenecks.

❸ Memory Assignment and Parameter Preparation: The system assigns memory to each request and prepares the necessary parameters required to process the batch. This ensures that each request has the allocated resources for efficient processing.

❹ Token Generation: The model computes the next token for each request in the batch according to the parameters prepared. This step involves running forward passes through the model's architecture to generate output predictions.

❺ Preloading Parameters: The necessary parameters are preloaded into the workspace, ensuring that the system is ready for efficient and fast access during the computation of the next token.

❻ Accessing Parameters for Computation: During the computation step, the system accesses the preloaded parameters to perform the required computations efficiently for each request in the batch.

❼ Post-Processing: Once the model generates a token, this newly generated token is immediately stored and appended to the respective request within the pool. By doing so, the context of the request is dynamically updated, allowing for the incorporation of the most recent information in subsequent token predictions.

❽ Request Completion: Once all tokens for a request have been generated, the system marks the request as complete and returns the final output to the user.

❾ Batch Adjustment: After processing the current batch, the system recalculates the memory and computational space required for the next round of token generation. Based on this analysis, the batch is dynamically adjusted, either by merging new requests from the pool or splitting current requests to maintain efficient memory usage and computational performance.

This iterative process ensures that FixGen maintains high throughput by efficiently processing requests, while simultaneously adapting to varying request loads and model constraints. By continuously updating the context and optimizing resource allocation at each step, FixGen can dynamically respond to fluctuations in workload demands, whether in terms of computational complexity or memory requirements. This adaptability is key to maintaining performance under diverse conditions, allowing the system to handle both high-volume, low-latency requests and more resource-intensive tasks with minimal overhead.

In the implementation, we use `batch_info` to hold context information for the current batch during the inference process. We introduce the parameters contained in `batch_info` here and will not repeat the explanation in the following sections. The `batch_info` contains following data:

- `input_ids`: The concatenation of the requests in the batch.
- `total_length`: The total number of tokens to be processed in the batch.
- `batch_size`: The number of requests in the batch.
- `prefill_length`: The number of tokens that require prefill operation.
- `kv_length`: The length of KV stored for each request.
- `batch_index`: The starting position of each request in `input_ids`.
- `blocks_ids`: A list of indices for the memory blocks required by each request.
- `blocks_index`: The starting position of the required memory blocks for each request in `blocks_ids`.
- `next_token_ind`: The expected position of the new token generated for each request.
- `request_length`: A list of the lengths of each request.
- `token_batch_ind`: The corresponding request number for each token in `input_ids`.
- `prefill_index`: The starting position of each request in the prefilling stage.
- `decode_index`: The starting position of each request in the decoding stage.
- `max_input_len`: The length of the longest request.
- `pet_idx`: The index of the PET model that needs to be used.
- `pet_len`: The number of tokens to be processed by each PET model.

This data structure will be used to store all the contexts of a batch. In the process of model inference, $batch\_info$ serves as the flow data.

CHAPTER 4

Memory Management: FixPool

In the aforementioned discussion, we highlighted that in LLM inference involving offloading, the performance bottleneck is often the prefetch operation. In addition, memory management should be user-driven to minimize memory wastage. Therefore, our memory management design focuses on reducing the prefetch of offloading resources, fully utilizing I/O resources such as PCIe, and efficiently using the limited available memory. We integrate dynamic memory allocation strategies to adapt in real-time to the changing demands of inference tasks, enhancing system responsiveness and efficiency.

4.1 Memory Structure

We extended unified paging in S-LoRA and the blocked KV cache in vLLM to FixPool to unify the management of all parameters that need to be offloaded and prefetched [Kwo+23; She+23b]. In contrast to the aforementioned memory management approaches, our FixPool places emphasis on minimizing I/O latency through innovative memory management strategies. Prioritization of reducing the frequency of I/O operations or enhancing their efficiency takes precedence over maximizing the effective utilization of memory resources.

Fig. 4.1 shows the structure and parameter distribution in FixPool. We design FixPool that uses memory layers to manage a section of logically contiguous memory. Each memory layer stores the parameters required by a transformer layer. The memory size in the memory layer is precalculated and allocated, including a fixed length section for model weights and two reserved sections for PET (Parameter Efficiency Transformer) weights and the KV cache, which size varies with requests. Reserving more memory allows for a larger batch size,

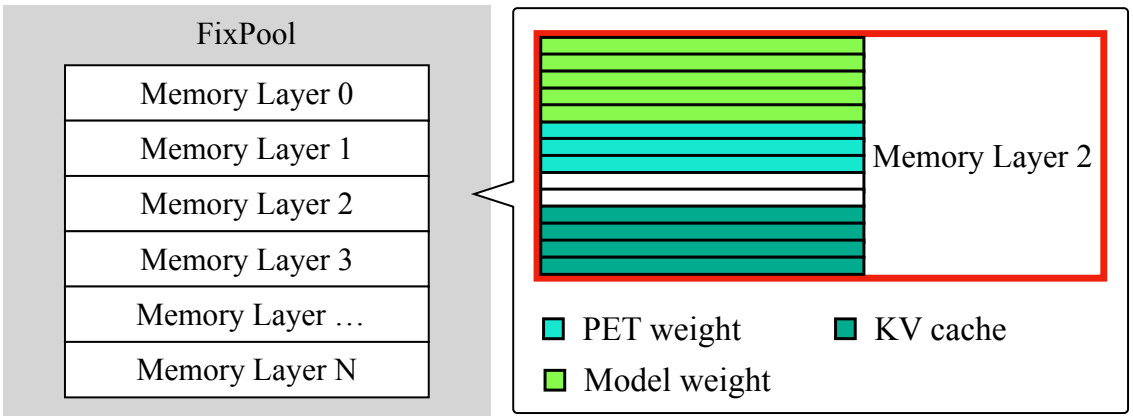

FIGURE 4.1: The Layer structure design of Fixpool and the parameter distribution for memory layers.

enhancing throughput. Pre-allocating these memory spaces avoids memory allocation during inference and prevents OOM errors during runtime.

The memory in each memory layer is actually stored in up to three proportionally divided continuous physical memory partitions located on the GPU, CPU memory, and disk, respectively. Users can adjust the proportions based on their hardware to maintain high GPU utilization. The memory layer is internally divided into fixed-size blocks, each of which can be assigned to parameters needed by the inference layer. When new requests require memory, assign free blocks to them to avoid reallocation of memory resources to reduce latency.

The code of the memory structure is shown below. The Fixpool contains the same number of memory layers as the hidden layers. It requires several parameters to construct the memory layer, including basic parameters such as `opt_config`, which represents the model's configuration and the expected size. A key parameter is `percent`, which represents the offload ratio of the memory layer. This is a list of three integers that represent the distribution ratio of memory blocks across GPU memory, CPU memory, and disk.

LISTING 4.1: Memory Layer

```
class Fixpool:
    def __init__(self, opt_config, block_num, block_size, percent
        =[100, 0, 0]):
```

```
        self.memory_layers = []
        for i in range(self.num_hidden_layers):
            self.memory_layers.append(
                MemoryLayer(opt_config, block_num * block_size,
                    percent, self.token_per_block, self.
                    working_spaces, layer_num = i)
            )
```

4.1.1 Parameter Distribution in Memory Layer

FixPool optimizes memory management performance by storing each type of parameter contiguously within memory layers. This design ensures that parameters requiring simultaneous prefetching are adjacent to each other in memory, reducing randomness and fragmentation of memory access. Consequently, it avoids additional overhead and access latency caused by memory fragmentation. This contiguous storage strategy significantly enhances the efficiency of batch prefetch operations, particularly in scenarios involving large-scale inference tasks or frequent memory swaps, effectively reducing I/O overhead. Through the refined management of memory partitioning and data layout, FixPool achieves higher storage utilization while providing reliable performance support for system operations.

During memory allocation, the memory occupied by all KV caches will be preferentially allocated to higher addresses, while PET will be allocated to lower addresses. This approach naturally separates the two and ensures that parameters of the same type are physically contiguous.

We conduct an analysis of the parameter prefetching process in our system, focusing on the memory consumption of various components during inference. The LLM utilizes three types of memory intensive parameters throughout the inference process: first, the model backbone weights, which have a fixed length and remain constant across different tasks. Second, the KV cache, whose length dynamically adjusts on the basis of the size and nature of incoming

requests. And third, the PET parameters, whose length varies depending on the specific task associated with the requests.

Prefetch usually occurs between inference layer switching and batch switching. Inference layer switching requires fetching all these three kinds of parameters, while batch switching requires fetching the corresponding KV cache only. Thus, the contiguous space for each layer is stored from higher to lower addresses, starting with the model parameters, followed by PET model parameters, and finally, the KV cache, to ensure the continuity of memory space for each part and as a whole, in order to maximize I/O efficiency (DMA burst reading) or to allow possible compression [NVI15].

The following code represents the block assigned to FixPool. It shows how the blocks are assigned to each PET and KV cache to maintain physical continuity and logically separate meanings.

LISTING 4.2: Assign memory blocks

```
def assign(self, rq):
    rq_block_need = self.block_need(rq)
    pet_block_need = 0
    if rq.pet_model.count == 0:
        pet_block_need = self.block_need(rq.pet_model)
    if pet_block_need + rq_block_need > len(self.free_block):
        return False

    rq.pet_model.count += 1
    rq.pet_model.blocks.extend([self.free_block.pop(0) for i in
       range(pet_block_need)])
    rq.blocks.extend([self.free_block.pop() for i in range(
       rq_block_need)])
    return True
```

In the code, the system first checks whether the corresponding PET model needs to be loaded into GPU memory. We maintain a reference count for the loaded PET model, incrementing the counter for each request in the batch that references the PET model. The counter is decremented when the request is removed from the batch. The PET model is only released from GPU memory when the counter reaches zero. This approach helps to avoid redundant loading of the same PET model, thus preventing resource wastage. Next, we allocate the available memory blocks to the request itself and its corresponding PET model, marking the involved memory blocks as nonidle.

The code uses the `block_need` function, which calculates the additional memory blocks required for each layer of the target request. Fixpool will allocate memory according to the required number of blocks.

LISTING 4.3: Block needed

```
def block_need(self, item):
        total_need = (item.length + self.token_per_block)
        current_hold = len(item.blocks)
        need = total_need - current_hold
        if need > 0:
            return need
        else:
            return 0
```

When the system does not use zigzag block scheduling [She+23a], the PET model and the KV cache can be stored together, as they are prefetched simultaneously. In this configuration, prefetching is relatively straightforward, as the system can directly load the blocks that contain the relevant data for the task at hand. Since both the PET model and the KV cache are prefetched in parallel, the process becomes more efficient with minimal overhead. The system can identify and retrieve only the blocks that are essential for the current operation, thus reducing unnecessary data retrieval and optimizing memory usage. The following code snippet illustrates the prefetching process.

LISTING 4.4: Prefetch

```
def prefetch(self, memory_layer):
    self.free_block.sort(reverse = True)
    cut = self.free_block[-1]
    for i in range(len(self.free_block) - 1):
        c = self.free_block[i]
        n = self.free_block[i+1]
        if n != c - 1:
            cut = c
    working_space = self.working_spaces.pop()
    memory_layer.prefetch(working_space.data, cut * self.block_size
        )
    self.working_spaces.insert(0, working_space)
```

The code snippet provided demonstrates the prefetching process for memory optimization. It begins by sorting the available memory blocks in descending order to identify the "cut" point for used memory blocks, which determines where the memory can be segmented for prefetching. The function then gets the working space, and the relevant data are obtained from the memory layer up to the cut point in the working space. This process ensures that only the necessary data blocks are fetched, enhancing memory efficiency and reducing unnecessary retrievals.

4.1.2 Working Space

Finally, we also have preallocated memory in the GPU to hold runtime resources, which is called the working space. The weights or KVs needed will be fetched into the working space for the operator to access. The meta-parameter controls the size of the working space. By default, it matches the size of the memory layer to load all the parameters of an inference layer at one time. Still, in some cases, it may be much smaller than the memory layer, making it impossible to simultaneously load all the necessary parameters for an entire decoder layer. In such situations, resources can be loaded in batches during inference of a decoder layer to avoid

out-of-memory (OOM) errors. This method allows the model inference to run successfully by trading time for memory. In the code implementation, we use a `MemoryLayer` fully stored in GPU memory as a working space.

LISTING 4.5: Construct Working Space

```
self.working_spaces = []
for i in range(num_working_space):
    self.working_spaces.append(
        MemoryLayer(opt_config, block_num * block_size, [100,0,0],
            self.token_per_block)
    )
```

Each MemoryLayer instance is initialized with specific configuration parameters such as `opt_config`, the size of each block `(block_num * block_size)`, and an offload distribution [100, 0, 0] which means 100 percent of parameter stores in the GPU memory.

4.2 Storage Strategy

The way in which parameters are stored, their locations, and their distribution between different hardware components can significantly impact the overall efficiency of the system. For example, by taking into account the memory hierarchy, we can optimize the placement of parameters, positioning them closer to the processing units. This reduces data access latency and improves processing efficiency, thus improving overall performance. In addition, the characteristics of various hardware components, such as cache size and access speed, must be considered when designing the memory and data distribution strategies of the system. By carefully tailoring the placement and distribution of parameters across the system's memory and processing units, we can strike an optimal balance between computational efficiency and resource utilization. This section will delve into the details of parameter storage strategies and highlight how different memory management techniques can be employed to maximize system performance.

4.2.1 Hyperparameter Policy

FixPool is initialized based on certain hyperparameters, including total length, block length, and offload ratio for each layer. The configuration of these parameters will directly impact the overall efficiency of the system:

1) The allocation of total memory resources impacts the trade-off between system throughput and latency. GPU memory is limited, and allocating more memory for storing model weights reduces the prefetch latency between layers, but leaves less memory available for the KV cache and PET weights, thereby supporting smaller batch sizes. In contrast, allocating more memory to the KV cache and PET weights allows larger batch sizes, better suited to a zigzag block schedule [She+23a], increasing throughput but also increasing latency.

2) The parameters required by each inference layer can be stored on the same device. However, this may lead to imbalances in prefetch latency between layers, as layers with parameters stored on the disk take longer time to load onto the GPU than those on the CPU memory. This also leads to wastage of I/O resources, as the I/O resources between the CPU memory and disk remain idle when layers stored on the CPU memory are being loaded.

Therefore, in memory management systems, to ensure optimal performance, the distribution of hyperparameters should adhere to the following strategies:

1) Minimizing Total Memory Parameters: Calculate the total memory parameters based on the model requirements and minimize overall memory usage. This approach reduces system overhead and enhances resource utilization efficiency.

2) Priority-Based Memory Utilization: Prioritize GPU memory usage, ensuring that it is fully utilized before gradually utilizing CPU memory. Once CPU memory is maximized, the transition to disk storage. This hierarchical utilization strategy effectively reduces performance bottlenecks and improves overall processing efficiency.

3) Memory Distribution: To ensure similar data prefetching latencies across layers, maintain a consistent storage proportion for the parameters of each Transformer layer. This prevents performance degradation caused by long-tail effects.

Implementing these strategies can significantly optimize computational efficiency and resource utilization in large-scale inference scenarios, providing robust support for improving system performance.

4.2.2 Task-specific Parameter Store

Optimizations can be made to the storage of the PET model parameters. By implementing an optimized memory allocation strategy in memory management systems, system performance can be significantly improved.

Specifically, prioritizing the preloading of frequently accessed parameters into faster storage layers, such as GPU memory or CPU memory, rather than slower storage mediums, such as disks, effectively reduces data access latency. We can analyze the access frequency and access patterns of different PET parameters, and allocate appropriate resources based on their specific characteristics to achieve higher computational efficiency. In addition, based on the real-time changes in the workload of different task types, we can prefetch certain PET parameters according to the frequency of different task types in the request pool.

This reduces computational latency and enhances the efficiency of model execution. We consider optimization in two aspects:

1) It is challenging to load numerous PET models onto GPU or CPU memory due to space constraints, and storing PET model weights on the disk incurs significant offloading costs. To mitigate this, part of the PET parameters can be permanently stored on the available GPU and CPU memory, reducing offloading and shortening the preparation time for processing the next batch of requests. Considering the different characteristics of downstream tasks, we prioritize storing PET models for high-priority or frequent tasks on the GPU. In contrast, models for low-priority or infrequent tasks are stored on the disk.

2) In resource-constrained LLM inference systems, there is typically an overlap achieved by parallelizing the fetching of the next layer's parameters with the ongoing computation. However, when the parameters to be loaded are already present in the GPU memory or CPU memory, the I/O resources across PCIe, CPU memory, and disk may go underutilized, resulting in resource wastage. Given the predictable execution order of offline requests, the system can utilize any idle I/O resources to prefetch the PET model parameters for subsequent requests, thereby optimizing the utilization of I/O resources.

4.3 KV Cache Update

During LLM inference, the KV cache is updated in the working space, but the memory layer that stores all parameters remains unchanged. To ensure coherence between GPU and CPU memory, the parameter on GPU needs to be updated to CPU. To efficiently update the KV cache to the memory layer, it is necessary to locate the KV cache position for each request and store newly generated K and V. Due to the design of FixGen, requests in both the prefilling and decoding stages may be processed simultaneously, so the position of each request in the KV cache and the required KV length are not fixed. It is challenging to store only the newly generated KV parts back to the CPU memory due to the varying KV lengths of each request. If the entire KV is transferred back to the CPU memory, redundant KV data will cause unnecessary consumption of I/O resources.

4.3.1 Update Process

As Fig. 4.2 shows, our solution to this challenge is to simultaneously update the KV cache in both the GPU working space and the CPU memory layer. We designed specialized update kernels for GPU and developed the corresponding traditional “naive" CPU algorithms to handle updates to the corresponding KV cache in CPU memory or on disk. After calculating the KVs in the LLM decoding layer, we only copy the newly generated KV back to the CPU memory rather than copying the entire KV. When the GPU updates the KV cache in GPU memory, the CPU can invoke its corresponding update algorithm to synchronize by copying

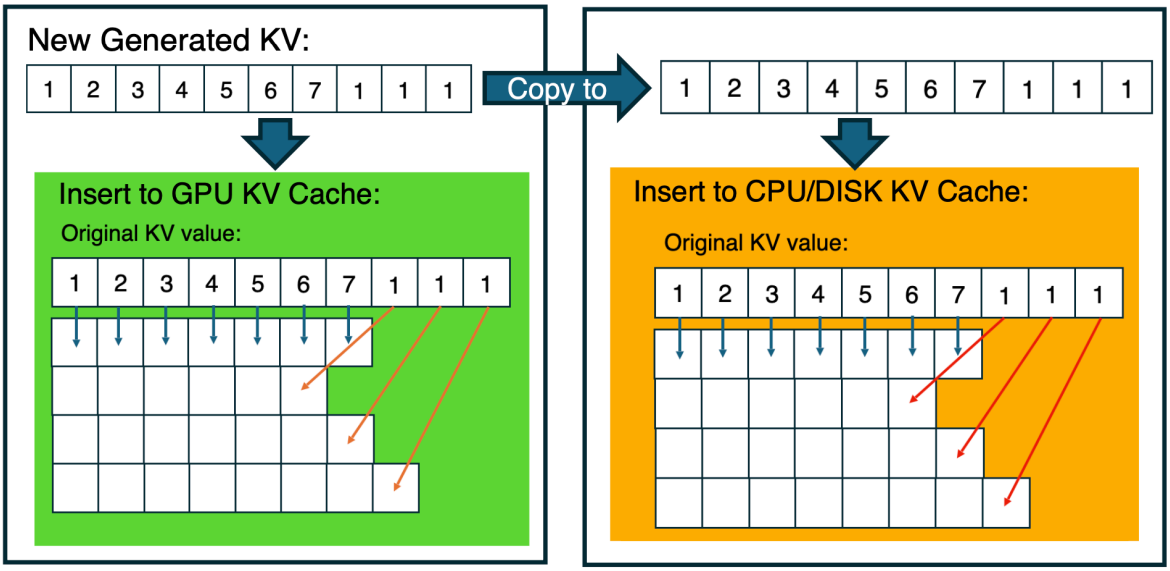

FIGURE 4.2: The process of storing the KV cache.

the KV back to CPU memory, thereby updating the KV cache stored in the CPU memory or on the disk. This approach completes the KV cache updates without consuming additional I/O resources.

4.3.2 Update Kernel Design

In this subsection, we discuss and introduce the GPU kernel used for KV updates. Unlike conventional copy operators, each request within the current batch may have different lengths, and the handling of KV pairs varies between stages. For the prefilling stage, it is necessary to store back all KV pairs associated with each request. However, during the decoding stage, only the KV pairs corresponding to the most recent token of each request need to be copied.

We have designed an operator named updatekv that, in addition to requiring K, V, and blocks as pointers for the memory layer, also needs additional parameters to pinpoint the target locations for storage. We designed an operator named updatekv, which requires not only KV and blocks as pointers for the memory layer, but also several additional parameters for precise location targeting within the memory. These parameters include `token_batch_ind`, which indicates the index of each token within its respective request in the current batch, `batch_index`,

marking the starting position of each request within the batch, and `request_length`, noting the length of each request. Additional details related to memory blocks are also needed, such as `blocks_ids` to identify which blocks are needed for the current batch and `blocks_index` to indicate the starting position of the blocks required by each request within the `blocks_ids`.

Alg. 1 shows the detail of updatekv. Each thread block is used to store one newly generated KV pair. From lines 8 to 11, the target request is for the current KV pair. Then, the position of the memory block in the memory layer is calculated from Line 12 to Line 15. Finally, the code from Line 16 to Line 19 does the pointer arithmetic and assigns each thread in the current thread block to copy KV to the target space.

The launch function is listed below. The launcher for the `updatekv` kernel includes the process of setting the grid size and thread block size and calling the `updatekv` kernel. As shown, the size of the grid is set to the externally passed `length`, which represents the number of new KV pairs generated in this round of inference. This ensures that each thread block corresponds to a set of KV pairs. The size of the thread block is set to $\min(\texttt{hidden\_dim}, 1024)$, where each thread in the block is responsible for copying a parameter, maximizing the parallelism capacity of the GPU threads. Since `hidden_dim` is often larger than 1024, the maximum number of threads that can be set, a loop is used to utilize all threads by copying the parameters in multiple passes.

LISTING 4.6: UpdateKV launcher

```
template <typename IType, typename DType>
void launchupdatekv(DType* Blocks, DType* K, DType* V,
        IType* token_batch_ind, IType* batch_index,
        IType* request_length,
        IType* blocks_index, IType* blocks_ids,
        IType token_per_block, IType hidden_dim, IType length
            ){
    int threadNumber = hidden_dim;
```

Algorithm 1 updatekv kernel

Input:
  $batch\_info$: information of the current batch
  $hidden\_dim$: Dimension of hidden layers
  K: Key matrix
  V: Value matrix
Output:
  $Blocks$: Base pointer for blocks in memory
Initialize the token identifier $token\_id$ using the block index along the x-dimension ($blockIdx.x$)
Retrieve the current batch index $current\_batch$ from the $token\_batch\_ind$ array using $token\_id$
Determine the batch shift $batch\_shift$ from the $batch\_index$ array using $current\_batch$
Compute the global token identifier $global\_token\_id$ by adding the sequence length of the current batch ($request\_length[current\_batch]$) to the difference between $token\_id$ and $batch\_shift$
Retrieve the block shift $block\_shift$ from the $blocks\_index$ array using $current\_batch$
Calculate the block index $block\_ind$ by adding $block\_shift$ to the integer division of $global\_token\_id$ by $token\_per\_block$
Load the specific block ID $block\_id$ from the $blocks\_ids$ array using $block\_ind$
Compute the pointer to the target block $target\_block$ by offsetting the base pointer $Blocks$ with the calculated $block\_id$, scaled by $2 \times token\_per\_block \times hidden\_dim$
Initialize the thread-specific index n using the thread index along the x-dimension ($threadIdx.x$)
Determine the token shift $token\_shift$ as the remainder of $global\_token\_id$ divided by $token\_per\_block$
Store the Key value by assigning $K[token\_id \times hidden\_dim + n]$ to the position $token\_shift \times hidden\_dim + n$ in $target\_block$
Store the Value value by assigning $V[token\_id \times hidden\_dim + n]$ to the position $(token\_shift + token\_per\_block) \times hidden\_dim + n$ in $target\_block$

```
if(hidden_dim > 1024){
    threadNumber = 1024;
}
updatekv<<<length, threadNumber>>>(Blocks, K, V,
                    token_batch_ind, batch_index,
                        request_length,
                    blocks_index, blocks_ids,
                    token_per_block, hidden_dim);
```

```
}
```

When calling the `updatekv` kernel function, the main parameters passed include K and V as sources of the KV pairs, and `Blocks`, which represents the memory blocks and serves as the address for the KV cache. In addition, parameters such as `token_batch_ind`, `batch_index`, `request_length`, `blocks_index`, and `blocks_ids` are used to compute the relative position of the tokens and their global positions within the `Blocks`.

CHAPTER 5

Inference Process and Batch Strategy

The FixGen batching process is designed to utilize available resources efficiently. The entire FixGen workflow operates within a loop structure, which is mainly made up of three distinct stages, as shown in Fig. 3.1. These stages work together to optimize the allocation of computational resources and the usage of fixpool during the inference process. By organizing tasks into batches, the system reduces overhead, improves throughput, and ensures efficient resource utilization for different types of requests. The following sections will describe each of these stages in detail, highlighting their role in achieving high performance and responsiveness in real-time and non-real-time tasks.

1) Batching strategy: select requests to batch. Due to the different nature of online and offline requests, they need to be handled separately. They will be elaborated in section 5.1.

2) Inference: Model Inference. The router will select appropriate operators to support requests in the prefilling and decoding stages being inference within the same batch. The details of the router will be elaborated in section 5.2.

3) Post-Inference: We adopted the concept of ORCA iteration-level scheduling [Yu+22] to maximize the utilization of limited computational resources. Update each request and re-organize the memory blocks when the distribution is disrupted. It will be elaborated in section 5.3.

5.1 Batching Stategy

In environments where online and offline requests coexist, a batch strategy must consider the differing latency requirements. Online requests should be processed and responded to as quickly as possible. Offline requests, while not requiring rapid responses, involve a large volume of work and require high throughput for efficient processing. To handle environments where both online and offline tasks exist, we divide the batching strategy into two steps: Step 1, online request strategy, and Step 2, offline request strategy.

The code for the Batch strategy is shown below. As seen, we use the `BatchManager` to manage batch-related functions and have designed user-friendly interfaces. These two functions use four different parameters in total: `batch`, which represents the current batch, `mempool` is an instance of FixPool, `online` representing the existing online requests, and `minibatches`, which refers to offline requests organized into `minibatches`.

LISTING 5.1: Batching strategy

```
# add online
if len(online) != 0:
     if BatchManager.add_online(batch, online.pop(0), mempool):
# add offline
BatchManager.add_offline(batch, minibatches, mempool)
```

5.1.1 Online Request Strategy

In Step 1, we add as many online requests received in the batch as possible. In cases of insufficient resources, offline requests in the batch can be suspended to free up enough space to handle online requests. There might be a scenario where the available space is insufficient. In such cases, temporarily suspending some offline requests may be necessary to free up the resources initially allocated to them. This allows online requests to utilize these resources for LLM inference, ensuring that the system prioritizes handling online requests first. Only

after all the online requests have been loaded into the execution batch are offline requests considered. This prioritization ensures a rapid response to online needs.

In the code implementation, we used a loop to manage the assignment of online requests. Specifically, when there is insufficient free memory to accommodate an online request in the current batch, the loop iteratively releases offline requests from the batch. This process continues until enough memory is freed to successfully insert the target online request into the batch.

LISTING 5.2: Add online request

```
def add_online(batch, request, fixpool:Fixpool):
    total_block_need = fixpool.total_block_need(request)
    for i in range(len(batch)):
        if total_block_need <= len(fixpool.free_block):
            fixpool.assign(request)
            batch.append(request)
            return True
        else:
            target = batch.pop(0)
            if target.online:
                batch.append(target)
                continue
            else:
                fixpool.free(target)
                target.add_to_requestpool()
    if total_block_need > len(fixpool.free_block):
        fixpool.assign(request)
        batch.append(request)
        return True
    return False
```

Finally, we also consider edge cases: when online requests spike, after all offline requests are released, we will attempt to re-add the target online requests to the current batch outside of the loop. This helps prevent situations where memory becomes available after the release of memory blocks but remains underutilized.

5.1.2 Offline Request Strategy

In Step 2, we add the offline request to the batch in case there is space left in the memory block after adding the online request in the first step. Since prefetch latency is dominant and does not vary significantly with increasing batch size, we should accommodate as many requests as possible.

To avoid repeatedly loading shareable resources, such as PET parameters, we first organize requests of the same task type into a minibatch, where the requests share PET parameters. We aim to place the entire minibatch in a single execution batch as much as possible to minimize the redundant loading of PET parameters. When selecting a minibatch, the initial step involves calculating the number of memory blocks required for that specific minibatch. If sufficient memory is available, these blocks are pre-allocated to serve as the addresses for both the PET and the corresponding KV cache for the requests. When the available memory is not enough to load the entire mini-batch, we split the mini-batch and load as many requests as possible, thereby improving space utilization. Offline requests whose PET parameters are already stored in the Memory Layer are prioritized for loading into the batch to reduce the offloading of PET parameters.

The code snippet provided demonstrates the process of adding offline requests to the current batch. It starts by iterating over the `minibatches` list, popping the last `minibatch` from it. For each request `rq` in the `minibatch`, it checks whether there is enough space in the Fixpool using the `assign` method. If the request cannot be assigned, for example in the case with insufficient space, the remaining requests from the current `minibatch` are added back to the `minibatches` list, and the function returns. If the request is successfully assigned, it

is added to the current batch. This ensures that only requests that can fit into the available space are added, while others are postponed for later processing.

LISTING 5.3: Add offline requests

```
def add_offline(batch, minibatches, fixpool:Fixpool):
    while len(minibatches) != 0:
        minibatch = minibatches.pop()
        for ind, rq in enumerate(minibatch):
            if not fixpool.assign(rq):
                remain = minibatch[ind: ]
                minibatches.append(remain)
                return
            else:
                batch.append(rq)
```

When a `minibatch` is fully loaded into the current batch, the system then attempts to load the next `minibatch`. This ensures the most efficient utilization of PET parameters already loaded onto the GPU.

5.2 Router

Traditional methods have shortcomings when handling online requests, which is to directly pause the ongoing offline requests in the decoding stage, process the online task's prefilling phase, and then merge it with the paused offline functions into the same batch for simultaneous inference once the online request enters the decoding stage. However, throughput will decrease significantly when handling online requests during the prefilling stage: the current batch needs to be suspended. This reduction is particularly noticeable during frequent online request bursts, which can significantly affect overall latency. To address this challenge in single GPU systems, we have made modifications to the context attention used for prefilling computations and the token attention used for decoding computations.

In environments where both online and offline requests coexist, scheduling strategies must take memory management into account: due to the unpredictability of online tasks, trigger OOM errors will arise when resource shortages, and it also potentially lead to increased overhead and delays in the memory system. When designing scheduling strategies, consideration must be given to how memory management can be utilized to avoid additional overhead and potential anomalies. In addition, there is a need to respond quickly to online requests without affecting the overall system throughput. Traditional LLM inference systems are designed with the assumption that the workload consists solely of offline or online tasks. In environments where both online and offline requests coexist, scheduling strategies must not only address memory management issues, but also ensure quick responses to online requests without negatively impacting overall throughput.

From the point of view of a single GPU system, each layer of the LLM computation involves two main operations: parameter prefetch from CPU memory to GPU and interference computation. The latency cost of one decoding layer L can be expressed as Equation 5.1 with $L_{offload}$ denoting the latency for prefetch operations and L_{comp} for the latency of computation operations.

$$L \approx max(L_{offload}, L_{comp}) \tag{5.1}$$

Prefetch operations and computation operations can be executed in parallel. In a traditional inference system, the model weights are all loaded into the GPU, or some of them are replaced in CPU memory, $L_{offload}$ will be very low, and L_{comp} dominates the overall latency. The $L_{offload}$ is a more decisive factor in a single GPU system. Mixing these requests with those currently being decoded only increases L_{comp}, but in single GPU systems, where $L_{offload}$ is relatively high, mixing phase inference does not affect overall latency when the computational latency is lower than the prefetch latency.

Therefore, it is advisable to mix requests from different stages in the same batch for computation. Fig. 1.1 shows the superiority of inferring requests in different phases in one batch while accepting new requests. In conventional systems such as FlexGen, since different operators are used for prefilling and decoding, the only way to quickly respond to an unexpected online request is to suspend the requests in the decoding phase and do a separate round of prefilling

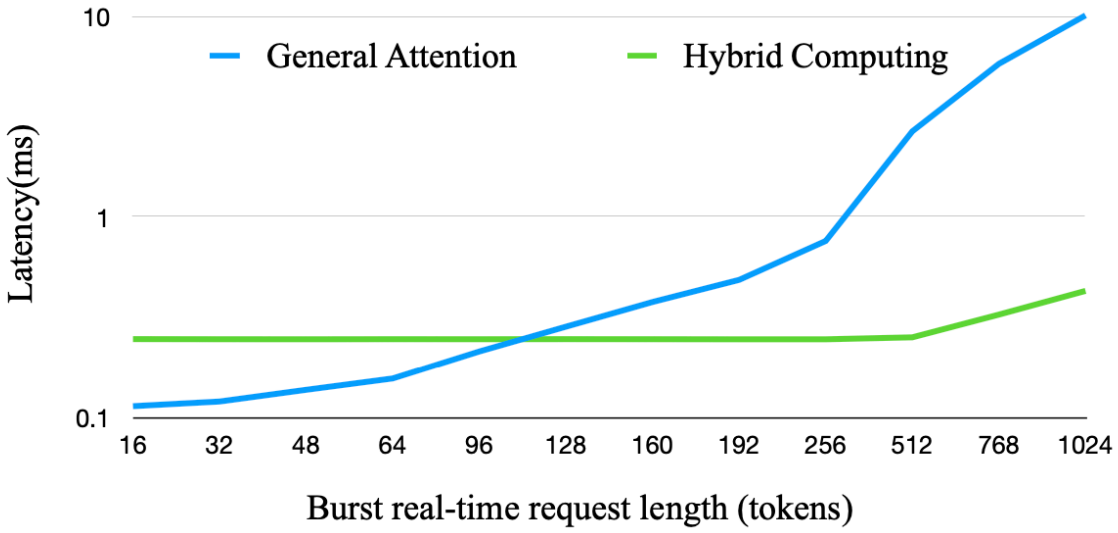

FIGURE 5.1: Comparison between general attention and hybrid computing.

for the online request. If users of a single GPU system frequently issue online requests during active periods, suspending the current batch to accommodate these requests can significantly impact inference efficiency. This interruption disrupts the continuous processing flow, leading to increased latency and reduced overall throughput. Mixed-phase batching avoids this suspension to keep the total throughput by ensuring parallelization of request inference at different stages.

5.2.1 Router Process

We recorded the latency for different lengths of sudden online requests using general attention and hybrid computing with the same batch size and the same request length in Fig. 5.1. This shows that when the length of sudden online requests reaches a certain threshold, the performance of hybrid computing significantly outperforms general attention, which has been mentioned in section 2.4.3. In real-world hardware environments, we will test both online and offline tasks of varying lengths to identify their latency intersection, which will serve as the threshold.

Fig. 5.1 shows the comparison between hybrid computing and general attention, which can enable the system to infer two phases in one batch. This indicates that general attention is

more suitable for requests with shorter lengths, while hybrid computing is better suited for longer requests that require full utilization of GPU performance. Using this characteristic, the system can select the most appropriate operators based on the length of sudden online requests for more efficient inference. To mix computation requests from different stages in the same batch, the system needs to utilize general attention or hybrid computing approaches.

We employ a router to select the most appropriate processing method based on the characteristics of the incoming requests. Specifically, for online requests that exceed a predefined length threshold, the router will opt for hybrid computation, which allows for more efficient handling of larger and more complex tasks. In contrast, for requests with lengths below the threshold, the router typically utilizes a focused attention mechanism to process the task, as it is more efficient for smaller-scale operations. The threshold value, which determines the decision point between hybrid computation and focused attention, is determined by the latency intersection point of these two approaches. This adaptive approach ensures that the system can optimize resource allocation and processing efficiency based on real-time data.

In some scenarios, where computation latency dominates over offloading latency, hybrid computing of prefilling and decoding can result in longer overall latency for sudden online requests. In our design, we maintain a scheduling priority to handle the prefilling stage of sudden online requests separately to ensure that the inference of these urgent requests is completed with the lowest possible latency.

The code below shows how the router is used. The code implementation of the attention kernel has three branches: hybrid computing, decode only, and general attention.

In the hybrid computing branch, the router uses the batch information, `blen`, to determine whether the current batch is suitable for entering the hybrid computing branch. Once entering the hybrid branch, we use the prefilling attention kernel and the decoding attention kernel. When all requests are in the prefilling stage, the decoding attention kernel does not need to be called.

LISTING 5.4: Hybrid computing branch

```
if Router.hybrid(blen):
```

```
    # hybrid computing
    if blen.decode_index.shape[0] > 0:
        fixgen_token_att_fwd(qhead, blocks, temp_buffer, blen.
           decode_index, blen.blocks_ids, blen.blocks_index, blen.
           batch_index, blen.token_batch_ind, blen.request_length,
           self.memory_layer.token_per_block, blen.max_input_len)
    fixgen_context_att_fwd(qhead, k, v, temp_buffer, blen.
       prefill_index, blen.blocks_ids, blen.blocks_index,  blen.
       batch_index, blen.token_batch_ind, blen.request_length, self
       .memory_layer.token_per_block, blen.max_input_len)
```

When no requests are in the prefilling stage, the system operates in a streamlined mode, executing only the decoding attention kernel for high-performance attention computation. Since there are no prefilling tasks, the system can focus solely on decoding, without the additional overhead of data retrieval or memory management. As a result, the efficiency of this phase is optimized. We refer to this phase as the decode-only branch.

LISTING 5.5: Decode only branch

```
elif blen.prefill_length == 0:
    # decode only
    fixgen_token_att_fwd(qhead, blocks, temp_buffer, blen.
       decode_index, blen.blocks_ids, blen.blocks_index, blen.
       batch_index, blen.token_batch_ind, blen.request_length, self
       .memory_layer.token_per_block,  blen.max_input_len)
```

Finally, when there are requests in both the prefilling and decoding stages within the current batch, and the Router determines that hybrid computing is not suitable, the general attention kernel is called to perform the attention operation calculations.

LISTING 5.6: General attention branch

```
else:
    # general attention
```

```
    fixgen_general_att_fwd(qhead, blocks, temp_buffer, blen.
        blocks_ids, blen.blocks_index, blen.batch_index, blen.
        token_batch_ind, blen.request_length, self.memory_layer.
        token_per_block,  blen.max_input_len)
```

5.2.2 Attention Kernel

In this section, we will provide a detailed explanation of the kernels used for general attention and hybrid computation, along with their corresponding kernel launchers.

5.2.2.1 General Attention

We modified token attention to the general attention kernel to support prefilling and decoding attention. In general attention, we assign each thread block to compute output for a single head of an input token. In a thread block, the computation of Q for the current token and the multiplication with K and V, as well as the softmax operation, are performed in a loop. Each iteration reads the KV pairs for `BLOCK_N` tokens to increase the degree of parallelism.

The pseudocode for this attention kernel is shown in Algorithm 2, it is written in Triton so that we only do thread block level control and let the triton compile the thread level control. In lines 6 to 9, the necessary variable is set up, including thread block id and Triton array required for parallelism. In the pseudocode, from line 10 to line 12, information pertaining to the request associated with the current token is computed, including the index and the length of the corresponding request. Lines 13 and 14 calculate the token shift and head shift for Q and V, and the shift for V is the same for K because they are stored in pairs. The shift for KV is a temporary value to avoid repetitive computations. The exact locations in the memory layer of the corresponding KV will be located in the loop. The `acc` in line 15 is used to accumulate temporary data to iteratively calculate the softmax.

In the loop starting at line 16, each iteration computes Q and the corresponding results for the `BLOCK_N` pairs of K and V. The softmax results are then stored in `acc`. First, we need

Algorithm 2 general attention kernel

Input:
  $batch\_info$: The information structure that contains $request\_length$, $blocks\_index$, $token\_per\_block$.
  Q: Query matrix
  $blocks$: Blocks containing key vectors
Output: Updated attention result stored in Out
Initialize the current batch index cur_batch using the first dimension of the program ID
Initialize the current head index cur_head using the second dimension of the program ID
Generate offset array $offs\_n$ ranging from 0 to $BLOCK\_N - 1$
Generate offset array $offs\_d$ ranging from 0 to $BLOCK\_DMODEL - 1$
Load the global batch index cur_global_batch from $token\_batch\_ind$ using cur_batch
Load the sequence length $cur\_batch\_seq\_len$ for the current global batch from $request\_length$
Load the starting block index $blocks\_start$ for the current global batch from $blocks\_index$
Compute the query vector offset $off\_q = cur\_batch \times stride\_obs + cur\_head \times stride\_oh + offs\_d \times stride\_od$
Compute the key vector offset $v\_offs = cur\_head \times stride\_oh + offs\_d[None, :] \times stride\_od$
Initialize the accumulator acc as a zero vector of size $[BLOCK\_DMODEL]$ with data type float32
while $start\_n < cur\_batch\_seq\_len$ **do**
  Align $start\_n$ to the nearest multiple of $BLOCK\_N$
  Calculate the current block number $blocknum = \frac{start\_n + offs\_n}{token\_per\_block}$
  Load the current block ID cur_block from $blocks\_ids$ using $blocks\_start + blocknum$
  Load the query vector q from Q using the offset off_q
  Determine the key vector location $k\_loc = cur\_block \times 2 \times token\_per\_block + (start\_n + offs\_n) \mod token\_per\_block$
  Load the key vectors k from $blocks$ using $v\_offs + k\_loc[:, None]$, applying a mask where $(start\_n + offs\_n[:, None]) < cur\_batch\_seq\_len$ and setting masked positions to 0.0
  Compute the attention scores $att\_value = \text{sum}(q[None, :] \times k, \text{axis} = 1)$
  Scale the attention scores by the scaling factor sm_scale
  Perform Blocked Softmax Processing on att_value
  Determine the value vector location $v\_loc = cur\_block \times 2 \times token\_per\_block + (start\_n + offs\_n) \mod token\_per\_block + token\_per\_block$
  Load the value vectors v_value from $blocks$ using $v\_offs + v\_loc[:, None] \times stride\_obs$, applying a mask where $(start\_n + offs\_n[:, None]) < cur\_batch\_seq\_len$ and setting masked positions to 0.0
  Accumulate the weighted value vectors into acc as $acc+ = \text{sum}(att\_value[:, None] \times v\_value, \text{axis} = 0)$
  Increment $start\_n$ by $BLOCK\_N$
end while
Convert the accumulator acc to float16 precision
Compute the output offset $off\_o = cur\_id \times stride\_obs + cur\_head \times stride\_oh + offs\_d \times stride\_od$
Determine the output pointer $out\_ptrs = Out + off\_o$
Store the accumulated attention result acc into out_ptrs

to compute `Q * K`. At line 20, Q is loaded. In line 21, we calculate the target position of K in the memory layer. This allows for loading of `BLOCK_N` corresponding values K for the target head on line 22, using the position and position shift of K. We did not change the attention process from flash attention, which iteratively calculates the approximate softmax value. After softmax operation, P * V is calculated. Finally, the final result is stored to the output pointer after the loop process.

To launch this kernel, the grid size is `(n, head_num)` where n is the length of input, and `head_num` denotes the number of heads. The triton can handle the management of each thread automatically.

This kernel can calculate attention for requests in different phases, but also bring drawbacks. The drawback of this kernel is that it cannot fully utilize the GPU due to the imbalance of thread assignment. In the model, the Q values for prefilling requests are contiguous, whereas those for decoding requests are non-contiguous. Consequently, prefilling attention may require more intensive processing. However, in our modified general attention mechanism, we have employed the same processing approach for prefilling attention as for decoding. Thus, efficiency is affected negatively.

5.2.2.2 Hybrid Computation

For hybrid computation, which means using the attention kernel in a prefilling state and decoding the attention kernel sequentially. We use two tensors, `prefill_index`, and `decode_index`, to record the index in the input matrix for each prefilling request and each decoding request relative. These tensors are then utilized in both context attention and token attention to selectively compute the necessary segments in batch. Additionally, we updated the pointer arithmetic to make it compatible with Fixpool. The procedure of pointer arithmetic is the same as the Algorithm 2, using additional parameters such as `token_batch_ind`, `blocks_ids` and `blocks_index` to locate the target KV to calculate. Pointer arithmetic takes O (1) time to execute, in practice, it only takes 1.01 to 1.10 times latency compared to the original, which is acceptable.

The prefilling attention launcher uses the length of `prefill_index` to replace the original batch. The code is listed below. To launch these prefilling attention kernels, the grid shape is set to `(len(prefill_index), head number, cdiv(max_input_len, BLOCK))` where `BLOCK` represents the number of KVs that will be processed parallel in a thread block. Apart from replacing the input length with the length of the `prefill_index`, there have been no other changes.

LISTING 5.7: Prefilling attention launcher: grid and warp

```
@torch.no_grad()
def fixgen_context_att_fwd(q, k, v, temp_buffer, prefill_index,
                blocks_ids, blocks_index,
                batch_index, token_batch_ind, request_length,
                token_per_block, max_input_len):
    BLOCK = 128
    # shape constraints
    Lq = q.shape[-1]
    assert Lq in {16, 32, 64, 128}
    sm_scale = 1.0 / (Lq**0.5)
    batch, head = prefill_index.shape[0], q.shape[1]
    grid = (batch, head, triton.cdiv(max_input_len, BLOCK))

    # batch, head,
    num_warps = 4 if Lq <= 64 else 8
```

When calling the prefilling attention kernel, we need to provide not only the parameters that were just determined, such as the grid size and warp size, but also the foundational inputs required for the attention calculation, such as Q, K, and V. Additionally, we need some additional information and internal memory space to assist in calculating pointers or temporarily storing intermediate results, such as `prefill_index`, `token_batch_ind`, `request_length`, `temp_buffer`.

LISTING 5.8: Prefilling attention launcher: call kernel function

```
    _fixgen_fwd_kernel[grid](
        q, k, v, sm_scale,
        prefill_index, token_batch_ind, request_length,
        temp_buffer,
        q.stride(0), q.stride(1), q.stride(2),
        BLOCK_M=BLOCK,
        BLOCK_DMODEL=Lq,
        BLOCK_N=BLOCK,
        num_warps=num_warps,
        num_stages=1,
    )
    return
```

Consistent with the marking attention mechanism in *lightLLM* [Ano23], the marking attention process is divided into two separate kernels to improve efficiency. Kernel A is responsible for calculating the dot product between the query matrix Q and the key matrix K, while kernel B handles the calculation of the softmax function and subsequent multiplication with the value matrix (P * V). To adjust the system's memory structure, we modified only the pointer arithmetic in these two kernels, ensuring that all other logical processes remain consistent with the original kernel implementation. This targeted modification allows us to integrate our memory management strategy without altering the core functionality of the attention mechanism, thereby optimizing for the specific memory requirements of our system while preserving the performance characteristics of *lightLLM*.

Since decoding attention uses two kernels for the complete attention calculation, in the launcher of the decoding attention kernel, we divide it into two parts. Part A manages the invocation of kernel A, while Part B handles the invocation of kernel B.

In launcher Part A, we begin by setting the grid size. Distinct from the original decoding attention, we adjust the token length parameter within the grid to reflect the decoding index length. This adjustment is necessary because, in the FixGen batch, only tokens in the decoding

stage require attention computation. Tokens associated with requests in the prefilling stage are excluded from consideration when determining the grid size. In the code, the variable representing the length of the decoding index is named `batch`, which corresponds to the length of the tokens in the batch that need to be processed by decoding attention. The `BLOCK` parameter is set to a default value of 32, indicating that 32 Ks are allocated for computation within each thread block. Since the length of q for each request is 1, and k is partitioned into blocks of length 32, the computation for each thread is small, allowing for rapid calculation of the results.

LISTING 5.9: Decoding attention part A launcher: grid and warp

```
@torch.no_grad()
def fixgen_token_att_fwd(q, blocks, temp_buffer, decode_index,
                blocks_ids, blocks_index,
                batch_index, token_batch_ind, request_length,
                token_per_block, max_input_len):
    BLOCK = 32
    # shape constraints
    Lq = q.shape[-1]
    # assert Lq == Lk
    assert Lq in {16, 32, 64, 128}
    assert q.stride(0) == blocks.stride(0)
    sm_scale = 1.0 / (Lq ** 0.5)
    batch, head_num = decode_index.shape[0], q.shape[1]
    grid = (batch, head_num, triton.cdiv(max_input_len, BLOCK))
    num_warps = 4
```

Subsequently, we use the grid and block parameters as input to invoke decoding kernel A, which is named `_fixgen_token_att1_kernel`. Unlike the original kernel, the decoding kernel used in FixGen requires additional parameters for computation. These extra parameters are essential for managing the specific conditions and optimizations within

the FixGen framework. We use `decode_index` to locate requests that are in the decoding stage. Additionally, parameters such as `token_batch_ind`, `request_length`, `blocks_ids`, and `blocks_index` are employed to locate the necessary information within the KV cache managed by FixPool. These parameters enable efficient indexing and retrieval of relevant data for decoding operations.

LISTING 5.10: Decoding attention part A launcher: call kernel function

```
    Att_out_shape = (head_num, batch, max_input_len)
    Att_out_size = (head_num * batch * max_input_len)
    Att_Out = temp_buffer.view(-1)[: Att_out_size].view(
       Att_out_shape)
    _fixgen_token_att1_kernel[grid](
        q, blocks, decode_index,
        token_batch_ind, request_length,
        blocks_ids, blocks_index,
        sm_scale, max_input_len,token_per_block,
        Att_Out,
        q.stride(0), q.stride(1), q.stride(2),
        Att_Out.stride(0), Att_Out.stride(1),
        BLOCK_DMODEL=Lq,
        BLOCK_N=BLOCK,
        num_warps=num_warps,
        num_stages=1,
    )
```

In part B of the launcher, the structure is similar to that in part A. First, we need to calculate the size of the grid. Since kernel B involves calculating $P \times V$, where both P and V have lengths equal to the length of their corresponding request (rather than the length of the newly generated token), the computation is more intensive. Therefore, it is difficult to avoid using loops within threads to derive the results. To address this, we set the grid size as $(\text{batch}, \text{head})$ and allocate the computation for each token's single head to a thread block. In the algorithm,

loops are used to iteratively compute the $P \times V$ and accumulate the final output. This design results in a reduction in the computational load per iteration. Therefore, to maintain high parallelism, we increase the `BLOCK` size.

LISTING 5.11: Decoding attention part B launcher: grid and warp

```
    prob = Att_Out
    out = q
    BLOCK = 128
    grid = (batch, head_num)
    num_warps = 4
    dim = q.shape[-1]
```

The following code demonstrates the invocation of kernel B. The kernel B uses the same name style, as `_fixgen_token_att2_kernel`. Similarly to kernel A, we not only pass `BLOCK` and grid size as input parameters, but also utilize `decode_index`, `token_batch_ind`, `request_length`, `blocks_ids`, and `blocks_index` to locate target requests and the corresponding V. In addition, to save space, we pass Q as the output location to the function. This causes the final attention result to overwrite Q, thus reusing the memory space of Q and avoiding additional memory allocation. Although this approach prevents traceability of Q's data during inference, it does not impact users who treat the LLM as a black-box model.

LISTING 5.12: Decoding attention part B launcher: call kernel function

```
_fixgen_token_att2_kernel[grid](
        prob, blocks, out, max_input_len,token_per_block,
        decode_index,
        token_batch_ind, request_length,
        blocks_ids, blocks_index,
        prob.stride(0), prob.stride(1),
        out.stride(0), out.stride(1), out.stride(2),
        BLOCK_DMODEL=dim,
        BLOCK_N=BLOCK,
        num_warps=num_warps,
```

```
        num_stages=1,
    )
    return
```

5.3 Post-Inference

Inspired by Orca [Yu+22], post-inference is a process used to update batch information and reorganize resources after the completion of each inference round. By performing post-processing on the inference results of each batch, the system can dynamically adjust resource allocation and optimize computational performance. Furthermore, post-inference also addresses potential anomalies or errors, ensuring data accuracy and stable operation of the system. This mechanism not only enhances the processing speed but also bolsters the system's adaptability to continuously varying data streams.

In the code implementation, we use `next_token_ind` to filter out newly generated tokens for each request, using the `update` function of the class `BatchManager` to update the status of each request, such as its length and whether it is completed. It will be introduced in detail in the following section.

LISTING 5.13: Post-inference

```
# update
out_tokens =  torch.argmax(
    out_ids[batch_info.next_token_ind], dim=-1
).cpu().numpy()
batch = BatchManager.update(batch, out_tokens, mempool)
```

5.3.1 Update Process

While Fixpool ensures the continuity of parameter storage across different layers, the continuity of KV cache and task-specific parameters can be disrupted due to some tasks being

released and reallocated. In Post-Inference, the newly generated tokens will be appended to the requests in the update step.

The code for the update process is as follows. The newly generated tokens are appended to each request, increasing their length, and marked as being in the decoding phase.

LISTING 5.14: Batch update

```
for ind, rq in enumerate(batch):
    rq.input_id.append(output_token[ind])
    rq.length += 1
    rq.prefill = False
    if rq.length == rq.output_len:
        # release completed request
        fixpool.free(rq)
    else:
        if not fixpool.assign(rq):
            # suspend
            fixpool.free(rq)
            rq.add_to_requestpool()
        else:
            # normal
            new_batch.append(rq)
```

Requests that reach their maximum output length or are completed will be released. The released memory can be reused for other requests. In the code implementation, an `if` statement is utilized to manage the memory allocation and the handling of requests. Specifically, when a request is completed, the system releases the associated memory resources by removing the request from the active batch. If a new block is required, but there is insufficient space available, the system first frees up memory by releasing the current request and subsequently adding it back to the request pool for computing it later. This mechanism ensures that no new resources are allocated to avoid OOM errors. Finally, requests that do not encounter any

exceptional circumstances are added to the next batch for processing, ensuring a seamless transition between request cycles and maintaining optimal system performance.

We then swap the contents of the memory blocks to organize them and restore the arrangement to the distribution shown in Fig. 4.1. We employ a two-pointer algorithm as reorganize algorithm for each memory block to minimize the number of swap calls. This algorithm arranges the pet model weights and KV cache at the upper and lower ends of the memory layer, respectively. We use an easy-to-use interface for memory swap:

```
fixpool.reorganize()
```

The details of the reorganized algorithm will be introduced in the next subsection.

Since invoking the reorganize algorithm involves numerous swap operations, calling it with every update would result in additional latency. Therefore, we have set a tolerance threshold, and the reorganize algorithm is invoked only when the number of memory fragments exceeds this threshold. When the system does not use a zigzag block schedule, the pet model and KV cache are always prefetched simultaneously. Therefore, the reorganize algorithm needs to be invoked only when the system employs a zigzag block schedule.

5.3.2 Memory Reorganization

After batch updates, the memory distribution may become disrupted. Therefore, an algorithm that can reorganize the memory distribution is necessary. Consequently, we developed the reorganize algorithm based on the two-pointer technique to address potential disruptions arising during the batch update steps. The pseudocode of the reorganize algorithm is shown in Alg. 3. In the reorganize algorithm, it has 3 main steps.

Step 1 is between lines 7 to 10, we initialize two pointers in the upper and lower bounds of the memory layer. This initialization step establishes a foundational framework, ensuring that the algorithm operates simultaneously from both ends of the memory. By positioning the pointers at the memory's extremes, we enable a balanced and efficient traversal of the available memory space. This dual-pointer approach facilitates more effective memory management and helps

optimize the algorithm's performance by reducing the time required for memory allocation and retrieval operations. The initialization of these pointers is crucial for ensuring that the algorithm can function in parallel and efficiently manage the available memory resources.

Step 2 is from line 11 to 23, two pointers move towards each other, traversing all memory blocks before they intersect. Blocks belonging to the pet model encountered by the lower pointer and blocks occupied by the KV cache encountered by the upper pointer are swapped. This step ensures that the pet model is stored in the upper memory and the KV cache in the lower memory. However, empty memory blocks may affect the continuity of storage of these parameters. Thus, the low pointer and high pointer are also used to track empty memory blocks at the lower and higher ends, respectively.

Step 3 is from line 24 to 35, we copy the memory blocks encountered into the recorded empty spaces after these two pointers intersect, thus managing similar-type parameters in contiguous spaces. This approach helps streamline memory management and ensures efficient use of memory resources.

This two-pointer algorithm ensures that memory pages are reorganized with as few swaps as possible. However, it also has its limitations. When most of the disordered data reside in slower memory, such as a disk, the swap operation can become very slow. Therefore, when the offload ratio between disk and CPU memory is high, we can skip the reorganization step to maintain higher efficiency.

Algorithm 3 Memory Layer re-organize

Input:
  $memory\_blocks$: Array of memory blocks
  n: Total number of blocks
  $model\_weight\_block\_number$: Number of blocks dedicated to model weights
Output:
  Re-organized memory blocks in $memory\_blocks$
$low\_ptr \leftarrow 0$
$high\_ptr \leftarrow n - model\_weight\_block\_number$
$high\_list \leftarrow list()$
$low\_list \leftarrow list()$
while $high\_ptr > low\_ptr$ **do**
  $high \leftarrow memory\_blocks[high\_ptr]$
  $low \leftarrow memory\_blocks[low\_ptr]$
  if $high.is\_KV()$ and $low.is\_PET()$ **then**
    SWAP($high$, low)
  else if not $high.is\_KV()$ **then**
    record high to high_list if high is empty block
    $high\_ptr = high\_ptr - 1$
  else if not $low.is\_PET()$ **then**
    record high to low_list if high is empty block
    $low\_ptr = low\_ptr + 1$
  end if
end while
while $low\_ptr < n - model\_weight\_block\_number$ **do**
  $low\_ptr = low\_ptr + 1$
  if $memory\_blocks[high\_ptr].is\_PET$ **then**
    Swap($high\_list.pop(0)$)
  end if
end while
while $high\_ptr > 0$ **do**
  $high\_ptr = high\_ptr - 1$
  if $memory\_blocks[high\_ptr].is\_KV$ **then**
    then Swap($low\_list.pop(0)$)
  end if
end while

CHAPTER 6

Experiments and Evaluation

In this section, we will introduce a comprehensive set of experiments designed to evaluate the performance and efficiency of our proposed FixGen solution. Our experiments aim to validate the effectiveness of the technologies and strategies we introduced, including request batching, model integration, scheduling of Attention operators, and parameter prefetching.

We conducted experiments on models of various sizes but with the same structure (such as OPT-6.7B, OPT-13B, and OPT-30B) on two machines with different hardware specifications to demonstrate FixGen's versatility and adaptability. We also compared FixGen's performance with that of other state-of-the-art inference service systems.

6.1 Setup

In this section, we will introduce the experimental setup, including the implementation of the code, the hardware used, the evaluation metrics, and the models adopted. These components provide the foundational context for assessing subsequent experimental results, ensuring reproducibility and reliability. The code implementation covers the design of core algorithms and modules, while the hardware setup includes key resources such as GPU, CPU, memory, and storage devices. Evaluation metrics such as inference latency, throughput, and memory utilization are selected to comprehensively assess system performance. The models used in the experiments include details of their architecture, parameter scale, and relevance to the experimental tasks, offering technical support to achieve the experimental objectives.

6.1.1 Implementation

The core functionality of FixGen is implemented using PyTorch 2.2.1, leveraging its flexible dynamic computation graph and efficient tensor operations. To meet the demands of high-performance computing, we have optimized the implementation utilizing CUDA stream technology. This allows for asynchronous overlapping of operations, significantly improving the efficiency of computation and data transfer. By carefully orchestrating these overlaps, the system minimizes idle time and maximizes throughput, ensuring optimal performance for large-scale and real-time applications.

For memory system operations, we chose PyTorch to handle memory allocation, leveraging its efficient memory management mechanisms and user-friendly interfaces. Based on this foundation, we further optimized various memory operation modes by designing a hierarchical memory allocation strategy to reduce GPU memory fragmentation. In addition, a cache coherency mechanism was implemented to ensure the efficiency and accuracy of data transfers between different computing units. For critical memory operations, we developed custom operators using CUDA and C++, whose highly optimized implementations significantly enhance overall system performance.

In the model's attention mechanism, we made extensive modifications to the flash-attention and token-attention modules based on Triton in LightLLM, ensuring seamless compatibility with our custom memory system. Specifically, we expanded the functionality of flash attention to support a wider range of input formats and improved the implementation of token attention to accommodate general attention mechanisms. These enhancements significantly improved the system's performance when handling LLMs.

In addition, for parallel computing, we integrated the Punica bgmv kernel and conducted experimental optimizations on several performance-critical PET (Parameter Efficiency Transformer) models. The Punica bgmv kernel incorporates the latest features of the CUTLASS library, such as efficient matrix multiplication algorithms for different adapters simultaneously [Che+24; Tha+23]. These optimizations delivered exceptional parallel acceleration across models of varying scales.

Finally, we integrated the framework written in Python with kernels written in other languages using pyBind11. The setup file listed below shows how we use pyBind to integrate FixGen.

First, in the C++ entry file, we use PYBIND11_MODULE to link each kernel launcher function with its corresponding PyTorch extension name. The subsequent Python files will then call the respective functions using their PyTorch extension names.

```
// fixpoolops.cc
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("updatekv", &updatekv, "");
  m.def("add_lora", &bgmv, "");
}
```

Next, we need to include information about this extension module in the Python setup file, including the source code files for the module and the required compilation settings. Since we are using both C++ and CUDA, we set cxx and nvcc as extra compile arguments in the setup configuration. The code is listed below.

```
# setup.py
ext_modules = []
ext_modules.append(
    torch_cpp_ext.CUDAExtension(
        name="fixgen._kernels",
        sources=[
            "csrc/fixpoolops.cc",
            "csrc/fixpool/kvupdate.cu",
            "csrc/bgmv/bgmv_all.cu",
        ],
        extra_compile_args={
            "cxx": ["-O3"],
            "nvcc": ["-O3"],
        },
```

```
    )
)
```

Finally, we integrate the information of this extension module into the overall setup framework, thereby combining programs from different languages into the same project. This allows for seamless interaction between the C++, CUDA, and Python components within the unified project structure.

```
# setup.py
setuptools.setup(
    ext_modules=ext_modules,
    name = "fixgen",
    version="0.0.1",
    cmdclass={"build_ext": torch_cpp_ext.BuildExtension},
)
```

6.1.2 Hardware

For our experiments, we used two distinct hardware sets to evaluate FixGen performance. The primary test system was equipped with an AMD Ryzen Threadripper PRO 3945WX 12-core CPU, 128GB of RAM, and an Nvidia A5000 GPU with 24GB of memory, connected via PCIe with a data transfer rate of 16GT/s. Additionally, we performed performance tests on a secondary hardware setup, which featured an AMD Ryzen Threadripper PRO 5955WX 16-core CPU, 128GB of RAM, and an Nvidia A6000 GPU with 48GB of memory, also utilizing PCIe with a data transfer rate of 16GT/s. Although we do not have strict requirements for the CPU, a GPU that supports CUDA programming is essential for efficient execution of our inference tasks. Both systems were operated on Ubuntu 20.04 and 22.04, providing a stable and efficient environment for running our experiments and benchmarking FixGen's performance across different configurations.

6.1.3 Model

In this study, we have chosen to utilize the OPT model [Zha+22a], developed by Facebook AI (now Meta AI). The OPT model is part of a suite of LLMs based on the Transformer architecture, designed to enhance the transparency and accessibility of artificial intelligence research through open-source sharing. These models have been pre-trained on a broad spectrum of textual data. By learning linguistic patterns within these data, the OPT model is capable of performing a variety of complex natural language processing tasks, ranging from text generation to semantic understanding.

TABLE 6.1: Models and Parameters

Model	Parameters
OPT-6.7b	num_layer=32, num_head=32, hidden_size=4096, intermediate_size = 16384
OPT-13b	num_layer=40, num_head=40, hidden_size=5120 , intermediate_size = 20480
OPT-30b	num_layer=48, num_head=48, hidden_size=7168 , intermediate_size = 28672

In our experiments, we used three different scales of the OPT model: OPT-6.7b, OPT-13b, and OPT-30b, where "b" denotes the number of parameters in billions, their detail is shown in Table 6.1. These models are designed to accommodate varying computational capabilities and performance needs by adjusting the model size, allowing researchers to select the most appropriate model based on specific tasks and available resources. The choice to use the OPT model for our experiments was influenced by its openness, high performance, and exemplary performance across multiple natural language processing benchmarks. These characteristics make the OPT model an ideal candidate for testing our code auto-generation correction tool, FixGen, as well as assessing the performance of other benchmark models.

In our experimental setup, we chose not to test LLMs, such as OPT-175b, primarily because the model size of OPT-30b already approaches the upper limit in a single GPU environment, providing sufficient computational power and processing speed. Employing a larger model, like OPT-175b, could potentially enhance certain performance metrics; however, its high demand for computational resources and the consequent significant latency would detract from the experiment's efficiency and practicality.

Moreover, we did not utilize other types of LLMs that feature different architectures or training methodologies. Our approach is designed to ensure compatibility with a broad range of LLMs, particularly those with similar structures and functions. Therefore, while our experiments directly used the OPT series models, the techniques and methods developed are theoretically applicable to models with similar transformer-based architectures, such as GPT [Ope+24] and Llama [Tou+23a]. This design not only validates the effectiveness of our methods but also demonstrates their potential for wide applicability.

6.1.4 Baseline

We used FlexGen [She+23a], DeepSpeed Inference [Ami+22], and HuggingFace Accelerate [Gug+22] as baselines, which are State-Of-The-Art systems that can run Facebook OPT models in a single GPU environment despite insufficient GPU memory. In addition, we modified FlexGen to adapt the conventional batching strategy and denote it as Flex+.

- “Flex" refers to FlexGen, an inference system specifically designed for LLMs to operate efficiently in resource-limited environments. FlexGen addresses the challenges of running computationally and memory intensive LLMs on devices with constrained resources, such as a single GPU setup. By leveraging advanced memory management and scheduling techniques, it optimizes resource utilization to enable high-throughput inference. It achieves this by strategically managing memory allocation and transfers, reducing memory bottlenecks typically encountered during LLM inference. Its scheduling mechanisms ensure that computation and memory operations are efficiently overlapped, allowing for smoother and faster inference even when operating on devices with limited GPU Memory.
- “Flex+" is an enhanced system built upon FlexGen, designed to better accommodate traditional batch processing strategies while optimizing response times for online requests. Compared to the original FlexGen, Flex+ introduces significant modifications to the batching strategy to improve responsiveness for online tasks. Fig. 5.1 shows the process of the conventional batching approach. In Flex+, when the system receives an online request, it temporarily pauses the ongoing batch processing to

prioritize the prefetching operation. After prefetching has been completed, the online request is dynamically integrated into the current batch processing. This mechanism allows online tasks to achieve faster response times, thereby delivering a smoother user experience for online requests.

- DeepSpeed Inference, abbreviated as "Deeps," is a high-performance serving system specifically designed for LLMs to address inference needs in resource-constrained environments. It leverages the innovative ZeRO (Zero Redundancy Optimizer) inference technology, which significantly reduces memory usage through distributed memory optimization, enabling LLMs to run on limited hardware resources. DeepSpeed Inference integrates advanced memory management strategies and efficient deep learning kernels, enhancing inference efficiency while maintaining model performance. By partitioning parameters across devices and intelligently scheduling memory access, Deeps maximizes hardware utilization. Additionally, its built-in optimizer supports mixed-precision inference and dynamic batch scheduling, further reducing memory requirements and computational costs.
- HuggingFace Accelerate, abbreviated as "Acc," is an efficient inference library designed to simplify and optimize device configuration and parallelization strategies for model inference. By automatically detecting hardware resources and intelligently adjusting distributed computing configurations, Acc offers a user-friendly and flexible solution, particularly well-suited for resource-constrained inference scenarios. One of Acc's key features is support for parameter offloading, which dynamically allocates model parameters across different devices (such as CPUs and GPUs), significantly reducing the memory load on individual devices. This capability provides users with a streamlined interface, eliminating the complexity of traditional inference system configurations. Moreover, Acc's built-in parallelization strategies enable efficient computational load distribution in multi-device environments, enhancing performance for LLM inference. In our experiments, we built a flexible model inference server based on Acc. This server allowed us to easily manage inference tasks, test various hardware configurations, and evaluate parallelization schemes.

This highly efficient experimental setup not only significantly shortened development cycles but also provided convenient tools for model performance tuning.

HeteGen [Zha+24] is a recently introduced LLM inference system focused on optimizing the efficiency of individual online requests, catering to latency-sensitive real-time applications. Its architecture simplifies batch processing logic by exclusively supporting inference tasks with a batch size of 1. This ensures that each request is processed immediately without waiting for batch filling, offering significant advantages in single online request scenarios. However, its ability to handle large-scale concurrent requests is relatively limited. Given HeteGen's restriction to batch size 1, it is not well suited for large-scale batch processing or mixed-task inference scenarios required in our experiments. Therefore, in our study, we excluded HeteGen from performance comparisons. This decision ensures the fairness and relevance of our experimental results, focusing on systems capable of supporting diverse batching strategies across various task scenarios.

6.1.5 Metrics

Our evaluation focused on two key performance metrics: latency and token throughput. Latency measures the time taken to process a request or a token, while token throughput quantifies the number of tokens handled per unit of time. These metrics are crucial for assessing the system's efficiency and capacity to manage varying workloads, providing insights into its responsiveness and operational speed under different testing scenarios.

6.2 End-to-End Results on Synthetic Workloads

We evaluate the average end-to-end response latency for sudden online requests that the systems can achieve on OPT-6.7b, OPT-13b and OPT-30b with the same request length of 512 without using the PET model on two hardware sets.

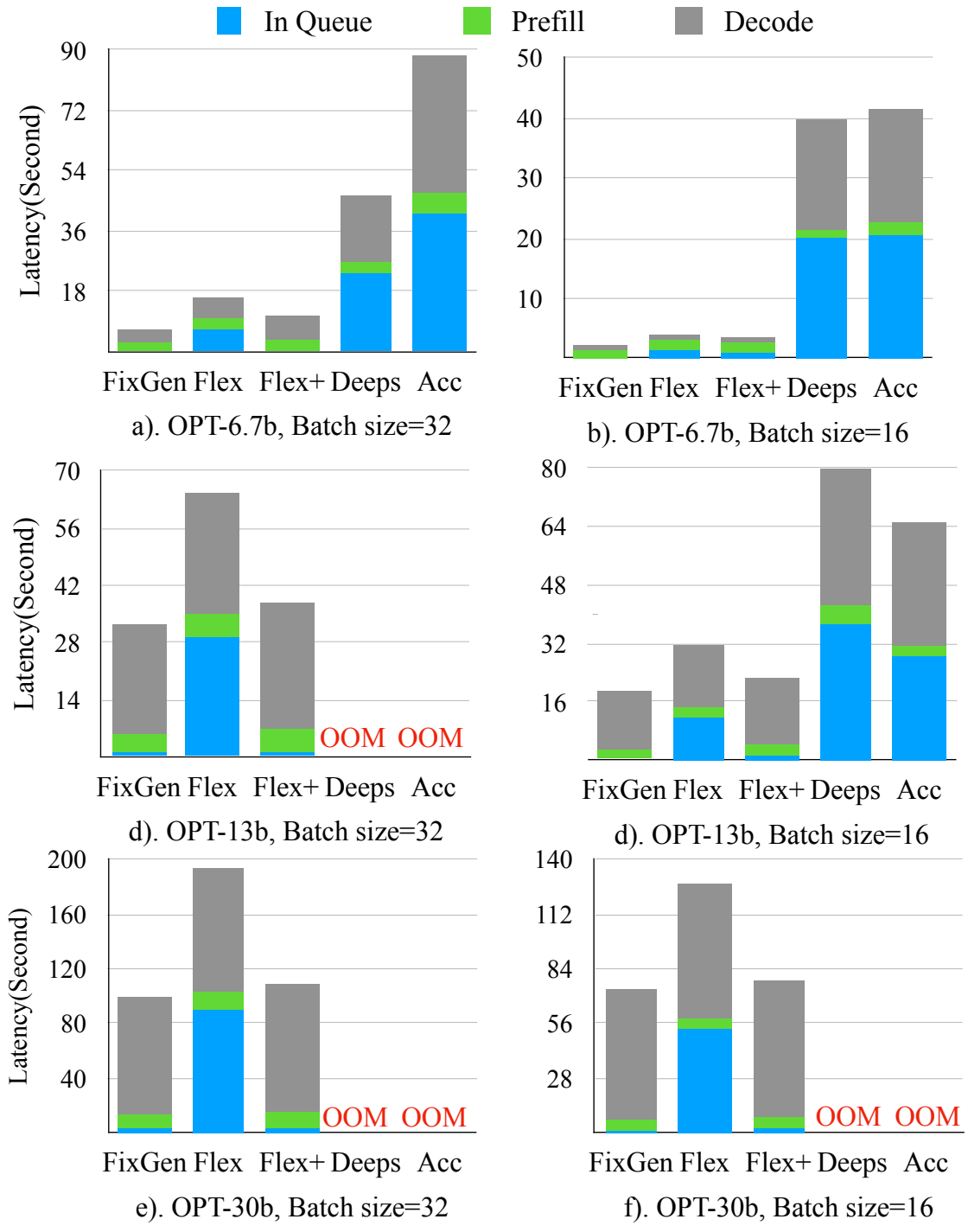

FIGURE 6.1: The average response latency (Second) for one sudden online request that can be achieved by a different system with different batch sizes from OPT-6.7b to OPT-30b. The prefill length=512 and decode length=32.

6.2.1 End-to-End Results on Primary Hardware Set

We first evaluate the average end-to-end response latency for sudden online requests that the systems can achieve on OPT-6.7b, OPT-13b, and OPT-30b with the same request length of 512 without using the PET model. We randomly generated 10 online requests in a scenario where the batch was fully loaded with offline requests and recorded the latency from the initiation to the completion of generating all tokens.

Fig. 6.1 shows the comparison of the average online request-response latency between FixGen and the baselines. In the diagram, we decompose the end-to-end latency into three parts: `in queue`, `prefill`, and `decode`, represented by blue, green, and gray colors, respectively. It is evident that compared to other systems, the `in queue` latency for online requests in FixGen is negligible. The response latency for online requests in FixGen is reduced by 1.82 compared to baseline FlexGen and reduced by 1.25 times that of FlexGen using a conventional strategy. In the case of high computation volume (large batch size, large model size), the boost is not as pronounced because FixGen, like other systems, cannot avoid many I/O operations, and the overall boost is diluted, only 1.06 times better than the baseline. This demonstrates that fixgen can improve performance by reducing `in queue` latency even in edge cases, and that these edge cases are not common in resource-limited environments.

The result for A6000 is given in Section 6.2.2. FlexGen+ directly processes sudden online requests, reducing `in queue` latency. However, this requires pausing the inference of requests that are in the decoding stage, theoretically reducing throughput. Therefore, we also compared the throughput of various systems under the same workload. We compared the throughput for OPT-6.7b, OPT-13b, and OPT-30b under different memory constraints across various systems to validate the throughput capabilities of FixGen where `batch size`= 32, `prefill length`=512 and `decode length`=32. The result is shown in Fig. 6.2. Since DeepSpeed and Accelerate do not allow modifications to the offloading ratio, each is depicted as a single point on the graph. FixGen and FlexGen are designed to dynamically adjust the offload ratio. Due to the different ways they control the offload distribution, we use the amount of GPU memory they occupy as a unified standard for comparison.

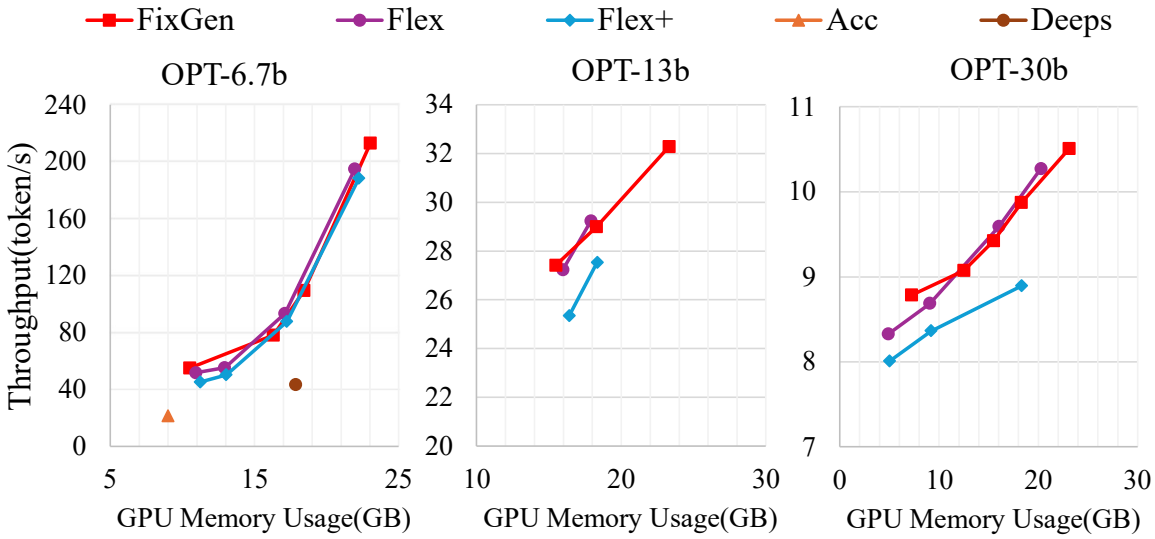

FIGURE 6.2: Throughput comparison under different memory constraints for OPT models from 6.7B to 30B with `batch size`=32, `prefill length`=512, `decode length`=32. DeepSpeed Inference cannot set setting the parameter offloading ratio. Accelerate and DeepSpeed Inference are out of memory for OPT-13B and OPT-30B.

Although Flex+ has improved the processing speed for online requests, its overall throughput has been negatively impacted. Meanwhile, FixGen can maintain a throughput comparable to FlexGen with a similar amount of GPU memory usage. When OPT-13b and OPT-30b are deployed using FlexGen, the maximum GPU memory usage is 17.11GB and 21.96GB, respectively, which does not fully utilize all available GPU memory. This limitation is due to FlexGen's method of allocating storage per network layer, which does not allow precise control over the offloading ratio. In contrast, FixGen offers more precise control to achieve greater utilization of GPU memory, which allows more efficient inference.

We compared the throughput for OPT-6.7b, OPT-13b and OPT-30b under different memory constraints across various systems to validate the throughput capabilities of FixGen where `batch size`=16, `prefill length`=512, and `decode length`=32 on A5000. The result is shown in Fig. 6.3. It gives a similar pattern as the result with batch size=32: The overall throughput of Flex+ has been negatively impacted while FixGen has a similar throughput to FlexGen.

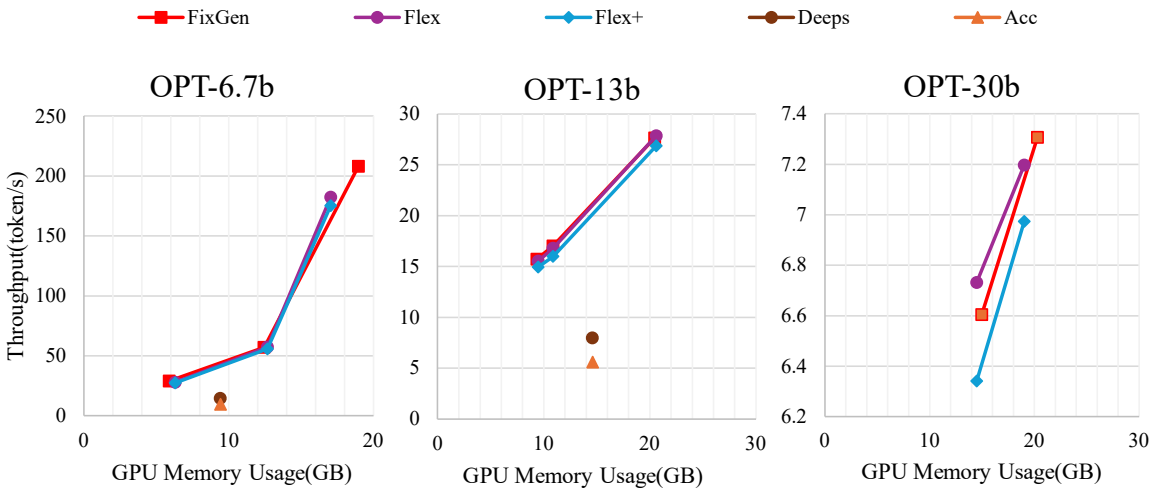

FIGURE 6.3: Throughput comparison under different memory constraints for OPT models from 6.7B to 30B with batch size=16, prefill length=512, decode length=32. DeepSpeed Inference cannot set setting the parameter offloading ratio. Accelerate and DeepSpeed Inference are out of memory for OPT-13B and OPT-30B.

The outcomes of the two experiments suggest that while FixGen has effectively reduced the response latency for online requests, it has not diminished the system's throughput.

6.2.2 End-to-End Latency on A6000 set

Fig. 6.4 shows the comparison of the average online request-response latency between FixGen and the baselines. The result shows a pattern similar to that in A5000. FixGen reduces total latency by reducing the `in queue` latency. Nvidia A6000 provides powerful performance that makes the computation faster, but I/O remains the same. Therefore, in scenarios that require extensive prefetching, the latency remains high. The latency for OPT-13b is low on the A6000 because this GPU has a larger memory capacity, which can fully accommodate OPT-13B and its associated parameters, thus eliminating the need for prefetching.

This result reflects that, on high-performance computing platforms, despite the enhancements in computational speed, the limitations in I/O processing speeds may still be a bottleneck affecting system response times. This is particularly pertinent when using LLM such as OPT-13b, where sufficient memory capacity allows the entire model and its parameters to

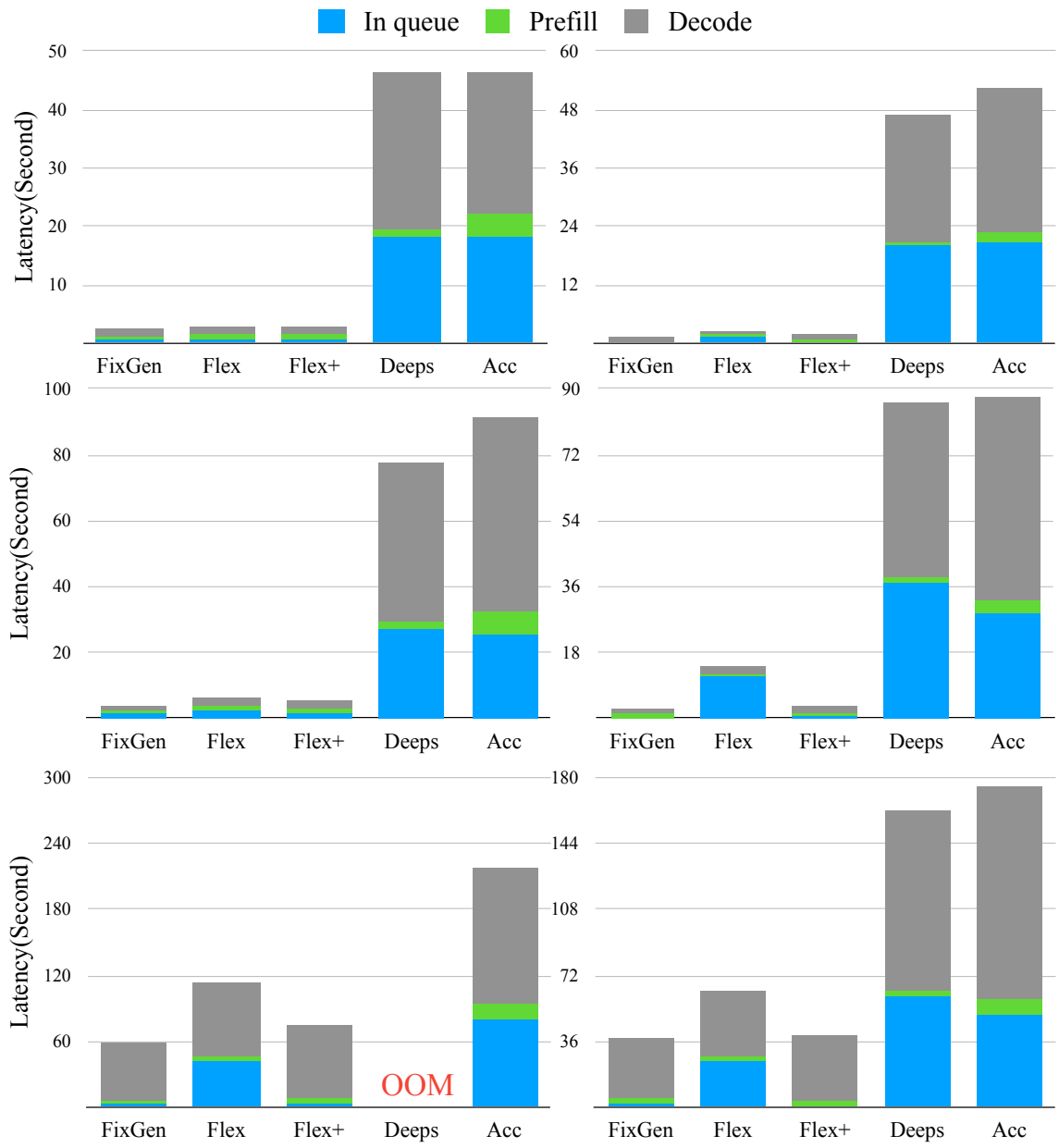

FIGURE 6.4: The average response latency (Second) for one sudden online request that can be achieved by a different system with different batch sizes from OPT-6.7b to OPT-30b in A6000. The prefill length=512 and decode length=32.

be loaded directly. This can significantly reduce the latencies caused by data transfers, thus enhancing the overall response efficiency of the system. This aspect is especially critical for the design of real-time application systems that require rapid responses, highlighting the crucial role of appropriate hardware resource allocation in performance optimization.

6.3 Prefetch Latency

TABLE 6.2: The Latency(Second) for prefetching the model from the CPU memory layer by layer when all parameters offload to the CPU memory.

MODEL	OPT-6.7B	OPT-13B	OPT-30B
TRADITIONAL	0.630	1.396	3.079
FIXGEN	**0.527**	**1.018**	**2.369**

We compared the total prefetch latency of traditional systems (including FlexGen, DeepSpeed, and Accelerate) that manage memory by network layer with that of FixGen. To ensure fairness of the experiments, we utilized the copy operator provided by PyTorch for different offloading strategies and stored all parameters in the CPU memory. The result is recorded in Table 6.2. FixGen requires only 2.369 seconds to prefetch all parameters, which is 1.30$\times$ faster than the traditional approach. This strategy optimizes data transfer and storage efficiency, reducing the overhead associated with handling large data sets and frequent parameter access.

6.4 Multi-tasks Request Response Latency

In FixGen, we recorded the average response latency under sudden online requests, simultaneously executing n burst online requests, where $n \in 1, 4, 6, 8$, covering various task types. The requests were generated randomly and assigned to the adapters. Each task used a LoRA model set to rank=16, assigned per request. The experimental results are shown in Table 6.3. When multiple online requests are initiated simultaneously within the same batch, the average latency does not exhibit significant changes as the number of task types increases. This indicates that handling multiple tasks does not introduce substantial additional computational stress, demonstrating that FixGen is suitable for multi-task environments and exhibits efficient memory management capabilities in handling parameters of different PET (Pre-trained Enhancement Transformer) models.

This experiment demonstrates that even under conditions of concurrent multi-task processing, FixGen effectively manages memory resources, ensuring that parameters of LoRA models for

different tasks are appropriately allocated and processed. This maintains low latency response times, showcasing its advantages in efficient processing and resource management.

TABLE 6.3: The Latency(Second) when efficiently use GPU memory with prompt length=512, batch size=16, decode length=32 with different number of task types

MODEL	OPT-6.7B	OPT-13B	OPT-30B
1	1.872	16.528	72.247
4	2.045	16.867	72.248
6	2.443	17.032	72.253
8	2.753	17.343	72.643

6.5 Additional Experiment

In this section, we present the results of additional experiments, which validate the effectiveness of each module discussed above and confirm the validity of the experiments.

6.5.1 Runtime Breakdown

Table 6.4 shows the breakdown of OPT-30b in FixGen when the number of PET models is 1 on A5000. In this experiment, all parameters are stored in the CPU memory. Since these three operations can be executed in parallel within the system, the one with the highest latency becomes the decisive factor for overall latency. The greatest latency arises from loading the Memory Layer from the CPU memory. Despite the hybrid computing approach, which includes prefilling and decoding, the latency from the computation operations does not add to the overall latency.

TABLE 6.4: Runtime break down Latency(second) for one decode layer

STAGE	COMPUTATION	LOAD	STORE
PREFILL	0.0588	0.0247	0.00524
DECODE	0.00243	0.0215	0.00014

6.5.2 Ablation Study

We tested the runtime of the phase aggregation component using hybrid computing only and general attention only under the same tasks when offloading all parameters to the CPU memory. We compared these with phase aggregation using a router. Table 6.5 shows the average latency for completing 16 requests with prompt length 512 in a batch on A5000. the `no router A` denotes the system is using general attention only, and the `no router B` means using hybrid computation only.

TABLE 6.5: Ablation study of proposed techniques. The latency (second) when efficiently using GPU memory with prompt length in range [32,512] batch size=16, decode length=32

MODEL	OPT-6.7B	OPT-13B	OPT-30B
ALL OPTIMIZATION	1.872	18.763	72.247
NO ROUTER A	2.431	20.896	73.020
NO ROUTER B	2.058	19.980	72.809
NO CONTINUOUS MEMORY	1.856	19.616	77.258

This result indicates that different computational strategies significantly impact system response times when processing batch requests. Systems employing a standard attention mechanism `no router A` versus those using hybrid computations `no router B` exhibit performance differences, reflecting the computational efficiency of different approaches when handling complex tasks. Additionally, by integrating a router to manage data flows and task distribution, the performance of the phase aggregation component is further optimized. This not only maintains or enhances processing speed, but also effectively manages data transfers between CPUs and GPUs, reducing latency. All modules involved in the ablation test positively contributed to system performance.

CHAPTER 7

Conclusion

In conclusion, our research introduces FixGen, a single-GPU online-offline hybrid inference service system for LLMs that supports multitasking inference. FixGen is specifically engineered to address the critical needs of latency reduction and throughput enhancement in multitasking with LLMs on resource-constrained devices. FixGen reduces online request-response latency by 1.06 to 1.84 times compared to traditional methods, while maintaining offline request throughput for models ranging from OPT-6.7b to OPT-30b on a single GPU. FixGen strategically conserves I/O resources and enhances batch synthesis capabilities, delivering a robust general-purpose solution even in scenarios where the hardware, such as advanced GPUs, cannot fully accommodate all model parameters.

In current LLM deployment solutions, models are typically configured in cloud environments, where LLM computations are performed remotely in response to user requests. However, local deployment of LLMs is emerging as both a trend and a necessity. Due to the limited GPU memory of consumer-grade personal computers and edge devices, which cannot accommodate all model parameters simultaneously, performing multitask LLM inference presents a significant challenge. One solution to this challenge is to offload unused parameters to main memory and prefetch them as needed. However, these methods suffer from significant latency due to data transfer overhead.

Thus, reducing latency for online requests and improving throughput for offline requests are two critical factors that determine whether LLMs can be successfully deployed and optimized on resource-constrained devices, such as personal computers and edge devices. This, in turn, enhances the practical utility of personal computers and opens up new possibilities. If the challenges of LLM deployment on resource-limited devices can be effectively addressed,

personal computers and other edge devices are more likely to enter a new era, capable of performing a wider range of essential tasks for users. This is the future that we are committed to creating and look forward to.

7.1 Future Outlook

As artificial intelligence technology continues to advance, personalized LLMs are expected to become a key trend in the future development of technology. Traditional LLMs are typically general-purpose, but with the diversification and personalization of application demands, there is an increasing tendency to favor customized models. This trend has led to a growing demand for customize LLMs, particularly in fields with specific needs, such as healthcare, education, and customer service. To address this demand, efficient deployment and operation of these models on resource-constrained devices will become a focal point of technological research.

The methods for personalized LLMs are also evolving, with approaches such as the mix of experts (MoE) model. In this approach, different parts of the model dynamically select various "expert" submodels based on the characteristics of the input data and task requirements, enabling more efficient inference and computation. This not only reduces redundant calculations, but also allows flexible adjustments according to different scenarios, maximizing the use of computational resources. For example, when handling text generation tasks, certain experts may specialize in generating text for a specific domain, while in other tasks, the system can automatically select a more appropriate expert for the computation. The use of MoE can effectively reduce the computational overhead of the model and provide more targeted solutions, facilitating the deployment of LLMs on personal devices. In the future, FixGen needs to adapt to advanced customized LLM methods, contributing to the construction of a robust ecosystem. The technology behind FixGen will not be limited to its current implementation, but will continue to optimize and adapt to more complex and diverse task requirements. Especially in scenarios such as consumer-grade AI personal computers, edge devices, and smart homes, LLMs will be able to perform more tasks, including automated

recommendations, natural language interaction, and intelligent monitoring, thus enabling more intelligent and personalized services.

In the other hand, many LLM inference tasks currently rely on the Retrieval-Augmented Generation (RAG) method, which separates the model from external knowledge bases. RAG allows for dynamic retrieval of relevant knowledge in specific scenarios based on task requirements, avoiding the enormous computational and storage burden of pre-storing all knowledge within the model. In the future, as technology continues to evolve, achieving efficient RAG implementation in a single GPU environment will be a key factor in advancing the widespread use of LLMs. By further optimizing data retrieval and processing workflows, personal computers could potentially provide low-latency, high-efficiency inference services without relying on powerful cloud computing resources.

Finally, as hardware costs continue to decline, future personal computers may be equipped with multiple GPUs, NPUs, or even inference accelerators from different manufacturers and models. Given the varying computational capabilities of these accelerators, exploring how to leverage them to deliver the best possible user experience remains a promising research direction.

Through the FixGen system, we envision a future where artificial intelligence technology enters every home and industry, ushering in a new chapter for an intelligent society.

Bibliography

[Ami+22] Reza Yazdani Aminabadi et al. 'DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale'. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022.

[Ang24] Nadezhda Angelova. 'The capabilities of the art-oriented artificial intelligence Adobe Firefly and its visual advantages and disadvantages'. In: *Journal" Fundamental Sciences and Applications"* 30.1 (2024), pp. 1–10.

[Ano23] Anonymous. *ModelTC. Lightllm: Python-based LLM inference and serving framework*. 2023. URL: https://github.com/ModelTC/lightllm.

[Ans+24] Jason Ansel et al. 'PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation'. In: *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. URL: https://pytorch.org/assets/pytorch2-2.pdf.

[BGR22] Elad Ben Zaken, Yoav Goldberg and Shauli Ravfogel. 'BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models'. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Ed. by Smaranda Muresan, Preslav Nakov and Aline Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 1–9. DOI: 10.18653/v1/2022.acl-short.1. URL: https://aclanthology.org/2022.acl-short.1.

[Bor23] Ali Borji. *Generated Faces in the Wild: Quantitative Comparison of Stable Diffusion, Midjourney and DALL-E 2*. 2023. arXiv: 2210.00586 [cs.CV]. URL: https://arxiv.org/abs/2210.00586.

[Bro+20] Tom Brown et al. 'Language Models are Few-Shot Learners.' In: *Advances in Neural Information Processing Systems*. 2020, pp. 1877–1901.

[Che+14] Sharan Chetlur et al. 'cuDNN: Efficient Primitives for Deep Learning'. In: *ArXiv* abs/1410.0759 (2014). URL: https://api.semanticscholar.org/CorpusID:12330432.

[Che+18] Tianqi Chen et al. 'TVM: An Automated End-to-End Optimizing Compiler for Deep Learning'. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2018. URL: https://api.semanticscholar.org/CorpusID:52939079.

[Che+24] Lequn Chen et al. 'Punica: Multi-Tenant LoRA Serving'. In: *MLSys*. 2024.

[Cho15] François Chollet. *keras*. https://github.com/fchollet/keras. 2015.

[Dao+24] Tri Dao et al. 'FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness'. In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. NIPS '22. New Orleans, LA, USA: Curran Associates Inc., 2024. ISBN: 9781713871088.

[Dev+19] Jacob Devlin et al. 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[Din+23] Ning Ding et al. 'Parameter-efficient fine-tuning of large-scale pre-trained language models'. In: *Nature Machine Intelligence* 5 (Mar. 2023), pp. 1–16. DOI: 10.1038/s42256-023-00626-4.

[Du+22] Jiangsu Du et al. *EnergonAI: An Inference System for 10-100 Billion Parameter Transformer Models*. 2022. arXiv: 2209.02341 [cs.LG].

[Ega+24] Kazuki Egashira et al. *Exploiting LLM Quantization*. 2024. arXiv: 2405.18137 [cs.LG]. URL: https://arxiv.org/abs/2405.18137.

[EM23] Artyom Eliseev and Denis Mazur. *Fast Inference of Mixture-of-Experts Language Models with Offloading*. 2023. arXiv: 2312.17238 [cs.LG]. URL: https://arxiv.org/abs/2312.17238.

[Fan+21] Jiarui Fang et al. 'TurboTransformers: an efficient GPU serving system for transformer models'. en. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Virtual Event Republic of Korea: ACM, Feb. 2021, pp. 389–402. ISBN: 978-1-4503-8294-6. DOI: 10.1145/3437801.3441578. URL: https://dl.acm.org/doi/10.1145/3437801.3441578.

[Gar+21] Stephan J. Garbin et al. 'FastNeRF: High-Fidelity Neural Rendering at 200FPS'. In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2021, pp. 14326–14335. DOI: 10.1109/ICCV48922.2021.01408. URL: https://doi.ieeecomputersociety.org/10.1109/ICCV48922.2021.01408.

[Git21] GitHub. *GitHub Copilot*. https://copilot.github.com/. Accessed: March 24, 2023. 2021.

[Gua+23] Lei Guan et al. 'PipeOptim: Ensuring Effective 1F1B Schedule with Optimizer-Dependent Weight Prediction'. In: *CoRR* abs/2312.00839 (2023). DOI: 10.48550/ARXIV.2312.00839. arXiv: 2312.00839. URL: https://doi.org/10.48550/arXiv.2312.00839.

[Gug+22] Sylvain Gugger et al. *Accelerate: Training and inference at scale made simple, efficient and adaptable.* https://github.com/huggingface/accelerate. 2022.

[He+21] Junxian He et al. 'Towards a Unified View of Parameter-Efficient Transfer Learning'. In: *ArXiv* abs/2110.04366 (2021). URL: https://api.semanticscholar.org/CorpusID:238583580.

[Hou+19] Neil Houlsby et al. 'Parameter-Efficient Transfer Learning for NLP'. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 2790–2799. URL: https://proceedings.mlr.press/v97/houlsby19a.html.

[Hu+22] Edward J Hu et al. 'LoRA: Low-Rank Adaptation of Large Language Models'. In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=nZeVKeeFYf9.

[Hu+23] Zhiqiang Hu et al. 'LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models'. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor, Juan Pino and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 5254–5276. DOI: 10.18653/v1/2023.emnlp-main.319. URL: https://aclanthology.org/2023.emnlp-main.319.

[Jou+17] Norman P. Jouppi et al. 'In-Datacenter Performance Analysis of a Tensor Processing Unit'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246. URL: https://doi.org/10.1145/3079856.3080246.

[KKK22] Donghyeon Kim, Inseo Kim and Jinsung Kim. 'Analysis of Sub-Routines in NVIDIA cuBLAS Library for a series of Matrix-Matrix Multiplications in Transformer'. In: *2022 13th International Conference on Information and Communication Technology Convergence (ICTC)*. 2022, pp. 618–620. DOI: 10.1109/ICTC55196.2022.9952498.

[Kwo+23] Woosuk Kwon et al. 'Efficient Memory Management for Large Language Model Serving with PagedAttention'. In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 611–626. ISBN: 9798400702297. DOI:

10.1145/3600006.3613165. URL: https://doi.org/10.1145/3600006.3613165.

[Li+20] Shen Li et al. 'PyTorch distributed: experiences on accelerating data parallel training'. In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 3005–3018. ISSN: 2150-8097. DOI: 10.14778/3415478.3415530. URL: https://doi.org/10.14778/3415478.3415530.

[Li+23] Zhuohan Li et al. 'AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving'. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pp. 663–679. ISBN: 978-1-939133-34-2. URL: https://www.usenix.org/conference/osdi23/presentation/li-zhouhan.

[Li+24a] Haohang Li et al. 'FinMem: A Performance-Enhanced LLM Trading Agent with Layered Memory and Character Design'. In: *ICLR 2024 Workshop on Large Language Model (LLM) Agents*. 2024. URL: https://openreview.net/forum?id=sstfVOwbiG.

[Li+24b] Shiyao Li et al. *Evaluating Quantized Large Language Models*. 2024. arXiv: 2402.18158 [cs.CL]. URL: https://arxiv.org/abs/2402.18158.

[Liu+22] Xiao Liu et al. 'P-Tuning: Prompt Tuning Can Be Comparable to Fine-tuning Across Scales and Tasks'. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Ed. by Smaranda Muresan, Preslav Nakov and Aline Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 61–68. DOI: 10.18653/v1/2022.acl-short.8. URL: https://aclanthology.org/2022.acl-short.8.

[Liu+23] Haotian Liu et al. *Visual Instruction Tuning*. 2023. arXiv: 2304.08485 [cs.CV]. URL: https://arxiv.org/abs/2304.08485.

[Liu+24a] Yuhan Liu et al. 'CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving'. In: *Proceedings of the ACM SIGCOMM*

2024 Conference. ACM SIGCOMM '24. Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 38–56. ISBN: 9798400706141. DOI: 10.1145/3651890.3672274. URL: https://doi.org/10.1145/3651890.3672274.

[Liu+24b] Zirui Liu et al. 'KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache'. In: *Proceedings of the 41st International Conference on Machine Learning*. Ed. by Ruslan Salakhutdinov et al. Vol. 235. Proceedings of Machine Learning Research. PMLR, 2024, pp. 32332–32344. URL: https://proceedings.mlr.press/v235/liu24bz.html.

[LL21] Xiang Lisa Li and Percy Liang. 'Prefix-Tuning: Optimizing Continuous Prompts for Generation'. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Ed. by Chengqing Zong et al. Online: Association for Computational Linguistics, Aug. 2021, pp. 4582–4597. DOI: 10.18653/v1/2021.acl-long.353. URL: https://aclanthology.org/2021.acl-long.353.

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[Nar+19] Deepak Narayanan et al. 'PipeDream: Generalized Pipeline Parallelism for DNN Training'. In: *ACM Symposium on Operating Systems Principles (SOSP 2019)*. Oct. 2019.

[Nar+21a] Deepak Narayanan et al. 'Efficient large-scale language model training on GPU clusters using megatron-LM'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476209. URL: https://doi.org/10.1145/3458817.3476209.

[Nar+21b] Deepak Narayanan et al. 'Memory-Efficient Pipeline-Parallel DNN Training'. In: *International Conference on Machine Learning (ICML 2021)*. July 2021.

[NVI15] NVIDIA. 2015. URL: https://developer.nvidia.com/gpudirect.

[NVI24] NVIDIA. 2024. URL: https://www.nvidia.com/en-au/geforce/graphics-cards/.

[Ope+24] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL]. URL: https://arxiv.org/abs/2303.08774.

[Pop+22] Reiner Pope et al. *Efficiently Scaling Transformer Inference*. 2022. arXiv: 2211.05102 [cs.LG]. URL: https://arxiv.org/abs/2211.05102.

[Rad+19] Alec Radford et al. 'Language Models are Unsupervised Multitask Learners'. In: (2019).

[Raj+20] Samyam Rajbhandari et al. 'ZeRO: memory optimizations toward training trillion parameter models'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[Ren+21] Jie Ren et al. 'ZeRO-Offload: Democratizing Billion-Scale Model Training'. In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 551–564. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/ren-jie.

[RN18] Alec Radford and Karthik Narasimhan. 'Improving Language Understanding by Generative Pre-Training'. In: 2018. URL: https://api.semanticscholar.org/CorpusID:49313245.

[She+23a] Ying Sheng et al. 'FlexGen: high-throughput generative inference of large language models with a single GPU'. In: *Proceedings of the 40th International Conference on Machine Learning*. ICML'23. Honolulu, Hawaii, USA: JMLR.org, 2023.

[She+23b] Ying Sheng et al. 'S-LoRA: Serving Thousands of Concurrent LoRA Adapters'. In: *arXiv preprint arXiv:2311.03285* (2023).

[Sil+16] David Silver et al. 'Mastering the game of Go with deep neural networks and tree search'. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961.

[Sti19] Sebastian U. Stich. 'Local SGD Converges Fast and Communicates Little'. In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=S1g2JnRcFX.

[Tha+23] Vijay Thakkar et al. *CUTLASS*. Version 3.0.0. Jan. 2023. URL: https://github.com/NVIDIA/cutlass.

[Tou+23a] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: https://arxiv.org/abs/2307.09288.

[Tou+23b] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[Vas+17] Ashish Vaswani et al. 'Attention is All you Need'. In: *Advances in Neural Information Processing Systems*. 2017.

[YTT23] Yi Yang, Yixuan Tang and Kar Yan Tam. *InvestLM: A Large Language Model for Investment using Financial Domain Instruction Tuning*. 2023. arXiv: 2309.13064 [q-fin.GN].

[Yu+22] Gyeong-In Yu et al. 'Orca: A Distributed Serving System for Transformer-Based Generative Models'. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 521–538. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/yu.

[Zha+22a] Susan Zhang et al. *OPT: Open Pre-trained Transformer Language Models*. 2022. arXiv: 2205.01068 [cs.CL]. URL: https://arxiv.org/abs/2205.01068.

[Zha+22b] Jie Zhao et al. 'Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization'. In: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 1–19.

[Zha+23] Yujia Zhai et al. 'ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs'. In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE

Computer Society, May 2023, pp. 344–355. DOI: 10.1109/IPDPS54959.2023.00042. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPS54959.2023.00042.

[Zha+24] Xuanlei Zhao et al. 'HeteGen: Efficient Heterogeneous Parallel Inference for Large Language Models on Resource-Constrained Devices'. In: *MLSys 2024, Proceedings of Machine Learning and Systems*. Ed. by P. Gibbons, G. Pekhimenko and C.M. De Sa. Vol. 6. 2024, pp. 162–172.

[Zho+22] Zhe Zhou et al. 'PetS: A Unified Framework for Parameter-Efficient Transformers Serving'. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 489–504. URL: https://www.usenix.org/conference/atc22/presentation/zhou-zhe.

1 Terminology List

- **KV Cache**: Key-Value Cache, typically used to store intermediate results generated during inference in a model, which helps accelerate subsequent computations, especially when processing sequential data.
- **Hybrid Batch**: A method that combines different sizes and phases of requests in a batch during inference to improve computational efficiency and resource utilization.
- **CPU**: Central Processing Unit, the primary processing unit of a computer responsible for executing program instructions and processing data.
- **GPU**: Graphics Processing Unit, a specialized processor designed to handle graphics and image processing, widely used in deep learning and parallel computing.
- **CUDA**: Compute Unified Device Architecture, a parallel computing platform and programming model developed by NVIDIA that allows developers to utilize GPUs for general-purpose computing.
- **NPU**: Neural Processing Unit, a hardware designed specifically to accelerate deep learning algorithms, optimizing the computational efficiency of neural networks.

- **TPU**: Tensor Processing Unit, a specialized integrated circuit developed by Google to accelerate machine learning tasks, particularly deep learning models within the TensorFlow framework.