Toward High-Performance LLM Serving: A Simulation-Based Approach for Identifying Optimal Parallelism

Yi-Chien Lin
yichienl@usc.edu
Univeristy of Southern California
Los Angeles, California, USA

Woosuk Kwon
woosuk.kwon@berkeley.edu
University of California, Berkley
Berkeley, California, USA

Ronald Pineda
ronaldp10@ucla.edu
University of California, Los Angeles
Los Angeles, California, USA

Fanny Nina Paravecino
fanny.nina@microsoft.com
Microsoft
Mountain View, California, USA

Abstract

Large Language Models (LLMs) have achieved significant success in various domains, and serving LLMs efficiently has become crucial. LLMs are often served with multiple devices using parallelism techniques like data, pipeline, and tensor parallelisms. Each parallelism presents trade-offs between computation, memory, and communication overhead, making it challenging to determine the optimal parallel execution plan. Moreover, input workloads also impact parallelism strategies. Tasks with long prompts like article summarization are compute-intensive, while tasks with long generation lengths like code generation are often memory-intensive; these differing characteristics result in distinct optimal execution plans. Since searching for the optimal plan via actual deployment is prohibitively expensive, we propose APEX, an LLM serving system simulator that efficiently identifies an optimal parallel execution plan. APEX performs dynamism-aware simulation, which captures the complex characteristics of iteration-level batching, a technique widely used in state-of-the-art LLM serving systems. APEX leverages the repetitive structure of LLMs to reduce design space, maintaining a similar simulation overhead, even when scaling to trillion scale models. APEX supports a wide range of LLMs, device clusters, etc., and it can be easily extended to new models or clusters through its high-level templates. We run APEX simulations using a CPU and evaluate the optimal parallel execution plans found by APEX using a cluster with 8 H100 GPUs. The simulations cover a myriad of LLM architectures, precision formats, cluster configurations, and input workloads. We show that APEX can find optimal execution plans that are up to 4.42× faster than heuristic plans in terms of end-to-end serving latency. APEX also reports a set of metrics used in LLM serving systems, such as time per output token (TPOT) and time to first token (TTFT). Furthermore, APEX can identify an optimal parallel execution plan within 15 minutes using a CPU. This is 71× faster and 1234× more cost-effective than actual deployment on a GPU cluster using cloud services. APEX will be open-sourced upon acceptance. Finally, we show that LLM service providers can utilize APEX to meet service-level objectives and explore hardware design space to build high-performance LLM serving platforms.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/xxx.xxx

CCS Concepts

• **Computing methodologies → Modeling and simulation**; **Parallel computing methodologies**.

Keywords

LLM Serving, Optimal Parallelism, Simulation

ACM Reference Format:
Yi-Chien Lin, Woosuk Kwon, Ronald Pineda, and Fanny Nina Paravecino. 2018. Toward High-Performance LLM Serving: A Simulation-Based Approach for Identifying Optimal Parallelism. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 14 pages. https://doi.org/xxx.xxx

1 Introduction

Large Language Models (LLMs) have been successfully applied to various applications, such as code generation [12, 45], question-answering [28, 49], and many more [40, 46, 48]. Serving an LLM is both memory- and compute-intensive; therefore, leveraging a cluster with multiple devices is essential to gain computing power and memory resources for achieving high performance. Several techniques have been proposed to parallelize LLM on multiple devices, such as data parallelism (DP) [24], pipeline parallelism (PP) [14], tensor parallelism (TP) [37], etc. Each parallelism has its pros and cons. Techniques like TP are more memory-efficient than DP as it does not produce model replicas, but they incur expensive collective communication overhead [37]. On the other hand, PP has lower communication overhead but can suffer from workload imbalance [14]. To balance the trade-offs, parallelism techniques can be adopted in a hybrid manner, potentially leading to better serving performance than relying on a single type of parallelism. However, determining the optimal parallel execution plan is challenging due to the many factors involved, including computation workload, network traffic, memory efficiency, etc. Furthermore, the optimal parallel execution plan also depends on the characteristics of the input requests. Some requests lead to long prompts with short generation lengths, such as summarizing an article, while some lead to short prompts and long generation lengths, like code generation and storytelling. The former type of request is compute-intensive, while the latter is memory-intensive. As a result, these two types of requests favor different parallelism techniques to optimize performance. A common practice is to adopt heuristic execution plans instead, such as applying TP within the same node and PP across different nodes [30, 39]. Yet, a recent study has shown that applying such heuristics approaches can be up to 2× slower than the optimal configuration [17]; this calls for the need for service providers to search for and adopt an optimal parallel execution plan rather than relying on heuristics.

To find an optimal execution plan, a straightforward approach is to deploy and evaluate various parallel execution plans directly; however, this approach is prohibitively expensive, as it could take thousands of GPU hours to assess multiple execution plans [1, 2]. Note that this high searching cost cannot be amortized, as the optimal configuration varies depending on the model and the workload characteristics of the input requests. An alternative is to search via modeling-based approaches, such as developing performance models [17] or building simulators [9, 42] to estimate the performance of a given parallel execution plan; nevertheless, existing solutions developed for Deep Neural Network [9, 42] or LLM training [4, 17] cannot be applied to LLM serving, as such systems introduce unique challenges: (1) **Dynamism of iteration-level batching:** Unlike conventional ML systems that adopt static batching, which waits for all requests in the current batch to be completed before batching new requests, state-of-the-art LLM serving systems [2, 21, 47] adopt *iteration-level batching* [41] to achieve high serving performance. With iteration-level batching, incoming requests are continuously added to the processing batch whenever memory becomes available. This makes it challenging to model the system, as the batch size changes dynamically during serving. Additionally, some requests within the batch may be in the prefill stage, while others are in the generation stage. This interleaving of stages further complicates the modeling, as these stages have significantly different computational characteristics (details in Section 2.1). (2) **Exponentially-growing design space:** Although modeling-based approaches provide a potentially time-efficient solution to search for the optimal parallel execution plan, they can still incur substantial overhead as the design space grows exponentially with respect to the model size and the number of devices in the cluster. Since LLM serving [2, 21, 47] often involves large models and clusters, it is non-trivial to develop a solution that can search for an optimal parallel execution plan within a reasonable timeframe. (3) **Adapting for continuously-evolving systems:** LLM serving system is an emerging area, with continuous evolution in model architecture, hardware platforms, and system optimizations such as quantization [13, 25] and parallelism techniques [26, 34]. Consequently, a performance model or a simulator can easily become obsolete and inapplicable to state-of-the-art LLM serving systems. Thus, keeping pace with the rapid evolution of such systems is necessary but also challenging.

To this end, we propose APEX, an extensible and dynamism-aware simulator for automated parallel execution in LLM serving. APEX takes an LLM, a set of input requests with various context and generation lengths, and a device cluster as inputs, and generates an optimal parallel execution plan for LLM serving. Specifically, for a given LLM model and device cluster, APEX first generates various parallel execution plans, each representing a unique way to parallelize the model by combining various parallelism techniques, such as tensor, data, pipeline, and expert parallelisms. APEX then evaluates each plan by estimating the execution time of serving the input requests through simulation. APEX performs dynamism-aware simulation that is capable of modeling the complex characteristics of iteration-level batching, which involves concurrently serving requests of varying lengths and stages (i.e., prefill and generation stage). After the simulation, APEX provides a comprehensive evaluation for each parallel execution plan, which includes multiple metrics used in LLM serving systems, such as time per output token (TPOT), time to first token (TTFT), end-to-end serving latency, P95 latency, among others [29]. A parallel execution plan with the lowest end-to-end serving latency is chosen as the optimal plan. Despite performing complex dynamism-aware simulations, APEX remains time-efficient by leveraging the repetitive structure of transformer layers in LLMs, which significantly reduces the design space; this allows APEX to scale to trillion-scale models on multi-node device clusters. APEX supports a broad range of LLMs, device clusters, quantization formats, and parallelism techniques, which are essential for modeling state-of-the-art LLM serving systems. Recognizing the rapid evolution of LLM serving systems, APEX is designed to be modular and extensible. We capture the high-level abstractions of LLMs and device clusters, and develop software templates based on these abstractions. This approach allows new models and device clusters to be easily supported with minimal coding effort in the templates. Additionally, APEX leverages operation-level profiling results to estimate the execution time of parallel execution plans on a given device cluster. Profiling-based estimation enables APEX to adapt to different clusters by collecting profiling data from the underlying platform. Note that the device cluster APEX supports is not limited to GPU clusters, but also AI-accelerator clusters such as TPU [20], Intel Gaudi [16], etc. We plan to open-source APEX upon paper acceptance. We further demonstrate that, in addition to identifying optimal parallel execution plans, APEX can also assist LLM service providers in meeting service-level objectives (SLOs) and provide valuable insights for building LLM-serving hardware platforms. The key contributions of this work are:

- We propose APEX, a dynamism-aware simulator that captures the complex characteristics of LLM serving. APEX automatically finds an optimal parallel execution plan for a given LLM, a device cluster, and a set of input requests.
- APEX supports a wide range of LLMs, parallelism techniques, quantization formats, and device clusters. APEX can also be easily extended to new models and device clusters.
- We evaluate APEX using four LLMs on three datasets with distinct workloads. APEX identifies optimal parallel execution plans that are up to 4.42× faster than heuristic plans in terms of the end-to-end serving latency.
- APEX provides high-fidelity simulation, achieving less than 10% relative error on average when predicting the speedup of optimal execution plans over the baseline plans.
- APEX is time-efficient and cost-effective, capable of finding an optimal parallel execution plan within 15 minutes on a CPU; this is 71× faster and 1234× cost-effective than actual deployment on a GPU cluster using cloud services.

- APEX is highly scalable, maintaining similar simulation overhead when scaling from billion-scale to trillion-scale models.
- APEX features comprehensive evaluation, which can help service providers meet SLOs. APEX also provides insightful suggestions for building LLM-serving hardware platforms.

2 Background

2.1 LLM Inference

Large Language Models (LLMs) are a type of ML models that are built upon transformer architecture [38]. LLM inference takes a user prompt as input, and generates a response, token by token, sequentially. The sequential token generation process is also known as the auto-regressive generation, which utilizes the previously generated tokens as input to predict the next token. The inference process of LLMs consists of two stages:

Prefill Stage: In the prefill stage, LLM processes the input request (i.e., prompt) to set up intermediate states (keys and values) that are used to predict the first token. Unlike token generation, the computation of prefill does not rely on previously generated output tokens, allowing the tokens of the input request to be processed in parallel all at once. This high degree of parallelism makes the prefill stage compute-bound [47].

Generation Stage: During the generation phase, LLM generates output tokens autoregressively until a stopping criterion is met. The generation of each token depends on all previous tokens' output states (keys and values). The generation phase is often memory-bound [25], as the latency mainly depends on the speed of data transfer for the output states from the memory, rather than on the computation itself.

2.2 LLM Serving and Iteration-Level Batching

LLMs are often hosted by service providers in the cloud [23]. Users send requests to these providers through APIs or chatbots and then receive responses generated by the LLMs. While a single LLM inference processes a single input request and generates one output response, LLM serving concurrently processes multiple input requests from the users and generates multiple output responses, i.e., multiple LLM inferences happen in parallel. For LLM serving, it is critical to achieve both low latency and high throughput: low latency is necessary to meet service-level objectives, while high throughput enables service providers to serve a large number of users simultaneously. Due to the autoregressive nature of LLMs, the lengths of the generated responses can vary significantly. Consequently, the commonly used static batching leads to suboptimal performance in LLM serving, as all the batched requests need to wait for the longest response to be generated to proceed to process new requests. To overcome this inefficiency, *iteration-level batching* [41] is proposed. Iteration-level batching continuously schedules newly arrived request(s) into the existing batch whenever GPU memory becomes available, rather than waiting for all the requests to be completed. Iteration-level batching significantly improves the serving throughput and is widely adopted in state-of-the-art LLM serving systems [2, 21, 32, 47].

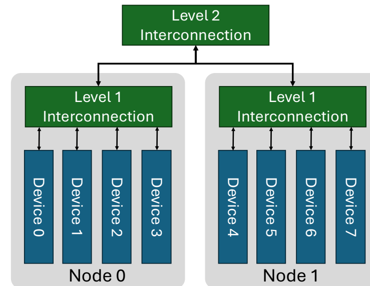

Figure 1: An example of a two-level device cluster hierarchy, where leaves represent the devices. The memory bandwidth and latency are uniform within the same level.

2.3 Device Clusters and Interconnections

Serving an LLM is both memory- and compute-intensive. To achieve high serving throughput and accommodate the large model size, LLMs are often deployed on a cluster of devices. The devices are connected in a tree-based network topology, which is one of the most popular topologies for node deployment. The memory bandwidth and latency is uniform within each level. Figure 1 illustrates an example of a commonly used two-level device cluster. Devices within the same node are typically connected by high-bandwidth interconnects like NVLink [22] at level 1, while devices across different nodes are connected via inter-node networks such as InfiniBand [27] at level 2. In addition to GPUs, clusters of AI accelerators are also emerging, such as TPU clusters [20] and Gaudi clusters [16]. These clusters also utilize tree-based network topologies for interconnection and can be abstracted similarly as GPU clusters [7].

2.4 LLM Parallelisms

LLM is often served with multiple devices (Figure 1), and various approaches have been proposed to parallelize LLM on the cluster. We discuss several representative and widely-used parallelisms.

Data Parallelism (DP): In DP [24], the model is replicated across multiple devices. The input requests are split into micro-batches and distributed to each model replica for processing. DP incurs no communication overhead as each micro-batch is processed independently. However, having model replicas incurs large memory overhead, and reduces the number of requests that can be batched concurrently.

Pipeline Parallelism (PP): PP [14, 31] divides the model in a layer-wise fashion. Each model partition consists of a subset of layers, and each subset of layers forms a pipeline stage. The input requests are split into micro-batches to flow through the pipeline stages. PP requires point-to-point (p2p) communications between the pipeline stages. In addition, PP suffers from pipeline stalls when the execution time of each stage is unbalanced.

Tensor Parallelism (TP): TP [37] divides the model in an intra-layer fashion, which splits individual layers across multiple devices. TP does not suffer from load imbalance like PP or produces model replicas like DP. Yet, TP incurs high collective communication overhead, such as AllReduce, which is prohibitively expensive for interconnection networks with limited bandwidth like PCIe.

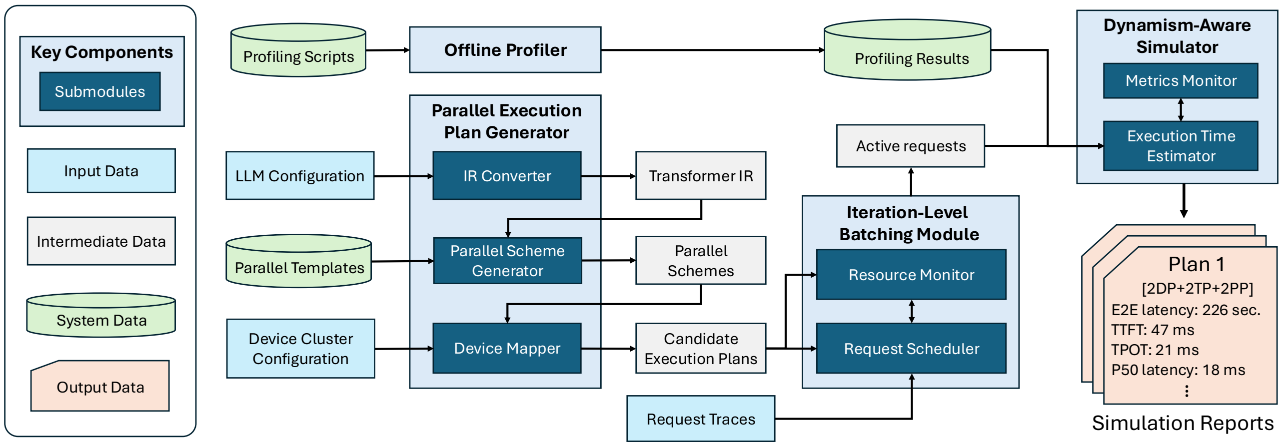

Figure 2: System overview of APEX

Expert Parallelism (EP): EP [34] is a special type of parallelism for Mixture of Experts (MoE) models. EP deploys different experts across various devices. For MoE models, only a subset of experts are activated for each input token. Thus, EP can result in workload imbalance and resource under-utilization if none of the experts on the device are activated; while this can be avoided by adopting TP, where each device is assigned with a partition of all the experts, EP incurs a lower communication overhead than TP.

Each parallelism has its own pros and cons, tradeing-off between computation, memory efficiency, collective communication overhead, etc. Thus, it's non-trivial to determine the optimal parallel execution plan, especially given that the parallelisms can be adopted in a hybrid manner. Heuristic plans are typically adopted for simplicity, such as applying TP to clusters with high-bandwidth interconnections like NVLink [22], and applying PP to clusters without such networks [30, 39]. APEX aims to evaluate the complex trade-offs among parallelism techniques to find optimal parallel execution plans that outperform heuristic approaches.

2.5 LLM Quantizations

Quantization is an essential technique for achieving performant LLM serving, as it reduces both memory and computation overhead. Various methods have been proposed to quantize one or more of the following components to lower precision while preserving high accuracy: (1) model weights, (2) activations, and (3) KV cache. For example, AWQ [25] quantizes the model weights, and KVQuant [25] focuses on quantizing the KV cache. State-of-the-art LLM serving systems like vLLM [21] and TensorRT-LLM [32] also support W8A8 quantization, which quantizes both the model weights and activations to FP8 format. Given the diversity of quantization methods, it is essential to flexibly support various techniques to find an optimal parallel execution plan for LLM serving systems.

3 APEX Simulator Design

3.1 System Overview

We depict the system overview of APEX in Figure 2. Initially, the Offline Profiler (Section 3.2) of APEX takes a set of profiling scripts to obtain the performance information of the underlying platform, and the results are stored for APEX's profiling-based simulation. Profiling is an offline process that is only performed once when porting to a new cluster with unknown devices (e.g., porting from an A100 to an H100 GPU cluster.) Next, given an LLM and device cluster, the Parallel Execution Plan Generator converts the model into an intermediate representation (Section 3.3) and generates various parallel execution plans (Section 3.4 and 3.5); each plan represents a unique way to map the LLM onto the device cluster. The Iteration-Level Batching Module (Section 3.6) takes the generated parallel execution plans and a set of request traces as inputs, and starts batching the input requests. The Batching Module determines whether a request should be added to or removed from the batch in each iteration, and reports the active requests to the Dynamism-Aware Simulator (Section 3.7). Based on the active requests in the batch, the Dynamism-Aware Simulator estimates the execution time for each iteration and keeps track of multiple LLM system metrics such as time to first token (TTFT), time per output token (TPOT), P95 latency, etc. After all requests have been processed, the Dynamism-Aware Simulator produces a Simulation Report for the corresponding parallel execution plan. This process is repeated for all the generated parallel execution plans, and the plan with the lowest end-to-end latency is selected as the optimal plan.

3.2 Offline Profiler

The Offline Profiler collects performance information of the underlying platform, where the results are used by the Dynamism-Aware Simulator (Section 3.7) to estimate the execution time of different parallel execution plans. The Offline Profiler leverages the fact that LLMs are based on transformer architecture, which consists of similar operations, and performs *operation-level profiling*, such as measuring the computation time of multi-head attention, which is used in the attention layers, and general matrix multiplication (GEMM), which is used in the feedforward layers. Profiling the key transformer operations allows APEX to support various LLMs without exhaustively profiling each model, as they can be broken down into the same set of operations. In addition to computation operation, the Offline Profiler also profiles the collective communication overheads on the cluster, e.g., the time to perform AllReduce,

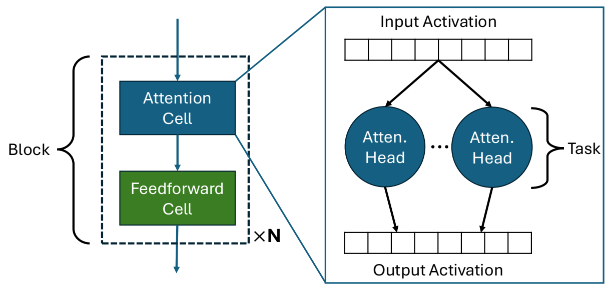

Figure 3: Transformer IR represents LLMs in a canonical way

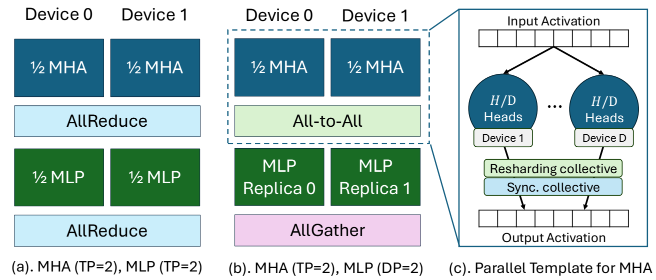

Figure 4: Examples of APEX's Parallel Templates for two devices. The parameterized templates can extend to D devices.

ReduceScatter, All-to-All. This allows APEX to support various parallelisms, which require different collective communications. For the computation operations, the profiling is performed by going through various sequence lengths, number of attention heads, hidden dimensions, etc. For the collective communication operations, the profiling is performed by varying the data transfer sizes and also the number of devices involved, both within a node and across nodes, in the collective operation. Operation-level profiling produces profiling results that can support various LLMs, request traces, parallelisms, and device clusters of different sizes. The profiling only needs to be performed once when porting to a new cluster with unknown devices, such as porting from an A100 GPU cluster to a H100 GPU cluster. Therefore, for the same type of device cluster, the profiling overhead is a one-time cost that can be amortized.

3.3 Transformer IR

APEX identifies an optimal parallel execution plan by generating and evaluating multiple candidate execution plans. Yet, it is non-trivial to generate parallel execution plans for a variety of LLMs as they have different model architectures. To address this, we introduce the *Transformer IR*, a unified abstraction for transformer-based models. APEX utilizes Parallel Templates, developed based on the Transformer IR rather than specific models, to generate parallel execution plans. This approach enables APEX to parallelize a wide range of LLMs that can be represented through the Transformer IR. We discuss the details of Parallel Templates in Section 3.4.

We depict the idea of Transformer IR in Figure 3. We define the key operations of transformers, such as the multi-head attention, as *cells*. An LLM can be represented as a chain of cells. For example, GPT-3 [5] models can be represented as a chain of multi-head attention cells and multi-layer perceptron cells, and Llama-3.1 [10] models can be represented as a chain of group query attention [3] cells and SwiGLU [36] cells. The Transformer IR represents an LLM canonically, which only captures the key cells in the model and ignores operations like tokenization and position embeddings, as they are less relevant for model parallelization; this reduces the search space for simulation. Given that LLMs often use the same set of cells repeatedly, we further define the smallest set of non-repetitive adjacent cells as a *block*. Each cell consists of multiple *tasks*, such as the attention heads in the multi-head attention cell, or the experts in the mixture-of-expert cell. Each task works independently from the other tasks (i.e., no inter-communication is needed), and their outputs are combined via operations like concatenation or AllReduce. To summarize, using the Transformer IR, an LLM can be unifiedly represented as multiple identical *blocks*, each block consists of numerous *cells*, and each cell consists of multiple *tasks*. We develop an IR converter, which parses the information of an LLM configuration file, such as the number of attention heads, number of layers, etc., and represents the model using Transformer IR.

3.4 Parallel Templates

We develop Parallel Templates to generate various parallel execution plans for LLM serving. As mentioned in Section 3.3, the templates are developed based on the Transformer IR, which allows the templates to support a wide range of LLMs. Each cell type is associated with a pre-defined Parallel Template that describes how it can be parallelized on multiple devices. We show some examples in Figure 4. If tensor parallelism is applied to both the multi-head attention (MHA) cell and the multi-layer perceptron (MLP) cell, as in Figure 4 (a), the tasks (e.g., attention heads) in the two cells are evenly distributed to each device. In addition to distributing the tasks, the Parallel Templates also handle the collective communications between adjacent cells. In the example of Figure 4 (a), an AllReduce is performed between the cells for data synchronization, which is similar to the case in Megatron-LM [37]. If the numbers of cell replicas vary between the adjacent cells, tensor resharding is required for the output activation; such cases happen when different data parallelism (DP) degrees are applied. For example, in Figure 4 (b), a 2-way DP is applied to the MLP cell, and a 1-way DP (i.e., no DP) is applied to the MHA cell, resulting in a different number of cells replicas. Consequently, All-to-All and AllGather operations are performed for tensor resharding. Figure 4 (c) depicts a Parallel Template associated with the MHA cell: H attention heads are evenly distributed to D device for computation, and collective communications for resharding and synchronization may also be performed, depending on the number of cell replicas of the adjacent cells. Note that while Figure 4 shows a simplified example of having only two devices, the Parallel Templates are parameterized and can support any number of devices.

3.5 Parallel Scheme Generator & Device Mapper

Given an LLM and pre-defined Parallel Templates, the Parallel Scheme Generator produces various *parallel schemes*, which maps the LLM onto a *logical device cluster* [43] with various combinations of parallelism techniques. A logical device cluster is a virtual

Algorithm 1 Parallel Scheme Generation

```
Input: LLM (in Transformer IR), Parallel Templates
Output: parallel_schemes
n = num_of_devices_in_cluster
for model_DP = 1, 2, ..., n: do                        ▷ model-level DP
    m = n ÷ model_DP                                   ▷ m = num_repica_devices
    for stage = 1, 2, .., m do                         ▷ inter-layer parallelism
        s = m ÷ stage                                  ▷ s = num_stage_devices
        for cell ∈ LLM.block do
            for cell_DP = 1, 2, ..., s: do             ▷ cell-level DP
                c = s ÷ cell_DP                        ▷ c = num_cell_devices
                task_mapping = templates(cell, c)
                cell_schemes.append(task_mapping)
        for r in 0,..., len(LLM.block.cells): do
            reshard = get_reshard_collective(cell_schemes, r)
            stage_schemes.append(cell_schemes, reshard)
        parallel_schemes.append(stage_schemes)
```

overlay that describes the number of devices in the cluster without the information of the network topology. Consequently, a parallel scheme only defines the parallelisms applied to the Transformer IR cells and their corresponding collective communications, such as the examples in Figure 4 (a) and (b), without assigning the cells to specific physical devices. We introduce a virtual overlay of a logical device cluster for design simplicity; this approach allows us to decouple the Parallel Template designs (Section 3.4) from the hierarchal network topology of a device cluster - jointly considering the network topology in the Parallel Templates would greatly increase design complexity. The Parallel Scheme Generator creates parallel schemes using a hierarchical top-down approach, beginning with the most coarse-grained model-level parallelism and gradually refining to fine-grained cell-level parallelism. The detailed procedure is outlined in Algorithm 1. First, the Generator decides the degree of model-level data parallelism (i.e., the number of model replicas), and also derives the number of devices assigned to each replica (i.e., number of replica devices). APEX requires the device cluster to be evenly partitioned by the number of model replicas, so the available degrees of parallelism are restricted to the divisors of the total number of devices in the cluster. Second, the Generator decides the number of pipeline stages within each replica. Similarly, APEX requires the cluster to be evenly partitioned by the number of stages, so the available parallelism degrees are the divisors of the number of devices assigned to the replica. Third, the Generator decides the degree of cell-level data parallelism for each cell, and generates task mappings using the pre-defined Parallel Templates. If the degree of cell-level data parallelism for a specific cell is less than the number of assigned devices, the Parallel Templates apply intra-layer parallelism like tensor or expert parallelism to parallelize the cell (Section 3.4). For example, if four devices are assigned to a cell with two replicas, two devices are assigned to each replica, and the cell can be parallelized using tensor parallelism. Afterward, for each pair of adjacent cells, the Generator inserts collective communications into the parallel schemes if tensor resharding is needed.

After generating the parallel schemes, the Device Mapper maps the logical devices onto the physical device cluster, producing *parallel execution plans*. While the parallel schemes are generated in a top-down approach, the Device Mapper operates in a bottom-up manner. The logical devices are first mapped to physical devices connected at the bottom level of the cluster (see Figure 1); if the number of logical devices exceeds the available physical devices at the current level, the Device Mapper moves to the next upper level to include additional physical devices for mapping. Since lower-level device connections generally offer higher bandwidth than upper-level connections, the Device Mapper prioritizes mapping logical devices assigned to the same cell, as fine-grained cell-level parallelisms tend to incur expensive collective communication overhead, such as the AllReduce in tensor parallelism. The Device Mapper then maps logical devices assigned to the same pipeline stage, followed by those assigned to the same model replica, progressing through increasingly coarse-grained parallelism levels. This bottom-up approach maps logical devices with potentially higher communication overhead to physical devices connected at the lower level, which have higher interconnection bandwidth, and vice versa.

3.6 Iteration-Level Batching Module

The Iteration-Level Batching Module takes the candidate execution plans and request traces as inputs, and simulates the behavior of a serving system using iteration-level batching [41]. Each request comprises three key attributes: context length, generation length, and arrival time. As mentioned in Section 2.2, iteration-level batching continuously adds arriving requests to the current batch whenever memory permits. Therefore, a Resource Monitor keeps track of the memory usage and determines whether a new request can be batched. The Request Scheduler also checks the arrival time of a request to determine whether a request has arrived and can be batched. The Request Scheduler maintains a list of *active requests*, which are the requests in the current batch. The active request list also tracks how many tokens have been generated for each request. If the number of generated tokens matches the generation length of a request, then the request is completed and should be removed from the active batch; this releases memory to batch new requests. One token is generated for each active request in one iteration. In each iteration, the Iteration-Level Batching Module sends the active request list to the Dynamism-Aware Simulator (Section 3.7) to estimate the execution time. The Request Scheduler greedily adds arriving requests to the batch whenever memory is available to accommodate the context length of a request, without considering the memory required for the KV cache of generated tokens. As a result, memory capacity may be exhausted before token generation for all requests is completed. In such cases, the most recently added requests and their generated tokens are temporarily removed from the batch to free up memory for earlier requests to complete their token generation. These removed requests are re-added to the batch as memory becomes available. Both the removal and re-addition are performed in order, prioritizing completing the earliest requests.

3.7 Dynamism-Aware Simulator

Given the active requests produced by the Iteration-Level Batching Module and the Profiling Results, the Dynamism-Aware Simulator

Table 1: Detail of the request traces used for evaluation

Traces	Context Lengths	Generation Lengths	# of requests
Summarization	2742.11±944.33	172.22±73.17	1188
Creation	306.82±81.03	1128.34±419.64	512
Chat	73.32±148.65	189.47±174.18	1024

estimates the execution time of each iteration. The active request list contains information such as the context length of the request and the number of tokens that have already been generated for a request. This allows the Simulator to identify whether a request is in the prefill stage or the generation stage. Specifically, a request is in the prefill stage if the number of generated tokens equals zero; otherwise, it is in the generation stage. For requests in the prefill stage, we calculate the execution time of the operations by setting the input sequence length as the context length, as the entire sequence is processed in parallel (Section 2.1). For requests in the generation stage, we calculate the execution time of the operations by setting the input sequence length as one; this is because the output states (i.e., key and value) of the previous tokens are stored in the KV cache and do not require recomputation. Thus, the computation is only performed on the current token, reducing the effective sequence length to one. Furthermore, requests in the generation stage are processed in parallel; thus, assuming n generation requests, the Simulator estimates the execution time by setting the sequence length as n, as each request has an effective sequence length of one. The Profiling Results contain the execution time of different attention heads, hidden dimensions, sequence lengths, etc., allowing the Simulator to estimate the total execution time for processing the prefill requests and the generation requests. If a specific data point is not presented in the Profiling Results, the Simulator estimates its value by leveraging linear interpolation between the nearest available profiling points. The Simulator is dynamism-aware as the Iteration-Level Batching Module continuously updates the list of active requests in every iteration. In addition, the Simulator utilizes the repetitive nature of LLMs to reduce simulation overhead. As discussed in Section 3.3, a *block* in Transformer IR is the smallest set of non-repetitive cells. Thus, the Simulator performs simulation using a single block and projects the execution time of the entire model, which consists of multiple repetitive blocks. The projection calculation, such as summing or taking the maximum of the execution times, depends on the connection pattern of the blocks (e.g., sequential or pipelined). In addition to estimating the execution time of each iteration, the Simulator also features a Metric Monitor, which calculates the value of several important metrics in LLM serving systems, such as time to first token (TTFT), time per output token (TPOT), P95 latency, Model FLOPs Utilization (MFU), Model Bandwidth Utilization (MBU), among others.

4 Experimental Results

4.1 Experimental Setup

To evaluate APEX, we choose a broad range of LLMs and datasets with distinct workloads. We also experiment with different data types. We discuss the setup details below.

Models: For evaluation, we choose four state-of-the-art LLMs: Qwen2.5-32B-Instruct [15], Llama-3.1-70B-Instruct [10], Mistral-Large-Instruct (123B) [18], and Mixtral 8x22B [19]; This covers a myriad of LLMs in different sizes, ranging from 32B to 123B, and also a Mixture-of-Expert (MoE) model. While the models use half-precision (FP16) by default, we also test cases where the model is quantized to FP8 format.

Datasets: We prepared three sets of request traces derived from distinct datasets: a paper abstract summarization dataset [8], a news abstract summarization dataset [11], and a conversational dataset, LMSYS-Chat-1M [44]. These datasets represent three distinct workloads. The paper abstract summarization [8] exemplifies a prefill-intensive workload, which has a long context length and short generation length. While some LLM workloads, such as paper summarization, are prefill-intensive, some (e.g., code generation and storytelling) are generation-intensive, involving short context lengths and long generation lengths. To create a generation-intensive workload, we adapt the news abstract summarization dataset [11], which consists of short summarizations. We modified the dataset by appending the following prompt to each news summarization: *"Please generate a long story using the provided abstract,"* and use them as input. This prompts the LLM to produce a long story from a short abstract, resulting in a generation-intensive workload. The LMSYS-Chat-1M [44] dataset collects a series of real-world conversations between users and LLMs. Most requests in this dataset feature short context and generation lengths, representing a lightweight conversational workload. For evaluation, we randomly subsampled between 512 and 1K requests from each dataset. To simplify terminology, we refer to these datasets as *Summarization* (paper abstracts), *Creation* (news generation), and *Chat* (LMSYS-Chat-1M) in the following sections. We assume a Poisson distribution for request arrival times. Details of the request traces are listed in Table 1. We report context lengths and generation lengths in terms of tokens, which vary for different tokenizers. We use the Llama-3.1 tokenizer as an example in Table 1; other tokenizers produce similar token counts.

Hardware Platform and Serving Systems: We run APEX simulation using an Intel Xeon 6530 CPU and validated the simulation results against the actual performance of LLM serving on a GPU cluster. We deployed vLLM [21] v0.6.0 to configure an LLM online server on a GPU cluster equipped with 8 Nvidia H100 SXM GPUs, each with 80 GB of GPU memory. Requests were sent to the server according to their arrival times. Since vLLM does not natively support data parallelism, we implemented data parallelism by setting up multiple vLLM servers and dispatching requests to them in a round-robin fashion.

4.2 Evaluation of APEX's suggestion

In this experiment, we address the following research question: **Can APEX improve LLM serving performance by identifying an optimal parallel execution plan?** For evaluation, we design multiple tasks, with each task comprising an LLM, request traces, and an arrival rate. We run APEX simulation for each task and compare the performance of the following three execution plans:

- **Baseline plan**: We follow the commonly-used heuristics [30, 39], which applies tensor parallelism within the same

Table 2: APEX simulation results: End-to-end latency (seconds). APEX demonstrates its ability to identify optimal parallel execution plans, achieving superior serving performance compared to heuristic baseline plans.

Traces	Model	Arrival Rate	Baseline	Feasible Optimal	APEX Optimal
Summarization	Qwen-2.5-32B	0.25	2153.27 (1×)	1137.70 (1.89×)	968.26 (2.22×)
		0.5	1823.85 (1×)	1116.84 (1.63×)	967.23 (1.89×)
	Llama-3.1-70B	0.25	1998.82 (1×)	1340.99 (1.49×)	1175.09 (1.70×)
		0.5	1565.11 (1×)	1225.38 (1.28×)	1140.19 (1.37×)
	Mistral-Large	0.25	1301.27 (1×)	498.32 (2.61×)	376.13 (3.46×)
		0.5	1140.36 (1×)	498.32 (2.29×)	376.13 (3.03×)
	Mixtral-8x22B	0.25	1841.90 (1×)	1290.97 (1.43×)	551.94 (3.34×)
		0.5	1429.46 (1×)	1137.87 (1.26×)	552.28 (2.59×)
Creation	Qwen-2.5-32B	0.25	5718.22 (1×)	2432.1 (2.35×)	1507.68 (3.79×)
		0.5	4950.96 (1×)	2421.82 (2.04×)	1512.95 (3.27×)
	Llama-3.1-70B	0.25	6549.85 (1×)	3742.77 (1.75×)	3159.96 (2.07×)
		0.5	5240.78 (1×)	4578.49 (1.14×)	3183.67 (1.65×)
	Mistral-Large	0.25	6379.09 (1×)	4063.11 (1.57×)	2684.37 (2.38×)
		0.5	5221.52 (1×)	4121.53 (1.27×)	2682.05 (1.95×)
	Mixtral-8x22B	0.25	2645.24 (1×)	2423.82 (1.09×)	1039.87 (2.54×)
		0.5	1442.15 (1×)	1417.75 (1.02×)	1035.62 (1.39×)
Chat	Qwen-2.5-32B	0.25	1251.75 (1×)	798.75 (1.57×)	513.38 (2.44×)
		0.5	1115.01 (1×)	786.08 (1.42×)	510.72 (2.18×)
	Llama-3.1-70B	0.25	1622.48 (1×)	1118.85 (1.45×)	824.43 (1.97×)
		0.5	1344.42 (1×)	1021.09 (1.32×)	808.96 (1.66×)
	Mistral-Large	0.25	973.16 (1×)	396.32 (2.46×)	254.26 (3.83×)
		0.5	890.71 (1×)	391.96 (2.27×)	254.26 (3.50×)
	Mixtral-8x22B	0.25	1757.68 (1×)	1504.88 (1.17×)	397.93 (4.42×)
		0.5	1406.75 (1×)	1253.22 (1.12×)	396.80 (3.55×)

node and pipeline parallelism across nodes. Since we only run on a one-node cluster, we apply tensor parallelism as the baseline plan.

- **APEX Optimal plan**: APEX identifies an optimal plan for each task. However, APEX's search space extends beyond the capabilities of current LLM serving systems, including advanced features like cell-level data parallelism. As a result, an execution plan may not be fully supported by existing LLM serving systems.
- **Feasible Optimal plan**: This is the optimal plan identified by APEX under the constraint that only parallelism techniques supported by current LLM serving systems are used. Due to the reduced search space, the feasible optimal plan may achieve lower performance compared to the unconstrained APEX Optimal plan.

While the APEX Optimal plan may not be realizable in current LLM serving systems, we include the results to showcase the potential performance gains that can be achieved by supporting a broader range of parallelism techniques.

We show the experimental results in Table 2. We experiment with the three request traces and four LLMs mentioned in Section 4.1. We also conduct experiments using two different arrival rates for the requests, assuming a Poisson distribution. To explore different quantization formats, we quantize the Mistral-Large model using FP8 for the KV cache, as well as the weights and activations (i.e., W8A8). APEX identifies optimal parallel execution plans for both dense and sparse LLMs (i.e, MoE models), and the Feasible Optimal plans consistently deliver performance improvement over the baseline, achieving up to 2.61× speedup in terms of end-to-end serving latency. Furthermore, assuming cell-level data parallelism is available, the APEX Optimal Plans can achieve a speedup of up to 4.42×. Cell-level data parallelism is particularly effective when the execution time varies among cells (e.g., MHA and MLP cells). This approach allows a specific cell to be parallelized further using data parallelism without requiring replication of the entire model. In this experiment, we demonstrate APEX is able to identify an optimal parallel execution plan that improves LLM serving performance under various LLMs and request traces.

APEX identifies an optimal parallel execution plan for each task. The optimal plans consist of various combinations of data, pipeline, and tensor parallelism, effectively balancing the trade-offs of various factors (e.g., compute, memory, network traffic) to outperform the baseline plan, which only relies on tensor parallelism. While the optimal plans are different from task to task, we observe that incorporating data parallelism (DP) often yields performance benefits. Many of the identified optimal plans set the degree of DP to 2 or even 4. Existing LLM serving systems often overlook DP, assuming it is prohibitively memory-intensive. Instead, our results demonstrate that trading memory efficiency for reduced communication overhead can lead to performance improvements. We also observe that the Feasible Optimal plan of Mixtral-8x22B shows marginal improvement over the baseline on the Creation dataset. This is due to the dominating role of memory utilization in this setup, leaving fewer trade-offs available for performance improvement. Specifically, Mixtral-8x22B, with its 141B parameters, heavily utilizes memory resources, while the Creation dataset's

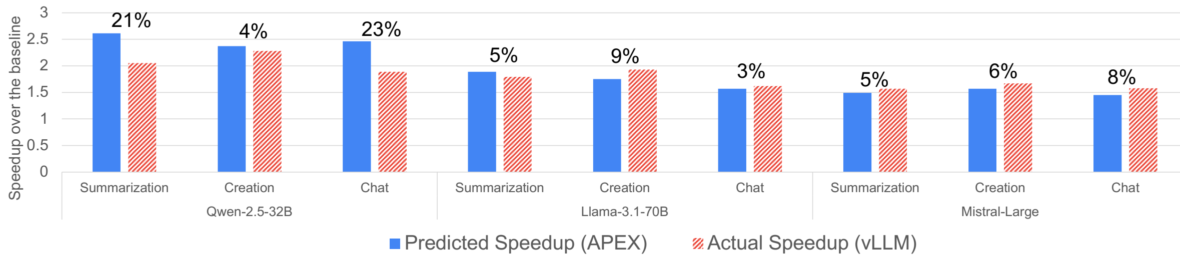

Figure 5: We compare the speedup predicted by APEX against the actual speedup achieved using vLLM. APEX accurately predicts the speedup of adopting an optimal execution plan over the baseline plan.

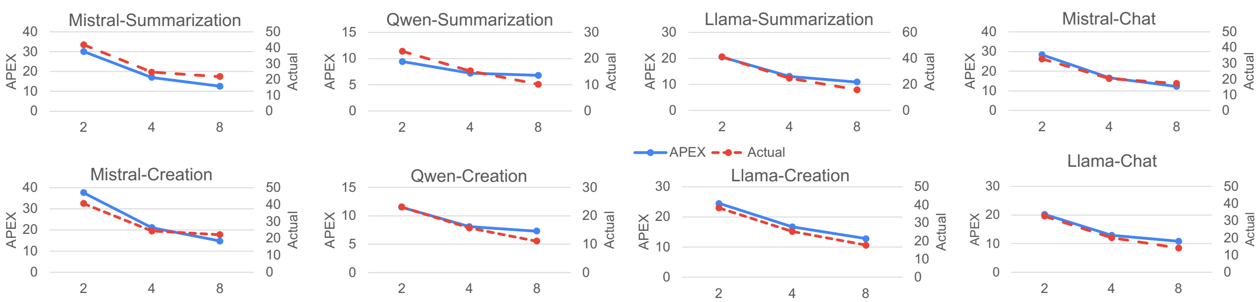

Figure 6: APEX accurately predicts the scalability trend across varying numbers of GPUs, as measured by TPOT (ms)

generation-intensive nature further increases memory demands due to extensive KV caching. Under these conditions, the baseline plan, which adopts pure tensor parallelism, performs well owing to its high memory efficiency. Nevertheless, the APEX Optimal plan still manages to deliver a 2.54× speedup by enabling cell-level data parallelism (DP), which is more memory-efficient than replicating the entire model.

4.3 Evaluation of Simulation Fidelity

In this experiment, we address the following research question: **Can APEX provide high-fidelity simulation for LLM serving?** While we demonstrate in Table 2 that APEX can identify optimal parallel execution plans that enhance LLM serving performance, it is also essential to verify the accuracy of its simulation results. We evaluate the simulation fidelity from two perspectives. First, under a fixed number of devices, we assess whether APEX can accurately predict the performance improvement of an optimal plan compared to a baseline plan. Second, for a given execution plan, we evaluate whether APEX can reliably predict performance improvements when scaling across different numbers of devices. We show the experimental results in Figure 5 and Figure 6. We set the arrival rate to 0.5 for both experiments as an example; setting the arrival rate of 0.25 yields similar results.

For the first experiment set, we compare the predicted speedup of Feasible Optimal plans with the actual speedup achieved on the device cluster (Figure 5). We do not evaluate Mixtral-8x22B as vLLM currently does not support expert parallelism. APEX achieves accurate predictions, with an average relative error of 9.5%. The largest discrepancies occur with the Qwen-2.5-32B model, the smallest model in our experiment, where the relative error reaches 23%. This is because, for smaller models, the overhead of operations such as RMSNorm becomes relatively significant, and APEX does not account for these operations in its simulations. In contrast, for larger models like Llama-3.1-70B and Mistral-Large (123B), the predictions are more accurate, as the execution time is dominated by attention and feedforward cells, which are well-modeled by APEX. The fidelity for smaller LLMs could be improved by incorporating profiling results for additional operations. However, as APEX primarily focuses on optimizing performance for serving large models, which are more computationally intensive, we leave this improvement for future work. For the second experiment set, we evaluate the time per output token (TPOT) across different numbers of GPUs using tensor parallelism. We show the results in Figure 6. The APEX prediction results are shown as blue solid lines, while the actual results are represented by red dotted lines. The high similarity between the two lines across all cases demonstrates that APEX accurately captures the scalability trend when scaling from 2 GPUs to 8 GPUs across various models and request traces. The estimated TPOT (ms) of APEX is plotted on the left-hand Y-axis, and the actual TPOT (ms) is plotted on the right-hand Y-axis. The actual measured TPOT is consistently higher than the predicted value, as evident from the larger values on the right-hand Y-axis compared to the left-hand Y-axis. This is because APEX primarily focuses on the overhead of key operations, such as attention and feedforward, while omitting the overhead of other operations. Nevertheless, APEX accurately captures the relative performance differences between various parallel execution plans and degrees of parallelism, as demonstrated

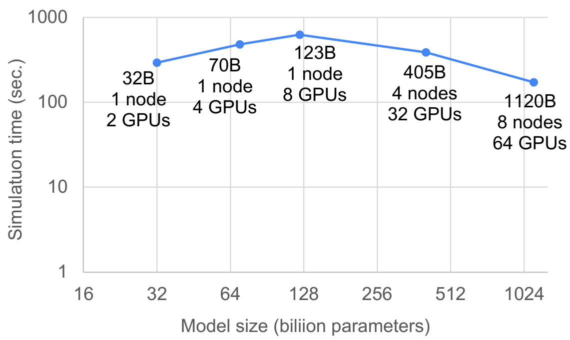

Figure 7: APEX maintains similar simulation overhead when scaling from billion-scale to trillion-scale models

in Figure 5 and 6. This capability enables APEX to effectively evaluate and compare different execution plans, thereby identifying an optimal parallel execution plan for LLM serving.

4.4 Evaluation of Efficiency

We evaluate the efficiency of APEX by comparing the actual execution time on the device cluster with the APEX simulation time. Evaluating all parallel execution plans across the setups in Table 2 (i.e., different models, traces, arrival rates) takes approximately 160 hours on 8 H100 GPUs. In contrast, the same evaluation is completed in less than 2.5 hours using APEX on CPU, making it 71× faster than the actual implementation. From a cost perspective, running the actual implementation would cost approximately $8,889 (based on the Azure NC40ads H100 cluster pricing), whereas the APEX simulation costs only $7.20 (assuming an Azure D64s v6 node). This translates to a 1234.5× cost reduction, demonstrating the significant efficiency and cost-effectiveness of APEX. Obtaining the operation-level profiling results to set up APEX takes approximately 40 GPU hours. Yet, this is a one-time cost that can be amortized for the same type of hardware device.

We also evaluate the simulation overhead when scaling to larger LLMs on larger device clusters. We use the following example models: Qwen2.5-32B, Llama-3.1-70B, Mistral-Large (123B), and Llama-3.1-405B. We synthesize a trillion-scale model by scaling the Llama-3.1-70B model 16 times; this can be done by modifying values in the LLM configuration file. As shown in Figure 7, APEX demonstrates high scalability, maintaining a similar simulation overhead when scaling from a 32B model to a trillion-scale model. This efficiency is achieved by leveraging the canonical representation in Transformer IR (Section 3.3) and the repetitive structure of the transformer architecture (Section 3.7) to greatly reduce the design space. This scalability highlights APEX's potential for future applications, as model sizes continue to grow and the cost of actual deployment may become even more expensive.

4.5 Evaluation of Extensibility

LLMs and their serving systems are evolving rapidly. While APEX supports a wide range of models, devices, etc., the simulator may still become outdated as new advancements emerge. To address this, APEX is designed for extensibility, enabling easy adaptation to support the latest developments. We evaluate APEX's extensibility by measuring the overheads to implement a new feature. We use two metrics: (1) programming overhead, measured in lines of code required for a feature extension, and (2) implementation time overhead, measured in hours, including time to write and execute the necessary code or scripts. We reported the overhead of various types of extensions in Table 3. Below, we provide details of the extensions implemented for the evaluation.

Table 3: Evaluating the overhead of extending APEX

Extension Type	Programming Overhead (Lines of Code)	Implementation Time Overhead (Hour)
LLM	0	~0
LLM (w/ unknown cells)	50 - 150	1-2
Device cluster	~20	4-8
Batching Mechansim	~100	1-2
Parallelism	50 - 200	1-2

Extending to New Models APEX can effortlessly support a new LLM without additional programming or implementation time by requiring only a configuration file of the model. However, for LLMs containing unknown transformer cells (Section 3.3), such as a novel feedforward network, additional effort is needed. The primary work involves implementing the Parallel Template (Section 3.4) for the new cells. For evaluation, we measured the effort to support SwiGLU cells [36] used in the Llama and Qwen models.

Extending to New Device Cluster APEX can easily adapt to new device clusters by providing the device name, memory capacity, and interconnection network (e.g., NVLink, PCIe). We evaluated the effort required to support a new type of GPU and an in-house AI accelerator. The implementation time overhead mainly involves running profiling scripts, which take 4 to 8 hours depending on the hardware's performance. While extending to new devices requires a relatively long script execution time, this effort is only necessary when a new device is released.

Extending to New Batching Mechanism APEX can also support new batching mechanisms. While this is not as straightforward as supporting a new model or a new device cluster, it can be done by modifying the Iteration-Level Batching Module 3.6. APEX adopts a vLLM-style [21] batching by default. As an example, we evaluate the overhead of extending to support a Sarathi-Serve-style [2] batching, which performs *chunk prefilling* to better interleave prefill and decode requests. This extension is done by adding a new variable, *chunk size*, to the batching module and a counter to each request to ensure all prefill chunks are completed before moving to the generation stage.

Extending to New Parallelism The overhead of extending APEX to support new parallelism types depends on the specific parallelism being added. For example, supporting a new type of intra-layer parallelism is analogous to supporting a new transformer cell, as it primarily requires adding a new Parallel Template. Similarly, Fully Sharded Data Parallelism (FSDP) can be integrated into APEX by extending the existing Data Parallelism implementation, incorporating additional collective communication steps, and adjusting memory usage to account for sharded model storage.

In summary, APEX is designed to support a wide range of extensions with minimal programming and implementation overhead, enabling it to stay up-to-date with advancements in LLMs, hardware, and serving systems as they continue to evolve.

4.6 Beyond Identifying Optimal Execution Plan

While APEX is primarily developed to identify optimal parallel execution plans for LLM serving systems, its high simulation fidelity makes it applicable to other use cases as well. Below, we present two examples of applications.

4.6.1 APEX provides insights for building LLM serving platform. When a new hardware device is released, service providers can use APEX to estimate the expected performance improvements from adopting the new hardware by adjusting the hardware parameters of device cluster in APEX. For example, the performance boost of upgrading from Nvidia A100 to H100 GPUs can be projected by scaling the compute time and data transfer time according to their relative compute power and memory bandwidth. Using this approach, APEX estimates that upgrading from A100 to H100 results in a 1.79× improvement in TTFT and a 1.66× speedup in TPOT; this estimation closely aligns with actual results reported in [33], which shows a 1.85× improvement in TTFT and a 1.43× speedup in TPOT. APEX also allows users to freely scale hardware parameters (i.e., creating synthetic hardware) to estimate the hardware upgrade required to achieve a specific performance boost in LLM serving.

4.6.2 APEX provides insights for meeting SLOs. In addition to minimizing end-to-end serving latency, service providers must also meet various service-level objectives (SLOs), such as maintaining a time per output token (TPOT). APEX can assist in achieving these SLOs. A common strategy to meet latency requirements is to adjust the batch size by setting a maximum batch size constraint. APEX begins by simulating a subset of requests to determine an upper bound for the batch size, denoted as m. It then divides this upper bound into n segments and simulates the request traces with various maximum batch size constraints, ranging from $\frac{1\times m}{n}$ to $\frac{n\times m}{n}$, where n is determined heuristically. Figure 8 illustrates two examples using the Llama-3.1-70B and Mistral-Large models on the Creation dataset. For simplicity, only results for pure tensor parallelism are shown. However, similar evaluations are performed across all parallel execution plans, resulting in numerous design points to choose from. Assuming a service provider aims to decrease the TPOT to meet the SLO, they can use the results to estimate the necessary adjustment to the maximum batch size constraint. For instance, reducing the maximum batch size constraint from 16 to 8 results in an 18% TPOT improvement for the Llama model and a 14% improvement for the Mistral model. However, overly restricting the maximum batch size can negatively impact end-to-end latency, as illustrated in Figure 8, where the batch size constraint is set to 4.

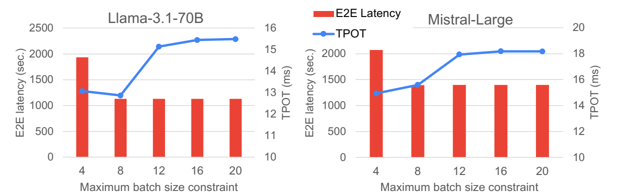

Figure 8: APEX can simulate with various maximum batch size constraints and suggest adjustments to meet SLOs.

5 Related Work

LLM Serving Systems: With the emergence of LLMs, numerous LLM serving systems have been proposed [2, 21, 32, 33, 41, 47]. Each system introduces innovations to address key challenges in LLM serving. Orca [41] introduced iteration-level batching, significantly improving serving throughput. This technique has since become a standard in LLM serving systems. vLLM [21] proposed PagedAttention, an efficient method to manage KV cache memory usage, enabling more concurrent request batching. Sarathi-Serve [2] addressed prefill-decoding interference by proposing *chunked prefill*, a technique that breaks the prefill stage into smaller stages to enhance batching efficiency; DistServe [47] tackled the same issue by disaggregating the prefill and decoding stages, processing each on separate device clusters optimized for their specific stages. Splitwise [33] also disaggregated the two stages, allocating requests to distinct device clusters to achieve higher throughput and cost-effectiveness. While these systems allow users to manually specify parallel execution plans (e.g., degrees of tensor and pipeline parallelism), they do not provide guidance on determining the optimal parallel execution plan. APEX complements these works. Users can utilize APEX to simulate and derive an optimal parallel execution plan tailored to their LLM serving setup, which can then be used to configure the serving system for improved performance.

Partition-Strategy-Search Tools: Several tools have been proposed to identify an optimal configuration for LLM training. Calculon [17] proposes a performance model that helps developers determine an optimal parallel execution plan for LLM training. vTrain [4] and ASTRA-sim [35] are simulators that guide users to find a system configuration that optimizes the LLM training time or the training cost. Nevertheless, these works cannot be applied to LLM serving due to the dynamism of iteration-level batching. LLMServingSim [6] and Vidur [1] are the few simulation frameworks designed for LLM serving systems. Both perform fine-grained simulations that account for the dynamism of iteration-level batching. However, LLMServingSim primarily focuses on NPUs and Processing-in-Memory (PIM) architectures, as it relies on their corresponding hardware simulators to estimate execution times. In contrast, APEX utilizes operation-level profiling data to estimate execution time and does not rely on other hardware simulators. Vidur requires a model onboarding step before simulation, which involves parsing the operations within the target LLM and profiling them. APEX bypasses this requirement by capturing the key operations of LLMs and utilizing pre-collected profiling results, allowing simulations to start immediately with amortized profiling costs. More importantly, the capabilities of both works are insufficient for simulating state-of-the-art LLM serving systems. Specifically, they do not support quantization techniques, their parallelism support is limited to pipeline and tensor parallelism, and they are constrained to traditional dense LLMs, lacking provisions for emerging architectures such as Mixture of Experts (MoE) models. While it is theoretically possible to extend these simulators to accommodate additional models and parallelism strategies, the required effort remains uncertain due to the lack of evaluations on their extensibility.

6 Conlusion

In this work, we developed APEX, an extensible and dynamism-aware simulator for LLM serving. We evaluated APEX across various LLMs and distinct workloads, demonstrating its high-fidelity simulation with less than 10% relative error on average. APEX successfully identified optimal parallel execution plans, achieving up to 4.42× speedup compared to heuristic plans. The simulation proved highly efficient, providing a 71× speedup and 1234× greater cost-effectiveness than deployment on actual hardware. APEX also showcased high scalability, maintaining a similar simulation overhead when scaling from billion-scale to trillion-scale models. Additionally, we demonstrated APEX's ease of extensibility to accommodate new models, device clusters, and more. In the future, we plan to extend APEX to support multimodal LLMs, which will involve incorporating parallel execution plans for encoders handling modalities such as vision and audio.

References

[1] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. VIDUR: A LARGE-SCALE SIMULATION FRAMEWORK FOR LLM INFERENCE. In *Proceedings of Machine Learning and Systems*.

[2] 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* (2024).

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv:2305.13245 [cs.CL] https://arxiv.org/abs/2305.13245

[4] Jehyeon Bang, Yujeong Choi, Myeongwoo Kim, Yongdeok Kim, and Minsoo Rhu. 2024. vTrain: A Simulation Framework for Evaluating Cost-effective and Compute-optimal Large Language Model Training. In *57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[5] 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. arXiv:2005.14165 [cs.CL] https://arxiv.org/abs/2005.14165

[6] Jaehong Cho, Minsu Kim, Hyunmin Choi, Guseul Heo, and Jongse Park. 2024. LLMServingSim: A HW/SW Co-Simulation Infrastructure for LLM Inference Serving at Scale. arXiv:2408.05499 [cs.DC]

[7] Jack Choquette. 2023. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro* (2023).

[8] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. 2018. A Discourse-Aware Attention Model for Abstractive Summarization of Long Documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*.

[9] Jiangfei Duan, Xiuhong Li, Ping Xu, Xingcheng Zhang, Shengen Yan, Yun Liang, and Dahua Lin. 2024. Proteus: Simulating the Performance of Distributed DNN Training . *IEEE Transactions on Parallel & Distributed Systems* (2024).

[10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco Guzmán, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shiva Shankar, Shuqiang Zhang,

Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vítor Albiero, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[11] Alexander Fabbri, Irene Li, Tianwei She, Suyi Li, and Dragomir Radev. 2019. Multi-News: A Large-Scale Multi-Document Summarization Dataset and Abstractive Hierarchical Model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

[12] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation. *IEEE Transactions on Software Engineering* (2024).

[13] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079* (2024).

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*.

[15] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL] https://arxiv.org/abs/2409.12186

[16] Intel. 2024. Intel Gaudi 3 AI Accelerator. In *Technical White Paper*.

[17] Mikhail Isaev, Nic Mcdonald, Larry Dennison, and Richard Vuduc. 2023. Calculon: a methodology and tool for high-level co-design of systems and large language models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[18] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825

[19] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] https://arxiv.org/abs/2401.04088

[20] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*.

[21] 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*.

[22] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* (2020).

[23] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. 2024. LLM Inference Serving: Survey of Recent Advances and Opportunities. arXiv:2407.12391 [cs.DC]

[24] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow.* (2020).

[25] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. *Proceedings of Machine Learning and Systems* (2024).

[26] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. arXiv:2310.01889 [cs.CL] https://arxiv.org/abs/2310.01889

[27] Peini Liu and Jordi Guitart. 2022. Performance characterization of containerization for HPC workloads on InfiniBand clusters: an empirical study. *Cluster Computing* (2022).

[28] Mary M Lucas, Justin Yang, Jon K Pomeroy, and Christopher C Yang. 2024. Reasoning with large language models for medical question answering. *Journal of the American Medical Informatics Association* (2024).

[29] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. 2023. Towards Efficient Generative Large Language Model Serving: A Survey from Algorithms to Systems. arXiv:2312.15234 [cs.LG] https://arxiv.org/abs/2312.15234

[30] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*.

[31] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*.

[32] NVIDIA. 2024. TensorRT-LLM [Online]. https://nvidia.github.io/TensorRT-LLM/overview.html. Accessed: 2024-11-13.

[33] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Inigo 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)*.

[34] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. In *Proceedings of the 39th International Conference on Machine Learning*.

[35] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2020. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 81–92. https://doi.org/10.1109/ISPASS48437.2020.00018

[36] Noam Shazeer. 2020. GLU Variants Improve Transformer. arXiv:2002.05202 [cs.LG] https://arxiv.org/abs/2002.05202

[37] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL]

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*.

[39] vLLM. v0.6.0. Distributed Inference and Serving [Online]. https://docs.vllm.ai/en/v0.6.0/serving/distributed_serving.html. Accessed: 2024-11-12.

[40] Qi Wang, Jindong Li, Shiqi Wang, Qianli Xing, Runliang Niu, He Kong, Rui Li, Guodong Long, Yi Chang, and Chengqi Zhang. 2024. Towards Next-Generation LLM-based Recommender Systems: A Survey and Beyond. arXiv:2410.19744 [cs.IR]

[41] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*.

[42] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.

[43] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. 2020. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[44] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. arXiv:2309.11998 [cs.CL]

[45] Li Zhong and Zilong Wang. 2024. Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence* 38 (2024).

[46] Ruizhe Zhong, Xingbo Du, Shixiong Kai, Zhentao Tang, Siyuan Xu, Hui-Ling Zhen, Jianye Hao, Qiang Xu, Mingxuan Yuan, and Junchi Yan. 2023. LLM4EDA: Emerging Progress in Large Language Models for Electronic Design Automation. arXiv:2401.12224 [cs.AR]

[47] 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. *Proceedings of 18th USENIX Symposium on Operating Systems Design and Implementation, 2024, Santa Clara* (2024).

[48] Hao Zhou, Chengming Hu, Ye Yuan, Yufei Cui, Yili Jin, Can Chen, Haolun Wu, Dun Yuan, Li Jiang, Di Wu, Xue Liu, Charlie Zhang, Xianbin Wang, and Jiangchuan Liu. 2024. Large Language Model (LLM) for Telecommunications: A Comprehensive Survey on Principles, Key Techniques, and Opportunities. arXiv:2405.10825 [eess.SY]

[49] Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. ToolQA: A Dataset for LLM Question Answering with External Tools. In *Advances in Neural Information Processing Systems*.