StitchLLM: Serving LLMs, One Block at a Time

Bodun Hu[1] *, Shuozhe Li[1] *, Saurabh Agarwal[1], Myungjin Lee[2], Akshay Jajoo[2], Jiamin Li[3], Le Xu[1], Geon-Woo Kim [1], Donghyun Kim[1], Hong Xu[4], Amy Zhang[1], Aditya Akella[1],

[1]The University of Texas at Austin, [2]Cisco Research, [3]Microsoft Research, [4]The Chinese University of Hong Kong

Abstract

The rapid evolution of large language models (LLMs) has revolutionized natural language processing (NLP) tasks such as text generation, translation, and comprehension. However, the increasing computational demands and inference costs of these models present significant challenges. This study investigates the dynamic and efficient utilization of pretrained weights from open-sourced LLMs of varying parameter sizes to achieve an optimal balance between computational efficiency and task performance. Drawing inspiration from the dual-process theory of human cognition, we introduce StitchLLM: a dynamic model routing framework that employs a powerful bottom model to process all queries, and uses a lightweight routing mechanism to allocate computational resources appropriately. Our novel framework optimizes efficiency and maintains performance, leveraging a trainable stitching layer for seamless integration of decoder layers across different LLMs. Experimental results demonstrate that StitchLLM improves system throughput while minimizing performance degradation, offering a flexible solution for deploying LLMs in resource-constrained settings.

1 Introduction

The rapid evolution of large language models (LLMs), such as GPT-4 (Achiam et al., 2023), has transformed natural language processing (NLP), enabling significant progress in text generation, translation, and comprehension. However, training LLMs remains computationally intensive, restricting foundation model development to organizations with massive compute resources. This bottleneck results in limited model size options. For example, Llama3 (Grattafiori et al., 2024) offers only five variants: 1B, 3B, 8B, 70B, and 405B.

This limited range of model sizes constrains the ability to balance accuracy and resource efficiency during inference. For example, a user requiring high accuracy must choose between Llama3 405B and Llama3 70B—two models with vastly different computational demands—without an intermediate option that allows balancing performance and efficiency. Such coarse granularity forces users into suboptimal trade-offs between accuracy and efficiency, as intermediate configurations are unavailable.

Existing techniques like distillation (Hinton, 2015; Gu et al., 2024; Liang et al., 2020) and pruning (Ma et al., 2023; Sun et al., 2023; Kurtic et al., 2022) try to create smaller models to address this issue. However, they come with substantial computational overhead, requiring extensive parameter updates and long training times. For instance, training a smaller model with Pythia (Biderman et al., 2023) can take over 24 GPU days, and generalization challenges persist (Gudibande et al., 2023).

To overcome these challenges, we propose a novel alternative: *dynamically composing pretrained LLM blocks of varying sizes*. This approach achieves fine-grained efficiency-accuracy trade-offs without retraining or finetuning, which is crucial for the scalable and sustainable deployment of LLMs in real-world applications, where both high throughput and accuracy must be maintained under resource constraints.

Our method draws inspiration from the dual-process theory of human cognition, which distinguishes between System 1 (fast, intuitive, automatic) and System 2 (slow, deliberate, computationally intensive) (Kahneman, 2011). Similarly, effective LLM deployment requires balancing lightweight, efficient inference (System 1) with computationally intensive, high-accuracy reasoning (System 2). To navigate this trade-off, we introduce StitchLLM, a serving system that seamlessly integrates pretrained models, dynamically allocat-

[*]Equal contribution.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 26887–26903

ing computational resources to mirror the adaptive interplay of Systems 1 and 2 in human cognition.

The core idea in StitchLLM is shown in Figure 1: incoming user requests are dynamically routed across "blocks" drawn from different models. The figure shows four block combinations that a request can traverse, spanning two bottom blocks and two top blocks, which open up various resource/accuracy trade-off points.

In developing StitchLLM, we overcome two key challenges: (i) Different models process query using unique intermediate representations, creating integration barriers. Additionally, identifying the ideal merge points—locations where models can be combined without compromising accuracy or efficiency—is complicated. (ii) Fragmenting models into smaller, reusable blocks, as shown in Figure 1, introduces communication overheads, and complicates various aspects such as managing GPU utilization and KV caches since different requests may use different model blocks.

To address the first challenge, we build on prior work in vision model stitching (Pan et al., 2023) by introducing a linear transformation that aligns the hidden dimensions of different LLMs (e.g., $4096 \rightarrow 2048$ for Llama-8B/Llama-1B), represented by the purple block in Figure 1). Training this lightweight layer requires updating only its parameters, minimizing overhead. We conduct extensive experiments—across 5 datasets and 12 models—to evaluate stitching at various locations and developed heuristics for optimal placement. Our findings indicate that stitching from a larger model to a smaller one (e.g., using earlier layers from Llama 8B and later layers from 1B) yields a better balance between performance and resource efficiency. Moreover, models within the *same family* exhibit similar stitching patterns, helping to reduce the search space for optimal stitching locations.

To overcome the second challenge, we develop end-to-end serving optimizations to enable effective model stitching. We employ greedy block-level scheduling and locality-aware placement to maximize GPU utilization while minimizing inter-server communication and KV cache management overheads. Unlike approaches taken by Claude (Priyanshu et al., 2024) and ChatGPT (Achiam et al., 2023), which switch to a lower-capacity model during high inference demand, StitchLLM can mitigate the stark trade-offs, enhancing overall user experience, and offering a fine-grained accuracy vs resource trade-off. As shown in Figure 3, StitchLLM bridges the accuracy-resource gap left by coarse-grained model sizes.

Using real cloud workloads, StitchLLM improves average response accuracy by 8% compared to state-of-the-art systems, while maintaining similar overall performance. It also reduces time-to-first-token by 18%. Our evaluation shows that StitchLLM enhances computational efficiency, lowering 95%ile latency by up to 33.5% and increasing GPU utilization by up to 20.1%. Further, StitchLLM excels under peak load scenarios, improving serving accuracy by 12.2%.

2 Related Work

We first provide a brief overview LLM inference and challenges.

LLM Workload Pattern. We next study the workload pattern observed when deploying LLMs, by looking at real-world traces. We first analyze the trace of request arrival patterns for Azure cloud services (Patel et al., 2024) released by Microsoft. As shown in Figure 2, we observe that the arrival patterns for user-facing applications are quite bursty. This pattern persists across various private deployments (Wang et al., 2024b,a; Patke et al., 2024; Khare et al., 2023; Agrawal et al., 2024), where unpredictable demand spikes force engineers to either over-provision resources or dynamically trade accuracy for efficiency via smaller models.

Accuracy and Resource Trade-off. More computationally intensive models generally provide better accuracy at the cost of high resource requirements. For example, ResNet-101 achieves higher accuracy over Resnet-50 (He et al., 2016a) on ImageNet (Deng et al., 2009) while requiring $2\times$ more FLOPs (4.1B vs 7.8B). This gap is even more pronounced in language models, *e.g.*, Llama3-70B shown 16% gain over Llama3-8B (Grattafiori et al., 2024) (the next smaller model) on MATH (Hendrycks et al., 2021b), with $8\times$ higher memory demands (260 GB vs 29 GB), as shown in Table 1. Current practice limits users to a few discrete model sizes-1B, 3B, 8B, 70B, and 405B-with no fine-grained control over accuracy vs resource trade-offs between these tiers. Due to the prohibitive costs of training foundation models, practitioners cannot simply train intermediate-sized variants. Next, we review existing approaches to create new models to navigate accuracy and resource trade-offs.

Creating New models Recent work has explored generating smaller LLMs from pre-trained mod-

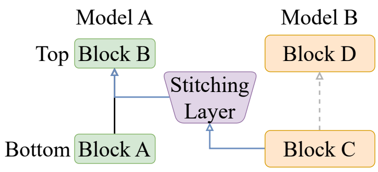


Figure 1: Requests can be routed from a Bottom block to a Top block, with different sizes, configurations, and originating from various LLMs. The Stitching Layer transformer intermediate output from bottom block to match that of the top block.

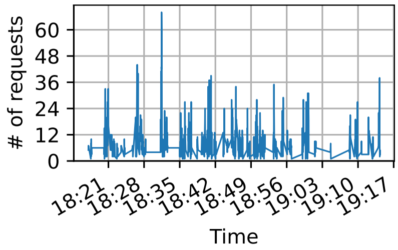


Figure 2: Temporal Heterogeneity: The above plot is the request arrival pattern observed from two Azure LLM inference services. There is a very high temporal heterogeneity as the number of requests can change from 1 to 68 in less than a minute.

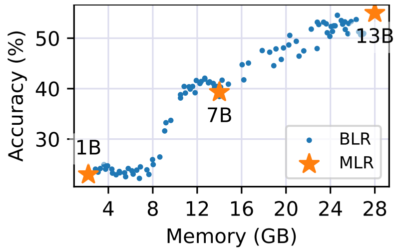


Figure 3: Tradeoff between Accuracy and Resources: accuracy and memory requirements for `StitchLLM`. Stars represent the trade-off space for existing models, while the blue dot represents the trade-off space for `StitchLLM`.

	1B	3B	8B	70B
Memory (GB)	3.8	11.2	29.8	260.4
Price (USD/1M tokens)	0.04	0.06	0.1	0.8
TTFT (ms)	45	53	69	1445
MMLU (%)	33	60	67	80
MATH (%)	10	38	44	60

Table 1: Llama 3 model metrics comparisons.

els. Distillation-based approaches (Hsieh et al., 2023; Yang et al., 2024; DeepSeek-AI et al., 2025; Sreenivas et al., 2024; Harper et al.) use a larger model as a teacher to train a smaller student model, but still demands significant resources and training time (Hsieh et al., 2023; Sreenivas et al., 2024; DeepSeek-AI et al., 2025). Pruning methods similarly reduce computation but often require extensive fine-tuning (Xia et al., 2024; Men et al., 2025; Gromov et al., 2024; Frantar and Alistarh, 2023). In contrast, with `StitchLLM` we aim to design a method which dramatically reduces the amount of computation required for generating new models.

Inference System Optimizations. LLM-based applications are being rapidly deployed on user-facing applications. However, LLMs' massive size (Touvron et al., 2023; Team et al., 2024) and high computational demands (Dao, 2023; Sheng et al., 2023) make inference challenging. In particular, the auto-regressive nature of LLMs makes inference stateful, requiring efficient caching of KV matrices to prevent redundant recalculations (Radford et al., 2019b; Kwon et al., 2023; Prabhu et al., 2024). Recent work has focused on optimizing KV cache prefill and generation (Agrawal et al., 2024; Zhong et al., 2024; Patel et al., 2024), improving compute density (Dao, 2023), and reducing memory fragmentation in KV caches (Kwon et al., 2023). These advances are crucial to alleviate memory and compute constraints, and thus any approach to improve accuracy/resource trade-offs should be compatible with them.

We introduce `StitchLLM`, our solution for creating models of various sizes with minimal compute and no fine-tuning. `StitchLLM` enables flexible accuracy–latency trade-offs while remaining fully compatible with existing optimizations.

3 `StitchLLM`

`StitchLLM` enables efficient creation of new models from existing models by *stitching* blocks of layers from different models without requiring parameter updates to the LLMs to maintain accuracy. We start by providing an overview of stitching.

3.1 Stitching in LLMs

Large Language Models (LLMs) consist of stacked decoder layers, where it is widely believed (Zhang et al., 2024b,c; Ju et al., 2024) that layers closer to the input capture broader input patterns, and those closer to the output encode entity-specific knowledge-a structural consistency observed across model sizes. This property enables the idea of *layer stitching*: combining lower layers from one LLM with upper layers of another to create a hybrid model.

We define a stitched model $\mathcal{M}_{(t,p)}$ as two components: the **bottom** blocks $\mathcal{M}_b$ and the **top** blocks $\mathcal{M}_t$. The bottom blocks are consecutive decoder layers selected from a model $\mathcal{B}$, while the top blocks are selected from another model $\mathcal{T}$. During inference (Figure 1), an input query q first traverses $\mathcal{M}_b$, producing an intermediate representation $\mathcal{A}_b$. This output is then processed by $\mathcal{M}_t$ to generate the final response: $\mathcal{A} = \mathcal{M}_t(\mathcal{M}_b(q))$, where $\mathcal{M}_b : \mathcal{Q} \rightarrow \mathcal{A}_b$ and $\mathcal{M}_t : \mathcal{A}_t \rightarrow \mathcal{A}$ map a query to intermediate representation, and from intermediate representation to response, respectively.

By selecting $\mathcal{M}_b$ and $\mathcal{M}_t$ from LLMs of differing sizes, `StitchLLM` achieves flexibility in balancing efficiency and performance.

Algorithm 1 Stitching Layer Training

Require: Given two LLMs $\mathcal{B}$ and $\mathcal{T}$, selects consecutive decoder layers from $\mathcal{B}$ and $\mathcal{T}$ as bottom block $\mathcal{M}_b$ and top block $\mathcal{M}_t$.

Initialize the stitching layer $\mathcal{S}$ based on the hidden size of $\mathcal{M}_b$ and $\mathcal{M}_t$.
Freeze the weights of $\mathcal{M}_b$ and $\mathcal{M}_t$.
for $i = 1, ..., n_{iter}$ **do**
    Get next batch of data q_i.
    $output = \mathcal{M}_t(\mathcal{S}(\mathcal{M}_b(q_i)))$.
    $loss = MSE(output,\ q_i)$.
    Update $\mathcal{S}$ using $loss$.
end for

3.2 Challenges

Stitching model blocks with heterogeneous representations leads to two challenges:

Intermediate Dimension Mismatch. Blocks from different models often have incompatible hidden dimensions. We address this by introducing a Stitching Layer (Section 3.3), which aligns intermediate representations across mismatched sizes.

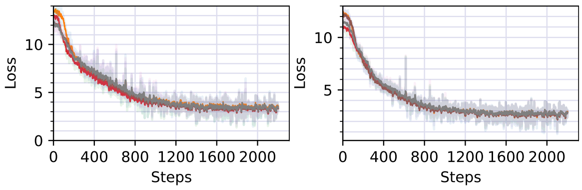
Optimal Layer Selection. Performance can depend on where and how many layers are stitched. To understand, analyze layer interactions across models in Section 3.4 to derive data-driven heuristics for identifying optimal stitching positions and layer counts.

3.3 Stitching Layer

We introduce the stitching layers as follows: Given a bottom block $\mathcal{M}_b$ producing intermediate representations $\mathcal{A}_b \in \mathbb{R}^{S \times H_b}$ (sequence length S, hidden size H_b) and a top block $\mathcal{M}_t$ requiring inputs $\mathcal{A}_t \in \mathbb{R}^{S \times H_t}$ with distinct hidden size H_t, direct compatibility is impossible due to dimensional mismatch. We propose a lightweight stitching layer $\mathcal{S} \in \mathbb{R}^{H_b \times H_t}$—implemented as a single MLP—to align hidden dimensions. The generation process becomes: $\mathcal{A} = \mathcal{M}_t(\mathcal{S}(\mathcal{M}_b(q)))$.

The stitching layer is trained using the same cross-entropy loss employed during the pre-training stage of the underlying models, leveraging the original pre-training dataset (e.g., C4 (Dodge et al., 2021) in our experiments). This ensures the routing layer generalizes effectively to diverse text representations and avoids becoming a bottleneck for information flow among bottom/top blocks.

The training process for the stitching layer is documented in Algorithm 1. We select bottom abd top blocks ($\mathcal{M}_b$, $\mathcal{M}_t$) from frozen base models $\mathcal{B}$ and $\mathcal{T}$, then insert *trainable* a stitching layer $\mathcal{S}$ between them. The training process is lightweight and takes < 2000 gradient steps (Figure 4). This process requires minimal resources (Table 2)—training even large models completes in < 6 GPU hours.



(a) Llama 3.1 8B and 3.2 3B. **(b)** Llama 2 7B and TinyLlama 1.1B.

Figure 4: Training losses for randomly sampled stitching layers on stitched Llama 3 and Llama 2 models. Convergence achieved after approximately 2000 gradient steps.

Stitching Block	GPU hours
(2048, 4096)	2.01
(4096, 5120)	4.33
(5120, 4096)	4.84
(4096, 8192)	5.32
(5120, 8192)	5.85

Table 2: The GPU hours consumed for training stitching layers of various sizes on A100 GPUs for Llama Models.

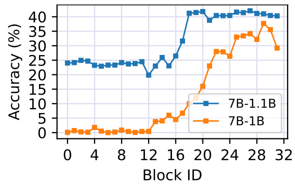


Figure 5: MMLU accuracy all across decoder layers: the blue line shows the accuracy of stitching Llama 2 7B with TinyLlama 1.1B; the orange line shows stitching Llama 2 7B with Llama 3.2 1B.

3.4 Choosing stitching location and models

The choice of location of stitching and the models and their number of layers to stitch can greatly impact performance. We explain our choices next.

Location of Stitching. Prior work shows adjacent layers share similar feature representations (Pan et al., 2023; Kornblith et al., 2019), motivating our *bilateral stitching* approach. Let $\mathcal{B}$ and $\mathcal{T}$ be two models with $\mathcal{L}_b$ and $\mathcal{L}_t$ decoder layers, respectively. For a bottom block comprising the first i layers of $\mathcal{B}$, we stitch it to the top blocks of $\mathcal{T}$ starting at layer: $j = i \times (\mathcal{L}_b/\mathcal{L}_t)$. We insert a stitching layer after each decoder layer in $\mathcal{B}$, resulting in $\mathcal{L}_b$ stitching layers overall.

Stitching Heuristics. To balance model capacity, we assemble blocks from pre-trained networks of varying dimensions. Prior work prioritizes stacking *smaller bottom blocks* with larger top blocks (Pan et al., 2023; He et al., 2016b; Huang et al., 2016); however, `StitchLLM` exclusively pairs **larger bottom blocks** with smaller top blocks. Empirical analysis (Section 5.2) shows small-to-large configurations under-perform their base models, while large-to-small assemblies retain performance. Larger bottom blocks preserve foundational representations by better extracting and retaining input information, reducing the load on subsequent layers, making them essential for maintaining accuracy. While the choice of top blocks also matters—smaller top blocks can degrade performance—the effect is less pronounced than that of bottom blocks (Section 5.2). This method also reduces the number of stitching models and the search space for optimal accuracy-latency tradeoffs. For example, larger-smaller stitching for Llama 2 13B and 7B cuts stitching candidates from 72 to 40, a 45% reduction.

Further, we restrict stitching to blocks from models within the same family, as cross-family combinations (e.g., Llama 2 with Llama 3) degrade performance due to structural incompatibilities in decoder blocks. Empirical results (Figure 5) confirm significant quality loss when mixing families, reinforcing the need for intra-family stitching for performance preservation. This constraint also reduces the number of stitching layers, lowering both training overhead and search space complexity.

Efficiency-Driven Block Optimization. To optimize block selection under a memory constraint $\mathcal{C}$, we propose a *greedy approach* that maximizes inference accuracy f_{acc} by selecting the number of bottom blocks n_b with embedding size N_b and top blocks n_t with embedding size N_t, where each decoder block takes $mem(N)$ amount of memory:

$$\begin{aligned} \max_{n_b, N_b, n_t, N_t} \quad & f_{acc}(n_b, N_b, n_t, N_t) \\ \text{s.t.} \quad & n_b \cdot mem(N_b) + n_t \cdot mem(N_t) \leq \mathcal{C} \\ & n_b \geq 1 \\ & n_t \geq 1 \end{aligned} \tag{1}$$

This method enables StitchLLM to dynamically adjust block selection based on available resources, efficiently handle request fluctuations, and naturally incorporate more parameters when resources allow—reinforcing our observations and aligning with prior work that larger models yield better performance (Radford and Narasimhan, 2018; Radford et al., 2019a; Brown et al., 2020).

Additionally, StitchLLM employs *accuracy-guided pruning*. Empirical observations (Section 5.3) show that choosing fewer bottom blocks can degrade performance due to feature incompatibility. Therefore, StitchLLM prunes suboptimal stitched models and retains only those within the accuracy range $\mathcal{M}_s = \{m \mid \alpha_t \leq \ accuracy(m) \leq \alpha_b\}$ where α_t is the weaker model's accuracy and α_b is the stronger model's accuracy. By integrating our greedy stitching heuristic and accuracy-guided pruning, StitchLLM further reduces stitching candidates (e.g., from 72 to 20 for Llama 2 13B and 7B, a 73% reduction). This efficiency is crucial for deployment in resource-constrained environments with strict latency requirements, where traditional methods are impractical.

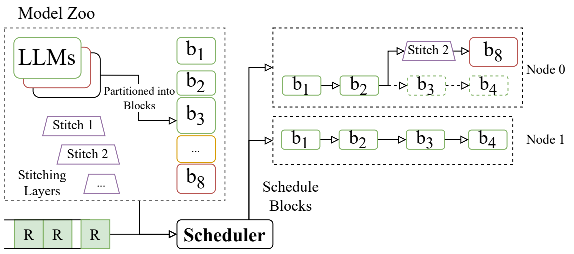


Figure 6: StitchLLM System Architecture. The framework enables adaptive model composition through two core mechanisms: (1) Stitching Layers that dynamically route computations between model blocks, and (2) Resource-Aware Scheduler that selects optimal blocks in real-time based on current system constraints (e.g., memory).

4 StitchLLM Serving

StitchLLM is an end-to-end serving system that helps realize the benefits of stitching models overcoming key challenges. In Section 4.1, we first analyze the limitations of existing approaches and demonstrate how we address these gaps. We then decribe the design.

4.1 Existing Serving System

To manage fluctuating workloads, LLM providers often use *Model-Level Routing* (MLR), where requests are routed to smaller models (e.g., Llama 70B to Llama 7B) during peak demand, prioritizing availability over accuracy. However, MLR has several inefficiencies: (1) the trade-off between accuracy and resource requirements is coarse. MLR forces providers to choose between discrete model sizes, resulting in abrupt accuracy drops (Figure 3). (2) During model transitions, GPU memory must store weights of both the original and smaller models, causing "memory bloat" forcing smaller batch sizes and reduced throughput. (3) Smaller batches during transitions degrade GPU utilization, worsening inefficiencies.

Our StitchLLM serving system addresses these inefficiencies by unlocking *Block-Level Routing*, decomposing models into reusable layer blocks, and routing among them. By storing only active blocks, StitchLLM eliminates memory bloat while ensuring high throughput, optimal cluster utilization, and adaptability to fluctuating workloads.

4.2 Overview

Figure 6 provides an overview of StitchLLM.

Model Zoo. StitchLLM's "block zoo" repository

Algorithm 2 Determining Stitching Configurations

Require: Stitching Candidates $\mathcal{M}_s$, $\mathcal{C}$
Require: $Configs \leftarrow []$
for all $(b_i,\ t_i) \in \mathcal{M}_s$ **do**
  append $(b_i,\ t_i)$ to $Configs$
end for
Sort $Configs$ s.t. $(m_{b_i}, m_{t_i}) \preccurlyeq (m_{b_j}, m_{t_j})$ if $m_{b_i} < m_{b_j}$ or $(m_{b_i} = m_{b_j} \wedge m_{t_i} \leq m_{t_j})$, **where** $m_{b_i} = n_{b_i} \cdot mem(b_i)$.
for all $(b_i,\ t_i) \in Configs$ **do**
  if $m_{b_i} + m_{t_i} \leq \mathcal{C}$ **then**
    return $(b_i,\ t_i)$
  end if
end for
return $null$

organizes LLMs by partitioning decoder layers into individual blocks. It not only serves as a storage interface but also integrates a profiler that records key performance metrics (e.g. memory usage, average latency per token, task accuracy, and architectural details, etc.) while evaluating the resource–accuracy trade-offs for each block.

Scheduler. During inference, the `StitchLLM` scheduler manages resource allocation and block placement, processing requests. It schedules blocks (denoted as "block instances") onto devices and decides how to route requests and what point in trade-off space should models "degrade to".

Agent. A `StitchLLM` agent on each device in a cluster of machines monitors block instances and request queues, handles requests, manages the KV cache, and transfers outputs among blocks. It provides compute and memory utilization statistics to the `StitchLLM` scheduler and enables request migration across nodes. Appendix H provides further details of `StitchLLM`'s serving implementation.

4.3 Online Serving

Figure 6 illustrates the steps in `StitchLLM`'s online serving process; we provide details below.

Request Scheduling. `StitchLLM`'s scheduling strategy prioritizes block instances that either *hold a request's KV cache* or are *already loaded* in GPU memory, provided the device memory can accommodate the request data. If the device memory is insufficient, `StitchLLM` estimates the latency of each potential block instance and greedily schedules the request to the instance with the smallest latency increase. Details can be found in Appendix K.

Block resource allocation. `StitchLLM`'s scheduler allocates resources for blocks, allowing independent per-block scaling using a queue-length-based policy. If the queue length exceeds $t\%$ of the maximum (configurable by the user), we scale onto more devices, starting with the heaviest-loaded instances. If an instance has requests' KV cache, we balance the load by moving the state along with rerouting requests to new instances (Appendix I).

When memory becomes constrained, `StitchLLM` dynamically prioritizes request throughput by trading accuracy for capacity. `StitchLLM` first identifies memory usage and searches for smaller stitching configurations to reduce load, enabling higher request volumes. Using the greedy strategy from Eq. 1, it sorts all stitching configurations by descending bottom-block size, then top-block size, and iteratively evaluates them until finding a configuration under the memory budget $\mathcal{C}$ (Algorithm 2). This ensures real-time adaptability while balancing efficiency and resource limits.

Locality-aware block placement. To mitigate transfer overhead between blocks, `StitchLLM` places blocks to prioritize locality. During placement, `StitchLLM` ensures blocks with frequent inter-dependencies are placed close together, ideally on the same server leveraging high-capacity intra-server connections like NVLink interconnects and avoiding constrained inter-server links.

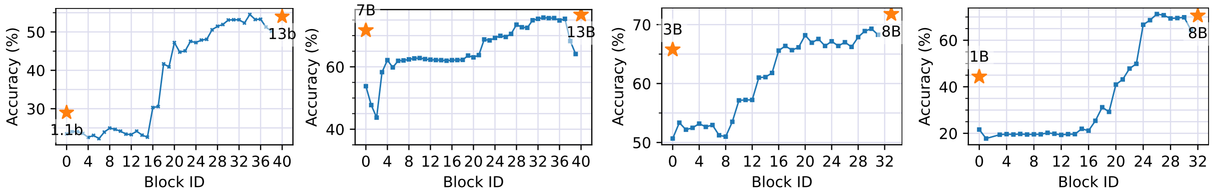
Locality is quantified by monitoring historical traffic and recording inter-dependency frequencies. High-locality block pairs are placed on the same server. Additionally, `StitchLLM`'s scheduler dynamically adapts to changing traffic patterns, migrating block instances as needed. Appendix F discusses the benefits of locality-aware placement.

5 Evaluation

Setup. All experiments were conducted on two servers equipped with an Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz, supplemented by four NVIDIA A100 GPUs, each with 80GB of RAM. The server runs on Ubuntu 22.04.4 LTS and uses PyTorch 2.4.0 (Paszke et al., 2019).

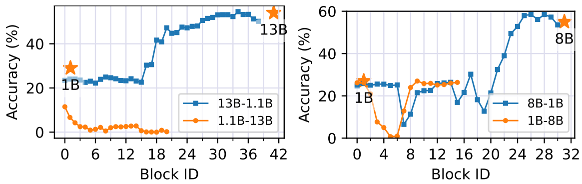
Models. We conduct all experiments using both Llama 2 and Llama 3 models. For Llama 2, we utilize TinyLlama 1.1B (Zhang et al., 2024a), as well as 7B and 13B variants. For Llama 3, we select the 1B, 3B, and 8B versions. In addition, we use Qwen 2.5 models (Bai et al., 2023) to analyze their block stitching behavior, using the 1.5B, 3B, 7B, 14B, and 32B variants.

Datasets. We choose five representative datasets to evaluate the effectiveness of our methods: MMLU (Hendrycks et al., 2021a), BoolQ (Clark et al., 2019), CommonsenseQA (Talmor et al.,



(a) MMLU performance on Llama 2 13B and TinyLlama 1.1B. **(b)** BoolQ performance on Llama 2 13B and Llama 2 7B. **(c)** Winogrande Performance on Llama 3.1 8B and Llama 3.2 3B. **(d)** CommonsenseQA performance on Llama 3.1 8B and Llama 3.2 1B.

Figure 7: Stitching Performance Across Various Decoder Layers.



(a) Llama 2 13B and TinyLlama 1.1B. **(b)** Llama 3.1 8B and Llama 3.2 3B.

Figure 8: MMLU Accuracy Across Different Decoder Layers: The orange line represents using smaller bottom blocks paired with larger top blocks, while the blue line depicts larger bottom blocks combined with smaller top blocks.

2019), Hellaswag (Zellers et al., 2019), and Winogrande (Sakaguchi et al., 2021).

Baselines: To understand fine-grained accuracy-resource trade-offs, we compare stitched models against their *base models*, where bottom and top blocks are selected. For end-to-end inference improvements, we compare `StitchLLM` with *model-level* routing (MLR), where requests are routed to available LLMs instead of different blocks. These baselines meticulously validate our observations.

5.1 Stitching LLMs

We first analyze the performance of `StitchLLM`, using Llama 2 and Llama 3 models. By applying Algorithm 2, we create 112 stitching layers for Llama 2 models (13B-1.1B, 13B-7B, and 7B-1B) and 92 stitching layers for Llama 3 models (8B-3B, 8B-1B, and 3B-1B). As illustrated in Figure 7, our stitched models provide fine-grained accuracy-latency tradeoffs on four different datasets: MMLU, BoolQ, Winogrande, and HellaSwag, filling the gap left by their base models. This indicates that `StitchLLM` successfully achieves stitching across a variety of tasks. Additional evaluations are included in Appendix A.

5.2 Stitching Direction

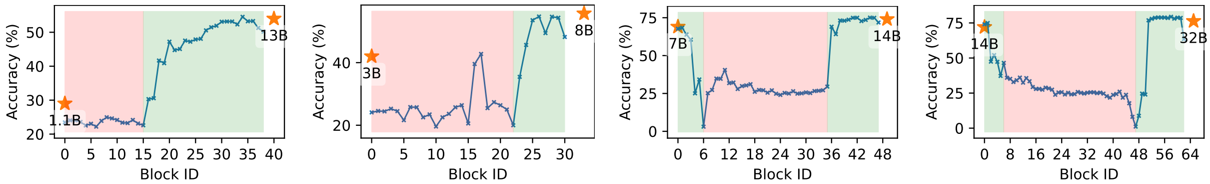
In Figure 8 we evaluate two neural network stitching strategies using Llama 2 and 3: (1) smaller bottom blocks paired with larger top blocks (2) larger bottom blocks combined with smaller top blocks. We observe that models with larger bottom blocks and smaller top blocks consistently outperform the reverse configuration across both architectures. This suggests that foundational lower layers play a disproportionately critical role in knowledge retention and reasoning. Our findings indicate that the size of bottom blocks is crucial for achieving maximum performance. Using larger models (Llama 2 13B and Llama 3 8B) as bottom blocks significantly outperforms using smaller ones, leading us to prefer larger models for bottom blocks. More evaluations can be found in Appendix A.

Furthermore, we identify a performance bounding effect, smaller bottom blocks act as an irreversible bottleneck, capping overall model performance at the level of their source architecture even when augmented with larger or more capable top blocks. We include more analysis in Appendix B.

5.3 Existence of Performance Boundary

For each model, we observe a distinct performance boundary—a specific decoder block, where, on one side, performance remains largely constant, and on the other side, it suddenly changes. Figure 9 demonstrates this phenomenon on the MMLU benchmark for both Llama 2 and Qwen 2.5 models. This boundary separates two regions: a *cold* region, where stitching positions yield low and stable performance, and a *hot* region, where performance improves dramatically. For example, the performance boundaries occur at block 15 for Llama 2 13B, and block 7 and 47 for Qwen 2.5 32B. We include more evaluations in Appendix C.

Each model family also exhibits a unique *performance boundary pattern*. First, the ratios of the hot and cold regions are remarkably consistent within each family: Llama 2 models show a hot-to-cold ratio of approximately 1.3:1, while Llama 3 models display a ratio of roughly 1:2.33. Second, models within the same family tend to share a similar boundary layout regardless of their overall size, as shown in Figure 9. Specifically, Llama 2 models typically have their performance boundary in the middle of the decoder layers, Llama 3 models near the end, and Qwen models feature two bound-



(a) Llama 2 13B and TinyLlama 1.1B. **(b)** Llama 3.1 8B and Llama 3.2 3B. **(c)** Qwen 2.5 14B and 7B. **(d)** Qwen 2.5 32B and 14B.

Figure 9: MMLU performance using differnt stitching block configurations on LLama 2, 3, and Qwen 2.5.

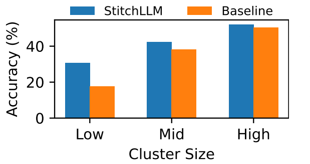


Figure 10: Accuracy advantage: When sizing the cluster to support peak load, average load and minimum load in the Azure trace, we observe StitchLLM outperforms MLR.

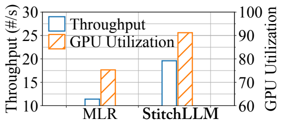


Figure 11: Throughput and GPU utilization comparison: We observe that `StitchLLM` when compared to MLR provides high throughput and GPU utilization.

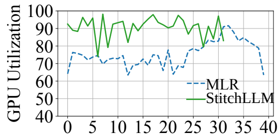


Figure 12: GPU Utilization: The above figure high GPU utilization change over time. We observe that compared MLR , `StitchLLM` consistently provides higher GPU utilization.

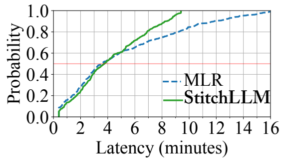


Figure 13: Latency CDF: The above figure show the CDF of request completion time of `StitchLLM`. `StitchLLM` improves both job completion and make span .

	Llama 2		Llama 3	
	MLR	`StitchLLM`	MLR	`StitchLLM`
TTFT (ms)	112	101.6	99	91
Accuracy (%)	30	38	39.7	42.7
Parameters used (B)	12.29	10.75	5.96	5.07

Table 3: Performance comparison between `StitchLLM` and MLR using Llama 2 (1.1B, 7B, 13B), and Llama 3 (1B, 3B, 8B).

aries—one near the beginning and another near the end. We include more evaluations in Appendix C.

5.4 Serving Performance

Accuracy. We first utilized production Azure traces to examine how accuracy is impacted by variations in the request arrival rate and perform evaluation on the standard MMLU benchmark. As shown in Figure 10, the average accuracy fluctuates over time under different cluster configurations. Here, "Low" represents the use of 2 GPUs, "Mid" represents 4 GPUs, and "High" represents 8 GPUs. The models evaluated in this study include Llama 2 at 1.1B, 7B, and 13B parameters. Incoming requests prioritize the highest-accuracy blocks (13B) unless those resources are unavailable.

Additionally, we analyze how `StitchLLM` improves accuracy and TTFT over MLR using the Azure production traces with 2 GPUs. Table 3 shows that the average accuracy achieved by `StitchLLM` is 38%, which is 8% higher than using MLR. Similarly, Table 3 shows that the average accuracy achieved by `StitchLLM` is 42.7%, which is 3% higher than using MLR. When request arrival rates are low, `StitchLLM` shares the same blocks to save memory as observed by the average number of parameters invoked per request. Which is lower when using `StitchLLM`. Higher request arrival rates cause `StitchLLM` to redirect traffic to faster blocks (1.1B) more frequently, but the accuracy degradation process is more gradual than MLR, resulting in a gradual decline in accuracy.

Latency and throughput. Figure 13 depicts the CDF of the latency of completing a request in `StitchLLM`. `StitchLLM` reduces the 95%ile latency by 33.5% compared with MLR. The throughput of `StitchLLM` is 1.71x of MLR. (Figure 11). By decomposing models into more granular blocks, `StitchLLM` enhances efficiency of processing larger batch sizes. This approach significantly reduces tail latency, especially under high request rates.

GPU utilization. In Figure 12 we monitor the end-to-end serving process, and observe that the average GPU utilization is improved by 20.1% compared with MLR. `StitchLLM` efficiently dispatches requests under the existing deployment status to avoid frequent model loading and unloading. We provide memory measurement and additional metrics in Appendix G.

6 Conclusion

We present `StitchLLM`, a finer-grained serving system tailored for LLM workloads. In `StitchLLM`, we show the effectiveness of improving throughput by allowing model component reuse with blocks. We enable adaptive serving, effectively coordinating multiple requests' KV cache, and mitigating the communication costs to improve serving efficiency. Our experiments show that `StitchLLM` achieve significant efficiency improvement.

Limitations

Our approach relies on empirically derived heuristics for greedy block selection and accuracy-guided pruning, which may not generalize to novel model families or emerging architectures. In addition, heterogeneous block execution poses challenges for GPU memory management, particularly when handling large batches. Furthermore, `StitchLLM` requires multiple model variants for effective stitching. Its ability to merge blocks from different models is contingent on specific compatibility factors—such as consistent tokenizers and vocabulary sizes—within the same model family. However, our observations indicate that these incompatibility issues are infrequent, suggesting that the use of routing layers remains broadly feasible.

References

Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. Infercept: Efficient intercept support for augmented large language model inference. *Preprint*, arXiv:2402.01869.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *Proceedings of 18th USENIX Symposium on Operating Systems Design and Implementation, 2024, Santa Clara*.

Sangmin Bae, Jongwoo Ko, Hwanjun Song, and Se-Young Yun. 2023. Fast and robust early-exiting framework for autoregressive language models with synchronized parallel decoding. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Yanxi Chen, Xuchen Pan, Yaliang Li, Bolin Ding, and Jingren Zhou. 2023. Ee-llm: Large-scale training and inference of early-exit large language models with 3d parallelism. *ArXiv*, abs/2312.04916.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2924–2936, Minneapolis, Minnesota. Association for Computational Linguistics.

Luciano Del Corro, Allie Del Giorno, Sahaj Agarwal, Bin Yu, Ahmed Awadallah, and Subhabrata Mukherjee. 2023. Skipdecode: Autoregressive skip decoding with batching and caching for efficient llm inference. *Preprint*, arXiv:2307.02628.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, and more authors. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255.

Jesse Dodge, Maarten Sap, Ana Marasović, William Agnew, Gabriel Ilharco, Dirk Groeneveld, Margaret Mitchell, and Matt Gardner. 2021. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. *Preprint*, arXiv:2104.08758.

Elias Frantar and Dan Alistarh. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot. *Preprint*, arXiv:2301.00774.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and more authors. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Andrey Gromov, Kushal Tirumala, Hassan Shapourian, Paolo Glorioso, and Daniel A. Roberts. 2024. The unreasonable ineffectiveness of the deeper layers. *Preprint*, arXiv:2403.17887.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. 2024. Minillm: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations*.

Arnav Gudibande, Eric Wallace, Charlie Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. 2023. The false promise of imitating proprietary llms. *arXiv preprint arXiv:2305.15717*.

Eric Harper, Somshubra Majumdar, Oleksii Kuchaiev, Li Jason, Yang Zhang, Evelina Bakhturina, Vahid Noroozi, Sandeep Subramanian, Koluguri Nithin, Huang Jocelyn, Fei Jia, Jagadeesh Balam, Xuesong Yang, Micha Livne, Yi Dong, Sean Naren, and Boris Ginsburg. NeMo: a toolkit for Conversational AI and Large Language Models.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016a. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016b. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021a. Measuring massive multitask language understanding. *Preprint*, arXiv:2009.03300.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021b. Measuring mathematical problem solving with the math dataset. *NeurIPS*.

Geoffrey Hinton. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*.

Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. 2016. Densely connected convolutional networks. *CoRR*, abs/1608.06993.

Ajay Kumar Jaiswal, Bodun Hu, Lu Yin, Yeonju Ro, Tianlong Chen, Shiwei Liu, and Aditya Akella. 2024. FFN-SkipLLM: A hidden gem for autoregressive decoding with adaptive feed forward skipping. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 16943–16956, Miami, Florida, USA. Association for Computational Linguistics.

Tianjie Ju, Weiwei Sun, Wei Du, Xinwei Yuan, Zhaochun Ren, and Gongshen Liu. 2024. How large language models encode context knowledge? a layer-wise probing study. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 8235–8246, Torino, Italia. ELRA and ICCL.

Daniel Kahneman. 2011. Thinking, fast and slow. *Farrar, Straus and Giroux*.

Alind Khare, Dhruv Garg, Sukrit Kalra, Snigdha Grandhi, Ion Stoica, and Alexey Tumanov. 2023. Superserve: Fine-grained inference serving for unpredictable workloads. *Preprint*, arXiv:2312.16733.

Bo-Kyeong Kim, Geonmin Kim, Tae-Ho Kim, Thibault Castells, Shinkook Choi, Junho Shin, and Hyoung-Kyu Song. 2024. Shortened llama: Depth pruning for large language models with comparison of retraining methods. *Preprint*, arXiv:2402.02834.

Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. 2019. Similarity of neural network representations revisited. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3519–3529. PMLR.

Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. 2022. The optimal bert surgeon: Scalable and accurate second-order pruning for large language models. *arXiv preprint arXiv:2203.07259*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.

Kevin J Liang, Weituo Hao, Dinghan Shen, Yufan Zhou, Weizhu Chen, Changyou Chen, and Lawrence Carin. 2020. Mixkd: Towards efficient distillation of large-scale language models. *arXiv preprint arXiv:2011.00593*.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems*, 36:21702–21720.

Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and weipeng chen. 2025. ShortGPT: Layers in large language models are more redundant than you expect.

Zizheng Pan, Jianfei Cai, and Bohan Zhuang. 2023. Stitchable neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16102–16112.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE.

Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Shengkun Cui, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2024. One queue is all you need: Resolving head-of-line blocking in large language model serving. *Preprint*, arXiv:2407.00047.

Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2024. vattention: Dynamic memory management for serving llms without pagedattention.

Aman Priyanshu, Yash Maurya, and Zuofei Hong. 2024. Ai governance and accountability: An analysis of anthropic's claude. *arXiv preprint arXiv:2407.01557*.

Alec Radford and Karthik Narasimhan. 2018. Improving language understanding by generative pretraining.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019a. Language models are unsupervised multitask learners.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019b. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Winogrande: an adversarial winograd schema challenge at scale. *Commun. ACM*, 64(9):99–106.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: high-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR.

Sharath Turuvekere Sreenivas, Saurav Muralidharan, Raviraj Joshi, Marcin Chochowski, Ameya Sunil Mahabaleshwarkar, Gerald Shen, Jiaqi Zeng, Zijia Chen, Yoshi Suhara, Shizhe Diao, Chenhan Yu, Wei-Chun Chen, Hayley Ross, Oluwatobi Olabiyi, Ashwath Aithal, Oleksii Kuchaiev, Daniel Korzekwa, Pavlo Molchanov, Mostofa Patwary, Mohammad Shoeybi, Jan Kautz, and Bryan Catanzaro. 2024. Llm pruning and distillation in practice: The minitron approach. *Preprint*, arXiv:2408.11796.

Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. 2023. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. 2019. CommonsenseQA: A question answering challenge targeting commonsense knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4149–4158, Minneapolis, Minnesota. Association for Computational Linguistics.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2024a. Burstgpt: A real-world workload dataset to optimize llm serving systems. *Preprint*, arXiv:2401.17644.

Yuxin Wang, Yuhan Chen, Zeyu Li, Zhenheng Tang, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2024b. Towards efficient and reliable llm serving: A real-world workload study. *arXiv preprint arXiv:2401.17644*.

Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. 2024. Sheared LLaMA: Accelerating language model pre-training via structured pruning. In *The Twelfth International Conference on Learning Representations*.

Chuanpeng Yang, Yao Zhu, Wang Lu, Yidong Wang, Qian Chen, Chenlong Gao, Bingjie Yan, and Yiqiang Chen. 2024. Survey on knowledge distillation for large language models: Methods, evaluation, and application. *ACM Trans. Intell. Syst. Technol.* Just Accepted.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800, Florence, Italy. Association for Computational Linguistics.

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024a. Tinyllama: An open-source small language model. *Preprint*, arXiv:2401.02385.

Yang Zhang, Yanfei Dong, and Kenji Kawaguchi. 2024b. Investigating layer importance in large language models. *arXiv preprint arXiv:2409.14381*.

Yang Zhang, Yanfei Dong, and Kenji Kawaguchi. 2024c. Investigating layer importance in large language models. In *Proceedings of the 7th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pages 469–479, Miami, Florida, US. Association for Computational Linguistics.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *Preprint*, arXiv:2401.09670.

A Performance across Tasks

We evaluate the commonsense reasoning (CommonsenseQA, Figure 14), coreference resolution (Winogrande, Figure 15), commonsense inference (HellaSwag, Figure 16), knowledge-intensive understanding (MMLU, Figure 17), and Boolean reasoning (BoolQ, Figure 18) capabilities of Llama 2 (13B/7B/1.1B) and Llama 3 (8B/3B/1B), with full results visualized in their respective figures.

B More on Stitching Direction

Figure 19 provides additional evaluations comparing large-small and small-large stitching using Llama 3 (3B and 1B) and Llama 2 (7B) with TinyLlama (1.1B). The dataset used is MMLU.

C More on Performance Boundary

Figure 20 presents additional evaluations of the performance boundary patterns for Llama 2, Llama 3, and Qwen 2.5. The dataset used is MMLU.

D Compare To Block Skipping

Block stitching and block skipping (Jaiswal et al., 2024; Chen et al., 2023; Men et al., 2025; Corro et al., 2023; Kim et al., 2024; Bae et al., 2023), are two approaches designed to lower the resource requirements of LLMs. In Figure 21, we compare their performance on MMLU. Our results show that block stitching delivers a more stable balance between accuracy and resource efficiency than block skipping. Moreover, combining these two strategies may yield even more favorable accuracy-resource tradeoffs.

E More Complex Stitching Layer

We investigate whether more complex stitching layer designs can further boost the accuracy of stitched models. In our experiments, we combine the bottom blocks from Llama 3 8B with the top blocks from both Llama 3 1B and 3B. For the 8B-to-3B stitching, we use a three-layer MLP with dimensions 4096×4096, 4096×3072, and 3072×3072, inserting ReLU activations between layers. For the 8B-to-1B stitching, we similarly use three MLP layers sized 4096×4096, 4096×2048, and 2048×2048. As shown in Figure 22 (evaluated on MMLU), the complex stitching layers improve overall performance and yield smoother accuracy-resource tradeoffs. This finding reinforces the potential of LLM stitching and opens up opportunities to create heterogeneous LLMs with decoders of varying sizes.

F Locality-aware Block Placement

We compare the communication costs between `StitchLLM`'s locality-aware placement and the widely adopted fragmentation-minimized (frag-min) placement. Figure 23 shows the average performance change of using the frag-min placement. The median and 95%ile latency is increased by 12.6% and 18.2%. The communication costs of one request sum up all the transfer costs, therefore presenting a significant inflation of 73.4%. The locality-aware placement has reduced 72.3% inter-server communications compared with the frag-min placement strategy.

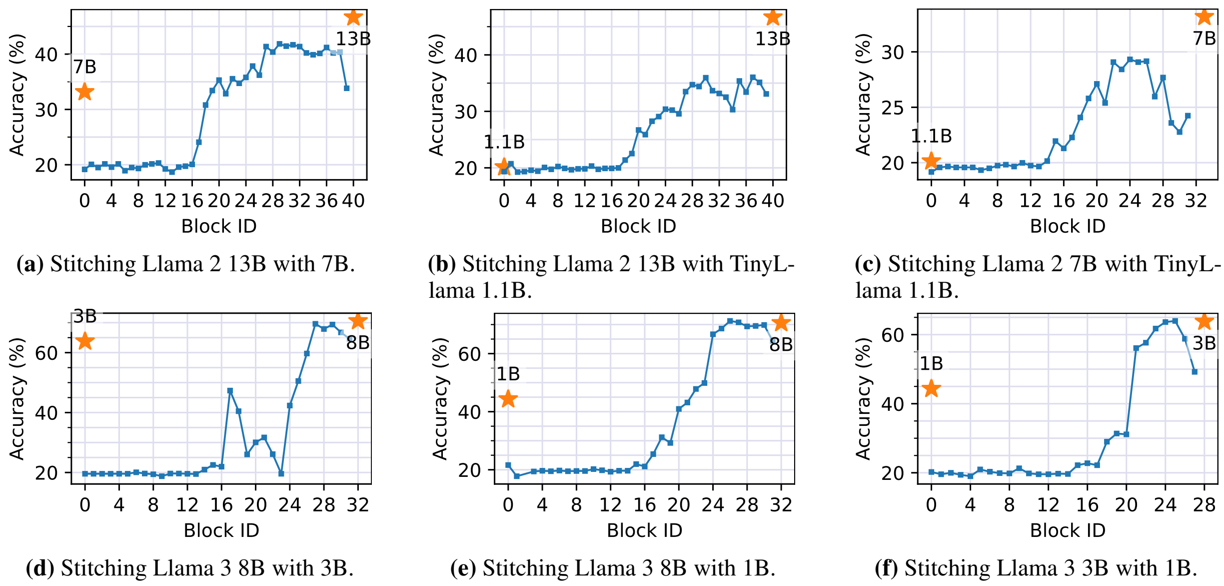
G More on GPU Utilization

In Figure 24, we show the memory consumption of model parameters and request-related data, including input, intermediate activations, output, and the KV cache. In the optimal scenario, BlockLLM utilizes 16.1% less space for model parameters and 24.1% more space for request-related data, indicating that more requests are being processed. This increase is attributed to our ability to share smaller top blocks among multiple top blocks, thereby freeing up more memory for request processing.

H Implementation Details

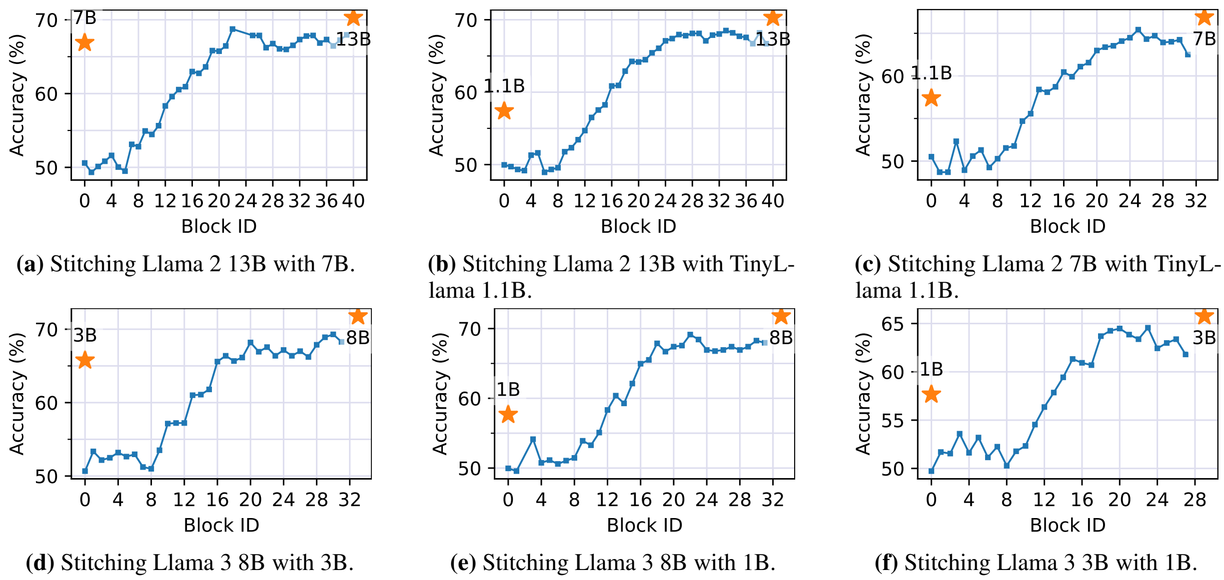
We have implemented a prototype of `StitchLLM` on top of vLLM (Kwon et al., 2023). It is compatible with HuggingFace models. We use NCCL for data transfer among servers.

Profiling. To support the online serving system,



(a) Stitching Llama 2 13B with 7B.
(b) Stitching Llama 2 13B with TinyLlama 1.1B.
(c) Stitching Llama 2 7B with TinyLlama 1.1B.
(d) Stitching Llama 3 8B with 3B.
(e) Stitching Llama 3 8B with 1B.
(f) Stitching Llama 3 3B with 1B.

Figure 14: CommonsenseQA performance across different decoder blocks.



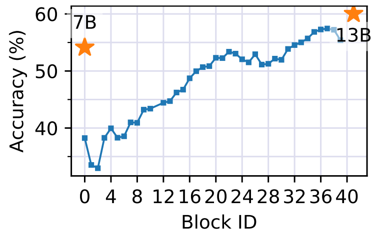
(a) Stitching Llama 2 13B with 7B.
(b) Stitching Llama 2 13B with TinyLlama 1.1B.
(c) Stitching Llama 2 7B with TinyLlama 1.1B.
(d) Stitching Llama 3 8B with 3B.
(e) Stitching Llama 3 8B with 1B.
(f) Stitching Llama 3 3B with 1B.

Figure 15: Winogrande performance across different decoder blocks.

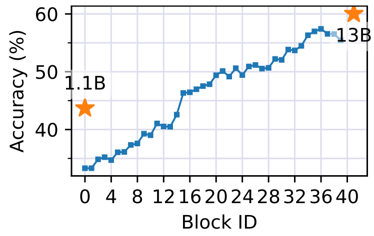
`StitchLLM` profiles blocks by measuring computation time across various batch sizes, including surrogate computations and multiplexing performance. It also evaluates communication latency between devices using NCCL primitives and quantifies the overhead of loading the block engine from disk into host and device memory.

Batching. While larger batch sizes improve computational efficiency, enforcing a fixed large batch size complicates request reorganization. To balance flexibility and efficiency, `StitchLLM` loosely encourages batching within each block instance. When a new batch arrives, `StitchLLM` 's agent queues it and attempts to merge it with neighboring requests, ensuring the combined batch remains within the upper batch size limit. If no queued requests are available, the agent processes the batch immediately. Requests reaching EOS are removed and forwarded to the scheduler.

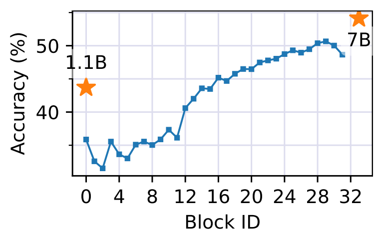
Request dispatching. `StitchLLM` 's agents employ a FIFO + priority queue, giving precedence to requests that have exited KV cache memory. Each block instance maintains a countdown clock for auto-regressive requests, ensuring their timely processing. The scheduler and agents handle dispatching differently: agents identify candidate blocks, pack requests, and broadcast them to available agents, while the scheduler maintains a live record of block placements, streamlining dispatching.



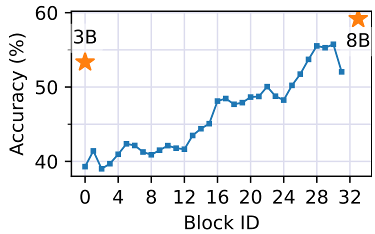
(a) Stitching Llama 2 13B with 7B.



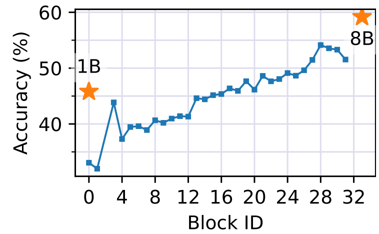
(b) Stitching Llama 2 13B with TinyLlama 1.1B.



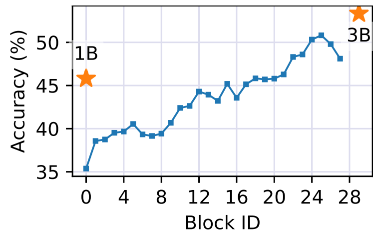
(c) Stitching Llama 2 7B with TinyLlama 1.1B.



(d) Stitching Llama 3 8B with 3B.

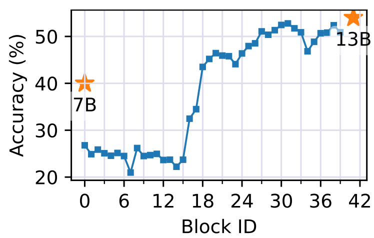


(e) Stitching Llama 3 8B with 1B.

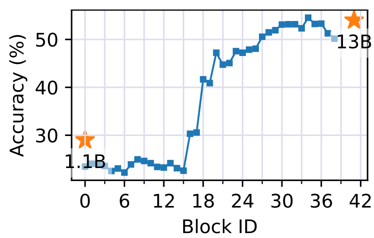


(f) Stitching Llama 3 3B with 1B.

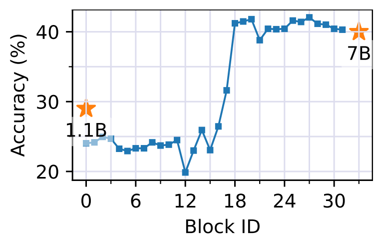
Figure 16: Hellaswag performance across different decoder blocks.



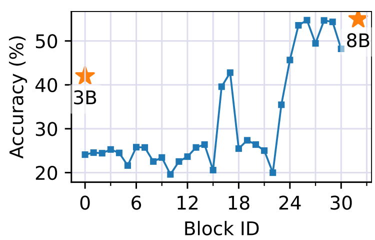
(a) Stitching Llama 2 13B with 7B.



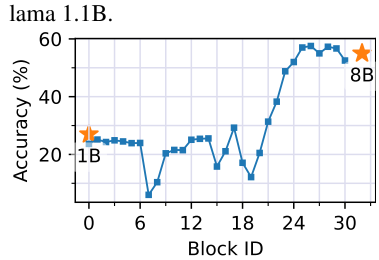
(b) Stitching Llama 2 13B with TinyLlama 1.1B.



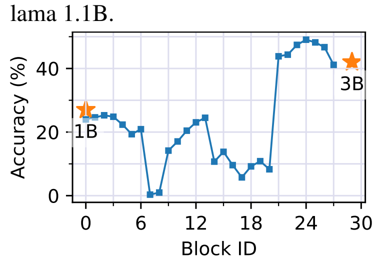
(c) Stitching Llama 2 7B with TinyLlama 1.1B.



(d) Stitching Llama 3 8B with 3B.



(e) Stitching Llama 3 8B with 1B.



(f) Stitching Llama 3 3B with 1B.

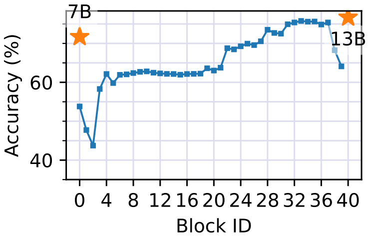
Figure 17: MMLU performance across different decoder blocks.

I KV Cache Coordination

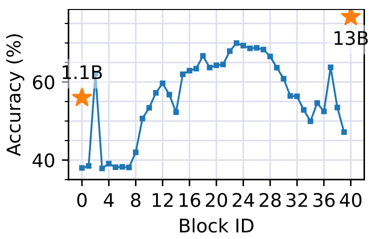
Memory bandwidth-bound KV cache. Efficient stateful coordination of the KV cache is crucial for auto-regressive LLMs in `StitchLLM`, as memory bandwidth constraints on the KV cache have been identified as a significant bottleneck in numerous studies. Existing systems process one batch of requests at a time, weighing the trade-off between recalculating the KV matrices and caching them in device memory. This trade-off reaches a point—determined by factors such as device type, model architecture, and request sequence length—where caching becomes more efficient than recalculation. However, as request sequences lengthen, memory bandwidth constraints become a performance-bounding factor when loading the KV cache (Kwon et al., 2023).

I/O and recalculation cost. As mentioned in Section 3, `StitchLLM`'s design complicates the problem. The assumption that requests are consistently processed by the same block instances no longer holds, making I/O costs for transferring KV caches between instances unavoidable.

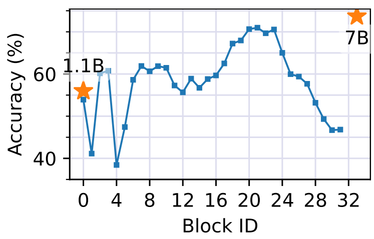
To migrate the KV cache from device d_i to d_j, we optimize the process by overlapping KV cache recomputation with copying, thereby minimizing migration time. Given the fully known context, we employ chunked pre-filling for efficient recomputation. For sequences to be migrated, denoted as



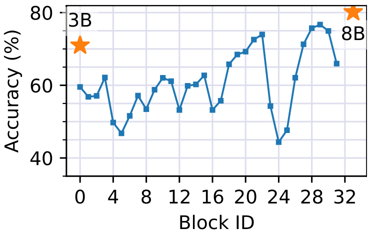
(a) Stitching Llama 2 13B with 7B.



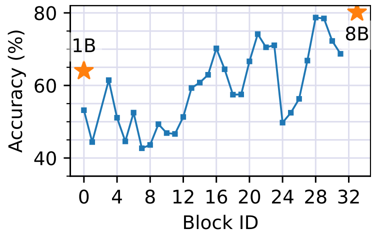
(b) Stitching Llama 2 13B with TinyLlama 1.1B.



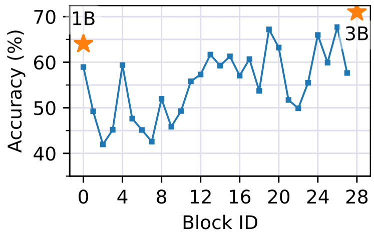
(c) Stitching Llama 2 7B with TinyLlama 1.1B.



(d) Stitching Llama 3 8B with 3B.

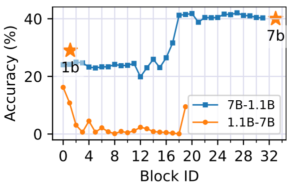


(e) Stitching Llama 3 8B with 1B.

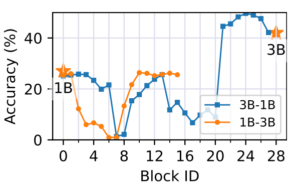


(f) Stitching Llama 3 3B with 1B.

Figure 18: BoolQ Performance across different decoder blocks.



(a) Llama 2 7B and TinyLlama 1.1B.



(b) Llama 3.2 1B and 3B.

Figure 19: MMLU Accuracy Across Different Decoder Layers: The orange line represents using smaller bottom blocks paired with larger top blocks, while the blue line depicts larger bottom blocks combined with smaller top blocks.

$S = s_{i1}^{n}$, we begin by recomputing the KV cache from the start of sequence s_1 while simultaneously copying the cache starting from the end of sequence s_n. The process concludes when recomputation encounters a KV cache page that has already been copied, indicating the completion of migration.

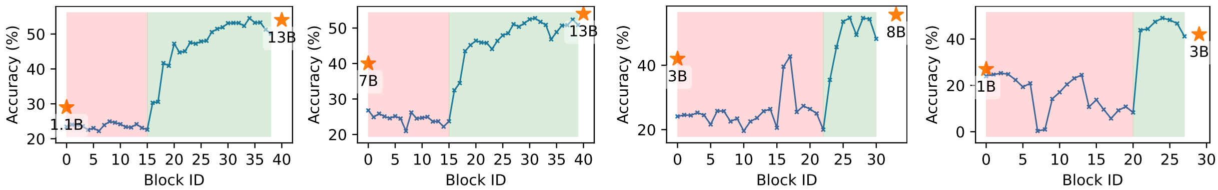
This approach is chosen for two key reasons. First, each token's KV cache depends on the KV caches of all preceding tokens. Recomputing the KV cache from the beginning of a sequence ensures the accuracy of the entire cache. In contrast, copying can take place independently, without relying on preceding caches. Second, ongoing requests on the target device may introduce latency during recomputation. By employing chunked prefill, we improve GPU utilization and mitigate the impact of KV cache recomputation on other tasks.

Proactive KV Cache Migration. As `StitchLLM` may redirect requests to blocks lacking the necessary KV caches, this can introduce additional migration latency. While this overhead cannot be completely eliminated, it can be mitigated by proactively migrating KV caches in advance, thereby removing it from the critical path.

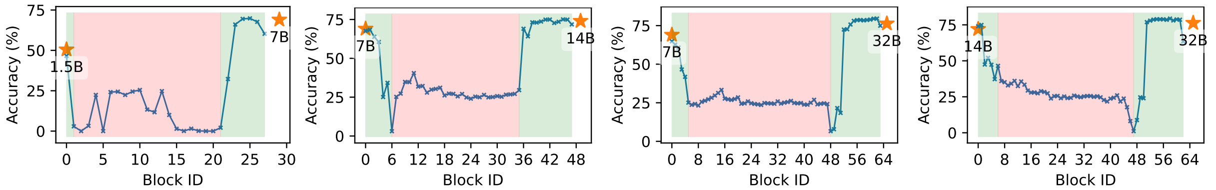
To ensure that migration does not introduce latency, it is essential to predict whether the KV cache will be used before the migration completes. The feasibility of predicting KV cache usage has been demonstrated in (Abhyankar et al., 2024). We adopt the method proposed in (Abhyankar et al., 2024) to estimate the interception time: $T_{INT} = t_{now} - t_{call}$, where t_{now} is the current time updated for each iteration, and t_{call} is the time when the last interception was initiated. Figure 25 shows an illustration of our approach.

Memory Efficiency. Modern LLM inference serving systems support paged attention (Kwon et al., 2023), a technique that partitions the KV cache into smaller pages. This approach eliminates the need to store the entire KV cache in contiguous memory and allows for the sharing of KV cache pages across multiple requests, thereby enhancing memory efficiency. However, dynamically routing requests to blocks that lack the required KV cache can result in the creation of new KV cache pages. Since each device maintains its own dedicated KV cache page table, generating the same KV cache page on a different device leads to the duplication of KV pages. This duplication, which otherwise would only require a single KV page with an incremented reference counter, undermines the advantages of memory sharing.

To prevent memory waste, we prioritize migrating pages referenced by fewer requests before those referenced by more. We denote all KV pages as

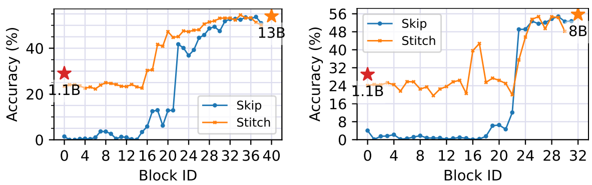


(a) Llama 2 13B and TinyLlama 1.1B. **(b)** Llama 2 13B and Llama 2 7B. **(c)** Llama 3.1 8B and Llama 3.2 3B. **(d)** Llama 3.2 3B and Llama 3.2 1B.



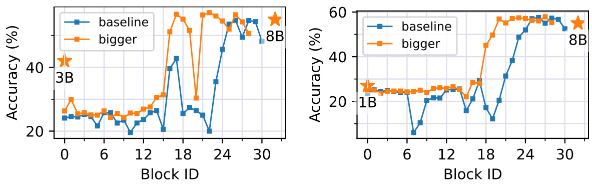
(e) Qwen 2.5 7B and 1.5B. **(f)** Qwen 2.5 14B and 7B. **(g)** Qwen 2.5 32B and 7B. **(h)** Qwen 2.5 32B and 14B.

Figure 20: MMLU performance using diffrent stitching block configurations on Llama 2, Llama 3, and Qwen 2.5.



(a) ⋆: Tiny Llama ⋆: Llama 2 13B **(b)** ⋆: Llama 3 1B ⋆: Llama 3 8B

Figure 21: Accuracy on MMLU: Block Stitching vs. Block Skipping. The orange line shows performance when stitching is applied at every decoder block, while the blue line represents skipping all subsequent decoder blocks after a given block.



(a) Stitching Llama 3 8B and 3B. **(b)** Stitching Llama 3 8B and 1B.

Figure 22: Accuracy on MMLU: Single Linear Stitching vs. Multi-Layer Stitching. The blue line shows performance with a single linear stitching layer, while the orange line reflects results using larger stitching blocks composed of multiple linear layers with activation functions.

$C = \{c_1, c_2, ...c_n\}$, where each c_i represents the underlying consecutive KV pages of request s_i, and n is the total number of requests tracked by the system. We use $ref(c_i)$ to calculate the total number of pages referenced by more than one request in sequence s_i. For KV cache pages $c_i \in C$, we have $ref(c_i) \leq ref(c_{i+1})$. If $ref(c_i) = ref(c_{i+1})$, then $resumeTime(c_i) <= resumeTime(c_{i+1})$, where $resumeTime(c_i)$ is the estimated time the pages c_i will be reused by the intercepted request s_i.

Prioritize KV cache owner. Transferring KV

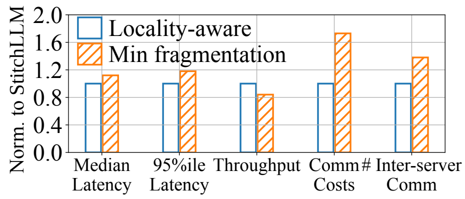


Figure 23: `StitchLLM` compared with fragmentation-minimized placement.

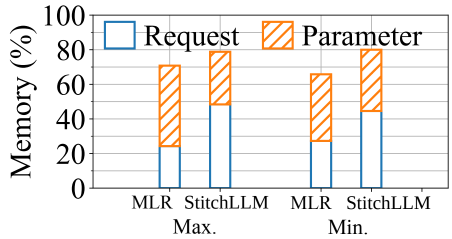
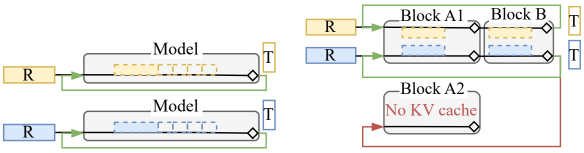


Figure 24: Memory used for parameters: Memory usage of parameters and request-related data. We observe that due to performing block level merging, `StitchLLM` is able to minimize memory used for parameters.

caches to a new block instance can introduce latency, impacting request processing times. This latency is subject to network bandwidth and varying network conditions. Although recalculating the KV cache can sometimes be more efficient than direct copying, it may still disrupt ongoing requests on the target device. Therefore, we prioritize block instances that already possess the required KV cache before redirecting requests to a new device. This approach follows the principle of best-effort coordination. When candidate block instances have the same status (e.g., queuing time), `StitchLLM`'s agent prioritizes dispatching the request to the block instance that holds the associated KV cache. In Appendix K, we provide a detailed discussion on how `StitchLLM`'agent selects the ap-



(a) Per-model provision. **(b)** Block provision.

Figure 25: Illustration of coordinating KV cache when using block-granularity provisioning.

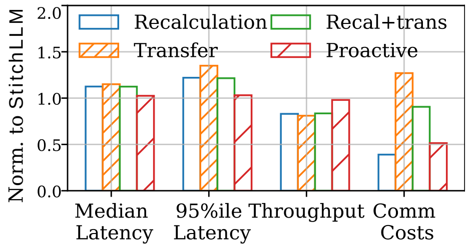


Figure 26: Ablation study of KV cache coordination strategy.

propriate block instance.

J Best-effort KV cache coordination.

`StitchLLM` performs best-effort dispatching by prioritizing the device with its KV cache memory. We consider two other solutions to verify if `StitchLLM`'s strategy is efficient: (1) All the KV cache are obtained using recalculation. (2) We always route the request back to the least busy device and let the KV cache owner transfer the cache to the instance. Figure 26 shows the median and 95%ile latency, throughput, and communication costs normalized to `StitchLLM`. The 95%ile latency is increased by 1.23x using recalculation and by 1.36x using least-busy-device routing. The communication costs are reduced significantly to 0.36 of `StitchLLM` when using recalculation and increased to 1.28x when using least-busy-device routing.

K Scheduler Formulation

$$Latency_{d_c} = T_{queue} + T_{compute} + T_{transfer} + T_{load},$$

$$T_{queue} = \sum_{i=1}^{n} Comp(req_i),$$

$$T_{compute} = Comp(req),$$

$$T_{transfer} = \frac{D_{req}}{B_{net}(s, d_c)} \quad \text{scheduler dispatches},$$

$$T_{load} = \begin{cases} \dfrac{D_b - D_{b'}}{B_{mem}(d_c)} & \text{Memory}, \\ \dfrac{D_b - D_{b'}}{B_{net}(d_c)} & \text{Network}. \end{cases}$$

We incorporate four key factors: queuing (T_{queue}), computation ($T_{compute}$), transfer ($T_{transfer}$), and the overhead associated with block switching (T_{load}). T_{queue} accounts for the duration required to process all n queuing request batches. $T_{transfer}$, denotes the time needed for scheduler s to send requests to target node. Here, D_{req} is the size of the request token, and $B_{net}(s,\ d_j)$ denotes the network bandwidth between two devices. T_{load} denotes the time needed to transfer the needed blocks to the target device. D_b is the total size of the blocks needed, and D'_b is the sizes of blocks already exists on target devices. The transfer will use CPU memory, or resort to network transfer if the needed block is not in memory. $B_{mem}(d_c)$ is the device memory bandwidth of candidate d_c.