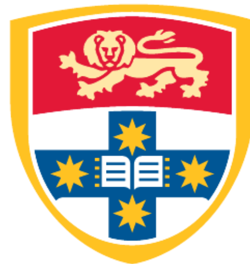
Integrated Transformers Inference Framework for Multiple Tenants on GPU

YUNING ZHANG

Master of Philosophy



THE UNIVERSITY OF
SYDNEY

Supervisor: Dong Yuan
Associate Supervisor: Wei Bao

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

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

14 August 2023

Abstract

In recent years, Transformer models have gained prominence in the deep learning domain, serving as the foundation for a wide array of applications, including Natural Language Processing (NLP) and Computer Vision (CV). These models have become essential for numerous inference tasks, but their implementation often faces challenges related to GPU utilization and system throughput. Typically, current GPU-based inference frameworks treat each model individually, which results in suboptimal resource management and decreased performance. Moreover, these frameworks do not consider the scenario of multiple tenants, which is different applications to share a single GPU.

To address these limitations, we introduce ITIF: Integrated Transformers Inference Framework for multiple tenants with a shared backbone. ITIF allows multiple tenants to share a single backbone Transformer model on a single GPU, consolidating operators from various multi-tenant inference models. This approach significantly optimizes GPU utilization and system throughput. Our proposed framework, ITIF, marks a considerable advancement towards enhancing the efficiency of deep learning, particularly for large-scale cloud providers hosting numerous models with a shared backbone.

In our experiments, we extensively evaluated the performance of ITIF in comparison with traditional baselines. We conducted tests on a variety of deep learning tasks, including NLP and CV tasks. We found that ITIF consistently outperformed the baselines, with improvements in performance by up to 2.40 times.

In conclusion, our research highlights the potential benefits of adopting the ITIF framework for improving the efficiency and scalability of Transformer-based deep learning systems. By enabling multiple tenants to share a single backbone model, ITIF provides an innovative solution to address the challenges faced by large-scale cloud providers in optimizing GPU utilization and system throughput. As such, ITIF presents a promising direction for further research and development in the field of deep learning.

Acknowledgements

I stand at the end of the fulfilling journey of completing this thesis, only to realize that this milestone has been a collective effort rather than an individual achievement. As I present my work, it is crucial for me to acknowledge the significant contributions of those who have been instrumental in making this journey both possible and successful.

Firstly, I extend my deepest gratitude to my supervisor, Dr. Dong YUAN, whose unwavering guidance and mentorship have been invaluable throughout my research journey. His boundless knowledge, thought-provoking critiques, and inspiring passion for our field have not only shaped this thesis but also my academic philosophy. I feel privileged to have learned under his tutelage, and I owe the success of this research in large part to his diligent supervision, unyielding support, and tireless commitment.

I would also like to express my appreciation to my associate supervisor, Dr. Wei BAO, whose professional guidance and constructive feedback were instrumental in the refinement of this thesis. His insightful inputs, although subtle, have had a significant impact on my research work.

Moving forward, I am profoundly grateful for my lab mates, Nan Yang and Zao Zhang. Their invaluable assistance, camaraderie, and consistent support were pivotal in the completion of my first paper and significantly contributed to my growth as a researcher. Our mutual exchange of ideas and collective problem-solving efforts have been a constant source of motivation and learning for me.

Furthermore, my sincere thanks goes to Fan Huang, Laicheng Zhong, and Yongkun Deng, whose companionship, shared insights, and assistance in various stages of my research have enriched this journey. Their contribution has been a key component in not only my academic pursuits but also in creating an enjoyable and supportive research environment.

I am fortunate to be blessed with loving and supportive parents whose unwavering belief in my capabilities has been my backbone throughout this journey. Their ceaseless

encouragement, endless patience, and unconditional love have been the foundation on which all my achievements rest.

In conclusion, although this thesis carries my name, it is the cumulative result of the constant guidance, untiring efforts, and relentless support of all the individuals mentioned above. I am deeply grateful for their contributions and consider it a privilege to have worked alongside them. Their collective wisdom and support have made this academic journey a rewarding and enlightening experience.

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.

Yuning Zhang Signature Date 14 August 2023

Authorship Attribution Statement

The content of this thesis has been accepted by the Fifty-Second International Conference on Parallel Processing. (ICPP'23). I designed the study, system, and algorithms and performed the experiments.

In addition to the statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

Student Name YUNING ZHANG Signature Date 14 August 2023

As the supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Supervisor Name DONG YUAN Signature Date 14 August 2023

List of Publication

Accepted Paper

Yuning Zhang, Zao Zhang, Wei Bao and Dong Yuan (2023) "ITIF: Integrated Transformers Inference Framework for Multiple Tenants on GPU" Accepted by the 52nd International Conference on Parallel Processing. ICPP'23. Salt Lake City, Utah, USA

Nan Yang, Dong Yuan, Yuning Zhang, Yongkun Deng and Wei Bao (2022) "Asynchronous Semi-Supervised Federated Learning with Provable Convergence in Edge Computing". Accepted by the IEEE Network, 2022, 36(5): 136-143.

Zao Zhang, Yuning Zhang, Dong Yuan, and Wei Bao (2022). A two-level architecture for deep learning applications in mobile edge computing. In Proceedings of the 17th ACM Workshop on Mobility in the Evolving Internet Architecture (pp. 43-48). Sydney, New South Wales, Australia

Papers that are under review

Zao Zhang, Yuning Zhang, Wei Bao, Changyang Li, and Dong Yuan (2023) "Coarse-to-Fine: A Hierarchical DNN Inference Framework for Edge Computing" Under review at the Future Generation Computer Systems.

Contents

Abstract ii

Acknowledgements iii

Statement of Originality v

Authorship Attribution Statement vi

List of Publication vii

Contents viii

List of Figures x

List of Tables xi

Chapter 1 Introduction 1

1.1 Motivation 1

1.2 Contributions 3

1.3 Thesis Overview 5

Chapter 2 Literature Review and Preliminaries 6

2.1 Deep Learning and Parallel Computing 8

2.1.1 Parallel Computing in Deep Learning 8

2.1.2 Accelerators for Parallel Computing 12

2.2 Inference Optimization for Multi-tenants in GPU 15

2.2.1 Kernel Level Optimization 16

2.2.2 Request Batching 18

2.2.3 Optimization for Multiple Tenants 20

2.3 The Enhancement for Transformers Inference 22

2.3.1 Transformer 23
2.3.2 Optimization for Transformer 25

Chapter 3 Integrated Transformers Inference Processing **28**
3.1 System Overview 28
3.2 Offline Preparation 29
3.3 Inference Processing 31
3.3.1 Request Handler 31
3.3.2 GEMM Scheduler 33
3.3.3 Inference Runtime 34

Chapter 4 Optimization Algorithms **37**
4.1 Requests Batching 37
4.1.1 Tenant Requests Batching 38
4.1.2 Compute-bound Operator Batching 40
4.2 Operation Level Optimization 42
4.2.1 CUDA Stream Optimization 42
4.2.2 Model Weights Rotation 43
4.2.3 Tensor Transformation 45

Chapter 5 Experiments **47**
5.1 Experiment setup 48
5.1.1 Experimental System 48
5.1.2 Models 49
5.1.3 Baselines 50
5.2 Overall Performance Evaluation 51
5.3 Scalability Evaluation 53
5.3.1 Requests Batching Evaluation 55
5.4 Evaluation of Operation Level Optimization 59

Chapter 6 Conclusion **62**

Bibliography **64**

List of Figures

1.1 Thesis Architecture 5
2.1 Multi Process Service [NVI21a] 21
2.2 Transformer Architecture [Vas+17] 24
2.3 BERT Architecture and Operators on GPU 25
3.1 The system architecture of ITIF. 29
3.2 Two-Stage Batching Strategy By Request Handler 32
3.3 GEMM Scheduler deploys the Operators into CUDA Stream 33
3.4 Operator level Optimization Methods in Inference Runtime 35
5.1 The performance evaluation for the four different transformer models. We evaluate the acceleration ratio with three baselines: Sequential execution, MPS, and multi-stream execution. The experiments contain four tenants for each model, and the number of total requests is 4. 52
5.2 Benchmarking the latency of runtimes with the variable number of requests. We use ALBERT as the model of evaluation, and the maximum sequence length of each request is 64. 55
5.3 Benchmarking the throughput of runtimes with the variable number of requests. We use ALBERT as the model of evaluation, and the maximum sequence length of each request is 64. 57
5.4 Benchmarking the throughput of runtimes with the variable number of requests. We use ALBERT as the model of evaluation, and the maximum sequence length of each request is 64. 58
5.5 "Get Q,K,V" Operator in Transformer. 60

List of Tables

5.1 Transformer models and parameters 49
5.2 Performance Evaluation for varied numbers of tenants (N. T.). We maximize the Input sequence length for BERT and AlBERT: 128, the input image size: 224×224×3. Latency (ms) 54
5.3 Execution time (ms) of memory-bound operators in BERT 59
5.4 cudaLaunchKernel analysis in BERT 60

CHAPTER 1

Introduction

1.1 Motivation

In recent years, Transformer models [Vas+17] have exhibited remarkable performance across multiple domains, with their unique multi-head attention mechanism making them a highly suitable choice. These models utilize pre-training and fine-tuning paradigms, such as BERT [KT19] and XLNet [Yan+19], to enhance performance in a wide array of natural language processing (NLP) tasks, encompassing text classification[Min+21], machine translation[RT18], question-answering[Zha+20], and more. Moreover, Transformer models have also emerged as frontrunners in the computer vision (CV) domain, with the implementation of image embeddings in models like ViT [Dos+20], MAE [He+22]. Owing to this paradigm, Transformer models can tackle diverse tasks after undergoing fine-tuning. Despite having fine-tuned weights for distinct tasks, these models share a common backbone. During inference applications, pre-trained Transformer models are employed initially, and subsequent tasks are executed based on the model's specialized processing. Consequently, the optimization of pre-trained Transformer models is of paramount importance.

In inference applications, Transformer models often necessitate substantial computing resources, which are typically deployed on cloud or edge servers equipped with GPUs. For example, Facebook [Haz+18] processes tens of billions of real-time inference requests, demanding a vast number of computing servers. Generally, there are two primary inference-serving methods in the industry: exclusive access and GPU sharing. To ensure high Quality

of Service (QoS), platforms like Amazon SageMaker [Amz17] and TensorFlow Serving [Ols+17] employ an exclusive access approach, wherein each model is allocated a dedicated GPU. This strategy eliminates resource contention, allowing inference requests to receive immediate responses, which benefits real-time tasks. However, the cost of exclusive access may be prohibitive for most service providers, and the majority of inference requests do not entail stringent latency requirements. In certain edge computing scenarios, only a single GPU is available as the computing resource and multiple tasks must be deployed on it. Consequently, these inference tasks must be accommodated on a single GPU. As a result, GPU sharing for multi-tenant solutions becomes essential.

Deploying inference applications presents challenges, regardless of whether they are situated on cloud or edge devices. Due to varying request frequencies from different tenants, the GPU can adequately handle requests while maintaining high QoS. Nevertheless, during periods of peak demand, resource contention for the GPU arises. Without optimization, the server system struggles to sustain reasonable latency for various tenants. Consequently, several methods have been proposed to optimize resource contention in GPU-sharing scenarios. NVIDIA has introduced spatial sharing approaches for general GPU tasks, such as Multi-Process Service [NVI21a] and Multi-Instance GPU [NVI21c]. These methods implicitly perform spatial multiplexing, which improves overall system throughput. However, these approaches are not specifically tailored for deep learning (DL) applications, and users may face difficulties in estimating inference runtime progress, especially when multiple tenants coexist in the system. This uncertainty can result in unstable inference latency, ultimately compromising the QoS.

Previous research has concentrated on targeted optimizations derived from the structure and characteristics of deep learning (DL) models, an approach that we also adopt. In these studies [Yu+21][Bai+21][Zha+22], DL models are abstracted as directed acyclic computation graphs (DAGs), wherein vertices represent operators and edges signify data flow between operators. By examining and optimizing these operators, the inference serving system can offer more effective inference strategies. In our work, we specifically analyze the structure of Transformer models and propose an inference optimization approach tailored to their

unique characteristics. Owing to the benefits of pre-trained Transformer models, they can handle various tasks following fine-tuning. When different Transformer models are deployed on a GPU, they can share the same backbone to process diverse tasks. However, without optimization, service providers treat them as distinct models and do not deploy them on the same device. Rather than implementing a complex resource scheduling mechanism, we can deploy these models with the same backbone on a single GPU and distribute the resources uniformly. This approach simplifies resource allocation while maximizing the potential of shared model components.

1.2 Contributions

In the present work, we introduce the **ITIF** (**I**ntegrated **T**ransformers **I**nference **F**ramework) for multi-tenants. This framework is meticulously crafted with the principal aim of maximizing GPU utilization, thereby significantly enhancing the overall system throughput on a single GPU unit. The unique characteristic of ITIF lies in its ability to consolidate models from multiple tenants into a single instance. It achieves this by implementing an innovative method of rotating model weights and integrating requests originating from various tenants.

In order to effectively implement this multi-tenant concurrent inference execution, we have strategically categorized operators in the Transformer model into two distinct groups. Furthermore, we have devised various approaches for this purpose. The ease of usability is another highlight of ITIF; users are required to only upload their respective model weights and input requests. Following this, ITIF takes charge and allocates suitable computing resources to each tenant. These allocations are based on individual request proportions, ensuring a balanced system.

Our first major contribution lies in the development of an innovative operator optimization method intended for sharing operators amongst different tenants. This method capitalizes on computational efficiencies by optimizing the execution of compute-bound operators using

CUDA streams [NVI21b] and BatchedGEMM [NVI22a]. Additionally, we address the issue of managing memory-bound operators by implementing a unique approach of rotating model weights. This method not only promotes resource sharing but also enhances the overall computational efficiency of the system.

Secondly, we introduce a novel batching strategy designed specifically to consolidate inference requests from a variety of tenants. This strategy leverages the use of a linked list to transmit the layout of requests from different tenants to the inference runtime. This systematic provision of the position information of individual tenant requests empowers the inference runtime to group these diverse requests into effective batches, facilitating their collective processing. This method drastically reduces latency and improves the throughput of the overall system, particularly when handling multiple simultaneous requests.

Finally, our third contribution pertains to the application of our developed methods to several popular Transformer models. We have conducted extensive experimentation on GPUs to validate the effectiveness of our Integrated Transformers Inference Framework (ITIF). The results from these experiments have been extremely encouraging, showcasing ITIF's remarkable capacity to enhance processing speed. In fact, ITIF has demonstrated a remarkable speedup of average $1.71\times$ and up to $2.40\times$ when benchmarked against established baselines, which is the acceleration ratio between ITIF and Sequential execution based on the latency metric. This clear performance enhancement underlines the practical benefits and potential applications of our novel multi-tenant framework.

Through these contributions, we aim to significantly enhance the efficiency and productivity of multi-tenant systems, making a noteworthy addition to the current body of knowledge in this area. We believe that the methods and strategies presented in our work open new avenues for further exploration and optimization in the realm of concurrent inference execution.

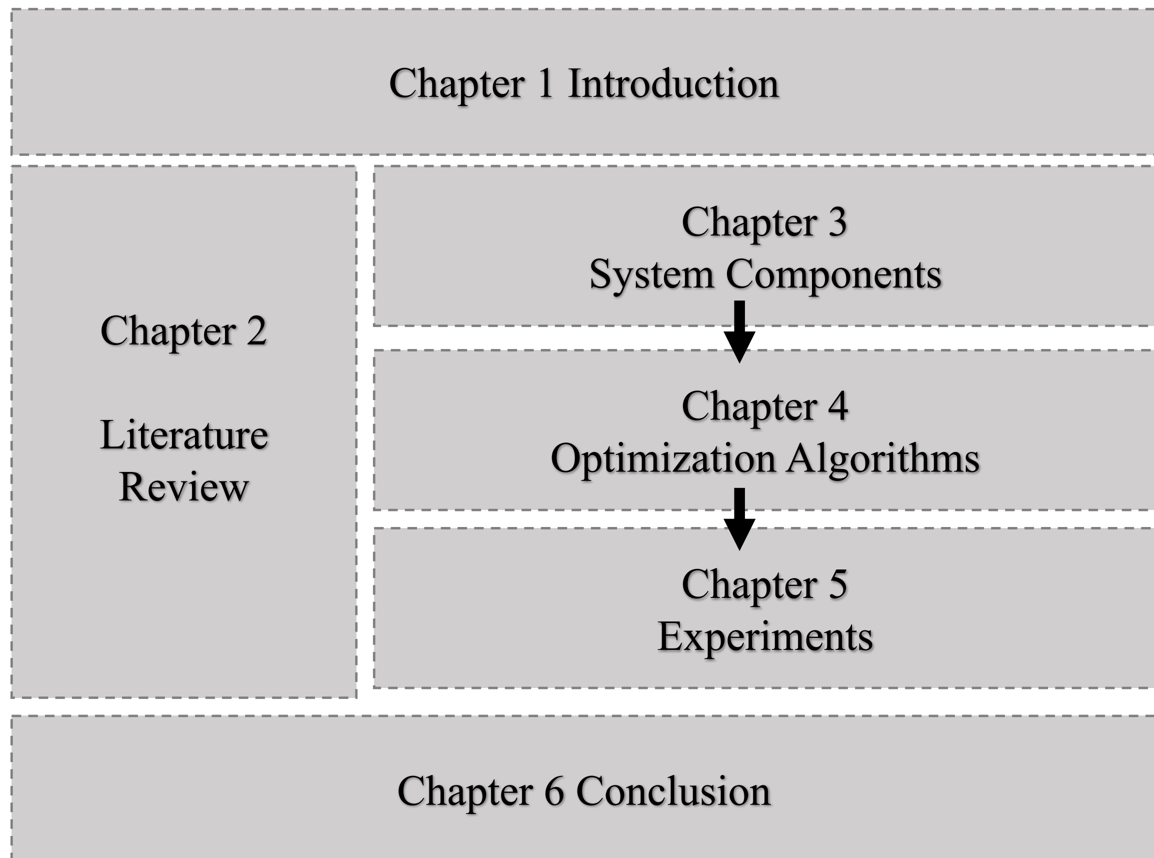


FIGURE 1.1. Thesis Architecture

1.3 Thesis Overview

Figure 1.1 shows the architecture of this thesis. In this thesis, we have six chapters to conclude my work. In chapter 1, we have presented our motivation and show our solution and contribution based on the motivation. In chapter 2, we review the relevant optimization methods and some existing solutions for multiple tenants inference. In chapter 3, we will introduce our solution ITIF and all the components in this framework. In Chapter 4, we provide the details of optimization algorithms in our framework, which are two parts: Requests Batching and some operation-level optimization approaches. In Chapter 5, we conduct different experiments and compare three different baselines. In the experiments, we analyze the advantages and usability of our framework from different perspectives.

CHAPTER 2

Literature Review and Preliminaries

With the development of the raising of Artificial Intelligent applications, People are already feeling the changes brought about by AI, such as chatGPT [Ope21], Stable Diffusion [Rom+21], and GitHub Copliot [Git21]. Behind these phenomenal applications is the rise of deep learning applications. Deep learning models are constantly optimized, and the number of parameters is increasing, making the models more powerful in performance. At the same time, a large amount of data is involved in the training of the models, making them more accurate. Behind the success of deep learning is the increase in the computing capability of accelerators and the optimization of parallel computing. Whether it is a more massive large language model or a huge amount of data, it needs enough computing capability of accelerators to support the training.

Deep learning models, especially convolutional neural networks (CNNs) [LeC+98] and Transformer [Vas+17], require a large amount of computation to train and perform inference. These computations primarily involve matrix multiplications and convolutions, which are highly parallelizable operations. Graphics Processing Units (GPUs) [Owe+08] are designed to handle parallel processing tasks, which makes them ideal for accelerating deep learning computations. GPUs are optimized for single-precision floating-point arithmetic, which is commonly used in deep learning calculations. Additionally, modern GPUs have hundreds or even thousands of cores, allowing for massively parallel processing of data. Several deep learning frameworks, such as TensorFlow [Aba+16], PyTorch [Pas+19], and Keras [Cho+15], have GPU acceleration built in, allowing developers to easily take advantage of GPUs for

deep learning. Training deep learning models on GPUs can be up to 100 times faster than training on CPUs, which greatly reduces the time required to train models and iterate on them. Furthermore, the scalability of deep learning models can be greatly increased by using GPUs. Large models can be distributed across multiple GPUs or even multiple machines, allowing for parallel processing of data and faster training times. This is particularly important for large-scale applications such as image and speech recognition, where large datasets [Den+09] and complex models are common.

In order for the model to be better applied, the inference process of deep learning becomes important. Training and inference are two distinct phases of the DL process that have different computational requirements and goals. During the training phase, the DL model is fed a large amount of data and its parameters are adjusted iteratively through an optimization process, such as stochastic gradient descent, to minimize a loss function. The goal of training is to find the optimal set of parameters that will enable the model to make accurate predictions on new data. In contrast, during the inference phase [Par+18], the DL model is given new data, and its parameters are fixed. The model then produces an output, which can be a class label or a probability distribution over classes, depending on the specific task. The goal of inference is to use the trained model to make accurate predictions on new data.

The computational requirements of training and inference are different. Training typically requires much more computational power than inference, as it involves multiple iterations of adjusting model parameters and computing gradients. This can require large amounts of memory and computational resources, which can be challenging to scale for large DL models. In contrast, inference requires less computational power than training, as it involves only a single pass through the trained model. However, the latency requirements for inference can be much stricter than for training, especially in real-time applications such as autonomous driving [Gri+20] or natural language processing (NLP) [LK17].

Therefore, DL training and inference are two distinct phases of the DL process that have different computational requirements and goals. Training is a computationally intensive

process that involves adjusting model parameters iteratively to minimize a loss function, while inference involves using the trained model to make accurate predictions on new data.

In summary, we will present a literature review on deep learning, parallel computing, inference optimization for deep learning, and the specific approaches for Transformers. In section 2.1, we will discuss the relationship between Parallel Computing and Deep Learning, which will include the specific algorithms and relevant accelerators, especially the GPUs. In section 2.2, we will analyze the inference optimization methods, such as kernel-level optimization, request batching, and some optimization approaches for multi-tenant. In section 2.3, we will illustrate the specific optimization methods for Transformer models in the inference stage.

2.1 Deep Learning and Parallel Computing

Deep learning often requires a huge amount of data to be put into the model for computation. These data often do not have dependencies on each other, so parallel computing can better optimize deep learning and improve the speed of training and inference. This section will discuss the relationship between deep learning and parallel computing, and first, we will analyze the importance of parallel computing in the field of deep learning. In the second half of this section, we will discuss different accelerators and the most popular GPUs and CUDA[NVF20] due to the different parallel computing algorithms derived from different hardware architectures.

2.1.1 Parallel Computing in Deep Learning

Deep learning is a subfield of machine learning that focuses on artificial neural networks with multiple layers, also known as deep neural networks (DNNs) [LBH15]. These networks are capable of learning hierarchical representations of data, enabling them to automatically extract and learn complex features from raw data, such as images, text, or audio signals. Deep

learning has achieved state-of-the-art performance on a wide range of tasks, including image recognition [Hon+21], natural language processing [CW14], speech recognition, [Li+22] and reinforcement learning [Lev+20].

Deep learning models consist of multiple interconnected layers, where each layer transforms the input data into a higher-level representation. The input layer receives raw data, while the output layer generates predictions based on the learned features. Intermediate layers, known as hidden layers, are responsible for learning hierarchical representations of the input data. The depth of a neural network, i.e., the number of hidden layers, plays a significant role in the model's ability to learn complex patterns and representations.

The most common deep learning architectures include:

Convolutional Neural Networks (CNNs): CNNs are primarily used for image recognition and computer vision tasks. A typical CNN architecture consists of multiple convolutional layers, each followed by an activation function, interspersed with pooling layers to reduce spatial dimensions[LeC+98]. The resulting feature maps are then flattened and connected to one or more fully connected layers to generate the final predictions. CNNs have more powerful performance with less computation due to the impact of convolutional layers. CNN models have achieved significant results in Computer Vision (CV) Domain, such as ResNet, VGGNet, and MobileNet [He+16; How+17; SZ14]. Compared to Transformer's large number of parameters, CNN's model structure is relatively small and is often used in edge devices or distributed learning [Lai+21].

Recurrent Neural Networks (RNNs): RNNs are designed for sequential data, such as time series or natural language processing tasks.[Yin+17] They have a built-in memory mechanism that allows them to maintain state information from previous inputs, enabling them to model long-range dependencies in the input data [HS97]. A popular variant of RNNs is the Long Short-Term Memory (LSTM) network, which addresses the vanishing gradient problem and enables more effective learning of long-range dependencies [GSC00].

Transformer Networks: Transformers are a more recent architecture designed primarily for natural language processing tasks but have also been applied to various other domains. They rely on self-attention mechanisms to capture dependencies between input elements without the need for recurrent connections. Transformers have become the foundation for many state-of-the-art models, such as BERT and GPT [Vas+17; KT19; Rad+19].

Training deep learning models typically involves a process called backpropagation, which adjusts the model's parameters (weights and biases) based on the gradients of a loss function that measures the difference between the model's predictions and the ground truth labels [RHW86]. Stochastic gradient descent (SGD) or one of its variants is often used to optimize the model's parameters iteratively.

Due to a large number of parameters and the computational complexity of deep learning models, parallel computing techniques, such as data parallelism and model parallelism, have become essential for reducing training time and enabling the development of larger and more complex models [KSH17; Dea+12]. Parallel computing, which involves distributing the computational workload across multiple processing units, has emerged as a key enabler of deep learning, significantly reducing training times and enabling the development of larger and more complex models [KSH17].

In deep learning, both data parallelism and model parallelism play important roles in reducing the time taken for training and inference, allowing researchers to experiment with larger and more complex models. We will discuss these techniques in more detail below.

Data Parallelism: Data parallelism involves dividing the training dataset into smaller batches and distributing them across multiple processing units, such as GPUs or CPUs. Each processing unit operates on its subset of the data and calculates the gradients independently. The gradients are then aggregated and applied to update the model parameters. This approach is particularly useful when the model fits within the memory of a single processing unit, and the primary challenge lies in processing a large amount of data.

Data parallelism can be further divided into synchronous and asynchronous methods. In synchronous data parallelism, all processing units synchronize their gradient updates after each iteration, ensuring global consistency. This approach is commonly used in distributed deep learning frameworks like TensorFlow [Aba+16] and PyTorch [Pas+19]. However, synchronization can lead to communication overhead and potentially slow down training [Dea+12]. Asynchronous data parallelism, on the other hand, allows processing units to update model parameters independently without waiting for other units. This reduces communication overhead but may lead to issues like delayed gradient updates and reduced convergence rates [Lia+15].

Model Parallelism: Model parallelism involves distributing the model's parameters and computations across multiple processing units. This technique is particularly useful when the model is too large to fit within the memory of a single processing unit or when certain layers of the model have high computational requirements. In model parallelism, different parts of the model are assigned to different processing units, and each unit is responsible for computing a portion of the forward and backward passes.

There are various strategies to achieve model parallelism, including pipelining, partitioning, and hybrid methods [BH19]. Pipelining involves splitting the model into several stages and assigning each stage to a separate processing unit. As one processing unit completes its assigned stage, it passes the intermediate results to the next processing unit, forming a pipeline of computation. This approach can help reduce memory usage and improve computational efficiency [Hua+19]. Partitioning, on the other hand, involves dividing the model into disjoint sections and assigning each section to a different processing unit. This can be done along different dimensions, such as layers or neurons, depending on the model architecture and the available hardware [Kri14]. Hybrid methods combine aspects of both pipelining and partitioning to further improve parallelization efficiency [Zho+20].

In summary, parallel computing plays a crucial role in the development and application of deep learning models by distributing computational workloads across multiple processing

units. Data parallelism and model parallelism are two key techniques that have enabled the training and deployment of larger and more complex models, leading to significant advances in the field of deep learning.

2.1.2 Accelerators for Parallel Computing

In this section, we discuss the various accelerators for parallel computing in the context of machine learning, focusing on GPUs and CUDA as the most widely used accelerator and programming models, respectively. We first provide an overview of other accelerators, including TPUs, FPGAs, neuromorphic computing, and optical computing, before delving into the details of GPUs and CUDA.

2.1.2.1 Tensor Processing Units (TPUs)

Tensor Processing Units (TPUs) were introduced by Google as custom application-specific integrated circuits (ASICs) specifically designed to accelerate deep learning workloads [Jou+17]. The primary motivation behind the development of TPUs was to optimize the performance and energy efficiency of tensor operations, which are fundamental to deep learning models. TPUs are specifically designed to execute matrix multiplications and other tensor operations in parallel at high throughput and low latency. They feature a large systolic array of processing elements that can efficiently perform these operations, making them more suitable for deep learning tasks compared to traditional CPU or GPU architectures. This unique architecture allows TPUs to significantly reduce the time required for training and inference in machine learning applications.

Machine learning practitioners and researchers can benefit significantly from using TPUs to accelerate their deep learning workloads. TPUs have been integrated with popular machine

learning frameworks like TensorFlow, making it easier for developers to leverage the power of TPUs without the need for extensive modifications to their existing code [WWB19].

When working with TPUs, developers should consider several factors to ensure the efficient utilization of these specialized accelerators. TPUs are designed to handle large-scale tensor operations with high throughput, so it is important to structure the deep learning models and training processes to take advantage of these capabilities. This can involve batching multiple samples together and optimizing the data pipeline to ensure that the TPUs are continuously fed with data, preventing idle time and maximizing their utilization [Jou+17]. One notable example of leveraging TPUs for large-scale machine learning is the work done by Google's DeepMind team on the AlphaGo and AlphaZero projects. The training of these sophisticated deep learning models, which demonstrated superhuman performance in playing Go and other board games, was significantly accelerated using TPUs, enabling the team to iterate more quickly and explore new ideas in the development of these groundbreaking AI systems [Sil+16; Sil+17].

2.1.2.2 Field-Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays (FPGAs) are reconfigurable hardware platforms that provide a flexible and high-performance solution for accelerating a wide range of applications, including deep learning tasks [CH02]. Unlike GPUs and TPUs, which have fixed architectures, FPGAs are composed of programmable logic blocks and interconnects that can be customized to implement specific functionalities tailored to the requirements of a particular deep learning workload.

FPGAs offer several advantages for deep learning acceleration. One of the main benefits of using FPGAs is their ability to provide fine-grained customization at the hardware level. This enables developers to design custom processing pipelines optimized for specific deep learning

models, resulting in higher performance and energy efficiency compared to general-purpose accelerators [NZK20].

Another advantage of FPGAs is their support for dynamic reconfiguration, which allows for the modification of the hardware implementation even after the FPGA has been programmed. This feature enables developers to rapidly update the hardware design to adapt to evolving deep learning models and algorithms without the need for a full chip redesign [CH02].

FPGAs also excel at handling irregular and sparse data structures, which are common in certain deep learning models, such as sparse neural networks and graph-based models. The flexibility and programmability of FPGAs make them well-suited for implementing custom dataflow architectures that can efficiently process these irregular data structures [Zha+15].

2.1.2.3 Graphics Processing Units (GPUs) and CUDA

Graphics Processing Units (GPUs) have become the de facto standard for accelerating deep learning tasks due to their highly parallel architectures and efficient handling of matrix and vector operations [KSH17]. NVIDIA's Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model that enables developers to harness the power of NVIDIA GPUs for general-purpose computing tasks, including deep learning [NVF20].

Deep learning frameworks such as TensorFlow, PyTorch, and Caffe have integrated support for CUDA, allowing developers to easily leverage the power of GPUs for training and inference tasks with minimal code changes [Aba+16; Pas+19; Jia+14]. CUDA provides a set of libraries and APIs for performing common deep learning operations, such as cuDNN for deep neural networks and cuBLAS for linear algebra, which is optimized for GPU execution [Che+14; NVI22a].

One of the main reasons behind the widespread adoption of GPUs and CUDA for deep learning is their ability to significantly accelerate the training process. Training deep learning models involves performing large-scale matrix multiplications and convolutions, which can be executed efficiently on the highly parallel architectures of GPUs [KSH17]. The use of GPUs can reduce the training time of deep learning models from weeks or months to just a few hours or days, enabling researchers to iterate more quickly and explore new ideas.

GPUs are also well-suited for accelerating the inference process in deep learning, which involves evaluating a trained model on new data. The high throughput and low latency of GPUs make them an ideal choice for real-time applications, such as object detection, speech recognition, and natural language processing [Han+15].

In summary, GPUs and CUDA have played a pivotal role in the advancement of deep learning by providing a highly parallel and efficient computing platform for training and inference tasks. As the field of deep learning continues to evolve, GPUs and CUDA are expected to remain an essential component of the deep learning ecosystem, driving future breakthroughs in the field.

2.2 Inference Optimization for Multi-tenants in GPU

As the discussion in the previous section, central to the development and deployment of deep learning models are two main processes: training and inference. Training is the process of learning the parameters (e.g., weights and biases) of a neural network by minimizing the error between the predicted output and the ground truth (i.e., the actual output) on a labeled dataset [Kel19]. Unlike training, inference requires only a forward pass through the network without requiring backpropagation or gradient calculations. The inference is less computationally intensive than training but must be efficient and fast, particularly for real-time applications.

Several techniques have been proposed to optimize inference, including model quantization, pruning, and knowledge distillation, which aim to reduce the model size and computational complexity without significantly compromising accuracy [Han+15; HVD15; Ras+16]. However, we focus on the system level of inference optimization methods. There are two main parts: operator optimization and request batching.

2.2.1 Kernel Level Optimization

In deep learning computation, the kernel, as the smallest unit in the deep learning model, is an important part of inference optimization. The traditional parallel computing algorithm can be leveraged into kernel optimization, such as loop tiling, loop unrolling, and thread blocking. These methods can significantly improve kernel computing efficiency, especially for the Matrix Multiply. Therefore, most Deep learning compilers integrate these methods into their kernel optimization, such as TVM [Che+18] and Tensorflow XLA [Goo17].

In TVM, the author designs a compiler that can automatically deploy the deep learning inference model to different devices. The compiler transforms the input model into a computational graph and rewrites it into an optimized one. In the rewriting stage, TVM not only uses basic parallel computing optimization (loop tiling, loop unrolling) but also uses an important operator optimization approach: kernel fusion. Kernel fusion, also called operator fusion, combines multiple operations into a single operation, which can reduce the memory footprint and speed up computing. In TVM, they recognize the operators into four categories: injective, reduction, complex-out-fusible, and opaque. Classifying the different kinds of operators present in different models allows the compiler to perform sufficient operator fusion for any model, thus improving efficiency.

Kernel fusion is also leveraged in other works. Some prior works are similar to the TVM, which is used in the Deep Learning compiler. Apollo [Zha+22] proposes an automatic partition-based operator fusion method. Apollo has two stages: partition the input model and

fuse the primitive operators. In Apollo, the operators are classified into two main categories: computed-bound operators and memory-bound operators, and each main category contains more sub-categories. The more detailed classification of operators also makes the performance of the optimization more outstanding. There are other Deep Learning compliers to use kernel fusion to optimize the inference processing, such as XLA [Goo17], AutoGTCO[Bai+21]. In AutoGTCO, authors leverage dynamic programming to choose the optimal kernel fusion solutions.

From another aspect, some prior work focuses on specific model optimization using kernel fusion. DNNFusion [Niu+21] is a rigorous and extensive loop fusion framework that can exploit the operator view of computations in DNNs and enable a set of advanced transformations. The core idea of DNNFusion is to classify operators into different types and develop rules for different combinations of the types, as opposed to looking for patterns with a specific combination of operations. Compared with the basic kernel fusion approach, DNN fusion rewrites the internal computation directly on the basis of fusing different kernels, which is a novel mathematical-property-based graph rewriting. Moreover, it uses a different standard to classify DNN operators, making kernel fusion more adaptable to DNN networks.

In ParallelFusion [LLL21], authors use kernel fusion to improve the throughput of single inference up to 195% to 218%. In this paper, the authors focus on the challenge of extremely under-utilized DNN inference processing in Mobiles GPUs. To overcome this challenge, ParallelFusion fuses the execution of kernels and active more GPU threads to work concurrently for the kernels. In traditional GPU core function scheduling, the same GPU does not serve multiple core functions simultaneously. Although we can deploy different core function tasks to the GPU simultaneously, the GPU will still execute in a serial manner. Parallelfusion, however, leverages OpenCL [20] to execute different core functions simultaneously, allowing more GPU threads to be called simultaneously, thus increasing GPU utilization. Furthermore, there is some prior work that focuses on the optimization of Transformer models with kernel fusion, which we will discuss in section 2.3.

2.2.2 Request Batching

Request batching is a crucial technique employed in machine learning inference pipelines to bolster efficiency and reduce latency. This strategy is particularly beneficial in scenarios where processing a single request is computationally expensive or when the system must handle a large volume of concurrent inference requests. Request batching operates by aggregating multiple requests and processing them collectively as a batch rather than handling each request individually. For the request batching, Clipper[Cra+17] provides a demonstrative guide for the serving system. Clipper employs an adaptive batching scheme to dynamically find and adapt the maximum batch size for each model container. This is achieved using an additive-increase-multiplicative-decrease (AIMD) scheme, where the batch size is incrementally increased until the latency to process a batch exceeds the latency objective. At this point, the batch size is reduced by 10%. This scheme is robust to changes in the throughput capacity of a model. It has been found to provide significant performance improvements over the baseline strategy of no batching, achieving up to a 26x throughput increase in some cases.

This approach permits systems to more effectively utilize computational resources by taking advantage of the parallel processing capabilities of modern hardware, especially GPUs. The LazyBatching [CKR21] considers both scheduling and batching at the granularity of individual graph nodes rather than the entire graph for flexible batching. In terms of GPU usage, LazyBatching is implemented on top of existing hardware/software stacks without requiring hardware modifications. The stack-based batch status tracking process is done purely in software, and task preemption and context switching are conducted at node execution boundaries (i.e., layer boundaries). The required input/output tensors for batched requests are allocated upfront to accommodate the model-allowed maximum batch size, which amortizes the runtime memory management overhead.

However, determining the optimal batch size is a complex task that has been the subject of various research studies. A larger batch size can enhance throughput but may also increase latency due to the time required to accumulate enough requests to form a batch. This trade-off

has been the focus of numerous studies in the field. Moreover, in the NLP domain, the system cannot directly pack all the requests to calculate uniformly because the length of each input varies. The traditional method is to add zero padding, but it also provides a significant overhead. To handle variable-length inputs and save GPU memory occupancy, LightSeq[Wan+20] pre-defines the maximum of dynamic shapes, such as the maximal sequence length. At the start of the service, each intermediate result in the calculation process is allocated GPU memory to its maximum. Besides, GPU memory is shared for non-dependent intermediate results. This strategy is referred to as dynamic GPU memory reuse.

Further, some prior works leverage dynamic programming to improve request batching. TurboTransformers[Fan+21] employs a sequence-length-aware batch scheduler, Turbo-DP-Batch, which uses dynamic programming to solve an optimization problem that maximizes the response throughput. The batch scheduler sorts the requests in the message queue (MQ), and the execution of long sequences is delayed, thus increasing their latency. However, this approach improves the serving throughput significantly. Based on TurboTransformer, PetS[Zho+22] provides a two-stage DP algorithm for request batching, focusing on the Parameter-Efficient Transformers. Compared with TurboTransformers, PetS have two standards in request batching, which are types of adapter and sequence length. Therefore, a two-stage DP algorithm can handle complicated situations.

The last one is TCB [Fu+22]. TCB is a Transformer inference system with a novel ConcatBatching scheme and an online scheduling algorithm. The ConcatBatching scheme minimizes computational redundancy by concatenating multiple requests and aligning batch rows with reduced padded zeros. This approach is particularly beneficial for handling variable-length input, a common challenge in Transformer-based models. TCB provides better performance than DP request batching because they do not need any padding for the requests.

In conclusion, request batching offers a promising solution for efficiently handling large volumes of inference requests. However, it also presents challenges, such as determining the optimal batch size and managing the trade-off between throughput and latency. These

challenges are likely to continue to be the focus of future research, along with exploring the implications of batching on various types of machine learning models.

2.2.3 Optimization for Multiple Tenants

In the previous section, some the-state-of-art techniques have been introduced, and they can excessively accelerate the execution of inference models in GPU. However, most are designed for single tenants and can not directly fit into multiple-tenant scenarios. We have introduced three approaches for inference deployment in the GPU in the introduction. Although the exclusive methods can provide the best performance, that is unaffordable and waste the computation resources. For the temporal sharing of GPU, the author proposes the limitation of the temporal sharing GPU approach in this article [Jai+18], which is a massive overhead for GPU context switching, which is unnecessary and a waste of resources. Meanwhile, kernel launch is also an overhead that can not be ignored. In GPU programming (CUDA by Nvidia) [CGM14], all the functions are kernel functions, and the GPU schedules them. When the kernel functions are executed, there is overhead for the launched and released, and GPU needs to allocate the computational resources for each kernel function. Therefore, the spatial sharing of GPU is an optimal choice for multiple tenants.

Firstly, NVIDIA provides some powerful frameworks for spatial sharing, which are Multi-Instance GPU(MIG) [NVI21c], Multi-Process Service(MPS) [NVI21a], and Nividia Multi-Stream [NVI21b]. MIG is designed to allow the partitioning of a single GPU into several instances, with each partition functioning as a fully independent GPU with its own dedicated resources, such as memory, cache, and compute cores. This results in the capability to effectively spatially share a single GPU among multiple users or workloads. By allowing multiple tasks to run concurrently on the same GPU, MIG increases the utilization efficiency of the GPU resources. This is especially beneficial in scenarios where many smaller tasks need to be run, and a single task would not be able to fully utilize the GPU's capacity.

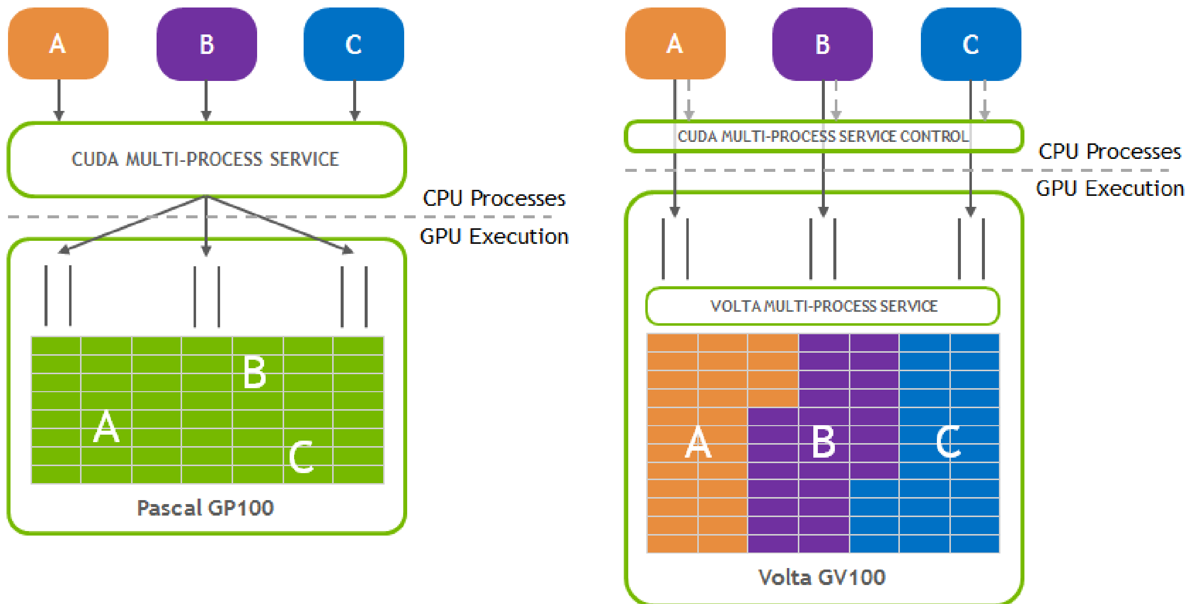


FIGURE 2.1. Multi Process Service [NVI21a]

The CUDA MPS is a CUDA Application Programming Interface (API). Figure 2.1 demonstrates the MPS workflow. The MPS runtime architecture is intended to enable transparent multi-process CUDA applications, typically as MPI [For94] programs. MPI processes are frequently assigned individual CPU cores in a multi-core CPU machine to allow CPU-core parallelization of potential limitations by Amdahl's law to balance workloads between CPU and GPU tasks. Therefore, when MPI processes are accelerated with CUDA kernels, the amount of work allotted to each MPI process may underutilize the GPU. While each MPI process may run more quickly, the GPU is inefficient. The MPS leverages inter-MPI rank parallelism to boost GPU utilization. Because of the task schedule, the MPS is a spatial sharing system, and it combines all the CUDA contexts into a single process, which reduces the overhead of the CUDA context switch.

CUDA stream is a tool to make the program to be concurrency in the GPU. With the CUDA stream, interleaving CUDA processes from distinct streams is possible. Therefore, the CUDA Multi-Stream could be the task concurrency in the CUDA programming, and the structure is a pipeline. Based on the CUDA stream, the kernel functions could be a pipeline. One stream

can deal with the computational kernel, and the other can execute the data movement, which is implicit in part of the memory management overhead.

Although these frameworks can provide significant performance improvement, they cannot suit every scenario. Therefore, some prior works design the scheduling systems for different tasks. In this paper [DKR20], the author proposes a system which is Controlled the Spatial Sharing of GPUs for a Scalable Inference Platform (GSLICE). In GSLICE, it considered MPS as the uncontrolled spatial sharing of GPU. It leads to unstable performance isolation and unpredictable throughput and latency when the number of tasks is significant. In GSLICE, the unit task is called Inference Functions (IF), which focuses on the inference models. GSLICE has a pre-process that can calculate the operation points for each IF to control the GPU's spatial sharing. According to the number of operation points, the system allocates the relevant computational resource to each IF to balance the workloads. For example, the operation points of ResNet-50[He+16] are much higher than VGG-19[SZ14]. If they equally share the GPU, that is unfair, and throughput is reduced. Therefore, controlling the spatial sharing of GPU is necessary.

Furthermore, the authors deploy the different tenant DNN inference in a CUDA stream and build structural search space to provide the scheduling strategy for operators to balance the resource contention in [Yu+21]. To handle the real-time and best-efforts tasks, REEF [Han+22] propose a scheduling approach to provide the microsecond-scale preemption for the kernel rotation to the concurrent DNN inferences execution. Therefore, when solving multi-tenant problems, it is necessary to design a scheduling system based on specific tasks.

2.3 The Enhancement for Transformers Inference

In the previous literature review, we introduced the relationship between deep learning and parallel computing and the common optimization approaches for inference models. In this

section, we will analyze, more specifically, the transformer model and the optimization approaches in a multi-tenant scenario based on our motivation.

2.3.1 Transformer

In 2017, Google proposed a deep learning model for Natural language processing (NLP), which name is Transformer[Vas+17]. Compared with the Convolutional Neural Networks (CNN) based model, the Transformer has better performance in the convergence of prediction accuracy. With the great success of the Transformer in NLP, other areas of deep learning have also widely applied the transformer mechanism. There are multiple deep learning models, which include BERT [KT19], GPT [Rad+18], ViT [Dos+20], and MAE [He+22]. The architecture of Transformers comprises two primary components: the Encoder and the Decoder, both of which contain multiple layers of transformers. The encoder is comprised of a stack of identical layers, each of which has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise, fully connected feed-forward network. A residual connection is employed around each of the two sub-layers, followed by layer normalization. Similar to the encoder, the decoder also stacks identical layers. However, in addition to the two sub-layers found in the encoder, the decoder has a third sub-layer that performs multi-head attention over the output of the encoder stack. Similar to the encoder, residual connections are employed around each of the sub-layers, followed by layer normalization. By utilizing the encoders and decoders in the Transformer, these models become State-of-the-art in their respective domains. Although the Transformer has excellent convergence, which makes it widely implemented in deep learning, its disadvantage is also very prominent: its computation intensity.

In Transformer, the Multi-Head Attention (MHA) mechanism provides significant strength and can be calculated parallel. At the same time, traditional NLP models such as RNN can only be computed sequentially. However, the calculation complexity of the Transformer is significant, and the structure of the Transformer based model is massive. In BERT[KT19]

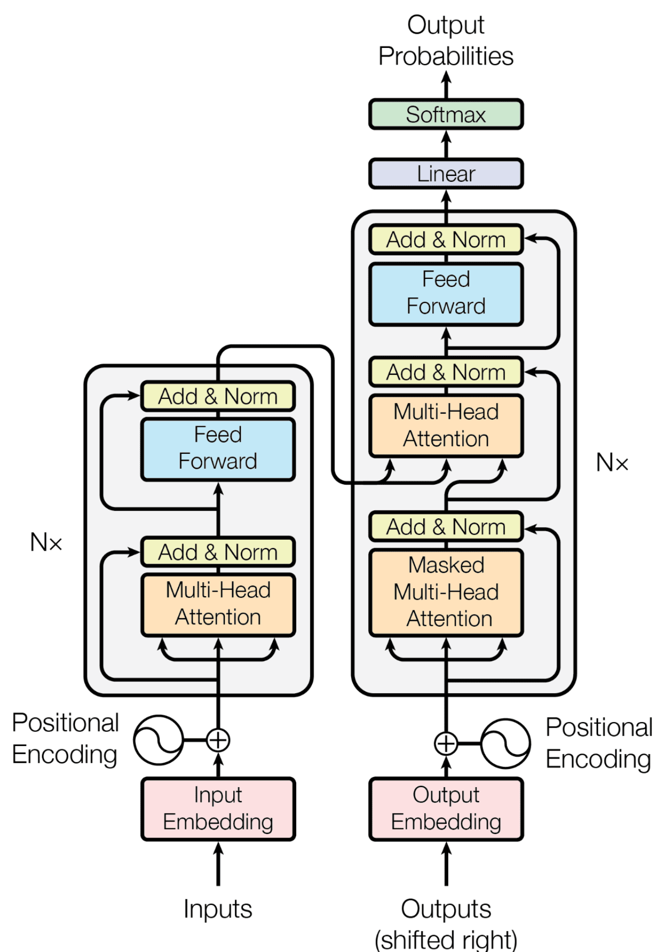


FIGURE 2.2. Transformer Architecture [Vas+17]

and ViT[Dos+20], the model contains 20 encoder layers which include approximately 110M parameters (BERT base), which is much higher than the CNN-based models, such as ResNet50 [He+16] (25M). Although the number of parameters of BERT or ViT is pretty, they are encoder-based models which can parallel compute the MHA. The decoder-based models must be sequentially computed and contain many more parameters, such as GPT-3[Bro+20], including 1750 Billion parameters. The transformer-based models have strict performance requirements for deployment devices, especially since several inference models are deployed in the same GPU based on many parameters.

2.3.2 Optimization for Transformer

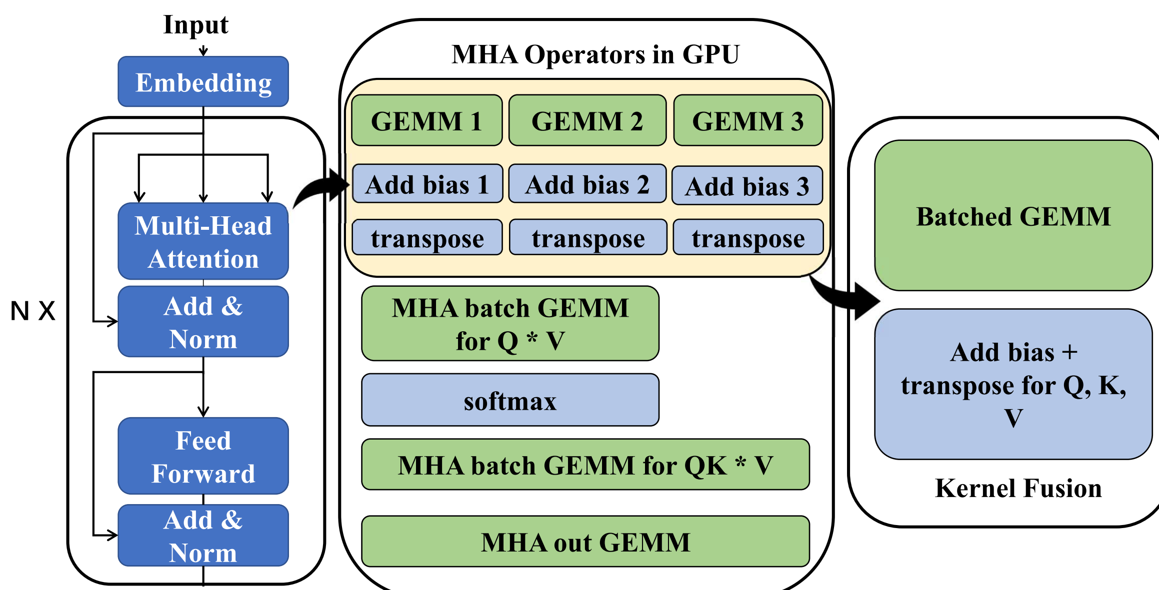


FIGURE 2.3. BERT Architecture and Operators on GPU

The left side of Figure 2.3 depicts the Transformer Models Architecture. Typically, Transformer models are composed of several homogeneous blocks, which include Transformer Encoders and Decoders. Both Encoders and Decoders consist of three layers: Multi-Head Attention (MHA), Add and Normalization layer, and Feed-Forward Network (FFN). Although the structure of Transformer blocks is constructed using only three distinct layers, numerous operators are required to implement these layers on GPUs.

The middle of Figure 2.3 illustrates the MHA operators. We classify them into two categories: compute-bound operators (green) and memory-bound operators (blue). Compute-bound operators generally include GEMM operators, while the remaining operators are memory-bound operators, such as SoftMax, LayerNorm, and AddBias. In CUDA programming [NVF20], operators are implemented by CUDA kernel functions. The considerable number of CUDA kernel executions can cause significant overhead from kernel launching, which has an even more pronounced impact on inference. Unlike model training, inference lacks a backpropagation phase, presenting an opportunity for optimization. The right side of Figure 2.3 displays the kernel fusion for several operators (inside the yellow box) in the MHA. Using

the BatchedGEMM technique in cuBLAS [NVI22a], the three compute-bound operators are fused into a single kernel, and the remaining memory-bound operators can be fused by operator rewriting. Kernel fusion is a mandatory optimization approach for Transformer models.

For Transformer-based models, there are some frameworks to provide the Kernel fusion. Because the most creative layer in Transformer is the MHA layer, most frameworks try to fuse the operators of the MHA layer. The TurboTranformer [Fan+21] provides a solution for the kernel fusion in the MHA layer. In the first step of MHA, the three different tensors, which are Q (query), K (keys), and V (values), needs to be calculated by a linear layer to align the multiple heads. In the operator levels, the linear layer is GEMM. TurboTranformer fuses the three linear layers into a single layer to reduce the number of kernel functions. Moreover, TurboTransformer also fuses all the "addbias" operators with the next operator (addbias $+$ transpose,addbias $+$ Layer Norm). Therefore, the core idea is to fuse all the non-GEMM kernels, which are element-wise kernels. According to the investigation of TurboTransformer, most of the execution time is for the GEMM kernels execution (61.8%), and the best part is for the Non-GEMM kernels.

Compared with the TurboTransformer, the TensorRT[NVI21d] is more creative and fuses the whole MHA layer. The new fused MHA kernel includes the GEMM for $Q \times K$, softmax, GEMM for $QK \times V$, transpose, and padding in BERT. This makes the entire model much lighter, with each encoder requiring only a few kernels to complete the computation. However, kernel fusion only can be used in the inference models because there is no backpropagation phase compared with the training process. Moreover, the kernel fusion needs to be careful, and the layer dependency needs to be considered, especially for the multiple-layer fusion, like the fused MHA kernel in TensorRT.

For the optimization of the Transformer model, much of the work also starts from the perspective of system scheduling. Both of them [BR21] [HPM19] are using the Multi-stream in the Transformer models. In Multi-Stream Transformers [BR21], they propose a Multi-stream

Transformer architecture. The transformer encoders are separated into multiple encoder streams to improve performance. Allowing the model to merge multiple representational hypotheses improves performance, with even more improvement possible when a skip connection is added between the first and last encoder layer. Compared with the Multi-Stream encoders, this article[HPM19] uses the Multi-stream for Self Attention mechanism. Each stream has layers of 1D convolutions with dilated kernels whose dilation rates are unique to each stream, followed by a self-attention layer in the proposed model architecture. Each stream's self-attention mechanism focuses on only one resolution of incoming speech frames, allowing for more efficient attentive computing. The outputs from all the streams are then concatenated and linearly projected to the final embedding at a later step. Although they change the model's architecture and the purpose is not to improve the model performance in GPU, the idea is still creative.

In Orca [Yu+22], authors design a distributed serving system for Generative Models, especially GPT-3. Due to the large size of the generative model, the output has a large number of iterations and therefore consumes a long time to complete the computation. At the same time, there is a large variation in the input length of generative models, and there is a lot of extra overhead in request batching during inference. To solve this problem, Orca converts the inference process from the request level to the iteration level. Orca deploys Large Language Models(LLMs) to multiple GPUs for distributed inference and employs selective batching to schedule the arithmetic, which greatly improves inference efficiency.

CHAPTER 3

Integrated Transformers Inference Processing

To address the aforementioned challenges, we design an inference serving framework, **ITIF**. The primary goal of ITIF is to integrate models from different tenants and provide an optimal scheduling strategy for operators, thereby improving GPU utilization and meeting specified latency SLOs. We integrate various inference services from different tenants based on our operator optimization methods and a novel batching algorithm. The system details are as follows.

3.1 System Overview

Figure 3.1 illustrates the system architecture of ITIF. ITIF consists of two phases: Offline Preparation and Inference Processing. The workflow of ITIF is as follows: ① Tenants upload all the information about their model to the server, which includes the model structure (backbone) and model weights. The server only accepts tenants with the same backbone, and ITIF performs the Offline Preparation. In ②, ITIF rewrites the memory-bound operators and concatenates the model weights with those of previous tenants. Then, ITIF records the memory addresses for each tenant's model weights and saves all the model weights into the GPU memory. Lastly, ITIF captures the model weights and all necessary tensors from GPU memory.

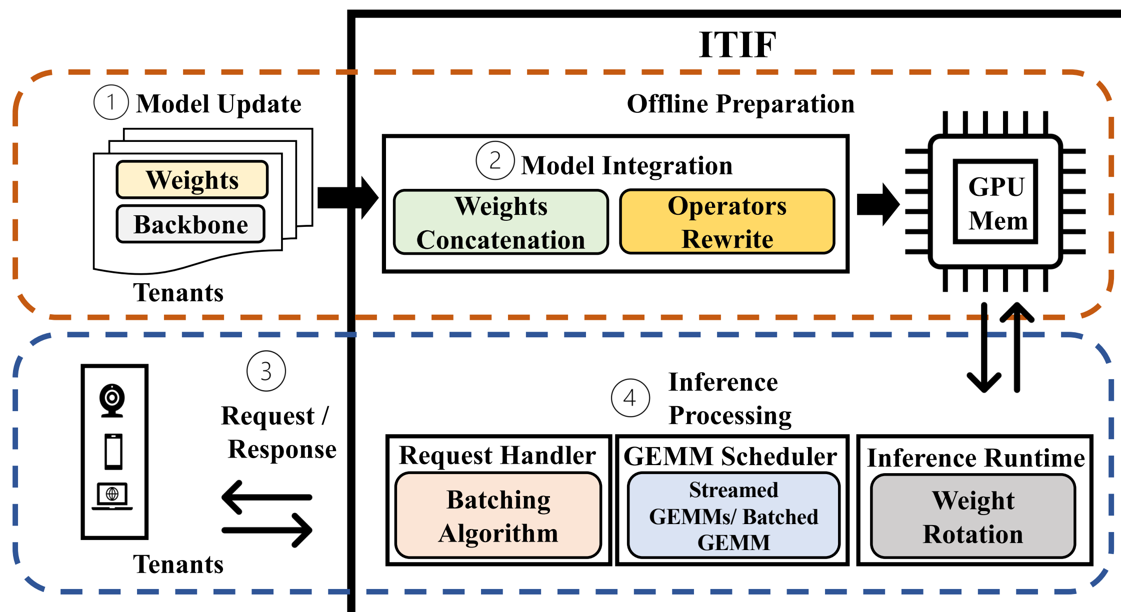


FIGURE 3.1. The system architecture of ITIF.

When ITIF completes the Offline Preparation, it begins serving inference requests. As tenants send requests to ITIF (③), ITIF pushes the requests to the Request Handler, and the Inference Processing stage (④) commences. The Request Handler employs a batching algorithm to integrate all requests from different tenants. The GEMM scheduler deploys various GEMM operators to the GPU, which can handle complex GEMM operator situations for multi-tenant scenarios. There are two options: GEMMs and BatchedGEMM; the GEMM scheduler's decision is based on the batching information. Finally, batched inputs are pushed to the Inference Runtime, and the results are returned to the tenants.

3.2 Offline Preparation

The Offline Preparation phase is designed to deploy the model from tenants to the server before the Inference Processing begins. The first step in this phase is the Model Update. In this step, tenants are required to provide model backbone information to the ITIF, such as BERT_base, ALBERT_large, ViT_huge. Based on the model backbone, ITIF selects the

corresponding operators, the number of Transformer Blocks, and the number of attention heads to construct a model structure.

Once all tenants have occupied the GPU, the ITIF initiates the Model Integration process. During this process, all tenants upload their model weights to the ITIF. Conventionally, memory allocation is based on the models, and different tenants are considered as separate entities, meaning each model occupies a specific memory slot. However, in our approach, ITIF serves multi-tenant inference requests as a single instance, and it uses a single operator to serve all tenants. When the Inference Runtime executes an operator, the corresponding tensors from different tenants are required in the same memory slot. Discontinuous memory storage cannot meet the requirements of CUDA.

To address this issue, we propose a method called Weights Concatenation to save the model weights in GPU memory. We calculate the overall memory workspace to reduce data movement overhead based on the backbone structure and the number of tenants in GPU memory. We allocate the memory for each type of model weight, ensuring that tensors of the same type are in the same memory slot. For instance, in the process of allocating GPU memory for the Multi-Head Attention (MHA) layer, we must determine the size of the hidden units. This size is computed by multiplying the number of MHA heads by the size per head, as given by the backbone information. In Figure 2.3, the MHA layer contains three GEMM operators. The kernel weights for these GEMMs are equal to the square of the hidden units, and this value is for a single tenant. To find the size of the kernel weight, we must multiply this value by the number of tenants, such that the equation 3.1:

$$Kernel_weight = num_tenants \times (num_head \times size_per_head)^2 \tag{3.1}$$

After the weights have been concatenated, the ITIF decomposes the uploaded models in units of different weights and then merges them in the GPU memory according to the upload order of the tenants to complete the deployment. This approach allows the ITIF to efficiently manage

the memory space of multiple tenants sharing the same model backbone, ensuring that each tenant's model weights are stored in the appropriate memory slots. This approach facilitates the seamless integration of different tenants' models into a single instance, allowing ITIF to optimize operator scheduling and improve overall GPU utilization and system throughput. Moreover, we have the operator rewrite for the memory-bound operators and align the inputs with the model weights in memory according to the tenants' order, which can fit all tenants into a single operator.

3.3 Inference Processing

The Inference Processing contains three components: Request Handler, GEMM scheduler, and Inference Runtime. In Figure 3.1, ITIF receives requests from the different tenants (③) and starts the Inference Processing (④) from Request Handler.

3.3.1 Request Handler

The Request Handler's primary function is to manage request batching. In single-tenant inference serving systems, request batching is a common strategy to improve GPU utilization and system throughput. This strategy is implemented in solutions like Clipper [Cra+17], Nexus [She+19], and BARM [QLC22] to enhance inference throughput and overall system performance.

As illustrated in Figure 3.2, our batching strategy in ITIF employs dynamic programming for the batching algorithm. The entire batch is divided into several mini-batches to accommodate different GEMM operators. This division is crucial as it allows for the efficient execution of different operations within the batch.

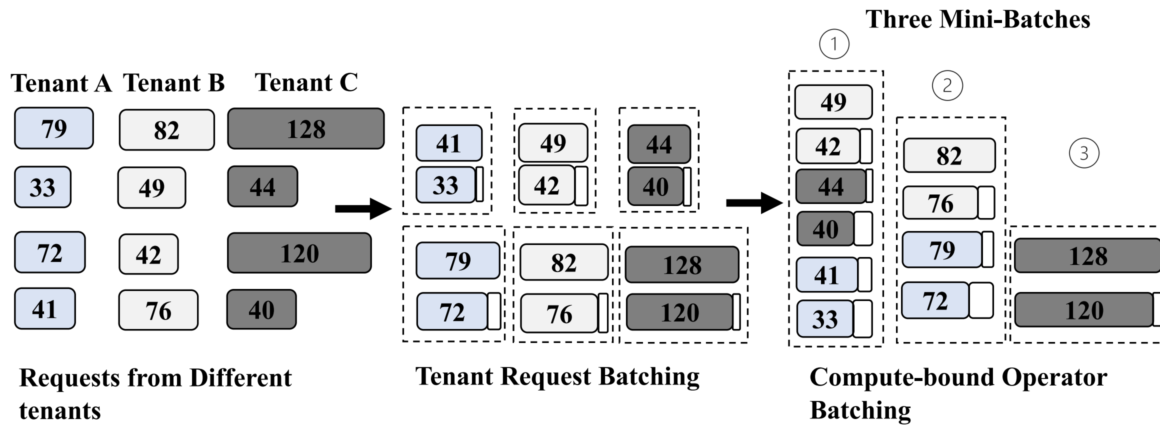


FIGURE 3.2. Two-Stage Batching Strategy By Request Handler

However, one challenge with this approach is the overhead associated with zero-padding. To address this, the Request Handler adjusts the batching algorithm to minimize padding while still maintaining efficient GPU utilization. This adjustment ensures that the GPU resources are used optimally without unnecessary padding that could lead to wasted resources. Following the request batching, zero padding is added to the input tensors. This padding is necessary to ensure that all tensors in the batch have the same dimensions, which is a requirement for many machine learning operations.

For instance, there are 12 requests from three tenants, and the sequence length is 33 to 128. Firstly, we batch the requests within each tenant. We form small batches of requests of similar length, as in the figure where tenant A forms a group of requests of lengths 41 and 33. ITIF completes zero padding for the shorter requests in these small batches, making all requests the same length. After the tenant Request Batching, the Request Handler enters the Compute-bound Operator batching stage. In this stage, the Request Handler batches the small batches with similar performance, which are profiled during the preprocessing. The batching process does not have limitations with different tenants, and the final batches can contain requests from different tenants. Similar to the first stage, zero-padding is added for the short requests. A more detailed explanation of this algorithm will be provided in the 4.1 section.

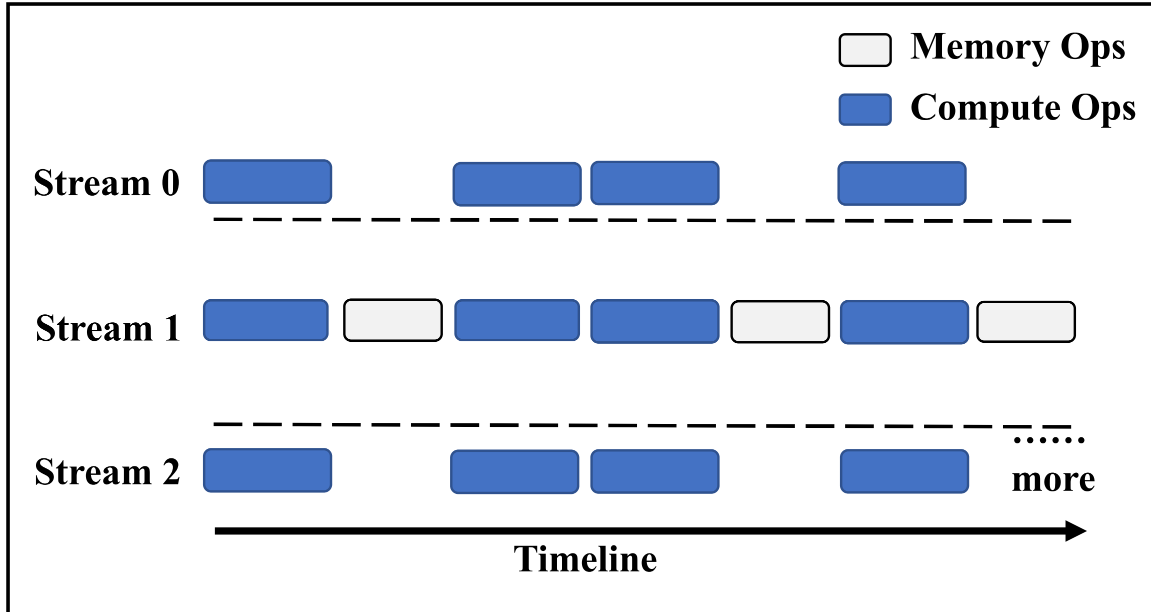


FIGURE 3.3. GEMM Scheduler deploys the Operators into CUDA Stream

Subsequently, ITIF allocates GPU memory for the input tensors. As shown in Figure 3.2, requests are categorized into different classes based on the input size. ITIF allocates memory according to these input sizes. It uses continuous memory allocation to store all requests, which fulfills the operators' requirements. This method of memory allocation ensures that all tensors are stored in a continuous block of memory, which can lead to more efficient memory access and improved performance.

3.3.2 GEMM Scheduler

In transformer models, matrix multiplication operations, specifically General Matrix to Matrix Multiplication (GEMM) operations, constitute a significant portion of the execution time. For example, in the FasterTransformer framework, GEMM operators account for 85.6% of the execution time for single-time Inference Processing with BERT on an NVIDIA RTX A5000, given a batch size of 1 and a maximum sequence length of 128. Given this, optimizing

GEMM operations is crucial for improving the efficiency of transformer models. The GEMM Scheduler in the ITIF is tasked with this optimization. It aims to exploit the GPU's capabilities to optimize the execution of GEMM operators.

In ITIF, we utilize BatchedGEMM and GEMM in the cuBLAS library to handle all GEMM operations. cuBLAS is a GPU-accelerated version of the BLAS (Basic Linear Algebra Subprograms) library, providing optimized routines for vector and matrices operations. However, there is a restriction in cuBLAS's BatchedGEMM: all matrices involved must have the same shape. This restriction means that for irregular inputs, we resort to using the standard GEMM operation.

To maximize parallelism and thus improve performance, we employ CUDA streams to spatially share the GPU. CUDA streams are sequences of operations that execute in order on the GPU. By deploying GEMM operators into different streams, we can have multiple operations executing concurrently, as shown in Figure 3.3. This deployment is managed by the GEMM Scheduler.

The specific method of CUDA Stream Optimization will be discussed in the 4.2 section. This method involves optimizing the scheduling and execution of operations in CUDA streams to improve the performance of the GEMM operations further.

3.3.3 Inference Runtime

Once the input requests are batched and the GEMM operators are dispatched on the target stream, the ITIF initiates the Inference Runtime. This is the phase where the actual computation of the machine learning model takes place.

As shown in Figure 3.4, the Inference Runtime involves several optimization methods. The Transformer model, which is the type of model we're dealing with, consists of a mix of

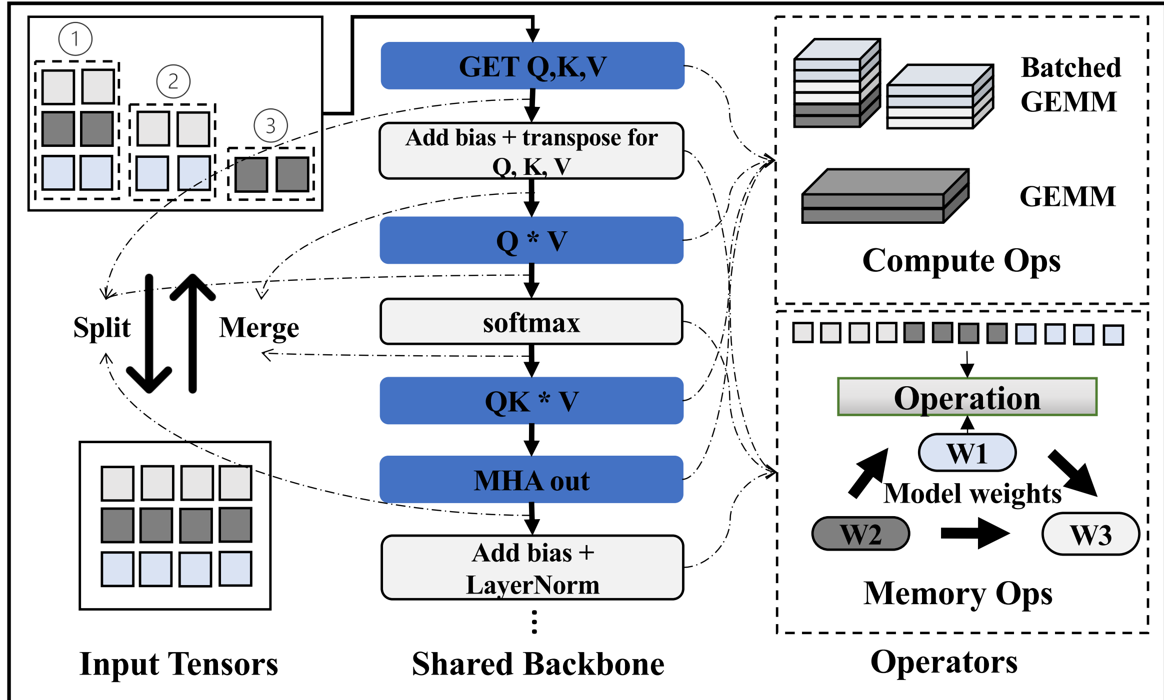


FIGURE 3.4. Operator level Optimization Methods in Inference Runtime

compute-bound and memory-bound operators. These operators run alternately during the Inference Processing.

Compute-bound operators are those where the computation time is the limiting factor, while memory-bound operators are those where memory access time is the limiting factor. Balancing these two types of operators is crucial for the efficient execution of the model. For compute-bound operators, we utilize BatchedGEMM and GEMM from the cuBLAS library to execute them. These are highly optimized routines for performing matrix multiplications, which is a key operation in Transformer models. For memory-bound operators, we propose a novel method called Model Weights Rotation. This method aims to fuse the operators from different tenants, which can help improve memory access patterns and reduce the overall memory footprint.

However, because compute-bound operators are executed separately and memory-bound operators are fused together, we face a challenge: the layouts of input tensors between these two categories of operators can be different. To address this, we designed a method called Tensor Transformation. This method adjusts the layout of tensors as needed for processing different operators.

Tensor Transformation helps balance the execution of compute-bound and memory-bound operators, ensuring that both types of operations can be performed efficiently. It also avoids the problem of having to deal with different layouts of input tensors between two categories of operators, which could otherwise lead to inefficiencies.

The details about Model Weights Rotation and Tensor Transformation, including how they work and the benefits they provide, are discussed in the 4.2 section. These methods are key components of our approach to optimizing the execution of Transformer models in a multi-tenant environment.

CHAPTER 4

Optimization Algorithms

In the last chapter, we have introduced ITIF and all components in this framework. These components are supported by several algorithms to gain optimization. Therefore, we discuss our optimization algorithms in this chapter. There are two parts: requests batching and operation level optimizations, regardless of our contributions. The requests batching is a two-stage Dynamic Programming based algorithm, referring to the request handler. The operation level optimizations include the CUDA stream Optimization, Model Weights Rotation, and Tensor Transformation, referring to the GEMM scheduler and Inference Runtime.

4.1 Requests Batching

In a multi-tenant system, the request format varies significantly due to the different tasks being solved. In the NLP domain, each input sequence has a different length. In the CV domain, the size of the input image varies across tenants. The conventional batching approach[NVI22b] packages all requests with a fixed length, resulting in considerable computation overhead. Turbotransformer[Fan+21] leverages dynamic programming to reduce batch size and make the sequence length of each request within the batch similar, reducing unnecessary computation costs. In Figure 3.2, if we package all the 12 requests as a single batch, the computational resource are wasted because of the zero-padding overhead. For example, there are four requests from two tenants, $\{x_1, x_2, y_1, y_2\}$, with sequence lengths $\{16, 64, 32, 128\}$. If we

fix all sequence lengths to 128, the computation cost for x_1 is four times higher, with three-quarters of the cost attributed to zero-padding.

The multi-tenant system has a more complex situation in ITIF. There are two forms of computational operators: BatchedGEMM and GEMM. When batching requests, ITIF must consider the impact of different compute-bound operators on system performance. Therefore, we design a two-stage batching algorithm for different operators and input sequence lengths using Dynamic Programming. This approach ensures that the system effectively handles different input sizes and operator requirements, ultimately improving the overall performance of the multi-tenant system.

The two-stage batching algorithm aims to optimize the utilization of GPU resources while minimizing zero-padding and unnecessary computations. In the first stage, the algorithm groups requests with similar sequence lengths to minimize the zero-padding overhead. In the second stage, the algorithm further divides the requests into mini-batches considering the constraints of BatchedGEMM and GEMM operators. All the batching decisions are saved in a linked list which is prepared for the Tensor Transformation. The batching process comprises two stages: Tenant Requests Batching in Algorithm 1 and Compute-bound Operator Batching in Algorithm 2.

4.1.1 Tenant Requests Batching

This approach focuses on individual tenant instances, and the details are shown in Algorithm 1. Initially, we have all the requests' information R. To enhance the batching algorithm's accuracy during the inference process, we collect performance information of the computational operator in the offline preparation stage through profiling. We obtain the profiling results of the GEMM operator by adjusting the input length and batch size, saving them using a two-dimensional array, denoted as A in Algorithm 1. ITIF sorts each tenant's requests in ascending order based on input size(line 2). Subsequently, ITIF creates a two-dimensional

Algorithm 1: Tenant Requests Batching with DP

1 **input** : Request list R, GEMM Profiling Result $A[][]$
  output : Batching location info $linked_list$
2 $T \leftarrow tenant_number$, $N \leftarrow sizeof(R)$;
3 sort sub-request list by tenants in increasing order with regards to the sequence length;
4 Create $mini_batch_idx_lists$ $states$ as lists of size T;
5 **for** $t \leftarrow 0$ **to** T **do**
6   Create $mini_batch_idx_lists[T]$ $states[T]$ as lists of size $N + 1$;
7   **for** $i \leftarrow 1$ **to** N **do**
8         $j \leftarrow i - 1$, $start_id = i - 1$, $seq_len = Request_list[t][i - 1].length$;
9         $optimal = A[seq_len][1] + state[t][j]$;
10   **while** $j > 0$ **do**
11   optimal = state[t][j - 1] + A[seq_len][i - j + 1];
12   **if** $tmp < optimal$ **then**
13                $optimal = tmp, start_id = j - 1$;
14   **end**
15            $j \leftarrow j - 1$;
16   **end**
17        $state[t][i] = optimal$;
18   update the Linked List;
19   **end**
20 **end**

array, denoted as $mini_batch_idx_list$, to store the starting request IDs for each mini-batch (line 3). The optimal mini-batches are then selected using Dynamic Programming (lines 4 - 18). We define $state[i]$ to represent the minimum cost for the first i requests, and the Bellman equation is given by Equation 4.1:

$$state[i] = \min_{0<j\leq i} (state[j - 1] + A[seq_len][i - j + 1]) \quad (4.1)$$

After the optimal solution has been chosen, all the results are stored and sorted in the linked list.

4.1.2 Compute-bound Operator Batching

In the second stage, ITIF performs the final sorting of the mini-batches generated in the first stage based on their input sizes. Similar to Stage 1, we obtain a three-dimensional profiling result array, denoted as B for BatchedGEMM in Algorithm 2. Consequently, requests from different tenants may be present within the same batch. In this stage, Dynamic Programming is also employed, using $state[i]$ to represent the minimum overhead for the first i requests. The corresponding Bellman equation is provided in Equation 4.2:

$$\begin{aligned} & state[i] = \min_{0<j\leq i}(state[j-1]+ \\ & B[cur_length][i-j+1][mini_batch[i].size]) \end{aligned} \tag{4.2}$$

For the second stage, the primary focus is on packaging the mini-batches from the first stage into the BatchedGEMM more effectively. Due to the varying lengths of requests, it is impractical to force all requests into the same batch. Additionally, the BatchedGEMM necessitates that all subGEMMs within the operator possess the same shape. Consequently, we examine the shapes of the next n mini-batches before batching and incorporate the mini-batches with identical shapes into the Dynamic Programming process to determine the optimal solution (lines 7 - 8). If a mini-batch exhibits a unique shape, it is treated as a separate GEMM operator for Inference Processing and not merged into the BatchedGEMM (lines 9 - 10).

Upon completion of the batching process, the linked list is updated for the final time. The relative position information stored in nodes for the batching, the input length after padding, and the corresponding operator type are updated. The nodes are then sorted based on the batching results and used for subsequent inference calculations (line 21). This approach

Algorithm 2: Compute-bound Operator Batching with DP

```
1  input  : Mini Batch List minibatch, GEMM Profiling Result A, BatchedGEMM
            Profiling Result B[][][]
   output : Updated batching location info linked_list
2  update the linked_list with the mini_batch_idx_lists,sorted the nodes of linked_list
    with sequence length;
3  M ← minibatch_size,create state[M], start_id_list[M];
4  for i ← 0 to M do
5      batched_idx ← i, j ← i − 1, start_id ← i − 1;
6      seq_len = mini_batch[i].length;
7      optimal = A[seq_len][mini_batch[i].size] + state[j];
8      while minibatch[batched_idx].size/mini_batch[i].size == 0 do
9          batched_idx = batch_idx + 1 ;
10     if batched_idx − i == 1 then
11         state[i] = min_cost; start_id_list[i] = start_id;
12     else
13         for i ← 1 to batched_idx do
14             j ← i − 1, start_id = i − 1;
15             seq_len = mini_batch[i].length;
16             optimal = A[seq_len][mini_batch[i].size] + state[j];
17             while j ← 0 do
18                 tmp = state[j − 1] + B[seq_len][i − j + 1][mini_batch[i].size];
19                 if tmp < optimal then
20                     optimal = tmp, start_id = j − 1;
21             state[i] = optimal, start_id_list[i] = start_id;
22 update the Linked List based on the start_id_list
```

allows for the completion of all requests batching without altering the memory address of the request.

4.2 Operation Level Optimization

4.2.1 CUDA Stream Optimization

In our batching algorithm, we categorize computational operators based on the types of requests they handle and arrange these operators into batches. This categorization and arrangement facilitate efficient execution during the Inference Runtime phase, where we further optimize these operators.

Consider the case of the Multi-Head Attention (MHA) operation, a critical component of Transformer models. The MHA operation involves obtaining Query, Key, and Value (Q, K, V) matrices through the GEMM calculations. To maximize parallelism and prevent race conditions, which could occur if multiple threads attempt to access or modify the same data simultaneously, we create three separate CUDA Streams to execute the corresponding GEMMs concurrently. CUDA Streams are sequences of operations that execute in order on the GPU, and by using multiple streams, we can have multiple operations executing concurrently, thereby improving performance.

When the system processes other compute-bound operators, we distribute the computational load evenly across the three different streams. This distribution is based on previous profiling results, which provide insights into the computational demands of different operators. By balancing the computational load across multiple streams, we can prevent any single stream from becoming a bottleneck, which could occur if it had to handle a disproportionately large amount of computation.

This approach not only balances the computational resources but also minimizes the synchronization time. Synchronization time refers to the time spent waiting for all threads to finish their tasks before proceeding. By balancing the computational load, we ensure that no stream takes significantly longer than the others, which would cause the other streams to spend time waiting and thus increase the synchronization time.

In summary, our approach promotes efficient utilization of GPU resources, ensuring that the computational power of the GPU is used to its fullest extent. This leads to improved performance and throughput in multi-tenant inference serving systems.

4.2.2 Model Weights Rotation

Algorithm 3: Model weights rotation

```
1  input  : hidden size: H, total size: T, input tensors: I, model weights : W, tenants
            index list: L, number of tenants: N
   output : output tensors: O
2  id ← CUDA threads id;
3  for id ← 0 to T do
4      finish the operations of unrelated model weights;
5      for tenants_idx ← 0 to N do
6          if id == L[tenants_idx] then
7              float val = O[id];
8              val = val + ldg(W[id ÷ H + H × tenants_idx]) Break
```

In this work, we propose a technique called Model Weight Rotation for optimizing memory-bound operators. While multiple tenants share the same operators, there is a need to manage the different model weights associated with each tenant. Given the variations in layout between input tensors and model weights for multiple tenants, we introduce the tenants' index list as an additional input parameter for these operators. This approach allows us to handle the different weights associated with each tenant.

Algorithm 3 provides an example of how Model Weight Rotation works in the context of add-bias operators. We use the CUDA thread ID to compare with the index number (line 1). When the threads execute the target token, the operator switches the model weights to the next tenant (lines 2 - 8). Although this rotation introduces some additional computation, the overhead is less than the overhead associated with multiple kernel launches. This makes it a more efficient approach for handling multiple tenants.

However, optimizing memory-bound operators is not just about managing different model weights. It's also about efficient resource allocation. During the Inference Runtime, memory-bound operators have exclusive access to the GPU, making resource allocation a critical factor in their performance. Ideally, we would allocate the same number of threads as the total size to maximize parallelism. However, when the number of allocated threads exceeds the maximum number of threads that the GPU can handle, the device switches to sequential mode. This switch causes the exceeding threads to wait for dispatch, leading to inefficiencies.

To address this issue, we adhere to the GPU's capability when allocating resources for kernel functions. In this work, we denote the number of threads per block as T, the number of blocks as B, the number of Stream MultiProcessors as SM, and the maximum number of resident blocks per SM as R. Our allocation strategy are as follows:

$$T = \min(H, 1024) \tag{4.3}$$

$$B = \min(R \times SM, \frac{Total\ size}{T}) \tag{4.4}$$

Based on device limitations, we only dispatch 1024 threads per block when the hidden size exceeds 1024. Furthermore, we set the maximum number of blocks to align with the device's

capability. This alignment reduces the overhead of redundant thread allocation, leading to more efficient use of GPU resources.

4.2.3 Tensor Transformation

As discussed in section 3.3, we utilize Tensor Transformation for Inference Processing. The concept of Tensor Transformation has been previously employed in ORCA, a work that leverages this method to process GPT-3. In the context of Inference Processing, when executing the Multi-Head Attention (MHA) layers, batched tensors are split, and tensors are merged with other layers.

In ITIF, we adopt a fine-grained selective batching approach. This approach involves splitting the requests into mini-batches for compute-bound operators and merging them for memory-bound operators. However, a challenge arises with ORCA in that the order of requests is different.

To overcome this challenge, we employ a linked list to store information during the batching stage. Each node in the linked list represents a request. Once the Request Handler completes the batching and padding process, the memory addresses and request sizes are written into the nodes. During the batching stage, sorting only alters the location information in the node, not its GPU memory address. This approach significantly reduces the GPU memory footprint.

As illustrated in Figure 3.4, we leverage the information from the linked list to perform the Tensor Transformation. When the Inference Runtime executes the compute-bound operators, the tensors are split into the mini-batch format. They are then merged back for the memory-bound operators.

One of the key advantages of using a linked list in ITIF is that it eliminates the need to actually merge or split tensors, and the memory address is not changed. This approach maintains

the integrity of the original data while allowing for efficient manipulation of the tensors. It provides a flexible and efficient way to manage the tensors during Inference Processing, leading to improved performance and reduced memory usage.

CHAPTER 5

Experiments

In this section, we present a comprehensive set of experiments designed to evaluate the performance and efficiency of our proposed ITIF. Our experiments aim to validate the effectiveness of the techniques and strategies we have introduced, including request batching, model integration, GEMM operator scheduling, Model Weights Rotation, and Tensor Transformation.

We conduct our experiments on various Transformer models, such as BERT, ALBERT, ViT, and decoding to demonstrate the versatility and adaptability of ITIF. We compare the performance of ITIF with other state-of-the-art inference serving systems.

Our evaluation metrics include throughput, latency, and GPU utilization. We also analyze the impact of different batch sizes, sequence lengths, and the number of tenants on the system's performance.

The following subsections describe our experimental setup, the datasets used, and the results obtained. We also offer a thorough discussion of the results, highlighting the strengths of our approach and potential areas for future improvement.

5.1 Experiment setup

In this section, we provide information about the experiment setup, including our experiment environment, used models, and the baselines we compared.

5.1.1 Experimental System

The implementation of our system is built upon the foundation of FasterTransformer [NVI22b], an NVIDIA inference runtime framework developed in C++. FasterTransformer is designed for high-performance inference of Transformer-based models, making it an ideal starting point for our system.

In order to adapt to the multi-tenant scenario, we have made several modifications to the original FasterTransformer framework. Specifically, we have rewritten memory-bound operators, including Layer Normalization (LayerNorm), Add bias, and activation functions. These modifications allow us to handle the different model weights associated with each tenant and to optimize the execution of these operators in a multi-tenant environment.

In addition to these modifications, we also leverage Cublas and CUDA streams to implement the compute-bound operators. Cublas is a GPU-accelerated version of the BLAS (Basic Linear Algebra Subprograms) library, which provides optimized routines for operations on vectors and matrices. CUDA streams, on the other hand, are sequences of operations that execute in order on the GPU. By using multiple streams, we can have multiple operations executed concurrently, thereby improving performance.

Regarding hardware configuration, our system is deployed on a GPU-based server. The server is equipped with an AMD Threadripper PRO 3945WX CPU, 128GB of RAM, and an NVIDIA RTX A5000 GPU. This high-performance hardware setup enables us to fully exploit

TABLE 5.1. Transformer models and parameters

Model	Parameters
BERT_base	num_layer=12, num_head=12, hidden_size=768
ALBERT_large	num_layer=24, num_head=16, hidden_size=1024
ViT	num_layer=12, num_head=12, hidden_size=768
Decoding	num_layer=6, num_head=8 beam_size=4 ,hidden_size=512

the capabilities of our system. The server runs on the Ubuntu 20.04 operating system, which provides a stable and efficient environment for our experiments.

5.1.2 Models

To evaluate the versatility and performance of our system across different domains, we employ a range of Transformer models. These models include BERT, ALBERT, ViT, and Decoding, which are representative of the state-of-the-art in Natural Language Processing (NLP) and Computer Vision (CV) tasks.

BERT (Bidirectional Encoder Representations from Transformers) [KT19] and ALBERT (A Lite BERT) [Lan+19] are models primarily used in NLP tasks, providing high performance in tasks such as language understanding and sentiment analysis. ViT (Vision Transformer) [Dos+20] is a model used in CV tasks, demonstrating the applicability of Transformer models beyond text-based tasks. Decoding [Vas+17] is a model that focuses on the evaluation of the Transformer decoder, which is a critical component of many Transformer-based models.

These models are constructed using the Transformer encoder, with the exception of Decoding, which focuses on the Transformer decoder. The details of these models and their parameters are presented in Table 5.1.

To assess the impact of model size on the performance of our system, we select different versions of these models, including BERT_base, ALBERT_large, ViT_base, and Decoding. These versions consist of 12 and 24 layers of encoders and six layers of decoders, respectively, offering a range of model complexities for evaluation.

Furthermore, the sizes of the heads in the Multi-Head Attention (MHA) component of these models also vary. The MHA is a key component of Transformer models, and its size can have a significant impact on the model's performance and resource requirements. By evaluating models with different head sizes, we can gain insights into how our system performs under different conditions.

5.1.3 Baselines

To evaluate the performance of our system, we construct three baselines for assessment and comparison. These baselines represent different strategies for handling multi-tenant scenarios. By comparing our system's performance with these baselines, we can gain insights into the effectiveness of our approach. To isolate the impact of other factors on performance, we exclusively choose FasterTransformer as the baseline and implement three different multi-tenant scenarios.

- **SEQ**: The first baseline represents a time-sharing strategy for the GPU, which is the default scheduling strategy for multi-tenant scenarios. In this strategy, known as sequential execution, each tenant is deployed on a corresponding CPU thread. These threads concurrently launch the inference, allowing multiple tenants to share the GPU over time.
- **Multi-stream**: The second baseline builds upon the SEQ implementation. In this strategy, we construct several CUDA streams and deploy each tenant on a corresponding CPU thread and CUDA stream. CUDA streams are sequences of

operations that execute in order on the GPU. Using multiple streams, we can have multiple operations executed concurrently, improving performance.

- **MPS**: The third baseline leverages Nvidia Multi-Process Service (MPS) as the scheduling strategy. MPS allows for the execution of multiple inference instances at the same time, which is a form of implicit spatial sharing of the GPU. Spatial sharing refers to the practice of having multiple tenants use different parts of the GPU at the same time, as opposed to time-sharing, where tenants use the entire GPU at different times.

These baselines provide a range of strategies for handling multi-tenant scenarios, from simple time-sharing to more complex spatial-sharing strategies. By comparing our system's performance with these baselines, we can evaluate the effectiveness of our approach in different scenarios.

5.2 Overall Performance Evaluation

Firstly, we provide an overall performance evaluation, which contains four tenants, and the number of requests is four. We build the input of BERT, ALBERT, and Decoding models as the randomly generated text with a maximum sequence length of 128. For the ViT, we conduct the inference on ImageNet [Den+09], which provides the image of size $224\times224\times3$. We profile three baselines for four models and record the average latency (ms). Based on the averaged latency, we calculate the acceleration ratio between ITIF and baselines in Figure 5.1.

The proposed system demonstrates a substantial acceleration in performance, ranging from a $1.35\times$ to a $2.40\times$ increase, consistently surpassing the Sequential (SEQ) approach in four distinct model tests. Furthermore, ITIF exhibits a more pronounced improvement compared to both the MPS and Multi-stream approaches. Upon examining experiments involving MPS, it was observed that its performance exhibits fluctuations in smaller single tasks with reduced

sequence lengths, attributed to disparities in the processing speeds of different processes. In contrast, our ITIF optimization yields more consistent results by integrating all tenants into a single instance, thereby contributing to enhanced stability in performance.

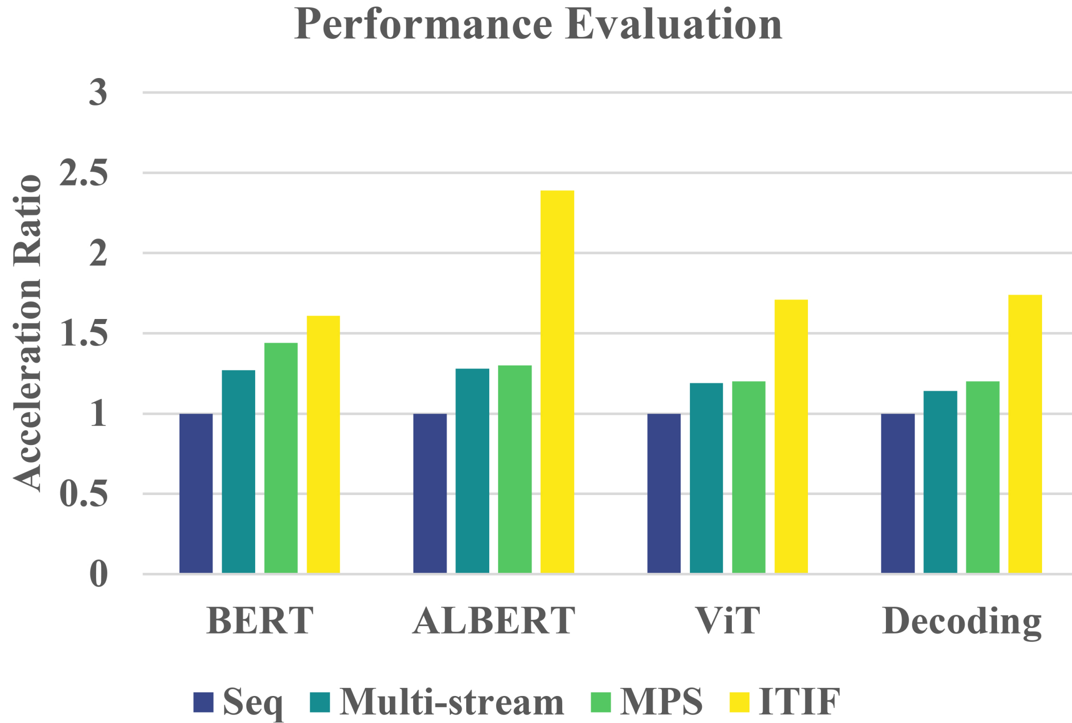


FIGURE 5.1. The performance evaluation for the four different transformer models. We evaluate the acceleration ratio with three baselines: Sequential execution, MPS, and multi-stream execution. The experiments contain four tenants for each model, and the number of total requests is 4.

Compared to CV models, our system delivers better performance for BERT and ALBERT because the input sizes of NLP and CV models differ. After patching and embedding, an image of size 224×224×3 is reconstructed into an input tensor with a sequence length of 197, which is larger than the input of NLP models. The increased input size allocates more computational resources for a single operator, leading to significant resource contention on the GPU. Therefore, smaller input sizes are more beneficial for spatial sharing of the GPU in a multi-tenant scenario.

On the other hand, although both BERT and ALBERT are constructed using the Transformer encoder, there is a notable performance gap between them. ALBERT has a deeper model structure, with twice the number of encoder layers as BERT, implying that ALBERT contains twice the number of operators to be executed. As a result, our system performs much better when handling models with a large number of operators. For Decoding, ITIF provides a 1.63× speedup compared to sequential execution because the number of heads is fewer than that in BERT, meaning that fewer computational resources are needed in MHA.

5.3 Scalability Evaluation

We evaluate the scalability of our system with varying numbers of tenants on a single GPU. We maintained the same settings as in the previous experiment and compared the results for different numbers of tenants. We set the maximum number of tenants to five due to GPU memory limitations.

The evaluation results are presented in Table 5.2. In general, ITIF outperforms the other three methods and becomes more effective as the number of tenants increases. However, when the number of tenants reaches 5, the optimization effect weakens due to the device's computing capability saturation. ITIF shows the best performance for different models and a speedup ranging from 1.87× to 2.40× compared to sequential execution for ALBERT. In comparison with Multi-stream execution, we still observe an improvement of up to 2×. Based on the results, we conclude that system performance improves with an increase in the number of tenants. There are two reasons for this performance improvement. Firstly, when the batch size is small, CUDA kernels require only a small number of CUDA threads to support execution. The GPU serves a single kernel function without optimization. The number of active CUDA threads does not meet the upper bound of overall CUDA threads, resulting in wasted computational resources. Secondly, as the number of tenants increases, the overall number of operators in the system also rises, causing the system to launch more

TABLE 5.2. Performance Evaluation for varied numbers of tenants (N. T.). We maximize the Input sequence length for BERT and AlBERT: 128, the input image size: 224×224×3. Latency (ms)

MODELS	N. T.	SEQ	MULTI-STREAM	MPS	ITIF	IMP.
BERT	2	6.90	6.06	5.14	**4.94**	**1.39**×
	3	9.95	7.56	7.03	**6.87**	**1.44**×
	4	13.51	10.68	9.68	**8.94**	**1.51**×
	5	16.22	12.32	12.43	**11.01**	**1.47**×
ALBERT	2	17.04	14.61	14.2	**7.82**	**2.17**×
	3	24.23	19.62	20.22	**10.63**	**2.27**×
	4	33.98	24.72	26.43	**14.14**	**2.40**×
	5	41.26	30.12	33.5	**22.05**	**1.87**×
VIT	2	9.92	8.26	9.28	**5.05**	**1.96**×
	3	14.19	11.92	13.71	**7.08**	**2.04**×
	4	19.21	16.21	18.28	**11.21**	**1.71**×
	5	23.41	18.21	22.55	**15.36**	**1.52**×
DECODING	2	46.10	32.74	30.06	**24.38**	**1.44**×
	3	70.13	57.13	50.88	**40.48**	**1.73**×
	4	94.42	82.32	78.31	**54.21**	**1.74**×
	5	117.77	101.82	95.32	**70.30**	**1.67**×

kernel functions on the GPU. This is also why the performance of ITIF is better than that of Multi-stream execution, as the overhead of kernel launching is significantly higher.

However, the performance of ITIF does not continue to improve as the number of tenants increases. According to the experimental results, the performance of BERT and ALBERT peaks when the number of tenants is 4, while the performance of ViT peaks when the number of tenants is 3. This is because when the number of tenants increases, the kernel needs more CUDA threads to improve the parallelism until increasing to the maximum number of CUDA threads. When the number of requests CUDA threads exceeds the maximum number, the kernel execution turns to the sequential execution, which affects the performance. Therefore,

the performance of ITIF is limited by the different capabilities of GPU and the corresponding number of CUDA cores.

5.3.1 Requests Batching Evaluation

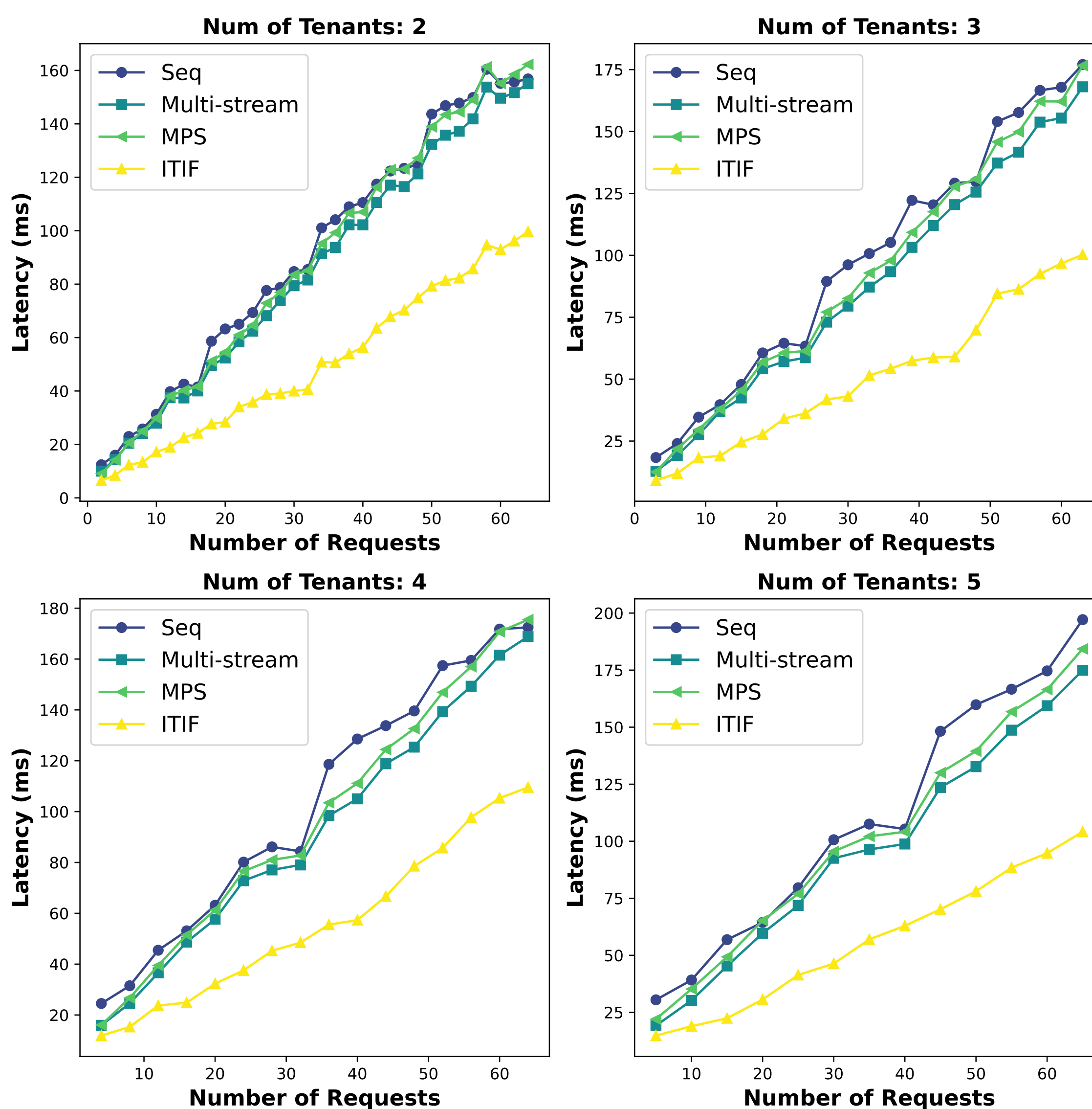


FIGURE 5.2. Benchmarking the latency of runtimes with the variable number of requests. We use ALBERT as the model of evaluation, and the maximum sequence length of each request is 64.

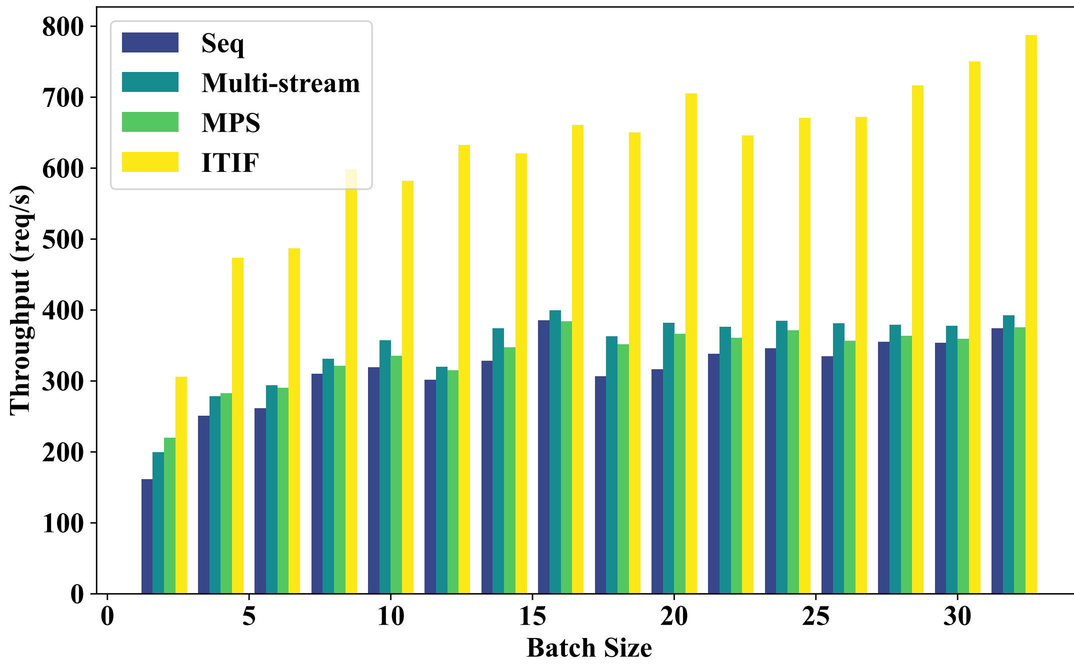
To evaluate the influence of requests batching on the system, we designed the requests batching evaluation experiment. We increased the number of requests from 2 to 64. We adjusted the number of tenants in our experiments from 2 to 5 to effectively observe the performance improvement from requests batching. For the model, we evaluated the large-scale model, ALBERT-large.

The performance results are shown in Figure 5.2. ITIF significantly improves performance over three baseline approaches in the large-scale model. In the case of ALBERT's inference, we achieve speedups compared to sequential execution ranging from 1.63× to 2.40×, with an average of 1.87×. Compared with multi-stream and MPS execution, ITIF still provides a considerable improvement in ALBERT, proving that ITIF is friendly with a large number of model layers.

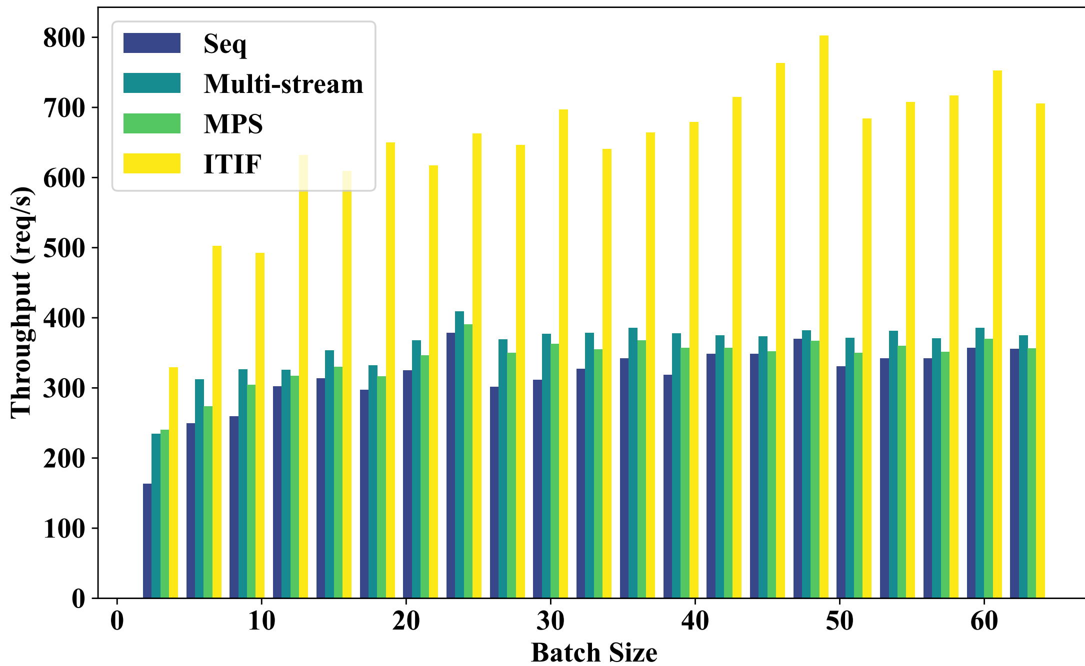
Therefore, the performance of ITIF is further improved with the optimization of requests batching when hardware computing capability allows it. Compared with requests batching for a single tenant, ITIF can use less time to receive enough requests. Although the number of requests is the same for the different tenants, the SEQ and Multi-stream have extra overhead from the rising number of tenants. ITIF is not impacted by it.

We further analyzed the throughput on different batch sizes. Similar to the previous experiments, we choose Albert as the experimental model to verify the throughput improvement in the multi-tenant scenario by continuously increasing the batch size. We increased the number of tenants from 2 to 5 and the batch size to a maximum of 64 to test the number of inference requests executed per unit time (one second). The reason for choosing these experiment parameters is that our experiment device (RTX A5000) has the capability bound. If there are more tenants or batch sizes in this device, ITIF can not provide further improvement.

As shown in Figure 5.3 and Figure 5.4, the throughput improvement of ITIF is more significant compared to the other three baselines. On the whole, ITIF has a more significant improvement in throughput. Especially when the number of tenants is small, ITIF has a significant upward

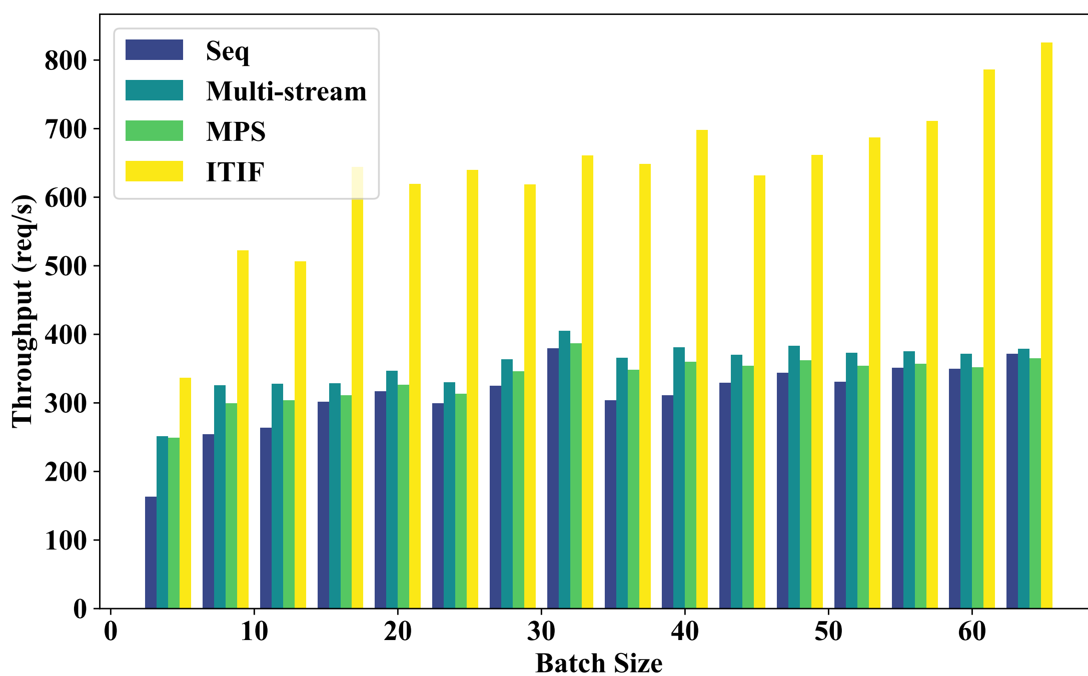


(a) Number of Tenants: 2

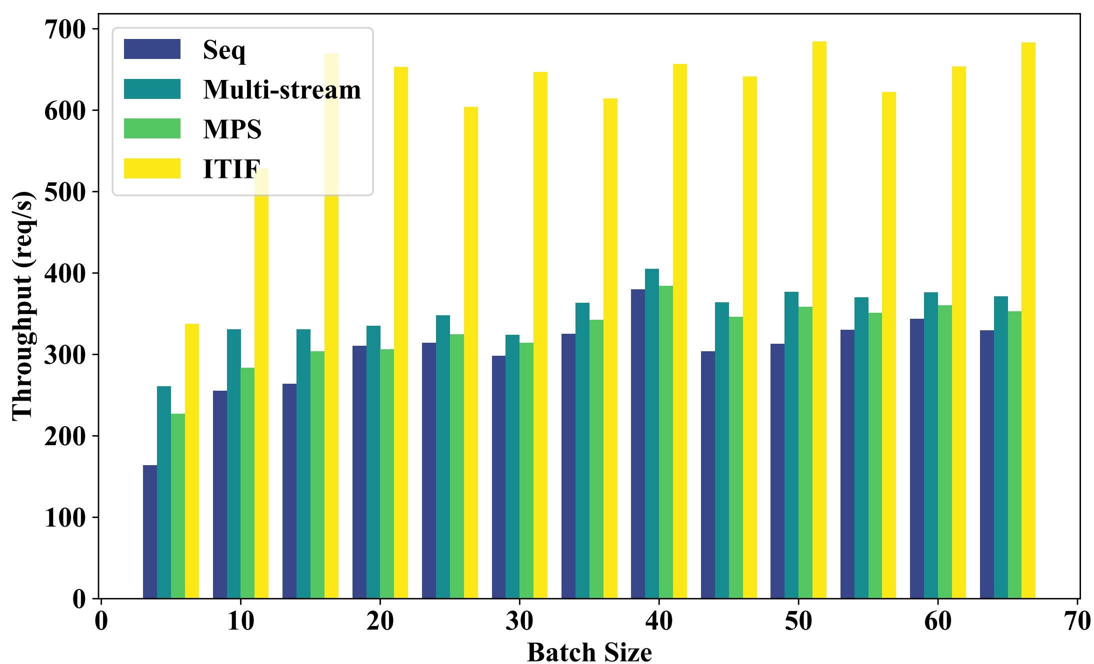


(b) Number of Tenants: 3

FIGURE 5.3. Benchmarking the throughput of runtimes with the variable number of requests. We use ALBERT as the model of evaluation, and the maximum sequence length of each request is 64.



(a) Number of Tenants: 4



(b) Number of Tenants: 5

FIGURE 5.4. Benchmarking the throughput of runtimes with the variable number of requests. We use ALBERT as the model of evaluation, and the maximum sequence length of each request is 64.

trend in throughput as the batch size increases. For other baselines, the change in batch size does not improve the throughput. This may be due to the fact that the number of inputs has reached its upper limit on GPU occupancy and cannot further improve performance. On the other hand, the parallelization of ITIF is better than several other baselines, so the performance improvement is obvious. And in the high-tenancy case (5 tenants), the number of inputs reached the upper limit of ITIF, so the throughput did not rise consistently.

5.4 Evaluation of Operation Level Optimization

We analyzed the performance of kernel functions in the BERT by the NVIDIA Nsight system [NVI22c] to profiling. Firstly, we focus on the performance of the memory-bound operators to evaluate our **Model Weight Rotation** approach.

TABLE 5.3. Execution time (ms) of memory-bound operators in BERT

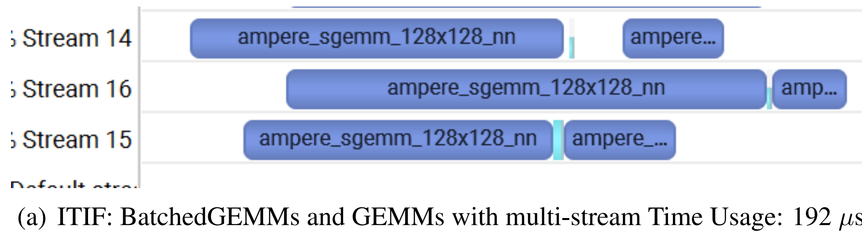
OPERATORS	SEQ	MULTI-STREAM	MPS	ITIF
SOFTMAX	0.333	0.790	0.432	**0.356**
ADDBIAS+LAYERNORM	0.291	0.606	0.325	**0.310**
ADDBIAS+GELU	0.245	0.847	0.349	**0.275**
TRANSPOSE	0.209	0.519	0.289	**0.220**

The execution of memory-bound operators is presented in Table 5.3. Compared to sequential execution, ITIF exhibits approximately 5% performance degradation, which stems from the overhead of Model Weight Rotation.

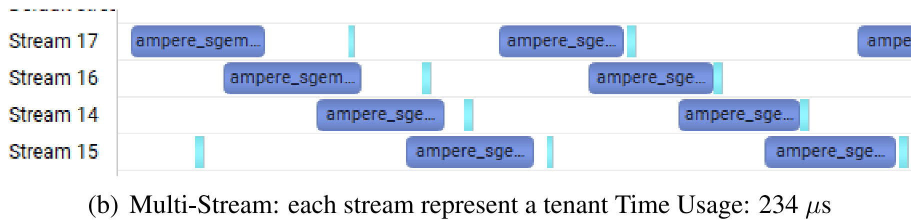
The poor performance of Multi-stream execution can be attributed to operators being concurrently executed in several CUDA streams. Although the overall throughput experiences a speedup, resource contention for concurrent kernel execution negatively impacts single kernel performance. Our approach fuses them into a single kernel to avoid the overhead of

TABLE 5.4. cudaLaunchKernel analysis in BERT

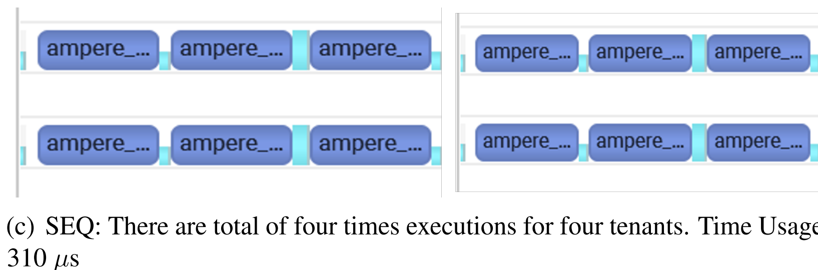
BASELINE	INSTANCES	TIME USAGE(MS)
SEQ	776	2.834
MULTI-STREAM	776	6.757
MPS	776	5.324
ITIF	**517**	**1.817**



(a) ITIF: BatchedGEMMs and GEMMs with multi-stream Time Usage: 192 μs



(b) Multi-Stream: each stream represent a tenant Time Usage: 234 μs



(c) SEQ: There are total of four times executions for four tenants. Time Usage: 310 μs

FIGURE 5.5. "Get Q,K,V" Operator in Transformer.

Multi-stream and enhance GPU utilization. Furthermore, our approach reduces the number of instances and the overhead of CUDA kernel launching. Table 5.4 demonstrates that our approach decreases the number of kernels by 33.4% and saves 1 ms in BERT's time usage for cudaLaunchKernel.

Figure 5.5 presents the profiling results of compute-bound operators obtained from the Nsight system, emphasizing the evaluation of our **CUDA Stream Optimization**. The experiment

involves four tenants, each with four requests and a sequence length of 64 for BERT, with the "GET Q,K,V" operator analyzed individually for SEQ, Multi-stream, and ITIF. Compared to the baseline approaches, Multi-stream and SEQ, the ITIF method demonstrates a more substantial performance improvement derived from the CUDA Stream. The findings reveal that excessive CUDA streams lead to increased overhead for Multi-stream, while SEQ cannot spatially share GPU resources effectively. ITIF's unique mechanism integrates the arithmetic of all tenants, enabling it to merge the GEMMs into BatchGEMMs, thereby reducing kernel launch overhead and capitalizing on the benefits provided by CUDA Streams.

CHAPTER 6

Conclusion

In this work, we address the challenge of spatial sharing for multiple tenants on a single GPU. Given the high demand for inference models on servers in cloud environments, we propose an optimization strategy that deploys the same backbone model on the same GPU. This approach differs from single-tenant model optimization, as our focus is on improving resource allocation and utilization of the GPU to maximize overall performance.

Our strategy involves integrating multiple tenant models into a single instance. We achieve this by leveraging request batching and model integration approaches, which effectively address the issue of unpredictable latency. Unpredictable latency can be a significant problem in multi-tenant environments, as the varying computational demands of different models can lead to inconsistent response times. By integrating multiple models into a single instance, we can better manage these demands and provide more consistent performance.

For the inference runtime, we design a scheduling system specifically for the GEMM operators, which are a key component of many machine learning models. We utilize CUDA streams and batchedGEMM methods to optimize these operators. CUDA streams allow for concurrent execution of operations, while batchedGEMM provides an efficient way to perform multiple matrix multiplications. In addition, we propose a novel approach called model weight rotation to combine the same memory-bound operators as a single CUDA kernel. This approach reduces the overhead of kernel launches, which can be a significant source of inefficiency in GPU computations. By combining multiple operations into a single kernel, we can reduce

the number of kernel launches and thus improve performance. Moreover, our novel batching request approach also provides significant performance improvement, which is the key to integrating the different tenants.

In summary, our ITIF provides a significant speedup compared to the sequential execution of the FasterTransformer. Based on our evaluation results, ITIF achieves a speedup of 1.12× to 2.40× in four different transformer models. This demonstrates the effectiveness of our approach in improving the performance of multi-tenant inference on a single GPU.

Bibliography

[20] *OpenCL*. Open Computing Language (OpenCL). Version 3.0. 2020. URL: %7Bhttps://www.khronos.org/opencl/%7D.

[Aba+16] Martıén Abadi et al. 'Tensorflow: a system for large-scale machine learning.' In: *Osdi*. Vol. 16. 2016. Savannah, GA, USA. 2016, pp. 265–283.

[Amz17] Amzon. *Amzon SageMaker*. 2017. URL: https://aws.amazon.com/sagemaker/.

[Bai+21] Yang Bai et al. 'AutoGTCO: Graph and Tensor Co-Optimize for Image Recognition with Transformers on GPU'. In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE. 2021, pp. 1–9.

[BH19] Tal Ben-Nun and Torsten Hoefler. 'Demystifying parallel and distributed deep learning: An in-depth concurrency analysis'. In: *ACM Computing Surveys (CSUR)* 52.4 (2019), pp. 1–43.

[BR21] Mikhail Burtsev and Anna Rumshisky. 'Multi-Stream Transformers'. In: *arXiv preprint arXiv:2107.10342* (2021).

[Bro+20] Tom Brown et al. 'Language models are few-shot learners'. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[CGM14] John Cheng, Max Grossman and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.

[CH02] Katherine Compton and Scott Hauck. 'Reconfigurable computing: a survey of systems and software'. In: *ACM Computing Surveys (csuR)* 34.2 (2002), pp. 171–210.

[Che+14] Sharan Chetlur et al. 'cudnn: Efficient primitives for deep learning'. In: *arXiv preprint arXiv:1410.0759* (2014).

[Che+18] Tianqi Chen et al. '{TVM}: An automated {End-to-End} optimizing compiler for deep learning'. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.

[Cho+15] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[CKR21] Yujeong Choi, Yunseong Kim and Minsoo Rhu. 'Lazy batching: An sla-aware batching system for cloud machine learning inference'. In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 493–506.

[Cra+17] Daniel Crankshaw et al. 'Clipper: A {Low-Latency} Online Prediction Serving System'. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 613–627.

[CW14] Erik Cambria and Bebo White. 'Jumping NLP curves: A review of natural language processing research'. In: *IEEE Computational intelligence magazine* 9.2 (2014), pp. 48–57.

[Dea+12] Jeffrey Dean et al. 'Large scale distributed deep networks'. In: *Advances in neural information processing systems* 25 (2012).

[Den+09] Jia Deng et al. 'Imagenet: A large-scale hierarchical image database'. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[DKR20] Aditya Dhakal, Sameer G Kulkarni and KK Ramakrishnan. 'Gslice: controlled spatial sharing of gpus for a scalable inference platform'. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 492–506.

[Dos+20] Alexey Dosovitskiy et al. 'An image is worth 16x16 words: Transformers for image recognition at scale'. In: *arXiv preprint arXiv:2010.11929* (2020).

[Fan+21] Jiarui Fang et al. 'TurboTransformers: an efficient GPU serving system for transformer models'. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021, pp. 389–402.

[For94] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. USA, 1994.

[Fu+22] Boqian Fu et al. 'TCB: Accelerating Transformer Inference Services with Request Concatenation'. In: *Proceedings of the 51st International Conference on Parallel Processing*. 2022, pp. 1–11.

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

[Goo17] Google. *XLA*. 2017. URL: https://www.tensorflow.org/xla.

[Gri+20] Sorin Grigorescu et al. 'A survey of deep learning techniques for autonomous driving'. In: *Journal of Field Robotics* 37.3 (2020), pp. 362–386.

[GSC00] Felix A Gers, Jürgen Schmidhuber and Fred Cummins. 'Learning to forget: Continual prediction with LSTM'. In: *Neural computation* 12.10 (2000), pp. 2451–2471.

[Han+15] Song Han et al. 'Learning both weights and connections for efficient neural network'. In: *Advances in neural information processing systems* 28 (2015).

[Han+22] Mingcong Han et al. 'Microsecond-scale Preemption for Concurrent {GPU-accelerated}{DNN} Inferences'. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 539–558.

[Haz+18] Kim Hazelwood et al. 'Applied machine learning at facebook: A datacenter infrastructure perspective'. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 620–629.

[He+16] Kaiming He et al. 'Deep residual learning for image recognition'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[He+22] Kaiming He et al. 'Masked autoencoders are scalable vision learners'. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 16000–16009.

[Hon+21] Danfeng Hong et al. 'SpectralFormer: Rethinking hyperspectral image classification with transformers'. In: *IEEE Transactions on Geoscience and Remote Sensing* 60 (2021), pp. 1–15.

[How+17] Andrew G Howard et al. 'Mobilenets: Efficient convolutional neural networks for mobile vision applications'. In: *arXiv preprint arXiv:1704.04861* (2017).

[HPM19] Kyu J Han, Ramon Prieto and Tao Ma. 'State-of-the-art speech recognition using multi-stream self-attention with dilated 1d convolutions'. In: *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE. 2019, pp. 54–61.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. 'Long short-term memory'. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[Hua+19] Yanping Huang et al. 'Gpipe: Efficient training of giant neural networks using pipeline parallelism'. In: *Advances in neural information processing systems* 32 (2019).

[HVD15] Geoffrey Hinton, Oriol Vinyals and Jeff Dean. 'Distilling the knowledge in a neural network'. In: *arXiv preprint arXiv:1503.02531* (2015).

[Jai+18] Paras Jain et al. 'Dynamic space-time scheduling for gpu inference'. In: *arXiv preprint arXiv:1901.00041* (2018).

[Jia+14] Yangqing Jia et al. 'Caffe: Convolutional architecture for fast feature embedding'. In: *Proceedings of the 22nd ACM international conference on Multimedia*. 2014, pp. 675–678.

[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*. ACM New York, NY, USA. 2017, pp. 1–12.

[Kel19] John D Kelleher. *Deep learning*. MIT press, 2019.

[Kri14] Alex Krizhevsky. 'One weird trick for parallelizing convolutional neural networks'. In: *arXiv preprint arXiv:1404.5997* (2014).

[KSH17] Alex Krizhevsky, Ilya Sutskever and Geoffrey E Hinton. 'Imagenet classification with deep convolutional neural networks'. In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[KT19] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: *Proceedings of NAACL-HLT*. 2019, pp. 4171–4186.

[Lai+21] Fan Lai et al. 'Oort: Efficient Federated Learning via Guided Participant Selection.' In: *OSDI*. 2021, pp. 19–35.

[Lan+19] Zhenzhong Lan et al. 'Albert: A lite bert for self-supervised learning of language representations'. In: *arXiv preprint arXiv:1909.11942* (2019).

[LBH15] Yann LeCun, Yoshua Bengio and Geoffrey Hinton. 'Deep learning'. In: *nature* 521.7553 (2015), pp. 436–444.

[LeC+98] Yann LeCun et al. 'Gradient-based learning applied to document recognition'. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[Lev+20] Sergey Levine et al. 'Offline reinforcement learning: Tutorial, review, and perspectives on open problems'. In: *arXiv preprint arXiv:2005.01643* (2020).

[Li+22] Jinyu Li et al. 'Recent advances in end-to-end automatic speech recognition'. In: *APSIPA Transactions on Signal and Information Processing* 11.1 (2022).

[Lia+15] Xiangru Lian et al. 'Asynchronous parallel stochastic gradient for nonconvex optimization'. In: *Advances in neural information processing systems* 28 (2015).

[LK17] Marc Moreno Lopez and Jugal Kalita. 'Deep Learning applied to NLP'. In: *arXiv preprint arXiv:1703.03091* (2017).

[LLL21] Jingyu Lee, Yunxin Liu and Youngki Lee. 'ParallelFusion: towards maximum utilization of mobile GPU for DNN inference'. In: *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*. 2021, pp. 25–30.

[Min+21] Shervin Minaee et al. 'Deep learning–based text classification: a comprehensive review'. In: *ACM computing surveys (CSUR)* 54.3 (2021), pp. 1–40.

[Niu+21] Wei Niu et al. 'DNNFusion: accelerating deep neural networks execution with advanced operator fusion'. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 883–898.

[NVF20] NVIDIA, Péter Vingelmann and Frank H.P. Fitzek. *CUDA, release: 10.2.89*. 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[NVI21a] NVIDIA Corporation. *CUDA Multi Process Service Overview*. 2021. URL: https://docs.nvidia.com/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[NVI21b] NVIDIA Corporation. *NVIDIA CUDA C/C++ Streams and Concurrency*. 2021. URL: http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf.

[NVI21c] NVIDIA Corporation. *NVIDIA Multi-Instance GPU*. 2021. URL: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[NVI21d] NVIDIA Corporation. *NVIDIA TensorRT*. url = https://developer.nvidia.com/tensorrt/. 2021.

[NVI22a] NVIDIA Corporation. *cuBLAS*. Accessed: 2016-05-14. 2022. URL: https://github.com/clMathLibraries/clBLAS.

[NVI22b] NVIDIA Corporation. *FasterTransformer*. Version 5.0. 2022. URL: https://github.com/NVIDIA/FasterTransformer.

[NVI22c] NVIDIA Corporation. *NVIDIA Nsight™ Systems*. 2022. URL: https://developer.nvidia.com/nsight-systems.

[NZK20] Supun Nakandala, Yuhao Zhang and Arun Kumar. 'Cerebro: A data system for optimized deep learning model selection'. In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2159–2173.

[Ols+17] Christopher Olston et al. 'TensorFlow-Serving: Flexible, High-Performance ML Serving'. In: (2017).

[Ope21] OpenAI. *GPT-3.5 [Language model]*. https://openai.com/research/. Accessed: March 24, 2023. 2021.

[Owe+08] John D Owens et al. 'GPU computing'. In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[Par+18] Jongsoo Park et al. 'Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications'. In: *arXiv preprint arXiv:1811.09886* (2018).

[Pas+19] Adam Paszke et al. 'Pytorch: An imperative style, high-performance deep learning library'. In: *Advances in neural information processing systems* 32 (2019).

[QLC22] Zhao-Wei Qiu, Kun-Sheng Liu and Ya-Shu Chen. 'BARM: A Batch-Aware Resource Manager for Boosting Multiple Neural Networks Inference on GPUs With

Memory Oversubscription'. In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 4612–4624.

[Rad+18] Alec Radford et al. 'Improving language understanding by generative pre-training'. In: (2018).

[Rad+19] Alec Radford et al. 'Language models are unsupervised multitask learners'. In: *OpenAI blog* 1.8 (2019), p. 9.

[Ras+16] Mohammad Rastegari et al. 'Xnor-net: Imagenet classification using binary convolutional neural networks'. In: *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV*. Springer. 2016, pp. 525–542.

[RHW86] David E Rumelhart, Geoffrey E Hinton and Ronald J Williams. 'Learning representations by back-propagating errors'. In: *nature* 323.6088 (1986), pp. 533–536.

[Rom+21] Robin Rombach et al. *High-Resolution Image Synthesis with Latent Diffusion Models*. 2021. arXiv: 2112.10752 [cs.CV].

[RT18] Alessandro Raganato and Jörg Tiedemann. 'An analysis of encoder representations in transformer-based machine translation'. In: *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. The Association for Computational Linguistics. 2018.

[She+19] Haichen Shen et al. 'Nexus: A GPU cluster engine for accelerating DNN-based video analysis'. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 322–337.

[Sil+16] David Silver et al. 'Mastering the game of Go with deep neural networks and tree search'. In: *nature* 529.7587 (2016), pp. 484–489.

[Sil+17] David Silver et al. 'Mastering chess and shogi by self-play with a general reinforcement learning algorithm'. In: *arXiv preprint arXiv:1712.01815* (2017).

[SZ14] Karen Simonyan and Andrew Zisserman. 'Very deep convolutional networks for large-scale image recognition'. In: *arXiv preprint arXiv:1409.1556* (2014).

[Vas+17] Ashish Vaswani et al. 'Attention is all you need'. In: *Advances in neural information processing systems* 30 (2017).

[Wan+20] Xiaohui Wang et al. 'LightSeq: A high performance inference library for transformers'. In: *arXiv preprint arXiv:2010.13887* (2020).

[WWB19] Yu Emma Wang, Gu-Yeon Wei and David Brooks. 'Benchmarking TPU, GPU, and CPU platforms for deep learning'. In: *arXiv preprint arXiv:1907.10701* (2019).

[Yan+19] Zhilin Yang et al. 'Xlnet: Generalized autoregressive pretraining for language understanding'. In: *Advances in neural information processing systems* 32 (2019).

[Yin+17] Wenpeng Yin et al. 'Comparative study of CNN and RNN for natural language processing'. In: *arXiv preprint arXiv:1702.01923* (2017).

[Yu+21] Fuxun Yu et al. 'Automated Runtime-Aware Scheduling for Multi-Tenant DNN Inference on GPU'. In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE. 2021, pp. 1–9.

[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)*. 2022, pp. 521–538.

[Zha+15] Chen Zhang et al. 'Optimizing FPGA-based accelerator design for deep convolutional neural networks'. In: *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. 2015, pp. 161–170.

[Zha+20] Xinyan Zhao et al. 'Condition aware and revise transformer for question answering'. In: *Proceedings of The Web Conference 2020*. 2020, pp. 2377–2387.

[Zha+22] 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.

[Zho+20] Qihua Zhou et al. 'Petrel: Heterogeneity-aware distributed deep learning via hybrid synchronization'. In: *IEEE Transactions on Parallel and Distributed Systems* 32.5 (2020), pp. 1030–1043.

[Zho+22] Zhe Zhou et al. '{PetS}: A Unified Framework for {Parameter-Efficient} Transformers Serving'. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 489–504.