Dynamic Operator Optimization for Efficient Multi-Tenant LoRA Model Serving

Changhai Zhou[1*†], **Yuhua Zhou**[2 4*], **Shiyang Zhang**[3*], **Yibin Wang**[1], **Zekai Liu**[1]

[1] School of Computer Science, Fudan University
[2] College of Computer Science and Technology, Zhejiang University
[3] Columbia University
[4] Zhejiang Lab
{zhouch23, zkliu23}@m.fudan.edu.cn, zhouyuhua@zju.edu.cn, sz3209@columbia.edu, yibinwang1121@163.com

Abstract

Low-Rank Adaptation (LoRA) has become increasingly popular for efficiently fine-tuning large language models (LLMs) with minimal resources. However, traditional methods that serve multiple LoRA models independently result in redundant computation and low GPU utilization. This paper addresses these inefficiencies by introducing Dynamic Operator Optimization (Dop), an advanced automated optimization technique designed to dynamically optimize the Segmented Gather Matrix-Vector Multiplication (SGMV) operator based on specific scenarios. SGMV's unique design enables batching GPU operations for different LoRA models, significantly improving computational efficiency. The Dop approach leverages a Search Space Constructor to create a hierarchical search space, dividing the program space into high-level structural sketches and low-level implementation details, ensuring diversity and flexibility in operator implementation. Furthermore, an Optimization Engine refines these implementations using evolutionary search, guided by a cost model that estimates program performance. This iterative optimization process ensures that SGMV implementations can dynamically adapt to different scenarios to maintain high performance. We demonstrate that Dop can improve throughput by 1.30-1.46 times in a SOTA multi-tenant LoRA serving.

1 Introduction

Low-rank adaptation (LoRA) (Hu et al. 2021) has emerged as a critical technique for efficiently fine-tuning LLMs with minimal training data and hardware resources. Traditionally, serving multiple LoRA models has required treating each model independently, resulting in the need for $k \times n$ GPUs for k different LoRA models, each requiring n GPUs. This approach overlooks the potential weight correlations between some LoRA models that originate from the same pre-trained model. Specifically, independently handling each LoRA model leads to redundant storage and computation, especially when multiple models share the same pre-trained backbone. Existing systems like vLLM (Narayanan et al. 2021) and FasterTransformer (Sheng et al. 2023) have made significant progress in optimizing LLM serving by addressing issues such as memory fragmentation and enhancing the efficiency of the attention mechanism. However, these systems struggle with the concurrent handling of multiple LoRA models due to their inability to batch GPU operations for different models effectively, resulting in suboptimal GPU utilization and increased latency.

Punica (Chen et al. 2024) tackles this issue by introducing a novel operator design called Segmented Gather Matrix-Vector Multiplication (SGMV), enabling the batching of GPU operations for LoRA models. This innovation allows GPU to hold only one copy of the pre-trained model while concurrently serving multiple LoRA models, thereby significantly improving GPU efficiency in terms of both memory and computation. Punica's approach has demonstrated substantial improvements in throughput and latency, establishing it as a robust solution for multi-tenant LoRA serving.

Despite significant advancements in existing systems, they all share a common limitation: their core operators are implemented using generalized methods, which may not be optimal for specific tasks and hardware architectures. This is because different implementations of the operator are logically equivalent, but they can yield vastly different performance outcomes due to factors such as loop structures, memory access patterns, parallelization strategies, and specific hardware features. Designing a custom operator for each specialized scenario not only requires extensive debugging time but also demands deep domain knowledge.

To address this challenge, we introduce Dynamic Operator Optimization (Dop), a method that leverages advanced automated search techniques to dynamically optimize operator implementations. Unlike traditional tensor program generation methods (Chen et al. 2018a; Zheng et al. 2020b; Haj-Ali et al. 2020b) that focus on standard matrix multiplication, Dop targets the efficient implementation of the SGMV operator, a non-standard matrix multiplication operation.

The primary components of Dop are the search space constructor and the optimization engine. The search space constructor creates a hierarchical search space, decoupling high-level program structures from low-level implementation details. This separation allows for comprehensive exploration of potential optimizations tailored to different hardware architectures and application scenarios. The optimization engine then employs evolutionary algorithms guided by a cost model to iteratively refine these implementations, automatically generating and fine-tuning complete operator imple-

[*] Equal contribution.
[†] Corresponding author.

mentations without the constraints of fixed templates or sequential construction methods.

This approach goes beyond simple parameter tuning by dynamically restructuring operator implementations to enhance the performance of multi-tenant LoRA systems in different scenarios, all while maintaining computational semantics and without requiring extensive domain knowledge. Our contributions are summarized as follows:

- We identify the problem of performance variability in static SGMV implementations when serving multiple LoRA models concurrently.
- We propose Dop to dynamically optimize the SGMV operator, tailored to various hardware architectures and application scenarios.
- Applying our method to state-of-the-art serving systems can achieve a 1.30-1.46 times improvement in throughput, showcasing the significant potential of dynamic operator optimization to enhance performance in practical applications.

2 Related Work

2.1 LLM Inference Optimization

Recent advancements in optimizing large language model (LLM) inference have significantly improved efficiency and performance. Orca (Yu et al. 2022) introduced a batching strategy that splits concatenated batch inputs during the self-attention process, enhancing computational efficiency. vLLM (Narayanan et al. 2021) reduced memory fragmentation in the key-value cache by utilizing virtual pages similar to those in operating systems. FlashAttention (Dao et al. 2022) optimized self-attention by minimizing data movement with block-wise computation. Punica (Chen et al. 2024) introduced a novel operator that allows batching of GPU operations for different LoRA models, enabling a GPU to hold only a single copy of the underlying pre-trained model, significantly enhancing GPU efficiency in terms of both memory and computation. FlexGen (Sheng et al. 2023) proposed an efficient swapping schedule to maximize throughput on a single GPU, although it increases latency. S-LoRA (Sheng et al. 2024) extends these efforts by presenting a system designed for the scalable serving of many LoRA adapters, employing unified paging, heterogeneous batching, and tensor parallelism strategies to efficiently manage memory and orchestrate parallelism across GPUs. While these methods significantly advance LLM inference optimization, they do not fully address the unique challenges of multi-tenant serving environments. Our approach leverages advanced automated search techniques to generate high-performance implementations of the SGMV operator, dynamically optimizing for specific hardware and task scenarios to ensure efficient multi-tenant LoRA serving.

2.2 Automatic Tensor Program Generation

Significant progress in automatic tensor program generation and auto-tuning has been achieved through the development of scheduling languages. Halide (Ragan-Kelley et al. 2013) introduced a scheduling language that describes loop optimization primitives, suitable for both manual and automatic optimization. Halide's auto-schedulers have evolved to employ techniques like beam search and learned cost models (Mullapudi et al. 2016; Li et al. 2018; Adams et al. 2019). TVM (Chen et al. 2018a) uses a similar scheduling language and includes AutoTVM (Chen et al. 2018b), a template-guided search framework. FlexTensor (Zheng et al. 2020b) offers general templates targeting individual operators. ProTuner (Haj-Ali et al. 2020b) improves estimation accuracy in Halide's auto-scheduler with Monte Carlo tree search, primarily for image processing workloads. Ansor (Zheng et al. 2020a) introduces new search spaces and optimizations specifically for deep learning. Additionally, search-based compilation and auto-tuning have proven effective in various domains. For instance, Stock (Schkufza, Sharma, and Aiken 2013) uses random search to optimize loop-free hardware instruction sequences, while OpenTuner (Ansel et al. 2014), a framework for program auto-tuning, uses multi-armed bandit approaches within user-specified search spaces. In contrast, Ansor constructs its search space automatically. High-performance libraries like ATLAS (Whaley and Dongarra 1998) and FFTW (Frigo and Johnson 1998) have long utilized auto-tuning. More recent advancements such as NeuroVectorizer (Haj-Ali et al. 2020a) and AutoPhase (Huang et al. 2019; Haj-Ali et al. 2020c) employ deep reinforcement learning to automate program vectorization and optimize compiler phase ordering, showcasing the evolving capabilities of auto-tuning in modern computational workloads.

3 Design

3.1 Background and Motivation

Background Punica (Chen et al. 2024) is a novel system designed to efficiently serve multiple Low-Rank Adaptation (LoRA) models on shared GPU clusters. It capitalizes on the fact that LoRA models share a common pre-trained backbone, with only small, low-rank adaptations differentiating them. This shared structure allows for significant optimization in multi-tenant serving scenarios.

At the core of Punica is a new operator called Segmented Gather Matrix-Vector Multiplication (SGMV). This operator enables efficient batching of operations across different LoRA models, a key innovation that sets Punica apart from traditional serving methods. By allowing a single GPU to process requests for multiple LoRA models simultaneously, SGMV dramatically improves GPU utilization and throughput.

The system's workflow is straightforward yet effective. When a request arrives, it is routed to one of the active GPUs in the cluster. Each GPU loads the backbone pre-trained model once and dynamically swaps in the necessary LoRA components as needed. This approach enables fast cold-starts and efficient memory usage. As the model generates tokens, they are streamed back to the user via the system's frontend.

Punica's scheduler plays a crucial role in optimizing resource usage. It consolidates multi-tenant workloads onto the smallest possible set of GPUs, periodically migrating requests to maintain high utilization. This dynamic schedul-

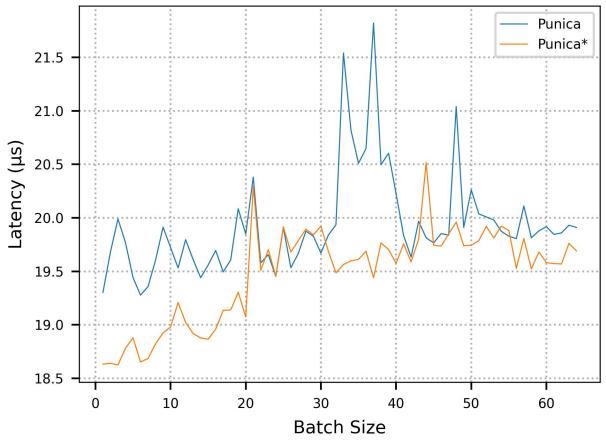

Figure 1: Latency comparison between the original SGMV implementation in Punica (blue line) and the hand-tuned SGMV implementation, referred to as Punica* (orange line), on an NVIDIA RTX 3090 GPU.

ing, combined with the SGMV operator, allows Punica to serve a large number of specialized LoRA models with significantly fewer GPU resources than conventional methods.

The Motivating Example As previously discussed, the static implementation of the SGMV operator in Punica has inherent limitations that constrain its performance across different hardware platforms. These issues become particularly evident when comparing its performance on GPUs with differing architectures. For instance, an implementation optimized for one GPU, such as the NVIDIA V100, might not perform as well on another GPU like the RTX 3090, due to differences in memory architecture and processing units. Additionally, varying request patterns for LoRA models, along with different batch sizes, can further impact the effectiveness of a static implementation.

To explore these limitations, we conducted a focused experiment comparing the original SGMV implementation in Punica with our custom hand-tuned version, referred to as Punica*. This experiment was carried out on an NVIDIA RTX 3090 GPU, specifically targeting scenarios where all requests are directed to the same LoRA model.

In this experiment, we evaluated the performance of different implementations using a pretrained model's weight matrix $\mathbf{W}$, with dimensions $[H_1, H_2]$ where $H_1 = 16$ and $H_2 = 4096$. The LoRA technique introduces two smaller matrices, $\mathbf{A}$ of shape $[H_1, r]$ and $\mathbf{B}$ of shape $[r, H_2]$, where $r = 16$ is the rank of the LoRA matrices. The model processes an input $\mathbf{x}$ as $\mathbf{y} := \mathbf{xW} + \mathbf{x}(\mathbf{AB})$. The experiments were conducted with batch sizes ranging from 1 to 64.

Each configuration was tested 1,000 times, and the average latency was recorded. As shown in Figure 1, our hand-tuned SGMV implementation (Punica*) outperforms the original Punica implementation across most batch sizes under the Identical request pattern. The orange line represents our implementation, which demonstrates lower latency compared to the original Punica implementation (indicated by the blue line).

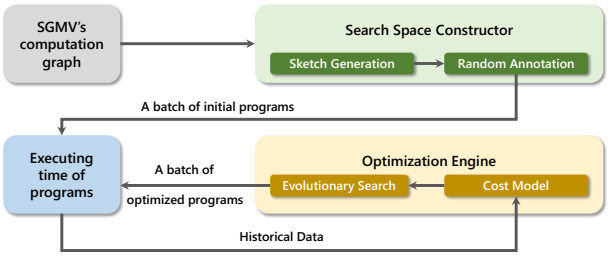

Figure 2: Overview of the Optimization Process. Dop dynamically adjusts program implementations based on predefined computation graphs and input dimensions, ensuring efficient and adaptable execution across various hardware and task scenarios.

These results highlight the necessity of a dynamic and adaptable approach to SGMV implementation, tailored to the specific characteristics of the hardware and the nature of the task. While domain experts can manually optimize these implementations, manually adjusting and testing every possible configuration is highly time-consuming and labor-intensive. The diversity of hardware platforms and task scenarios further complicates this approach, making it impractical in real-world applications. This underscores the need for an automated method that can dynamically optimize operators, efficiently exploring and evaluating a wide range of implementation possibilities. The following sections will elaborate on how we achieve this dynamic optimization in our approach.

3.2 Overview

As shown in Figure 2, Dop starts by generating a batch of initial programs from the predefined SGMV computation graph using the search space constructor. The generation of these initial programs is hierarchical, ensuring both flexibility and adaptability. These programs are then tested to measure their execution time, which is used to initialize the cost model within the optimization engine. The cost model guides the direction of the evolutionary search, generating a new batch of optimized programs. These optimized programs are subsequently tested for execution time, and the cost model is updated accordingly. This process iterates until the specified number of iterations is reached.

3.3 SGMV Operator

A simple approach to implementing LoRA operations involves iterating over each batch element and applying matrix multiplications sequentially. While straightforward, this method fails to take advantage of the parallel processing capabilities of modern GPUs, leading to significant performance bottlenecks. Another approach involves gathering LoRA matrices into a temporary tensor for pre-processing, enabling more efficient batched matrix multiplication. Although faster, this method still incurs additional memory operations, which can become a limiting factor in large-scale applications.

To address these inefficiencies, the SGMV operator di-

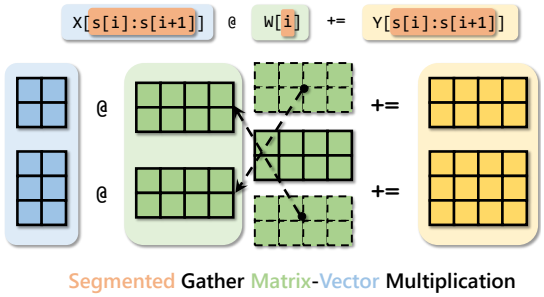

Figure 3: The semantics of the SGMV operation. Inputs are divided into segments $\mathbf{X}[s[i] : s[i+1]]$, where $s[i]$ and $s[i+1]$ denote the boundaries of each segment corresponding to the i-th LoRA model in the batch. These segments are then multiplied by the corresponding weight matrices $\mathbf{W}[i]$ and accumulated into the output segments $\mathbf{Y}[s[i] : s[i+1]]$.

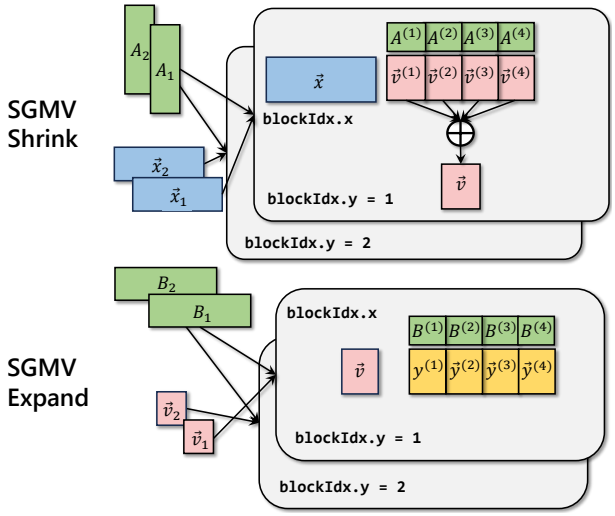

Figure 4: CUDA kernel scheduling for SGMV operations. SGMV-Shrink compresses high-dimensional input features into low-rank outputs using parallel matrix multiplications across thread blocks. SGMV-Expand expands low-rank inputs into high-dimensional outputs by distributing the computation across multiple thread blocks, utilizing the Split-K strategy for efficient parallelism.

rectly integrates the gathering and matrix multiplication steps into the CUDA kernel. This integration minimizes memory overhead and improves throughput by allowing the operations to be managed within a single kernel execution. The LoRA operation can be represented as $\mathbf{y} \mathrel{+}= \mathbf{xAB}$, and can be decomposed into two distinct steps using SGMV. First, we initialize $\mathbf{v} := \mathbf{0}$, then perform $\mathbf{v} \mathrel{+}= \mathbf{xA}$, followed by $\mathbf{y} \mathrel{+}= \mathbf{vB}$. This decomposition allows the operator to efficiently handle both gathering and multiplication tasks, reducing memory access and boosting overall performance.

SGMV is further classified into two categories based on input and output feature dimensions: SGMV-shrink and SGMV-expand. The SGMV-shrink operation ($\mathbf{v} = \mathbf{xA}$) reduces a high-dimensional input feature to a low-rank output, making it memory-bound and necessitating optimizations in memory access patterns and data locality. Conversely, the SGMV-expand operation ($\mathbf{y} = \mathbf{vB}$) expands the low-rank input feature to a high-dimensional output, making it compute-bound and benefiting from maximized throughput and parallelism. Figure 3 (Chen et al. 2024) illustrates the semantics of the SGMV operator, while Figure 4 shows the CUDA kernel scheduling for these two cases.

3.4 Search Space Constructor

The Search Space Constructor effectively partitions the search space into hierarchical levels. This hierarchical search space comprises high-level sketches and low-level annotations. Unlike sequential construction that generates complete programs directly, this decoupling enables us to efficiently create a comprehensive program space, ensuring better scalability and avoiding the limitations of inflexible optimization. It also allows for more efficient continuous updates to accommodate new hardware platforms and task scenarios.

High-level sketches capture the overall program structure and are created by directly applying derivation rules to the semantics of the SGMV operator. The process of generating high-level sketches begins by interpreting the semantics of the SGMV operator, identifying key computational patterns such as matrix multiplication, tiling for data reuse, loop unrolling, and memory access optimizations. For each identified pattern, derivation rules are applied to generate all possible high-level structures. These derivation rules are fundamental in creating a flexible and comprehensive search space, enabling the exploration of various optimization strategies. For the SGMV operator, the key derivation rules include:

- *Multi-level Tiling:* Apply multi-level tiling to nodes with data reuse potential, such as partitioning matrices into submatrices to enhance data reuse and reduce memory access latency.
- *Node Fusion:* Fuse adjacent computation nodes to minimize memory transfers, for instance, combining matrix multiplication with subsequent addition operations into a single operator.
- *Add Cache Stage:* Introduce caching nodes to store intermediate results in faster, on-chip memory, further improving performance.

The high-level sketches generated by these derivation rules form the initial framework for our optimization processas, illustrated in Figure 5. However, these sketches are not yet complete programs; they lack specific low-level details such as tile sizes, parallelization strategies, and memory access patterns. To transform these sketches into complete programs suitable for fine-tuning and evaluation, we annotate them with low-level details.

The annotation process involves randomly selecting specific values for tile sizes, parallelization of outer loops, vectorization of inner loops, and unrolling of inner loops. By randomly sampling these low-level details from a uniform

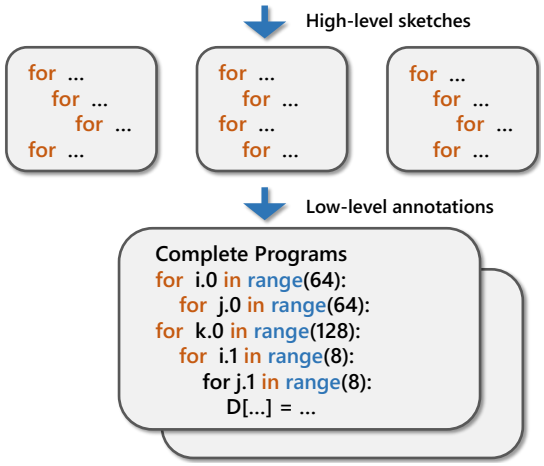

Figure 5: Illustration of the hierarchical search space construction, showing the transition from high-level sketches to low-level annotations.

distribution over all valid values, we ensure a diverse exploration of potential program configurations. Additionally, we may tweak the computation location of some nodes to further optimize the tile structure.

This combination of high-level sketches and low-level annotations generates a comprehensive search space that serves as the foundation for further optimization. The Search Space Constructor provides the potential for extensive and flexible exploration, but it is the Optimization Engine that refines and optimizes these programs to achieve the best possible performance.

3.5 Optimization Engine

The Optimization Engine is pivotal in fine-tuning the performance of the programs generated by the Search Space Constructor. Given the complexity and variability inherent in multi-LoRA scenarios, this engine employs evolutionary search and a cost model to iteratively optimize SGMV implementations.

The optimization process begins by selecting an initial program generated by the Search Space Constructor. This initial program serves as the starting point for the evolutionary search. The evolutionary search algorithm refines these programs by applying mutation and crossover operations, guided by the predictions of the cost model.

Evolutionary Search Evolutionary search is a metaheuristic algorithm inspired by the principles of natural selection and genetics. It begins with an initial population of programs, including both newly sampled programs and high-performing candidates from previous iterations. The search process iteratively generates new program candidates through mutation and crossover operations.

Mutation operations explore the search space by making small changes to specific aspects of a program, enabling the discovery of more optimized configurations. These operations include:

- *Tile Size Mutation:* Randomly adjusts the tile sizes in loops while maintaining the overall loop length, helping to identify an optimal tile size that balances cache utilization and memory access latency.
- *Parallel Mutation:* Modifies the parallel granularity of loops by either fusing or splitting them, potentially enhancing performance on different hardware architectures.
- *Pragma Mutation:* Randomly adjusts compiler-specific pragmas, such as those controlling loop unrolling or vectorization, to effectively leverage specific compiler optimizations.
- *Computation Location Mutation:* Changes the computation location of flexible nodes to improve data locality, reducing data movement and thus enhancing performance.

Crossover operations create new programs by combining elements from two or more existing programs. These elements, often referred to as "genes", are the specific transformations or optimizations that have been applied to the programs. The crossover process operates at the node level within the computational pattern, carefully preserving the dependencies between nodes. This ensures that when parts of different programs are combined, the resulting program remains functionally correct and structurally sound.

Throughout the evolutionary search, the cost model plays a crucial role by estimating the *fitness* of each program—its expected throughput. Programs with higher predicted fitness are more likely to be selected for mutation and crossover, driving the search towards more efficient implementations. The ability of the cost model to provide rapid and reasonably accurate predictions allows the Optimization Engine to explore tens of thousands of potential program configurations in a matter of seconds, vastly improving the efficiency of the search process.

Cost Model A cost model is essential for quickly estimating the performance of programs during the search. In our model, the variable y represents program throughput, which is a critical measure of performance. The cost model is trained on real-world performance data to predict the execution time of programs based on their structural and annotation features, such as arithmetic intensity and memory access patterns.

The loss function for the cost model is designed to emphasize the importance of accurately predicting the performance of faster programs. Specifically, it uses a weighted squared error where the weights are proportional to the program's throughput:

$$\text{loss}(f, P, y) = y \left(\sum_{s \in S(P)} f(s) - y \right)^2$$

where $S(P)$ represents the set of innermost non-loop statements in program P, and $f(s)$ is the model's prediction for each statement.

This weighting ensures that high-throughput programs have a greater influence on the training process, enabling the model to prioritize the optimization of efficient programs.

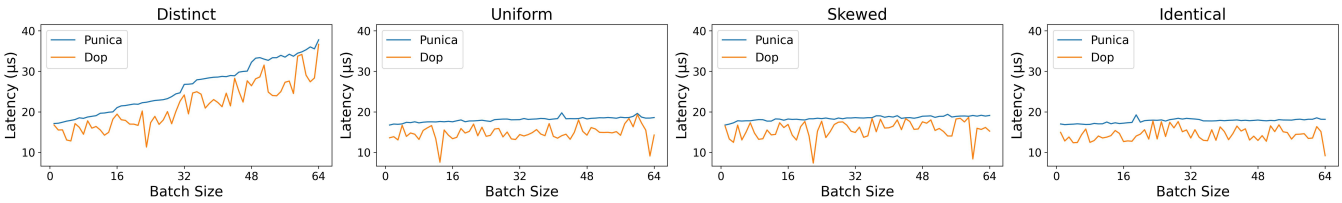

Figure 6: Latency comparison between Dop and Punica across four different request patterns on NVIDIA RTX 3090.

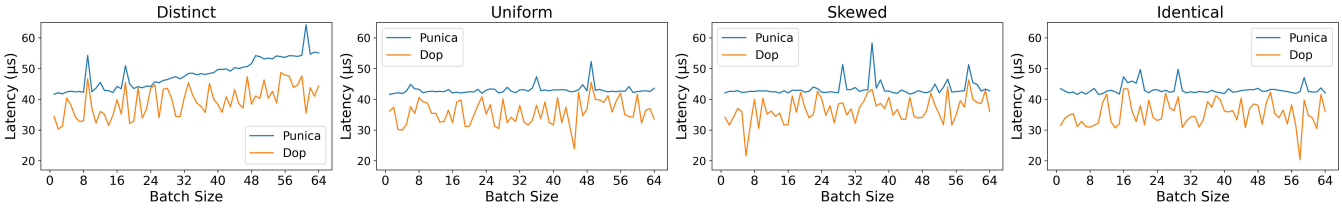

Figure 7: Latency comparison between Dop and Punica across four different request patterns on NVIDIA A100.

As a result, the cost model provides more accurate predictions for high-performance candidates, which are crucial for guiding the evolutionary search towards optimal implementations.

4 Experiments

4.1 Experimental Setup

LLMs and Benchmarks. In this study, we evaluate our method using the Llama-2 models (Touvron et al. 2023) with 7B and 13B parameters. The benchmarks (Chen et al. 2024) are divided into two main categories: microbenchmarking the LoRA operator latency and measuring the throughput during text generation. Due to the memory limitations of the RTX 3090 GPU, the Llama-2 model is only tested on the A100 40GB GPU.

Software and Hardware Configuration. All experiments are conducted on Ubuntu with PyTorch 2.1.2 and CUDA 12.4. The Llama-2 models are implemented using the HuggingFace Transformers (Wolf et al. 2020) library, with LoRA weights integrated via the HuggingFace PEFT library (Mangrulkar et al. 2022). The hardware used includes NVIDIA A100 40GB and NVIDIA RTX 3090 GPUs.

Baselines. To assess the performance of our approach, Dop is compared against several state-of-the-art LLM backbone serving systems, including HuggingFace Transformers (Wolf et al. 2020), FasterTransformer (Hsueh 2021), vLLM (Kwon et al. 2023), and DeepSpeed (Aminabadi et al. 2022). Since these systems do not inherently support multiple LoRA models, we use the HuggingFace PEFT library (Mangrulkar et al. 2022) to add LoRA weights to the baseline models. Additionally, the Punica system (Chen et al. 2024) is included as a key baseline for comparison. All settings, except for the SGMV operator implementation, are kept consistent across all systems. To ensure a fair comparison, model switching costs are excluded, and only backbone performance is considered.

Workload Types. To comprehensively evaluate our method, we considered four types of request distributions, each reflecting different operational scenarios. The *Distinct* distribution represents a highly diverse workload, where each request is linked to a different LoRA model. The *Uniform* distribution assumes that all LoRA models have an equal probability of being selected, simulating a balanced workload with $\lceil \sqrt{n} \rceil$ models used for n requests. The *Skewed* distribution follows a Zipf distribution with a parameter $\alpha = 1.5$, capturing scenarios where a few models dominate the workload. Lastly, the *Identical* distribution directs all requests to the same LoRA model, allowing for full optimization in a single-model scenario.

4.2 Main Results

We compare the performance of the Dop-optimized version of Punica (hereafter referred to as Dop) against the baseline systems. The evaluation covers various workload distributions, including the four request patterns described above. For each scenario, Dop was executed with 300 mutation iterations, with the entire Dop execution taking approximately 1.5 hours. Our results consistently show that Dop outperforms the existing solutions in both the LoRA operator microbenchmark and text generation throughput.

LoRA Operator Microbenchmark. We conducted experiments using the setup described in Section 3.1 (**The Motivating Example**) to evaluate the performance of different SGMV implementations across four request distributions. The experiments were performed on both NVIDIA RTX 3090 and NVIDIA A100 GPUs.

The results, illustrated in Figures 6 and 7, show that the Dop-optimized SGMV consistently outperforms the original Punica implementation across all request distributions. On the NVIDIA RTX 3090, Dop achieved a maximum improvement of 56.21% at batch size 31 under the Distinct request pattern. Similarly, on the NVIDIA A100, the highest improvement was 54.03% at batch size 45 under the Skewed

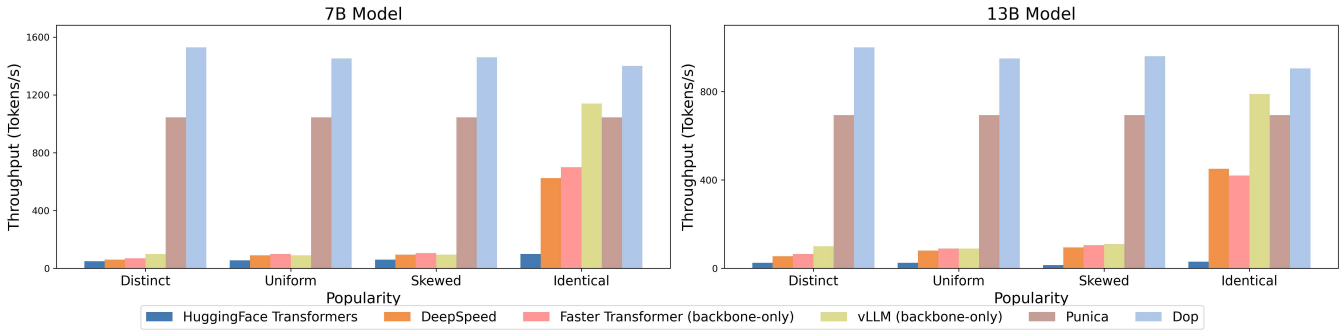

Figure 8: Throughput comparison across various systems using the 7B and 13B Llama-2 models on NVIDIA A100 GPU.

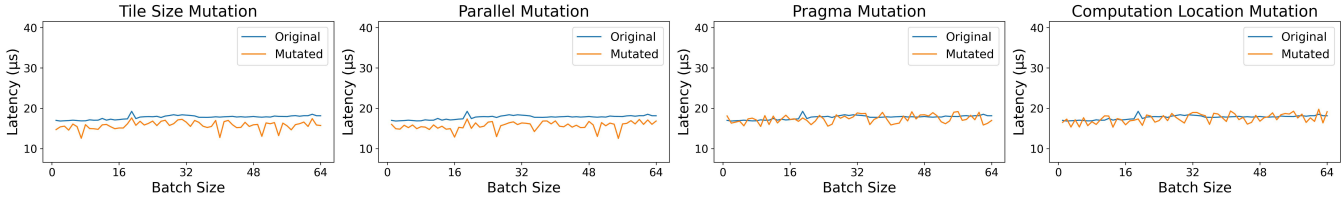

Figure 9: Impact of individual mutations on throughput for the 7B and 13B models on NVIDIA RTX 3090.

pattern. These results highlight the significant performance gains of the Dop-optimized SGMV, particularly in scenarios where LoRA model request pattern varies.

Text Generation Throughput. We evaluated text generation throughput on a single NVIDIA A100 GPU using the 7B and 13B Llama-2 models. Each system was tested with 1,000 requests, producing approximately 101k tokens. The maximum batch size was set to 32, and all systems processed requests in a first-come, first-served manner.

The results, as shown in Figure 8, demonstrate that Dop consistently achieves higher throughput compared to the baseline systems across all workload patterns. For example, under the *Skewed* workload, Dop achieved up to 1460 tokens/second for the 7B model, surpassing Punica's 1044 tokens/second, with even more pronounced improvements over systems that do not support batching. These improvements are primarily due to the optimizations applied to the SGMV operator, which allow Dop to more effectively manage different request patterns than Punica.

However, in the *Identical* request pattern, where all requests are directed to the same LoRA model, Dop's improvements over other systems (aside from Punica) are less pronounced. In this scenario, the operation effectively reduces to a Batched Matrix Multiplication (BMM), a highly optimized operation on GPUs like the A100. Thus, the gains achieved by Dop in this case are mainly due to adjustments in parameters such as tiling sizes, rather than leveraging SGMV's full advanced batching capabilities. Despite this, Dop still outperforms other systems, demonstrating its robustness even in scenarios where the opportunities for optimization are more limited.

4.3 Ablation Study

We conducted an ablation study to evaluate the impact of individual mutations on the performance of the SGMV operator. By selectively disabling other optimizations and isolating the effect of each mutation, we assessed their contributions to overall performance. The results, shown in Figure 9, indicate that not all mutations have a uniformly positive effect on performance. For instance, *Computation Location Mutation* offered more modest improvements, and in some cases, even led to slight performance degradation. These findings highlight the complexity of optimizing operators for diverse hardware and workloads. Specifically, *Tile Size Mutation* and *Parallel Mutation* significantly enhance SGMV performance by improving data locality, cache utilization, and GPU utilization. In contrast, mutations like *Computation Location* have less impact, as modern compilers and GPUs already optimize these aspects effectively.

5 Conclusion

In this paper, we introduced Dynamic Operator Optimization (Dop) to further enhance the performance of current systems in efficiently serving multiple LoRA models. By automating the search and generation of operator implementations, we identified the most suitable operator implementation for the current environment and tasks, eliminating the time-consuming and labor-intensive manual compilation and testing, and without requiring deep domain knowledge. Our experiments demonstrate that after Dop optimization, the operator's latency is significantly reduced in microbenchmark tests, and correspondingly, the system achieves a notable improvement in text generation throughput. This approach highlights the importance of dynamic optimization in modern multi-tenant serving environments.

Acknowledgements

This project is based upon work supported by National Natural Science Foundation of China Grant No. U22A6001 and "Pioneer" and "Leading Goose" R&D Program of Zhejiang No.2024SSYS0002. The computations in this research were performed using the CFFF platform of Fudan University.

References

Adams, A.; Ma, K.; Anderson, L.; Baghdadi, R.; Li, T.-M.; Gharbi, M.; Steiner, B.; Johnson, S.; Fatahalian, K.; Durand, F.; et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4): 1–12.

Aminabadi, R. Y.; Rajbhandari, S.; Zhang, M.; Awan, A. A.; Li, C.; Li, D.; Zheng, E.; Rasley, J.; Smith, S.; Ruwase, O.; and He, Y. 2022. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale.

Ansel, J.; Kamil, S.; Veeramachaneni, K.; Ragan-Kelley, J.; Bosboom, J.; O'Reilly, U.-M.; and Amarasinghe, S. 2014. OpenTuner: an extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 303–316.

Chen, L.; Ye, Z.; Wu, Y.; Zhuo, D.; Ceze, L.; and Krishnamurthy, A. 2024. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6: 1–13.

Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. 2018a. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 578–594.

Chen, T.; Zheng, L.; Yan, E.; Jiang, Z.; Moreau, T.; Ceze, L.; Guestrin, C.; and Krishnamurthy, A. 2018b. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, 3389–3400.

Dao, T.; Fu, D. Y.; Ermon, S.; Rudra, A.; and Ré, C. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness.

Frigo, M.; and Johnson, S. G. 1998. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 3, 1381–1384. IEEE.

Haj-Ali, A.; Ahmed, N. K.; Willke, T.; Shao, Y. S.; Asanovic, K.; and Stoica, I. 2020a. NeuroVectorizer: end-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 242–255.

Haj-Ali, A.; Genc, H.; Huang, Q.; Moses, W.; Wawrzynek, J.; Asanović, K.; and Stoica, I. 2020b. ProTuner: tuning programs with monte carlo tree search. *arXiv preprint arXiv:2005.13685*.

Haj-Ali, A.; Huang, Q.; Moses, W.; Xiang, J.; Wawrzynek, J.; Asanovic, K.; and Stoica, I. 2020c. AutoPhase: juggling HLS phase orderings in random forests with deep reinforcement learning. In *Third Conference on Machine Learning and Systems (ML-Sys)*.

Hsueh, B. Y. 2021. FasterTransformer.

Hu, E. J.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; Chen, W.; et al. 2021. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*.

Huang, Q.; Haj-Ali, A.; Moses, W.; Xiang, J.; Stoica, I.; Asanovic, K.; and Wawrzynek, J. 2019. Autophase: compiler phase-ordering for hls with deep reinforcement learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 308–308. IEEE.

Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C. H.; Gonzalez, J.; Zhang, H.; and Stoica, I. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23*, 611–626. Association for Computing Machinery. ISBN 9798400702297.

Li, T.-M.; Gharbi, M.; Adams, A.; Durand, F.; and Ragan-Kelley, J. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics (TOG)*, 37(4): 139.

Mangrulkar, S.; Gugger, S.; Debut, L.; Belkada, Y.; Paul, S.; and Bossan, B. 2022. Peft: State-of-the-art parameter-efficient fine-tuning methods.

Mullapudi, R. T.; Adams, A.; Sharlet, D.; Ragan-Kelley, J.; and Fatahalian, K. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4): 83.

Narayanan, D.; Shoeybi, M.; Casper, J.; LeGresley, P.; Patwary, M.; Korthikanti, V.; Vainbrand, D.; Kashinkunti, P.; Bernauer, J.; Catanzaro, B.; Phanishayee, A.; and Zaharia, M. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21*. Association for Computing Machinery. ISBN 9781450384421.

Ragan-Kelley, J.; Barnes, C.; Adams, A.; Paris, S.; Durand, F.; and Amarasinghe, S. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6): 519–530.

Schkufza, E.; Sharma, R.; and Aiken, A. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1): 305–316.

Sheng, Y.; Cao, S.; Li, D.; Hooper, C.; Lee, N.; Yang, S.; Chou, C.; Zhu, B.; Zheng, L.; Keutzer, K.; Gonzalez, J. E.; and Stoica, I. 2024. S-LoRA: Serving Thousands of Concurrent LoRA Adapters. arXiv:2311.03285.

Sheng, Y.; Zheng, L.; Yuan, B.; Li, Z.; Ryabinin, M.; Chen, B.; Liang, P.; Ré, C.; Stoica, I.; and Zhang, C. 2023. Flexgen: High-throughput generative inference of large language models with a single GPU. In *International Conference on Machine Learning, ICML 2023*, 31094–31116.

Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; Bikel, D.; Blecher, L.; Ferrer, C. C.; Chen, M.; Cucurull, G.; Esiobu, D.; Fernandes, J.; Fu, J.; Fu, W.; Fuller, B.;

Gao, C.; Goswami, V.; Goyal, N.; Hartshorn, A.; Hosseini, S.; Hou, R.; Inan, H.; Kardas, M.; Kerkez, V.; Khabsa, M.; Kloumann, I.; Korenev, A.; Koura, P. S.; Lachaux, M.-A.; Lavril, T.; Lee, J.; Liskovich, D.; Lu, Y.; Mao, Y.; Martinet, X.; Mihaylov, T.; Mishra, P.; Molybog, I.; Nie, Y.; Poulton, A.; Reizenstein, J.; Rungta, R.; Saladi, K.; Schelten, A.; Silva, R.; Smith, E. M.; Subramanian, R.; Tan, X. E.; Tang, B.; Taylor, R.; Williams, A.; Kuan, J. X.; Xu, P.; Yan, Z.; Zarov, I.; Zhang, Y.; Fan, A.; Kambadur, M.; Narang, S.; Rodriguez, A.; Stojnic, R.; Edunov, S.; and Scialom, T. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288.

Whaley, R. C.; and Dongarra, J. J. 1998. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, 38–38. IEEE.

Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; Davison, J.; Shleifer, S.; von Platen, P.; Ma, C.; Jernite, Y.; Plu, J.; Xu, C.; Scao, T. L.; Gugger, S.; Drame, M.; Lhoest, Q.; and Rush, A. M. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 38–45.

Yu, G.-I.; Jeong, J. S.; Kim, G.-W.; Kim, S.; and Chun, B.-G. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 521–538. USENIX Association. ISBN 978-1-939133-28-1.

Zheng, L.; Jia, C.; Sun, M.; Wu, Z.; Yu, C. H.; Haj-Ali, A.; Wang, Y.; Yang, J.; Zhuo, D.; Sen, K.; et al. 2020a. Ansor: generating high-performance tensor programs for deep learning. https://arxiv.org/abs/2006.06762.

Zheng, S.; Liang, Y.; Wang, S.; Chen, R.; and Sheng, K. 2020b. FlexTensor: an automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 859–873.