MaverIQ: Fingerprint-Guided Extrapolation and Fragmentation-Aware Layering for Intent-Based LLM Serving

Dimitrios Liakopoulos
The University of Texas at Austin
Austin, Texas, USA
dimliak@utexas.edu

Prasoon Sinha
The University of Texas at Austin
Austin, Texas, USA
prasoon.sinha@utexas.edu

Tianrui Hu
The University of Texas at Austin
Austin, Texas, USA
tianrui@utexas.edu

Myungjin Lee
Cisco Systems
Bellevue, Washington, USA
myungjle@cisco.com

Neeraja J. Yadwadkar
The University of Texas at Austin
Austin, Texas, USA
neeraja@austin.utexas.edu

Abstract

Large Language Models (LLMs) are becoming ubiquitous across industries, where applications demand they fulfill diverse user intents. However, developers currently face the challenge of manually exploring numerous deployment configurations—combinations of parallelism and compression techniques that impact resource usage, latency, cost, and accuracy—to meet these intents. Previous works automate configuration selection and deployment, but, they (a) rely on expensive profiling, and (b) suboptimally utilize the fragmented resource availability in multi-tenant GPU clusters, inflating operational costs for the provider. Moreover, none of these solutions tailors deployment configuration decisions to diverse user intents.

We present MaverIQ, an LLM inference serving system that exposes an intuitive interface allowing users to express their intent (e.g., minimize latency, meet cost target). MaverIQ automatically translates the intent into the best configuration and deploys it on behalf of the user, while minimizing the operational cost for the provider. To reduce profiling costs, MaverIQ introduces and observes a compact proxy of the LLM, called *fingerprint*, under a few configurations, and uses novel analytical models to extrapolate the observed fingerprint data to the full LLM. To optimize the cost for the provider, we leverage our key observation that, unlike training, distributing LLM layers across GPUs in an uneven manner has little impact on the overall inference latency. Our novel deployment algorithm exploits this insight to enable MaverIQ to utilize the fragmented cluster resources. Through rigorous empirical evaluation, we show that MaverIQ reduces profiling cost by 7-15× compared to state-of-the-art baselines. Lastly, across various LLMs, traces, and loads, MaverIQ best meets user intents while reducing operational cost for the provider by 3.8-8.3×. Our code is available at https://github.com/UT-SysML/MaverIQ.

This work is licensed under a Creative Commons Attribution 4.0 International License.
SC '25, St Louis, MO, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1466-5/2025/11
https://doi.org/10.1145/3712285.3759867

CCS Concepts

• **Computer systems organization**; • **Networks → Cloud computing**; • **Computing methodologies → Neural networks**; • **Mathematics of computing → Mathematical analysis**;

Keywords

Large Language Model Inference, Cloud Computing, Fingerprints

ACM Reference Format:
Dimitrios Liakopoulos, Prasoon Sinha, Tianrui Hu, Myungjin Lee, and Neeraja J. Yadwadkar. 2025. MaverIQ: Fingerprint-Guided Extrapolation and Fragmentation-Aware Layering for Intent-Based LLM Serving. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3712285.3759867

1 Introduction

Large Language Models (LLMs) are set to become ubiquitous across industries where applications demand that they meet varied user intents. For instance, real-time chatbots and voice assistants prioritize low latency to deliver instant responses while education platforms and automated FAQs need cost efficiency. Meanwhile, mobile apps and IoT devices afford minimal GPU usage. To satisfy these diverse intents, experts introduced various (a) compression techniques that reduce the size of the models (e.g., quantization [11, 12, 20, 47, 102, 108], pruning [19, 54, 86]) and (b) parallelism techniques that distribute the models across multiple GPUs (e.g., data [77], pipeline [31], and tensor parallelism [95]). The hundreds of resulting deployment configurations — combinations of these techniques — for each LLM vary across many dimensions. For example, Llama-2-70B can be deployed in 180 ways on an 8-GPU cluster: 15 parallelization options (4 tensor, 8 pipeline) combined with 4 weight quantization methods and 2 KV-cache quantizations, or 4 pruning strategies. To generate 100 tokens, inference latency varies by 8× (2.7-21.5 seconds), memory consumption by 4.6× (33.7-153.9 GB), and GPU time by 25× (6.9-171.5 GPU-seconds). Finally, the user cost for serving queries, based on the Azure LLM serving trace [67], varies by 25.4× ($10.4-$264). Thus, *the choice of a deployment configuration for an LLM is key in meeting varied user intents.*

Existing work falls into three main categories. (a) The first category consists of systems that ignore user intents and fix the configuration across different models [15, 64, 114]. These result in

suboptimal choices as the best deployment configuration is LLM-specific, intent-specific, and depends on the availability of resources in the GPU cluster (see § 3). (b) The second category consists of systems that leave it to users to specify the deployment configuration [5, 6, 40, 48, 61, 65]. However, this requires expertise in LLM architectures and hardware accelerators, which many users lack. Moreover, the search space of configurations is large and complex, making it non-trivial for users to make this choice. (c) The third category consists of systems that use profiling techniques to map user-intents to deployment configurations [1, 42, 46, 57, 64, 71, 84]. However, these systems are inefficient, as they (1) profile the full LLM [1, 46], inflating profiling resource consumption, and (2) profile the LLM under all deployment configurations [46], inflating profiling time. For example, it took us ~ 2.5 hours and 8 GPUs to profile a subset (100) of Llama2-70B's configurations. To select a deployment configuration tailored to users' intent, we need new techniques that *accurately estimate the implications of all deployment configurations for a given LLM, without expensive profiling.*

The chosen configuration now needs to be deployed in a way that optimizes the operational cost for the provider. This is challenging in a multi-tenant environment given the varying load and fragmented resource availability. Previous works cannot optimally utilize the fragmented resources, as they evenly distribute LLMs across GPUs [2, 5, 24, 40, 46, 61]. To reduce operational cost for the provider, we must develop new deployment techniques that *utilize the fragments* of the GPU cluster while still meeting user intents.

Our work. To this end, we build MaverIQ, an automated intent-based LLM inference serving system that first translates user-expressed intents into LLM deployment configurations and deploys the chosen configurations to improve cluster throughput and reduce operational cost for the provider.

To translate the user's intent to the best configuration, we develop new profiling techniques that heavily reduce the profiling costs seen in previous works. First, to reduce profiling resource demand, we make a key observation about commonly used LLMs: a majority of the layers within an LLM are repeated, leaving only a few unique layers. So, instead of profiling the full LLM, we profile only a compact proxy of the LLM containing a few unique layers, which we introduce as LLM *fingerprint*. Second, to reduce profiling time, we develop novel analytical models to accurately estimate an LLM's latency/memory footprint under any deployment configuration from just a few observations. Combining both, *we show that observing the lightweight fingerprints under just a few configurations is enough to efficiently and accurately understand the implications of all deployment configurations for the full LLMs.*

To efficiently utilize the fragmented resources in the GPU cluster, we make a key observation that goes against the commonly held assumption that LLM layers need to be evenly distributed across GPUs to avoid delays due to synchronization. We note that while this assumption is true for training that requires a forward and a backward pass, it does not hold for inference as inference only needs a forward pass. Thus, *distributing LLM layers across GPUs in an uneven manner has little impact on the overall inference latency and can be leveraged to efficiently utilize fragmented resources* in a multi-tenant cluster. MaverIQ's novel deployment algorithm leverages this insight to reduce operational costs for the provider.

We summarize our contribution as follows:

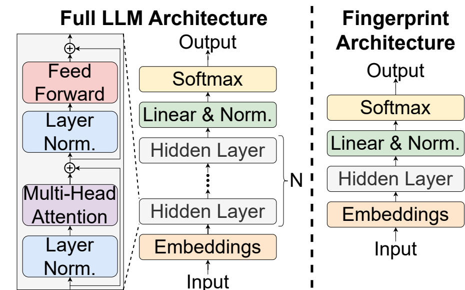

Figure 1: Architecture of a decoder-only LLM (left) and its fingerprint (right). The fingerprint has the same components as the full LLM, but only 1-2 hidden layers. See § 2.

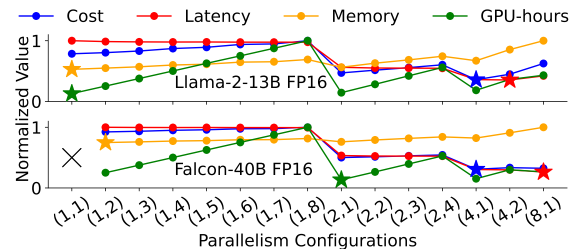

Figure 2: The best parallelism configuration (d_{TP}, d_{PP}) is user-intent- and LLM-specific. d_{TP}, d_{PP}: degrees of tensor and pipeline parallelism. See § 3.1.

- We introduce LLM fingerprints, a compact proxy that captures the essence of the model, enabling lightweight profiling.
- We are the first to formulate an analytical model that accurately captures the effects of parallelism techniques on inference latency for LLMs. Combining both fingerprints and analytical models, MaverIQ reduces profiling cost by 7-15× while reducing estimation error by 1.3-1.7×.
- We are the first to show that we can unevenly distribute LLM layers across GPUs to utilize fragmented resources without harming inference latency. This technique reduces operational cost by up to 2×.
- We build MaverIQ atop TensorRT-LLM and show that under strict accuracy requirements, MaverIQ reduces latency by 28-45% for the user while reducing operational cost by 3.8-8.3× for the provider. Under lower accuracy requirements, MaverIQ can exploit compression techniques to further reduce both user cost and latency by about 72%.

2 Background

Model architecture of LLMs. LLMs are based on transformers. The original transformers (e.g., Bart [44], Pegasus [111], T5 [73], UL2 [90]) used encoder-decoder architectures where the encoder created fixed-sized vectors for the decoder to generate outputs. Such encoder-decoder-based architecture was shown to be rigid, as only fixed-sized outputs could be generated. Instead, GPT-2 [72] introduced a more flexible decoder-only architecture that leverages previously generated tokens to generate outputs of arbitrary length. Most modern LLMs now follow this decoder-only structure, stacking hundreds of identical layers to extract features (Figure 1).

Deployment configuration for LLMs. To efficiently deploy modern LLMs, expert users leverage techniques from two categories:

parallelism and compression. Parallelism techniques deploy LLMs across multiple GPUs: (1) data parallelism (DP) [77] replicates LLMs across GPUs, (2) tensor parallelism (TP) [95] splits tensor operations across GPUs, and (3) pipeline parallelism (PP) [31] distributes layers across GPUs. Parallelism techniques can be combined (e.g., TP and PP), however their extent is limited by the available GPUs.

Compression (quantization and pruning) reduces an LLM's memory footprint. Quantization [11, 12, 20, 47, 102, 108] lowers the bit-width of LLM weights, activations [101], and KV-caches [14, 51]. Pruning removes redundant weights/layers of an LLM or enhances sparsity for acceleration [19, 54, 69, 86]. While both slightly affect output quality, they are widely used for LLM deployment [37, 54].

The combination of parallelism and compression techniques comprise an LLM's deployment configuration. Hence, throughout this paper, we refer to an LLM's deployment configuration as a selection of TP and PP degrees, the quantization applied to weights, activations, and the KV-cache, as well as the pruning strategy.

3 Motivation & Characterization

In this section, we study how the deployment configuration parameters impact four different user intents: minimize inference latency, GPU memory consumption, cost (memory × latency), and GPU-hours. We then describe the challenges in selecting the optimal deployment configuration for an LLM that can meet a user's intent. For this measurement study, we use an 8-GPU cluster with NVIDIA RTX A6000 (48GB) GPUs and six state-of-the-art LLMs: Falcon-7B, Falcon-40B, GPT-J-6B, Llama-2-7B, Llama-2-13B, and Llama-2-70B.

3.1 Effect of Parallelism

Figure 2 shows the cost, latency, total GPU memory, and GPU-hours as we vary the parallelism degrees for Llama-2-13B and Falcon-40B.

Impact of PP alone. Increasing PP degree (d_{PP}) from 1 to 8 ((1,1) to (1,8)) has minimal impact on inference latency (0.04-0.11 sec difference), as the time to propagate intermediate data layer-by-layer between high-bandwidth devices is negligible (< 0.05% the inference latency). Increasing d_{PP} increases the total memory usage (1.01-1.3×), but per-GPU memory decreases (1.5-6.1×). Thus, larger PP degrees may be useful in multi-tenant environments where many GPUs have smaller fragments of free memory.

Impact of TP alone. Increasing d_{TP} from 1 to 8 decreases latency while increasing memory consumption and GPU-hours: Falcon-40B's latency reduces by 1.8-2.1×, but memory and GPU-hours increase by 1.1-1.3× and 1.2-1.9×, respectively. For smaller LLMs like Llama-2-13B, the increased memory footprint and all-reduce synchronization overheads [80] of higher TP degrees (4 to 8) raise user cost by 1.8×, while Falcon-40B only experiences a 1.05× cost increase, making TP's impact on cost LLM-specific.

Impact of both TP & PP. Similar to our observations when analyzing TP and PP alone, the optimal combination of TP and PP is both LLM- and intent-specific: For example, while Llama-2-13B achieves the lowest latency with (4,2), Falcon-40B needs (8,1). However, these configurations do not minimize other intents: for both LLMs, cost is minimized with (4,1), while memory is lowest with (1,2) for Falcon-40B and (1,1) for Llama-2-13B.

Takeaway #1. Increasing PP inflates an LLM's total memory footprint but lowers per-GPU footprint. Increasing TP reduces inference latency, but excessive TP can harm latency due to synchronization overheads. *The best combination of TP/PP is LLM- and intent-specific.*

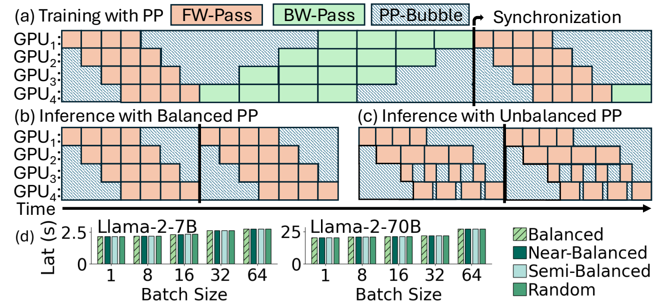

Figure 3: Unlike LLM training, the impact of unbalanced PP on latency is negligible in LLM inference. See § 3.2.

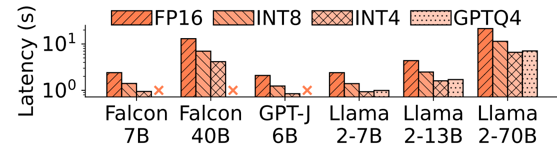

Figure 4: The impact of quantization on latency is LLM-specific. See § 3.3.

3.2 Effect of *Unbalanced* Pipeline Parallelism

LLM training consists of two passes: (1) the forward pass processes input tokens to generate outputs, and (2) the backward pass computes gradients and updates weights to minimize loss. Pipeline parallelism (PP) splits an LLM layer-wise across GPUs to enable microbatches of training data to be processed in a pipelined fashion. Although PP is meant to increase efficiency, it is rarely used in LLM training, especially with large batch sizes. PP is known to introduce GPU processing stalls [31, 58] since all GPUs must wait for the forward pass to process the microbatches before initiating backpropagation (Figure 3a). Unevenly placing layers of an LLM across GPUs (unbalanced PP) exacerbates stalls and degrades efficiency due to imbalanced microbatches. Hence, if PP is used during training, systems evenly distribute layers across GPUs to mitigate the undesired ramifications of unbalanced PP [49].

Inference only needs a forward pass (no backpropagation) [23]. We observe that the pipeline stalls in the inference phase are significantly reduced compared to training (Figures 3a, 3b). Thus, the impact of PP on inference latency is negligible (Figure 2). We also find that even if we distribute LLM layers across GPUs in an unbalanced manner, latency is not harmed regardless of the batch size (Figure 3d). Instead, unbalanced PP fluctuates the per-GPU memory consumption across an LLM's PP partitions: deploying Llama-2-7B with unbalanced d_{PP}=2, where one GPU hosts 10 layers and the other 22 layers, results in one partition using 4.53GB of memory while the other consumes 9.97GB. We leverage this insight to utilize GPUs with fragmented available memory, enabling MaverIQ to deploy more LLMs and increase cluster throughput (§ 8.5).

Takeaway #2. We can leverage unbalanced PP in LLM inference to utilize fragmented available GPU memory, deploy more LLMs, and increase cluster throughput without degrading inference latency.

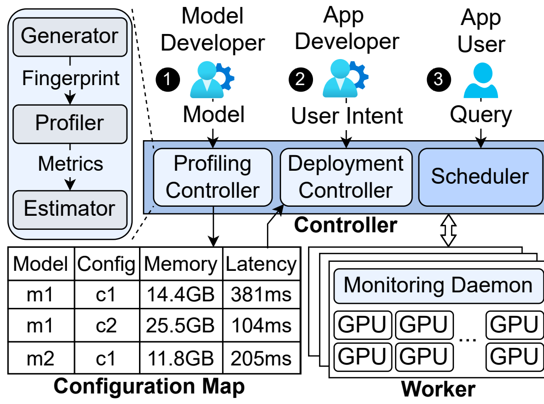

Figure 5: Architecture of `MaverIQ`. See § 4.

3.3 Effect of Quantization

Quantization predictably reduces memory consumption: across LLMs, INT8 cuts memory consumption by 50%, while INT4/GPTQ4 reduces it by 75%, compared to FP16. While quantization also reduces inference latency (the lower-precision data representations decrease GPU compute and memory bandwidth demand), its impact is LLM-specific (Figure 4). For example, quantizing with INT8 reduces Falcon-40B's latency by 1.9× but only 1.7× for Falcon-7B. Similarly, quantizing with GPTQ4 reduces LLama-2-7B's latency by 2.4×, compared to Llama-2-70B's 3× reduction. Lastly, while quantization reduces memory consumption and latency, it comes at the expense of LLM output quality [20, 37].

Takeaway #3. Quantization predictably reduces memory consumption across LLMs, however, its effect on latency is LLM-specific. Moreover, quantization presents a tradeoff between memory/latency reduction and output quality. Ultimately, there is no single optimal quantization technique: it is LLM- and intent-specific.

3.4 Effect of Pruning

We study the impact of pruning using two state-of-the-art techniques: (1) Wanda 2:4 [86], and LLM-Pruner [54] (we omit the figure for brevity). Both techniques reduce memory footprint, either by zeroing out weights (Wanda 2:4, 40% reduction in memory) or completely removing weights/layers (LLM-Pruner, 19-37% reduction). However, pruning increases latency (due to matrix conversion overheads or poor utilization of the hardware), albeit its effect is LLM-specific. For example, pruning increases Llama-2-7B's latency by 21-42%, while Llama-2-13B's latency only increases by 3-24%, as the gains from reducing computational load with pruned matrices outweigh the overheads. Finally, pruning degrades output quality [54, 86] with varying impacts across models.

Takeaway #4. Pruning techniques present a trade-off between accuracy, latency, and memory consumption, however, the effect of pruning on accuracy and inference latency is LLM-specific.

4 `MaverIQ` Design

Given a user intent, `MaverIQ` finds the best deployment configuration for the LLM without expensive profiling. `MaverIQ` then deploys the LLM in a cost-efficient manner by well utilizing fragmented resources in the GPU cluster. Figure 5 presents `MaverIQ`'s system architecture. We first describe `MaverIQ`'s intent-based interface, followed by its key components and workflow.

Table 1: `MaverIQ`'s APIs. See § 4.

API	Parameters
register_model	model, [compressionTechnique: accuracy]
deploy_model	model, maxOutputLength, intent <minCost, minLat, latSLO, costSLO, accSLO>
inference_model	model, inputText

`MaverIQ`'s intent-based interface. `MaverIQ` is the first intent-based LLM inference serving system. Table 1 outlines `MaverIQ`'s intent-based interface. Users register LLMs via *register_model*, specifying the LLM and a list of compression techniques it supports with their respective MMLU accuracy scores. The accuracy scores enable `MaverIQ` to meet accuracy needs, as it only considers configurations with scores above accuracy requirements; if none exist, we notify users and optimize secondary metrics (e.g., latency or cost). Users deploy an LLM via *deploy_model*, specifying the maximum output length and user intent. Notably, users are not required to specify any LLM configuration parameters or the number of GPUs to use. Instead, they simply specify their intent to minimize cost or latency (unburdening users from determining appropriate SLOs), or meet SLOs (latency, cost, and/or accuracy). `MaverIQ` defaults to minimizing cost if no intent is specified. As the cost formulation can vary from provider to provider, we evaluate `MaverIQ`'s efficacy with three cost formulations: (1) the traditional GPU memory × latency [59, 92], (2) GPU memory consumed as cost, and (3) GPU-hours consumed as cost. Finally, users can send inference requests with prompt inputs to a deployed LLM via *inference_model*.

Profiling Controller. `MaverIQ`'s Profiling Controller is responsible for accurately translating the user's intent into the best deployment configuration without expensive profiling. To do so, we efficiently profile LLMs with their lightweight *fingerprints* and accurately estimate each configuration's latency/memory using our novel analytical models (detailed in § 5).

Deployment Controller. `MaverIQ`'s Deployment Controller is responsible for automatically deploying the chosen configurations in a way that utilizes the fragmented resources in the cluster while still meeting the user intents. The Deployment Controller implements our novel deployment algorithm that leverages unbalanced LLM partitioning to achieve this goal (detailed in § 6).

Scheduler. Finally, the Scheduler routes inference requests to the Workers hosting the LLMs in the GPU cluster.

`MaverIQ` triggers each component based on the persona interacting with it: LLM developers, application developers (e.g., ChatGPT), and application users (e.g., ChatGPT users).

❶ **`MaverIQ`'s profiling workflow** is triggered when a *model developer* registers an LLM. Upon registration, the Profiling Controller takes charge: (a) First, the Profiling Controller's Generator creates the LLM's *fingerprint*, a compact proxy that encapsulates the LLM's unique model architecture in a smaller memory footprint (§ 5.1). (b) Its Profiler then deploys the generated fingerprint under a few deployment configurations and issues dummy requests to observe its latency/memory consumption. Observing an LLM's lightweight fingerprint greatly reduces the resource demand of `MaverIQ`'s profiling workflow. (c) Finally, its Estimator uses the profiled data and

Config	Per GPU Balanced Footprint	Total Memory
No Parallel	W | A	mem_{base} = (W+A)
TP	W/d_{TP} | A × d_{TP}	mem_{base} + (d_{TP}-1) × A
PP	W/d_{PP} | A/d_{PP} | A_P × d_{PP}	mem_{base} + (d_{PP}-1) × A_P
TP + PP	W/d_{TP}·d_{PP} | A/d_{PP} | A_P × d_{TP}·d_{PP}	mem_{base} + (d_{TP}-1) × A + + d_{TP} × (d_{PP}-1) × A_P

Figure 6: LLM parallelism memory overheads. Weights (W), activations (A), and pipeline activations (A_p) are replicated on d_{TP} (TP degree) and/or d_{PP} (PP degree) GPUs. See § 5.2.

our novel analytical models to estimate the LLM's latency and memory footprint for all deployment configurations (§ 5.2/ 5.3).

❷ **`MaverIQ`'s model-deployment workflow** is triggered when an *application developer* requests to deploy an LLM. They provide `MaverIQ` a user intent to optimize for. Using the configuration map with the estimated memory and latency for all configurations, `MaverIQ`'s Deployment Controller chooses the configuration that best meets the user intent and is feasible to deploy given resource availability. It uses (a) a load-aware policy to select which GPUs to deploy an LLM on, and (b) a fragmentation-aware algorithm for mapping layers of an LLM to best utilize the potentially fragmented memory resources in the selected GPUs. These two algorithms enable `MaverIQ` to efficiently utilize and reduce the operational costs of the GPU cluster. We detail this workflow in § 6.

❸ **`MaverIQ`'s inference-serving workflow** is triggered when *application users* submit inference requests. The Scheduler dispatches them to worker nodes. We use a simple FCFS scheduler, as our focus in this work is on efficient profiling and deployment for LLMs to meet user intents. We design `MaverIQ` in a modular fashion to enable future work on developing advanced scheduling algorithms.

5 Inexpensive yet Accurate Profiling

`MaverIQ`'s Profiling Controller inexpensively yet accurately estimates an LLM's memory footprint and latency under all deployment configurations. We introduce LLM fingerprints and develop novel analytical models to do so. In this section, we describe the need for inexpensive profiling and our fingerprint- and analytical-modeling-based profiling methodology, along with our intuition behind it.

5.1 LLM Fingerprints

Directly loading and profiling an LLM across all deployment configurations is prohibitively expensive; it requires tens to hundreds of GBs of GPU memory and can take hours to profile. We make a key observation: the main components that bloat an LLM's memory footprint are the hidden layers that are absolutely identical to each other (§ 2). So, instead of profiling the full LLM, we profile its compact proxy containing a few unique layers, which we introduce as LLM *fingerprints*. Fingerprints capture the key model architecture components: the embedding layer, hidden layers, normalization layer, and activation function (Figure 1). Unlike the LLM, the fingerprint has minimal hidden layers, reducing memory and time needed for profiling: Llama-2-70B's fingerprint is 41× smaller and 63× faster to profile. We find a linear relationship between the number of hidden layers and the LLM's memory footprint (figure omitted for brevity), enabling us to easily extrapolate the LLM's memory footprint from the fingerprint.

5.2 Estimating Memory

We detail how we profile and extrapolate the memory requirements from fingerprint to LLM for every deployment configuration.

Breaking down an LLM's memory footprint. Parallelism techniques replicate components across GPUs, thereby increasing an LLM's memory footprint. Figure 6 depicts the memory footprint under different parallelism strategies [52]. Three components define the footprint across parallelism schemes: weights (W), activations (A), and PP activations (A_p). The memory footprint without parallelism (mem_{base}) is simply the weights and activations ($W + A$). TP inflates this by $(d_{TP} - 1) \times A$, where d_{TP} is the TP degree, as each layer's activations are replicated on each GPU. PP replicates the intermediate PP activations, inflating the base footprint by $(d_{PP} - 1) \times A_P$, where d_{PP} is the PP degree. Finally, combining TP and PP incurs both overheads. *Knowing W, A, and A_p allows us to estimate the LLM's memory footprint for any configuration.*

Candidate memory profiling techniques. We use our memory footprint understanding to devise two candidate methods (M1 and M2) for estimating an LLM's memory footprint per configuration.

M1. LLM under selected configurations: To reduce profiling costs compared to brute force (observing all LLM configurations), we leverage our insight that knowing W, A, and A_p allows us to estimate any configuration's memory footprint. To estimate these values, we profile three configurations and solve a 3×3 system of equations (Figure 6). These values change with quantization. Thus we profile the LLM with three parallelism schemes per quantization: d_{TP}/d_{PP}=1, d_{TP}=1 d_{PP}=2, and d_{TP}=2 d_{PP}=1, which require the least resources. This method reduces profiling time by only profiling three configurations but still requires notable GPU memory.

M2. Fingerprint under selected configurations: To reduce profiling resource demands, we profile the LLM's fingerprint using M1's approach. However, solving the system of equations using the fingerprint's footprint yields W, A, and A_p specific to the fingerprint, not the LLM. Thus, we first linearly extrapolate the LLM's footprint from the fingerprints, using two fingerprints per configuration, under the three parallelism schemes. Then, we solve the equations with the extrapolated footprints to estimate W, A, and A_p.

5.3 Estimating LLM Inference Latency

Unlike memory, an LLM's inference latency scales with output length, which is unknown until deployment time. Therefore, during profiling, we capture key metadata (e.g., time to first token) that enables `MaverIQ` to trivially estimate latency during deployment time. We outline our insights and latency profiling strategies.

Breaking down an LLM's inference latency. LLMs generate outputs token by token using autoregressive decoding. Aside from the first output token, which an LLM generates by processing all input tokens, an output token depends on previously generated tokens. The KV-cache avoids recomputing preceding tokens, ensuring consistent latency for each new token generation. Thus, LLM inference latency is formally broken down into the time to first token (TTFT), time per output token (TPOT), and the output length:

$$L = TTFT + output_length \times TPOT \quad (1)$$

$TTFT/TPOT$ are *specific to an LLM's deployment configuration.* Hence, to estimate an LLM's latency under any configuration, we would

need to obtain the *TTFT* and *TPOT* for every configuration during profiling. However, we formulate novel analytical models that estimate the impact of parallelism (TP/PP) on inference latency. This enables us to estimate the latency for any configuration by determining the *TTFT*/*TPOT* for *only three configurations*.

Equation 2 presents our analytical model that captures how TP and PP affect inference latency. Increasing the TP degree distributes the computational workload across GPUs. Therefore, we divide the base latency without parallelism, L(1,1), by d_{TP} in the first term. However, TP introduces overhead (O_{TP}) with all-reduce synchronization operations, which we empirically observe is non-linear with respect to d_{TP}, captured in the second term. Using PP introduces its own overheads (O_{PP}) because of its send-recv operations, which we empirically observe scales non-linearly with respect to d_{PP}, captured in the third term. However, increasing PP reduces the per-GPU work by partitioning the LLM's layers across GPUs. This decreases the overall latency non-linearly by d_{PP}, captured in the first term. Moreover, PP reduces the amount of data that traverses the GPU interconnect, reducing the overhead of TP non-linearly by d_{PP}, captured in the second term. With this understanding, we can estimate inference latency as parallelism changes with:

$$L(d_{TP}, d_{PP}) = \frac{L(1,1)}{d_{TP} \times d_{PP}^{\alpha}} + \frac{(d_{TP}-1)^{\beta}}{d_{PP}^{\gamma}} \times O_{TP} + (d_{PP}-1)^{\delta} \times O_{PP} \quad (2)$$

where α, β, γ, and δ are constants that modulate the scaling and overhead effects of parallelism. To determine the constants, we collect a couple hundred data points (a one-time cost) across diverse LLMs and configurations and employ a Gradient Descent algorithm [76] with a Mean Absolute Error (MAE) loss function. MAE when applied to Equation 2 with respect to these constants is a non-convex function, precluding direct analytical solutions.

With fixed scaling constants applied across LLMs, the only remaining unknowns in Equation 2 are O_{TP} and O_{PP}. We can calculate these values using Equation 2 and the latency of three reference configurations: L(1,1), L(1,2), and L(2,1):

$$O_{TP} = L(2,1) - \frac{L(1,1)}{2} \quad \& \quad O_{PP} = L(1,2) - \frac{L(1,1)}{2^{\alpha}} \quad (3)$$

However, we cannot determine the latencies of these three reference configurations during profiling time, as they depend on the output length. Instead, our goal is to *efficiently determine the metadata (TTFT/TPOT) for the three reference configurations via profiling*. Then, during deployment time, we calculate L(1,1), L(1,2), and L(2,1) for a given output length using Equation 1, estimate the overheads O_{TP} and O_{PP} with Equation 3, and then estimate the latency under any configuration using Equation 2. We detail our two strategies that obtain this metadata for the three reference configurations next.

Candidate latency profiling techniques. We devise two profiling strategies for latency estimation.

L1. LLM under selected configurations. This method observes the latency for the three reference configurations (d_{TP}/d_{PP}=1, d_{TP}=1 d_{PP}=2, and d_{TP}=2 d_{PP}=1) to obtain their metadata. For each configuration, we measure the latency for two output lengths. We then solve a 2×2 system of equations with Equation 1 to estimate the unknowns $TTFT$ and $TPOT$. This method reduces profiling time but still requires significant GPU memory.

L2. Fingerprint under selected configurations. To reduce resource consumption, we profile the fingerprint (not the LLM) to determine $TTFT/TPOT$ for the three reference configurations. However, these values we measure are specific to the fingerprint, not the LLM with more hidden layers. Therefore, we break down $TTFT$ and $TPOT$ into the latency through the LLM's building blocks: the hidden layers and the other layers (§ 2). Formally,

$$TTFT = num_layers \times TTFT_{layer} + TTFT_{other} \quad (4)$$

$$TPOT = num_layers \times TPOT_{layer} + TPOT_{other} \quad (5)$$

where num_layers is the number of hidden layers in the LLM, $TTFT_{layer}$ is the latency for one hidden layer to produce the first token, $TPOT_{layer}$ is the latency for one hidden layer to produce an output token, and $TTFT_{other}$ and $TPOT_{other}$ are the latencies of the other basic building blocks for their respective token generation. With knowledge of these four variables in Equations 4 and 5, we can compute the LLM's $TTFT$ and $TPOT$ with num_layers hidden layers per configuration. Hence, in this method, we profile the LLM's fingerprints to efficiently obtain $TTFT_{layer}$, $TPOT_{layer}$, $TTFT_{other}$, and $TPOT_{other}$ per selected deployment configuration.

We observe two fingerprints per LLM, each with one and two hidden layers. We send each two requests with different output lengths. Then, for each fingerprint, we solve a 2×2 system of equations with Equation 1 to obtain its $TTFT/TPOT$. We use these values to solve a 4×4 system of equations with Equations 4/ 5 to estimate $TTFT_{layer}$, $TPOT_{layer}$, $TTFT_{other}$, and $TPOT_{other}$ per configuration. We then extrapolate the $TTFT/TPOT$ using Equations 4 and 5.

5.4 Design Exploration of Profiling Techniques

We evaluate the efficacy of our proposed profiling methods by analyzing profiling accuracy and cost (profiling memory, time, and GPU-hours) in Figure 7. We combine methods that use the full LLM (L1 and M1) and those that use fingerprints (L2 and M2).

Full-model profiling provides more accurate memory estimations, as it uses the base model directly to solve the system of linear equations. However, its latency estimation accuracy is either comparable to the fingerprint-based method (GPT-J-6B) or inferior, particularly for larger LLMs (Falcon-40B, Llama-2-70B). The size of the large models exceeds the capacity of a single GPU; L1 thus profiles the LLM at higher degrees of TP and PP. Consequently, the derived equations grow increasingly complex, incorporating all the hyperparameters (α, β, γ, δ) as opposed to the single parameter α required by Equation 3 used by L2. This amplifies the estimation error due to compounded dependencies and cascading errors.

Fingerprint-based profiling (L2,M2) greatly reduces profiling cost. It only requires a fraction of the memory the full-model profiling (L1,M1) consumes (3.8-18× less). For small models (e.g., Llama-2-7B), the profiling time and GPU-hours of the fingerprint-based method are slightly larger compared to the full-model profiling. This occurs, as the fingerprint-based profiling requires more runs than the full-model profiling to enable accurate extrapolation. Therefore, for small LLMs, where fingerprint and full-model latencies are similar, this overhead accumulates, making full-model profiling slightly faster. Nonetheless, the extra profiling time is minimal (∼23−124 seconds, ∼36−252 GPU-seconds), and offset by memory savings due to fingerprints. However, with larger LLMs (e.g., Llama-2-70B), the advantages of fingerprint-based profiling become apparent: profiling the Llama-2-70B's fingerprint reduces profiling time and GPU-hours by 2.5× and 3.8×, respectively, compared to profiling

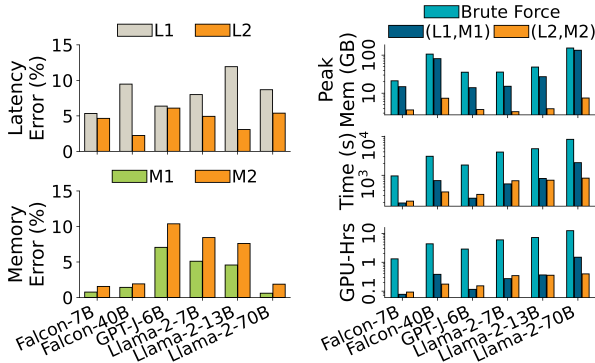

Figure 7: Design exploration of profiling methods. We show profiling error (col. 1) and cost (col. 2) for each method. MaverIQ uses (L2,M2) to reduce profiling cost. See § 5.4.

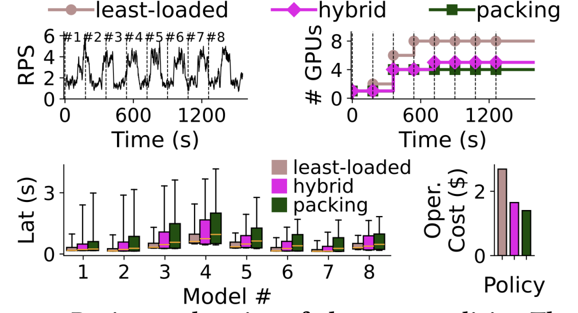

Figure 8: Design exploration of placement policies. The hybrid policy balances packing LLMs to reduce the # of GPUs consumed/operational cost and dispersing LLMs across GPUs to reduce latency. See § 6.

the full LLM. Thus, we deploy MaverIQ with the (L2,M2) profiling methods due to its lower profiling cost and relatively similar accuracy to full-model profiling. Our modular design allows operators to easily swap techniques.

5.5 Support for Various Batch Sizes

Batched requests to the same model can increase GPU utilization and cluster throughput [40, 46]. While we described our profiling techniques for batch size 1, MaverIQ supports larger batch sizes: we observe that the latency and memory overheads of batched requests, O^L_{BS} and O^M_{BS}, inflate latency and memory linearly when batch size is > 1. We develop a model capturing this observation:

$$L(BS) = L(1) + (BS - 1) \times O^L_{BS} \quad \& \quad M(BS) = M(1) + (BS - 1) \times O^M_{BS} \quad (6)$$

where $L(BS)$ and $M(BS)$ are the latency and memory for batch size BS. To estimate the overheads with batch sizes > 1, we follow our profiling procedure (§ 5.2 and 5.3) to estimate the latency and memory for two batch sizes (one and two) and solve a 2×2 system of linear equations using Equation 6. Then, for any given batch size, we can estimate the LLM's latency and memory consumption.

6 Cost-Efficient Deployment

The chosen configuration now needs to be deployed in a way that optimizes the operational cost for the provider. Multi-tenant environments naturally lead to GPUs with varying load and fragmented resource availability throughout the cluster. Existing systems are unable to sufficiently utilize fragmented resources as they assume that the LLM layers must be distributed evenly across GPUs to avoid impacting latency due to synchronization delays. While this assumption is true for training that requires a forward and backward pass, it does not hold for inference which only needs a forward pass. We find that distributing LLM layers across GPUs unevenly has little impact on inference latency (§ 3.2). MaverIQ's Deployment Controller leverages this insight to efficiently utilize GPUs with varying load and fragmented resources to reduce operational costs for the provider while meeting the intents of the user. It makes two key decisions: (1) which GPUs in the cluster to deploy the LLM on, and (2) how to distribute layers across the selected GPUs.

Overview. With the profiling-based estimations (§ 5), we rank configurations that best meet the given user intent (e.g., minimize cost). However, fragmented GPU memory may not allow us to fit the best configuration. So, we traverse the ranked list and find the first feasible configuration to deploy using our load-aware GPU selection and fragmentation-aware layer mapping strategies.

Load-aware GPU selection. For the configuration under consideration, requiring $d_{TP} \times d_{PP}$ GPUs, we generate all GPU sets of that size whose total free memory matches the configuration's memory needs. If no set exists, we consider the next configuration in the ranked list. Otherwise, we iterate over the GPU sets and use our load-aware GPU selection policy to select the GPUs for deployment.

We evaluate the efficacy of three deployment policies as load varies: least-loaded, packing, and hybrid. Least-loaded disperses LLMs across GPUs, choosing those with the least load. Packing places LLMs on the same GPUs until saturation. Hybrid is a flexible policy that balances packing and dispersal. It first packs LLMs on GPUs whose load is closest to but below a threshold θ. If no set satisfies this criterion, the LLM is deployed on the set of GPUs with the lowest average GPU load. We define a GPU's load as the number of seconds it is busy over a window screen period. This definition is more expressive than using memory utilization or number of models loaded; it captures the meaningful work a GPU completes. We track load every second over a 2-minute window, matching the rate at which load changes in production LLM clusters [67, 84].

Figure 8 shows the number of GPUs used (top right), inference latency (bottom left), and operational cost (bottom right). Following the methodology of previous works [38, 46], we generate a 30 min trace with fluctuating load using a Poisson process and deploy an LLM every 3 min to assess policy responses to load spikes/valleys (Figure 8 top left). Least-loaded reduces latency by dispersing LLMs but inflates the provider's operational cost by 1.63× compared to hybrid. Packing only uses four GPUs, increasing GPU contention and latency by 1.35× while only reducing operational costs by 1.18×. Hybrid uses five GPUs: during high load (e.g., #3 deployment), it deploys LLMs on idle GPUs rather than previously packed ones; during low load (e.g., #4, #6-8 deployment), it deploys on packed GPUs. While this slightly increases latency compared to least-loaded and operational cost compared to packing, hybrid best balances meeting the user intent while reducing operational cost for the provider. We use the hybrid policy ($\theta = 0.7$) to select GPUs for deployment but provide a knob to switch policies as needed.

Fragmentation-aware layer mapping. With the selected configuration and GPUs for deployment, our fragmentation-aware layer

mapping algorithm maps individual memory blocks of the LLM to each GPU. Memory blocks are the minimum unit of deployment, representing the memory requirement of a single layer of the LLM:

$$mem_{block} = \frac{mem_{total}}{d_{TP} \times num_{layers}}$$

where mem_{total} is the estimated memory footprint and num_{layers} is the number of hidden layers in the LLM. This basic memory block is configuration-specific, as it captures the memory scaling for a single layer due to the parallelism and quantization used.

We attempt to keep adjacent layers close and distribute blocks evenly. However, if the chosen GPUs have fragmented available memory, we adapt our layer mappings to utilize these fragmented resources. Specifically, MaverIQ unevenly disperses memory blocks across the fragmented GPUs, as unbalanced partitioning has little impact on inference latency (§ 3.2). This enables MaverIQ to pack more LLMs on fewer GPUs, maximizing total cluster throughput and minimizing operational cost for the provider.

7 Implementation

We build MaverIQ in Python (3K LoC) atop TensorRT-LLM [61], a widely-used LLM inference serving framework [60, 62, 63]. The Controller is a persistent process that manages a pool of threads to handle requests made to MaverIQ.

Fingerprint Controller. The Fingerprint Controller (a CPU process) generates and profiles a registered LLM's fingerprint. To create fingerprints, we use PyTorch's API to read a file containing LLM weights/architecture information (provided by the LLM developer in a safetensors format [32]) and remove redundant layers. We deploy the fingerprint on GPUs, issue it dummy requests to measure metadata, and extrapolate the memory footprint/latency per deployment configuration. We store this data in a configuration map locally for efficient access by MaverIQ's Deployment Controller.

Deployment Controller. The Deployment Controller selects the configuration, GPUs, and layer-to-GPU mapping to deploy an LLM with. On the server hosting the selected GPUs, we launch a TensorRT-LLM process and deploy the LLM under the chosen configuration. We store the worker's IP/port in an in-memory map that the Scheduler uses to dispatch requests. We extend TensorRT-LLM to allow manual mapping of LLMs layers to GPUs; this allows MaverIQ to unevenly distribute layers and fully utilize fragmented resources.

Monitoring Daemon. MaverIQ launches a lightweight, monitoring daemon per worker (runs on CPU) that tracks GPU memory and load via the PyTorch CUDA API. It sends updates to MaverIQ's Controller every second via Linux TCP sockets [34]. The Controller maintains a map of the available GPU memory and GPU load per server, which it leverages during model deployment.

8 Evaluation

Workload. We follow the methodology of prior work [3, 25, 46, 57] to create our workload.

We use two Azure LLM inference traces [67] that provide real-world query arrival patterns: (1) the *code* trace with bursty requests, and (2) the *conversation* trace with a steady, high query arrival pattern. We scale the traces to different query arrival rates while preserving the original pattern (e.g., burstiness). We compress the trace by discretizing it into intervals, aggregating request counts

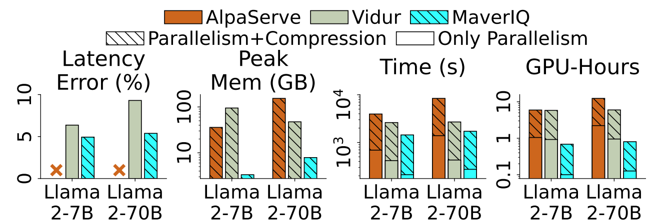

Figure 9: MaverIQ greatly reduces latency estimation error (col. 1) and LLM profiling time and resources (col. 2-4) compared to Vidur and AlpaServe. See § 8.1.

from adjacent intervals, and scaling by a rate factor (0.1-0.4) to generate low to high load where GPU utilization is 30-100%. The scaled traces consist of 861-7722 requests, providing sufficient queries per model. The traces only contain a timestamp and input/output size per query, but do not include the query input and intended LLM; so, we use the ShareGPT dataset [78] to pick the LLM and the input (max 512 tokens) uniformly at random. We use six state-of-the-art LLMs (Llama-2-7B, 13B, 70B, GPT-J-6B, and Falcon-7B, 40B) varying in size and support for parallelism and compression.

Baselines. We compare MaverIQ to three baselines.

Accelerate [24], a library used by prominent ML frameworks (e.g., HuggingFace [33]), automates LLM deployment by leveraging Fully Sharded Data Parallel (FSDP) [113] to partition LLMs across GPUs.

AlpaServe [46], a state-of-the-art LLM serving system, autoselects parallelism to meet SLOs. It assumes prior knowledge of request patterns, unrealistic for dynamic general-purpose inference-serving systems [75, 110]. For fairness, we provide AlpaServe with our full trace and deploy its configurations (DP, TP, PP) on TensorRT-LLM. AlpaServe may drop some LLMs to improve overall SLO attainment [45]; we also compare against AlpaServe*, a variant not allowed to drop LLMs.

Vidur [1] is an LLM profiling/simulation tool that predicts the most cost-effective parallelism configuration for an LLM using random forest regressors. Like AlpaServe, we provide Vidur our trace, as required. As Vidur is not a full inference serving system, we use it as a baseline solely to evaluate profiling methodologies.

Testbed. Our testbed resembles that of prior work [2, 22, 56]. We use 8 × NVIDIA RTX A6000 (48GB) GPUs. Pairs of GPUs are connected with NVLink, and the other connections are PCIe. The server has an AMD EPYC 7763 CPU (64 cores). We use Ubuntu 22.04 and NVIDIA Driver 535.171.04 with CUDA Version 12.2.

Metrics. We show MaverIQ's efficacy with both user and provider metrics. User metrics include latency, user cost, GPU memory consumption, GPU-hours, and SLO attainment. Provider metrics include cluster throughput and operational cost. We report actual values (Figures 10-12, 14) and full distributions (Figures 10-14).

8.1 MaverIQ's Profiling Accuracy and Cost

As MaverIQ heavily relies on its novel profiling techniques, we first evaluate its profiling accuracy and cost against Vidur and AlpaServe (Figure 9), two state-of-the-art works that autoselect LLM deployment configurations. We omit Accelerate, as it does not profile LLMs; it deploys LLM with maximum parallelism.

Profiling error analysis. Figure 9 column 1 compares MaverIQ's accuracy in estimating latency with Vidur's. AlpaServe makes no predictions, as it requires ground truth latency as input. We report

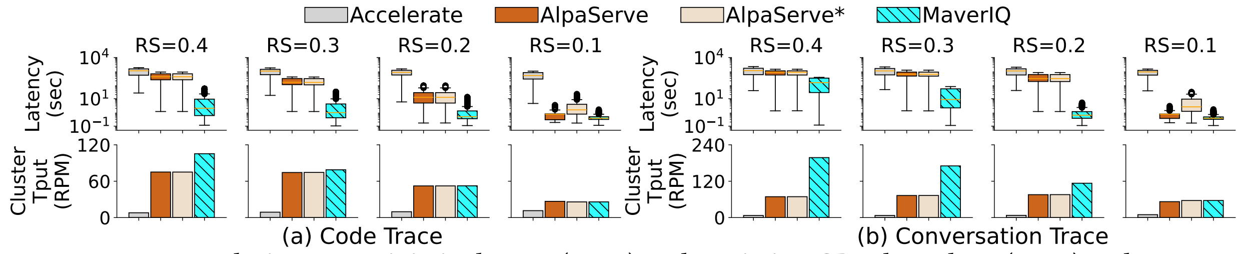

Figure 10: MaverIQ meets the intent to minimize latency (row 1), and maximizes GPU throughput (row 2) as the query arrival rates (RS) fluctuate under Azure's two LLM inference traces: (a) code and (b) conversation trace. See § 8.2.

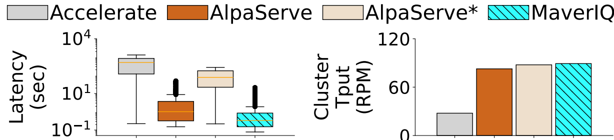

Figure 11: Latency and GPU throughput when deploying several smaller LLMs under high load with the code trace. MaverIQ continues to minimize latency while maximizing throughput (results hold for other loads and traces). See § 8.2.

the median absolute percentage error between ground truth and each system's predicted latency. MaverIQ reduces estimation error by $1.3 - 1.7\times$ compared to Vidur. Vidur trains a regressor per LLM to predict latency, however, MaverIQ's analytical modeling is more accurate than Vidur's black-box ML-driven approach.

Profiling cost analysis. We compare MaverIQ's profiling cost to Vidur and AlpaServe (Figure 9, columns 2-4). The baselines only navigate parallelism techniques. For a fair comparison, we ensure all systems navigate the same search space size: (1) parallelism and compression (we implement support for compression in Vidur/AlpaServe), and (2) only parallelism, where MaverIQ only navigates the parallelism search space.

Figure 9 shows that MaverIQ reduces profiling memory, time, and GPU-hours, regardless of the search space size, by 6-28×, 1.6-5×, and 7-15×, respectively, compared to Vidur and AlpaServe. AlpaServe inflates profiling cost with brute-force profiling to obtain the latency and memory requirements per LLM configuration. Vidur slightly reduces profiling time compared to AlpaServe by profiling fewer configurations, however, it still profiles large LLMs under many configurations that inflate profiling cost. MaverIQ's fingerprints greatly reduce the profiling GPU memory demands, while our analytical models reduce profiling time and GPU-hours, as we only need to observe fingerprints under a few configurations.

> MaverIQ makes accurate estimations (1.3-1.7× less error) while reducing profiling time (1.6-5×), GPU memory consumption (6-28×), and GPU-hours (7-15×) compared to Vidur and AlpaServe.

8.2 MaverIQ with the Minimize-Latency Intent

We now show MaverIQ's efficacy in meeting user intents. We first show that MaverIQ minimizes latency across loads, arrival patterns, and accuracy requirements. As Vidur is only a profiling tool and not a serving system, we omit it from the rest of our evaluation.

MaverIQ under consistent high load. Figure 10a column 1 evaluates MaverIQ under high load with the code trace. MaverIQ reduces median latency by $> 90\%$ compared to all baselines. Accelerate maximizes parallelism for each LLM, increasing GPU contention, queuing delay, and ultimately latency since every GPU hosts all LLMs. Moreover, Accelerate degrades latency for smaller LLMs, as the increased communication overheads outweigh any acceleration from reduced computation each GPU needs to complete with increased parallelism: Accelerate inflates Falcon-40B's latency by 11.5× compared to MaverIQ's configuration of $d_{TP} = 4/d_{PP} = 1$.

AlpaServe and AlpaServe*, despite autoselecting configurations per LLM, choose to maximize parallelism for all LLMs with d_{TP}=4 d_{PP}=2, increasing queuing delay due to GPU contention: MaverIQ reduces queuing delay by $> 10\times$ compared to AlpaServe. MaverIQ tailors configurations: it deploys Llama-2-70B with d_{TP}=8/d_{PP}=1, Llama-2-7B with d_{TP}=2/d_{PP}=4, and the two Falcon-40B's with d_{TP}=4 d_{PP}=1. Moreover, MaverIQ's hybrid, load-aware policy reduces GPU contention: it places the two Llama models across all 8 GPUs, and places the two Falcon-40Bs on a separate set of 4 GPUs, reducing GPU contention and inference latency by 2.26×.

MaverIQ under varying query arrival rates and patterns. Figure 10a columns 1-4 show that MaverIQ is robust to varying query arrival rates: it reduces latency by $\geq 49\%$ while maintaining peak throughput across rate scales compared to the baselines. Figure 10b shows that MaverIQ is robust to varying querying arrival patterns using Azure's conversation trace. Cluster throughput increases for all systems under this trace, as it provides consistent work with its steady, high arrival pattern. MaverIQ provides the best throughput and latency: at high load, it increases throughput by 65-97% while reducing latency by 83-87% compared to the baselines; at low load, it maintains peak throughput while reducing latency by $\geq 48\%$.

MaverIQ under varying model sets. Figure 11 shows MaverIQ's performance as it deploys a larger set of smaller LLMs under high load with the code trace (findings hold for other loads and traces): 2×Llama-2-13B, 2×Llama-2-7B, 2×Falcon-40B, and 2×GPT-J-6B. MaverIQ continues to outperform the baselines, reducing median latency by 67-98% and increasing cluster throughput by 26% on average. AlpaServe makes sub-optimal configuration (d_{TP}=4/d_{PP}=1 for all LLMs) and deployment decisions: despite its brute force algorithm, it greedily skips over MaverIQ's better configuration and placement decisions to reduce its search space navigation time. Hence, AlpaServe inflates GPU contention and overall inference latency by 67%. AlpaServe achieves comparable cluster throughput to MaverIQ despite its sub-optimal configuration decisions. This is because (1) the load is not high enough to showcase MaverIQ's peak throughput, and (2) smaller LLMs have less latency variation

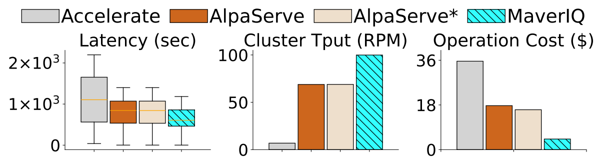

Figure 12: MaverIQ performs best under strict accuracy SLOs. All LLMs deployed without compression. See § 8.2.

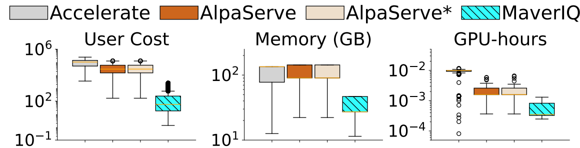

Figure 13: MaverIQ meets the intent, minimizing cost (col. 1), memory (col. 2), and GPU-hours (col. 3) at high load with the code trace (results hold across loads and traces). See § 8.3.

across configurations, making sub-optimal choices less impactful. Nonetheless, MaverIQ is robust to any LLM type (small or large).

MaverIQ under strict accuracy SLOs. We also evaluate MaverIQ under strict accuracy SLOs (Figure 12), which forces MaverIQ to only consider deployment configurations *without compression*. Thus, we navigate only the parallelism search space, like AlpaServe. MaverIQ continues to make better configuration and deployment decisions, reducing median latency by 28-45% for the user while increasing cluster throughput and reducing operational costs by 31-93% and 3.8-8.3× respectively, for the provider. AlpaServe greedily skips over MaverIQ's better configurations and placement decisions during its simulation, degrading latency, cluster throughput, and operational costs compared to MaverIQ. Regardless of the accuracy requirements, MaverIQ provides the best performance for both the user and the provider compared to state-of-the-art baselines.

MaverIQ is robust to load, arrival patterns, models, and accuracy SLOs. Under strict accuracy SLOs, MaverIQ reduces latency by 28-45% for the user while increasing cluster throughput by 31-93% and decreasing operational costs by 3.8-8.3× for the provider. Under loose accuracy SLOs, it further reduces latency by 72%.

8.3 MaverIQ with Other User Intents

We next evaluate MaverIQ's ability to meet diverse intents: minimize user cost, memory, or GPU-hours. Figure 13 shows MaverIQ's performance at high load under the code trace (findings hold for other loads/traces). MaverIQ accurately estimates the impact of parallelism and compression across intents. For example, MaverIQ quantizes and reduces parallelism to lower memory footprint by 69-80% compared to the baselines. Minimizing user cost and GPU-hours requires balancing resource allocation and latency. Accelerate's configurations that maximize parallelism inflate user cost and GPU-hours by increasing resource consumption. AlpaServe configuration decisions attempt to optimize for latency SLO attainment, rendering it inflexible to other intents. In contrast, MaverIQ learns LLM-specific traits to adapt to the intent; for example, quantizing Llama-2 variants to INT4 reduces latency by 60% compared to FP16, as reduced-bit precision formats lower memory traffic and computational latency [13, 101, 106]. Thus, MaverIQ reduces cost by 98% and GPU-hours by 78-96% compared to the baselines.

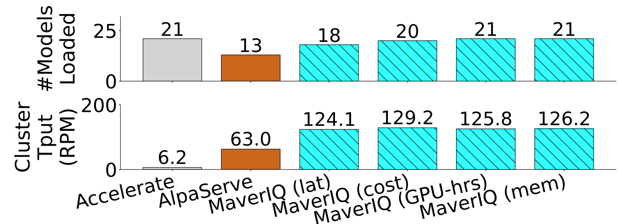

Figure 14: MaverIQ maximizes the number of models deployed in a fixed cluster size (top) while improving cluster throughput (bottom) across intents. See § 8.4.

We also analyze MaverIQ's ability to meet latency SLOs (figure omitted for brevity). Following Proteus [3], we set SLOs as multiples (2.5-7.5×) of the LLM's latency under minimum parallelism and no compression. MaverIQ consistently outperforms the baselines in SLO attainment across different SLOs and query arrival rates while increasing cluster throughput by 55.2% on average. Accelerate defaults to maximum parallelism, inflating SLO violations due to GPU contention and intermediate overheads (all-reduce, all-gather). At low load, AlpaServe matches MaverIQ. However, at high load, AlpaServe violates several SLOs with its greedy configuration selection and placement algorithm.

8.4 MaverIQ Scales with LLMs and Cluster Size

Scale with number of LLMs. MaverIQ scales as the number of LLMs to deploy in a fixed-sized cluster increases. We use a set of 21 LLMs on our 8-GPU cluster with Azure's conversation trace at high load. Figure 14 shows the number of models deployed and cluster throughput for each system. Accelerate serves all 21 LLMs by offloading weights to CPU but incurs PCIe transfer overheads, degrading throughput. AlpaServe can only deploy 13 LLMs with its poor configuration and placement decisions. MaverIQ deploys all 21 LLMs when minimizing memory or GPU-hours by quantizing and minimizing parallelism. When minimizing latency or GPU-hours, MaverIQ increases parallelism and consumes more GPU memory to reduce latency; thus, it deploys at most 20 LLMs. However, regardless of the intent, MaverIQ scales effectively as the number of LLMs grows, boosting throughput by 2-20× compared to the baselines.

Scale with cluster size. (a) In a homogeneous cluster (e.g., all A6000 GPUs), MaverIQ's profiling cost remains constant as the cluster size scales. While more GPUs increase opportunity for parallelism (e.g., d_{TP} and/or $d_{PP} > 8$), we can estimate latency and memory consumption for each new configuration without extra profiling using our analytical models (§ 5.2/ 5.3). (b) In a heterogeneous cluster (e.g., A6000s, A100s, H100s), latency and memory profiles vary by GPU type, requiring profiling for each. However, MaverIQ's novel profiling techniques keep costs manageable. To reduce cost, we could extrapolate profiles across GPU types by adjusting for architectural differences; we leave this for future work.

8.5 Deployment Controller Ablation Study

We assess the efficacy of MaverIQ's fragmentation-aware layer mapping algorithm in reducing the operational cost to serve multiple LLMs. We replicate the experiment in § 8.4 with the minimize-latency intent, comparing the number of models we can load in a fixed-sized cluster with and without our unbalanced layer mapping algorithm. Without it (all layers deployed evenly), MaverIQ can only loads 17 LLMs (one less than with our algorithm) before

requiring 8 more GPUs for the 18th LLM. Thus, without our algorithm, serving cost rises by 1.4-2× (~6.5 $/hour [41]), showing that MaverIQ's fragmentation-aware layer mapping policy effectively packs LLMs on GPUs to reduce operational costs for the provider.

9 Related Work

Estimating LLM performance metrics. Previous work attempts to estimate different end metrics for LLM inference. LLMCarbon [16] uses expensive profiling that measures an LLM's latency under all configurations to estimate its carbon footprint. MaverIQ's inexpensive, accurate estimations via fingerprint-based profiling and analytical modeling can further improve LLMCarbon's efficiency. Vidur [1] uses ML regressors to predict inference latency, however, we show MaverIQ achieves better accuracy with much lower profiling costs (§ 8.1). MAD-Max [29] and NeuSight [43] estimate LLM inference latency, but do not account for the effects of TP and PP in combination. We also empirically observe that Neusight is severely inaccurate: it ignore output length and only supports older LLMs (e.g., BERT, OPT) with learned positional embeddings; NeuSight does not generalize to modern LLMs using rotary embeddings [85]. Finally, no previous works consider the full LLM configuration search space with various compression techniques.

Navigating the deployment configuration search space. Previous works attempt to traverse the deployment configuration search space [42, 46, 57, 64, 71, 84]. However, these works ignore compression and only navigate parallelism. DynamoLLM [84] finds the energy-optimal TP degree per LLM by profiling full LLMs to obtain their energy-latency profiles. SpotServe [57] adapts LLM parallelism to spot instance availability using latency data profiled offline. MaverIQ's fingerprint-based profiler can be integrated into both systems to improve the efficiency of offline profiling. LLM-Pilot [42] predicts the best hardware type to meet the workload demands for an LLM. AlpaServe [46] uses a greedy ILP algorithm to make parallelism decisions. μ-Serve [71] builds on AlpaServe to make power-efficient LLM deployment decisions All these systems require brute-force expensive profiling; we evaluate against AlpaServe and show lower profiling cost (Figure 9).

Serving runtime optimizations. An LLM's serving latency and memory footprint vary depending on the framework and optimizations leveraged. A plethora of works propose new frameworks and optimizations for inference serving [5, 9, 10, 17, 18, 24, 28, 35, 40, 61, 66, 70, 79, 83, 89, 103, 105, 109, 115], with enhancements in (1) KV-cache memory management [5, 28, 40, 61, 70, 74, 103, 109], (2) computation graph compilation [115], and/or (3) efficient GPU utilization via custom kernels [5, 17, 28, 50, 61, 74], to name a few. MaverIQ's portable design makes it easy to accurately understand the impact of such optimizations without expensive profiling.

LLM deployment optimizations. Several works propose optimizations to handle the scale of LLMs: model parallelism [31, 77, 82, 87, 94, 95, 99, 100, 112, 113], quantization [8, 11–13, 20, 26, 39, 47, 51, 53, 96], pruning [19, 54, 86, 97], distillation [27, 91], low-rank adaptation [30], or a combination of those [21]. These methods add to the configuration search space that MaverIQ efficiently navigates.

Execution and scheduling optimizations. Several works focus on optimizing request scheduling [2, 3, 7, 22, 55, 67, 81, 88, 104, 107], reconfiguration [94], and speculative execution [56, 98]. These works are complementary to, and can be used with MaverIQ.

10 Discussion

Extensibility to new models. We show MaverIQ's efficacy for decoder-only LLMs, which are commonly used today [93]. MaverIQ's fingerprint-based profiling and analytical formulations can be used for other model architectures that have repetitive structures such as MoE-models like Mixtral-8-7B [36], that repeats experts, non-transformer models like RWKV [68], that repeats residual blocks, and multimodal models like Flamingo [4], that repeats LM blocks. For instance, Mixtral-8-7B has 32 MoE-layers, each containing 8 structurally-identical experts, of which only 2 are activated per input token. A fingerprint involving only a single expert, a single layer, and a single gating network can be used for profiling (instead of 8 experts and 32 MoE-layes), thereby theoretically saving 99.61% in GPU memory consumption. Extrapolating metrics from fingerprints to full MoE-based models will require updating our analytical formulations. While the proposed formulations lay a foundation for such extensions, this warrants further work that we plan to pursue.

Extensibility to new intents. MaverIQ readily supports intents derived from latency and memory data (e.g., throughput). For new intents (e.g., energy or carbon efficiency), the Controller can be extended to profile additional metrics (e.g., power consumption), though further work is needed to accurately extrapolate these from the fingerprint to the LLM. These techniques can be easily integrated into MaverIQ.

Extensibility to new hardware. MaverIQ supports clusters with the same GPU type (e.g., A100s, A6000s), as its fingerprint-based profiling and analytical modeling treat hardware as an opaque box. Deploying LLMs across heterogeneous devices introduces new challenges (e.g., latency variation across LLM stages deployed on different GPU types). We leave this investigation to future work.

11 Conclusion

We present MaverIQ, the first intent-based LLM inference serving system that, through its novel fingerprint-based profiling and analytical models, translates a user's intent to the best deployment configuration. Moreover, MaverIQ efficiently utilizes fragmented GPU resources by unevenly distributing LLM layers across GPUs without impacting latency, reducing operational cost for the provider. Across diverse models, traces, and loads, MaverIQ best meets user intents with minimal profiling costs and reduces operational costs for the provider compared to state-of-the-art baselines.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. We thank the members of the UT-SysML research group for their insightful discussions to improve this work. We also thank Vivek Chawda for his helpful feedback on the introduction. This work was supported by the UT ECE junior faculty start-up fund, UT iMAGiNE consortium and its industrial affiliates, an award from the UT Machine Learning Lab (MLL), a Cisco Research Award, and an Amazon Research Award.

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. *Proceedings of Machine Learning and Systems* 6 (29 May 2024), 351–366.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[3] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. 2024. Proteus: A High-Throughput Inference-Serving System with Accuracy Scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (, La Jolla, CA, USA,) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 318–334. doi:10.1145/3617232.3624849

[4] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. 2022. Flamingo: a Visual Language Model for Few-Shot Learning. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=EbMuimAbPbs

[5] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. 2022. DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. *arXiv preprint arXiv: 2207.00032* (2022).

[6] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 472–487. doi:10.1145/3492321.3519584

[7] Branden Butler, Sixing Yu, Arya Mazaheri, and Ali Jannesari. 2024. PipeInfer: Accelerating LLM Inference using Asynchronous Pipelined Speculation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 40, 19 pages. doi:10.1109/SC41406.2024.00046

[8] Shiyang Chen, Shaoyi Huang, Santosh Pandey, Bingbing Li, Guang R. Gao, Long Zheng, Caiwen Ding, and Hang Liu. 2021. E.T.: re-thinking self-attention for transformer models on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 25, 18 pages. doi:10.1145/3458817.3476138

[9] Shaoyuan Chen, Yutong Lin, Mingxing Zhang, and Yongwei Wu. 2024. Efficient and Economic Large Language Model Inference with Attention Offloading. *arXiv preprint arXiv:2405.01814* (2024).

[10] Yidong Chen, Chen Zhang, Rongchao Dong, Haoyuan Zhang, Yonghua Zhang, Zhonghua Lu, and Jidong Zhai. 2024. MixQ: Taming Dynamic Outliers in Mixed-Precision Quantization by Online Prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 74, 15 pages. doi:10.1109/SC41406.2024.00080

[11] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2198, 15 pages.

[12] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Article 441, 28 pages.

[13] Tim Dettmers and Luke Zettlemoyer. 2023. The case for 4-bit precision: k-bit inference scaling laws. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 307, 25 pages.

[14] Shichen Dong, Wen Cheng, Jiayu Qin, and Wei Wang. 2024. QAQ: Quality Adaptive Quantization for LLM KV Cache. *arXiv preprint arXiv:2403.04643* (2024).

[15] Jiaang Duan, Shiyou Qian, Dingyu Yang, Hanwen Hu, Jian Cao, and Guangtao Xue. 2024. MOPAR: A Model Partitioning Framework for Deep Learning Inference Services on Serverless Platforms. *arXiv preprint arXiv:2404.02445* (2024).

[16] Ahmad Faiz, Sotaro Kaneda, Ruhan Wang, Rita Chukwunyere Osi, Prateek Sharma, Fan Chen, and Lei Jiang. 2024. LLMCarbon: Modeling the End-to-End Carbon Footprint of Large Language Models. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=aIok3ZD9to

[17] Ruibo Fan, Xiangrui Yu, Peijie Dong, Zeyu Li, Gu Gong, Qiang Wang, Wei Wang, and Xiaowen Chu. 2025. SpInfer: Leveraging Low-Level Sparsity for Efficient Large Language Model Inference on GPUs. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 243–260. doi:10.1145/3689031.3717481

[18] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 389–402. doi:10.1145/3437801.3441578

[19] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 414, 15 pages.

[20] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. In *International Conference on Learning Representations*.

[21] Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. 2024. MARLIN: Mixed-Precision Auto-Regressive Parallel Inference on Large Language Models. *arXiv preprint arXiv:2408.11743* (2024).

[22] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: low-latency serverless inference for large language models. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 8, 19 pages.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[24] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. 2022. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate. Accessed: 2024-08-17.

[25] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: performance predictability from the bottom up. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 25, 20 pages.

[26] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2023. OliVe: Accelerating Large Language Models via Hardware-friendly Outlier-Victim Pair Quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23, Vol. 36)*. ACM, 1–15. doi:10.1145/3579371.3589038

[27] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. *ArXiv* abs/1503.02531 (2015). https://api.semanticscholar.org/CorpusID:7200347

[28] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. *arXiv preprint arXiv:2401. 08671* (2024). arXiv:2401.08671 [cs.PF] https://arxiv.org/abs/2401.08671

[29] Samuel Hsia, Alicia Golden, Bilge Acun, Newsha Ardalani, Zachary DeVito, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2024. MAD-Max Beyond Single-Node: Enabling Large Machine Learning Model Acceleration on Distributed Systems. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 818–833. doi:10.1109/ISCA59077.2024.00064

[30] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=nZeVKeeFYf9

[31] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao 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 *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, Article 10, 10 pages.

[32] HuggingFace. 2022. Safetensors. https://github.com/huggingface/safetensors. Accessed: 2024-10-08.

[33] HuggingFace. 2024. Hugging Face. https://huggingface.co/. Accessed: 2024-10-20.

[34] IBM Developer. 2020. Socket programming in C. https://developer.ibm.com/tutorials/l-sock/ Accessed: 2025-04-13.

[35] Jinwoo Jeong and Jeongseob Ahn. 2025. Accelerating LLM Serving for Multi-turn Dialogues with Efficient Resource Management. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3676641.3716245

[36] 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 preprint arXiv:2401.04088* (2024).

[37] Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. 2024. A Comprehensive Evaluation of Quantization Strategies for Large Language Models. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 12186–12215. doi:10.18653/v1/2024.findings-acl.726

[38] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2022. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 289–305. doi:10.1145/3542929.3563468

[39] H. T. Kung, Bradley McDanel, and Sai Qian Zhang. 2020. Term quantization: furthering quantization at run time. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 96, 14 pages.

[40] 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* (, Koblenz, Germany,) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[41] Lambda Labs. 2025. GPU Cloud Pricing. https://lambda.ai/service/gpu-cloud/pricing. Accessed: 2025-04-13.

[42] Malgorzata Lazuka, Andreea Anghel, and Thomas Parnell. 2024. LLM-Pilot: Characterize and Optimize Performance of your LLM Inference Services. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 16, 18 pages. doi:10.1109/SC41406.2024.00022

[43] Seonho Lee, Amar Phanishayee, and Divya Mahajan. 2025. Forecasting GPU Performance for Deep Learning Training and Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 493–508. doi:10.1145/3669940.3707265

[44] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 7871–7880. doi:10.18653/v1/2020.acl-main.703

[45] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe [Source Code]: Function in model_parallelism.py, Line 323-353. https://github.com/alpa-projects/mms/blob/dba47b18e95f037aadc8aff336e2e7e337010495/alpa_serve/placement_policy/model_parallelism.py#L323.

[46] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 663–679. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[47] 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 LLM Compression and Acceleration. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 87–100. https://proceedings.mlsys.org/paper_files/paper/2024/file/42a452cbafa9dd64e9ba4aa95cc1ef21-Paper-Conference.pdf

[48] Zhiqi Lin, Youshan Miao, Guodong Liu, Xiaoxiang Shi, Quanlu Zhang, Fan Yang, Saeed Maleki, Yi Zhu, Xu Cao, Cheng Li, Mao Yang, Lintao Zhang, and Lidong Zhou. 2023. SuperScaler: Supporting Flexible DNN Parallelization via a Unified Abstraction. *arXiv preprint arXiv:2301.08984* (2023).

[49] Weijie Liu, Zhiquan Lai, Shengwei Li, Yabo Duan, Keshi Ge, and Dongsheng Li. 2022. AutoPipe: A fast pipeline parallelism approach with balanced partitioning and micro-batch slicing. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 301–312.

[50] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Ré, and Beidi Chen. 2023. Deja Vu: contextual sparsity for efficient LLMs at inference time. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 919, 40 pages.

[51] LMDeploy. 2024. INT8 KV Cache. https://lmdeploy.readthedocs.io/en/v0.4.0/quantization/kv_quant.html. Accessed: 2024-10-20.

[52] Cheng Luo, Tianle Zhong, and Geoffrey Fox. 2023. RTP: Rethinking Tensor Parallelism with Memory Deduplication. *arXiv preprint arXiv:2311.01635* (2023).

[53] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The era of 1-bit llms: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764* (2024).

[54] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. LLM-Pruner: On the Structural Pruning of Large Language Models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Article 950, 19 pages.

[55] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2025. Helix: Serving Large Language Models over Heterogeneous GPUs and Network via Max-Flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 586–602. doi:10.1145/3669940.3707215

[56] 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 932–949. doi:10.1145/3620666.3651335

[57] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. SpotServe: Serving Generative Large Language Models on Preemptible Instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 1112–1127. doi:10.1145/3620665.3640411

[58] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. doi:10.1145/3458817.3476209

[59] NetApp. 2024. Fractional GPU Allocation for Less Demanding or Interactive Workloads. https://docs.netapp.com/us-en/netapp-solutions/ai/osrunai_fractional_gpu_allocation_for_less_demanding_or_interactive_workloads.html. Accessed: 2024-10-18.

[60] NVIDIA. 2020. Optimizing NVIDIA TensorRT Conversion for Real-time Inference on Autonomous Vehicles. https://developer.nvidia.com/blog/optimizing-nvidia-tensorrt-conversion-for-real-time-inference-on-autonomous-vehicles/. Accessed: 2024-10-15.

[61] NVIDIA. 2023. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM. Accessed: 2024-10-20.

[62] NVIDIA. 2024. Amazon Accelerates Customer Satisfaction With NVIDIA Triton Inference Server and NVIDIA TensorRT. https://resources.nvidia.com/en-us-inference-customer-story/nvidia-amazon-custom. Accessed: 2024-10-17.

[63] NVIDIA. 2024. American Express Prevents Fraud and Foils Cybercrime With NVIDIA AI Solutions. https://resources.nvidia.com/en-us-inference-customer-story/american-express-prevents-fraud. Accessed: 2024-10-17.

[64] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du seong Chang, and Jiwon Seo. 2024. ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. 369–384.

[65] OpenNMT. 2019. CTranslate2. https://opennmt.net/CTranslate2/. Accessed: 2024-10-06.

[66] Xiurui Pan, Endian Li, Qiao Li, Shengwen Liang, Yizhou Shan, Ke Zhou, Yingwei Luo, Xiaolin Wang, and Jie Zhang. 2024. InstInfer: In-Storage Attention Offloading for Cost-Effective Long-Context LLM Inference. *arXiv preprint arXiv:2409.04992* (2024).

[67] 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)*. 118–132. doi:10.1109/ISCA59077.2024.00019

[68] Bo Peng, Eric Alcaide, Quentin Gregory Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Nguyen Chung, Leon Derczynski, Xingjian Du, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen

Hou, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartłomiej Koptyra, Hayden Lau, Jiaju Lin, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Johan S. Wind, Stanisław Woźniak, Zhenyuan Zhang, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. 2023. RWKV: Reinventing RNNs for the Transformer Era. In *The 2023 Conference on Empirical Methods in Natural Language Processing*. https://openreview.net/forum?id=7SaXczaBpG

[69] Jeff Pool, Abhishek Sawarkar, and Jay Rodge. 2021. Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT. https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/. Accessed: 2024-10-20.

[70] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2024. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. *arXiv preprint arXiv:2405.04437* (2024).

[71] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2024. Power-aware Deep Learning Model Serving with μ-Serve. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 75–93. https://www.usenix.org/conference/atc24/presentation/qiu

[72] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[73] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21, 1, Article 140, 67 pages.

[74] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 20, 16 pages.

[75] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[76] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[77] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. 2019. Measuring the Effects of Data Parallelism on Neural Network Training. *Journal of Machine Learning Research* 20 (2019), 1–49.

[78] ShareGPT. 2023. ShareGPT. https://sharegpt.com/. Accessed: 2024-10-16.

[79] 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 *Proceedings of the 40th International Conference on Machine Learning* (, Honolulu, Hawaii, USA,) *(ICML'23)*. JMLR.org, Article 1288, 23 pages.

[80] 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 preprint arXiv:1909.08053* (2020). arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[81] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. 2024. USHER: holistic interference avoidance for resource optimized ML inference. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 51, 18 pages.

[82] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. 2023. A Hybrid Tensor-Expert-Data Parallelism Approach to Optimize Mixture-of-Experts Training. In *Proceedings of the 37th International Conference on Supercomputing (ICS '23)*. ACM. doi:10.1145/3577193.3593704

[83] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 590–606. doi:10.1145/3694715.3695964

[84] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. *arXiv preprint arXiv:2408.00741* (2024). https://arxiv.org/abs/2408.00741

[85] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. RoFormer: Enhanced transformer with Rotary Position Embedding. *Neurocomputing* 568 (2024), 127063. doi:10.1016/j.neucom.2023.127063

[86] Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. 2024. A Simple and Effective Pruning Approach for Large Language Models. In *The Twelfth International Conference on Learning Representations*.

[87] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. 2024. AdaPipe: Optimizing Pipeline Parallelism with Adaptive Recomputation and Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 86–100. doi:10.1145/3620666.3651359

[88] Zhenbo Sun, Shengqi Chen, Yuanwei Wang, Jian Sha, Guanyu Feng, and Wenguang Chen. 2025. MEPipe: Democratizing LLM Training with Memory-Efficient Slice-Level Pipeline Scheduling on Cost-Effective Accelerators. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1263–1278. doi:10.1145/3689031.3717469

[89] Bowen Tan, Yun Zhu, Lijuan Liu, Hongyi Wang, Yonghao Zhuang, Jindong Chen, Eric Xing, and Zhiting Hu. 2024. RedCoast: A Lightweight Tool to Automate Distributed Training of LLMs on Any GPU/TPUs. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: System Demonstrations)*, Kai-Wei Chang, Annie Lee, and Nazneen Rajani (Eds.). Association for Computational Linguistics, Mexico City, Mexico, 137–147. doi:10.18653/v1/2024.naacl-demo.14

[90] Yi Tay, Mostafa Dehghani, Vinh Q. Tran, Xavier Garcia, Jason Wei, Xuezhi Wang, Hyung Won Chung, Siamak Shakeri, Dara Bahri, Tal Schuster, Huaixiu Steven Zheng, Denny Zhou, Neil Houlsby, and Donald Metzler. 2023. UL2: Unifying Language Learning Paradigms. In *International Conference on Learning Representations*.

[91] Inar Timiryasov and Jean-Loup Tastet. 2023. Baby Llama: knowledge distillation from an ensemble of teachers trained on a small dataset with no performance penalty. *arXiv preprint arXiv:2308.02019* (2023).

[92] TrueFoundry. 2023. Using Fractional GPUs. https://docs.truefoundry.com/docs/using-fractional-gpus. Accessed: 2024-10-18.

[93] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[94] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. 2024. Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 195–210. doi:10.1145/3694715.3695975

[95] Boxiang Wang, Qifan Xu, Zhengda Bian, and Yang You. 2022. Tesseract: Parallelize the Tensor Parallelism Efficiently. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP '22)*. ACM. doi:10.1145/3545008.3545087

[96] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. 2023. BitNet: Scaling 1-bit Transformers for Large Language Models. *arXiv preprint arXiv:2310.11453* (2023). arXiv:arXiv:2310.11453

[97] Tuowei Wang, Kun Li, Zixu Hao, Donglin Bai, Ju Ren, Yaoxue Zhang, Ting Cao, and Mao Yang. 2024. Long Exposure: Accelerating Parameter-Efficient Fine-Tuning for LLMs under Shadowy Sparsity. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 75, 18 pages. doi:10.1109/SC41406.2024.00081

[98] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An Efficient Multi-Level Inference System for Large Language Models. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 233–248. doi:10.1145/3552326.3587438

[99] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 640–654. doi:10.1145/3694715.3695948

[100] Hao Wu, Shiyi Wang, Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2023. A Generic, High-Performance, Compression-Aware Framework for Data Parallel DNN Training. *IEEE Transactions on Parallel and Distributed Systems* (2023), 1–20. doi:10.1109/TPDS.2023.3266246

[101] Xiaoxia Wu, Cheng Li, Reza Yazdani Aminabadi, Zhewei Yao, and Yuxiong He. 2023. Understanding INT4 quantization for language models: latency speedup, composability, and failure cases. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1562, 16 pages.

[102] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*. PMLR, 38087–38099.

[103] Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina. 2024. Moe-infinity: Activation-aware expert offloading for efficient moe serving. *arXiv preprint arXiv:2401.14361* (2024).

[104] Fan Yang, Zehao Wang, Haoyu Zhang, Zhenhua Zhu, Xinhao Yang, Guohao Dai, and Yu Wang. 2024. Efficient Deployment of Large Language Model across Cloud-Device Systems. In *2024 IEEE 37th International System-on-Chip Conference (SOCC)*. 1–6. doi:10.1109/SOCC62300.2024.10737825

[105] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 94–109. doi:10.1145/3689031.3696098

[106] Zhewei Yao, Xiaoxia Wu, Cheng Li, Stephen Youn, and Yuxiong He. 2023. ZeroQuant-V2: Exploring Post-training Quantization in LLMs from Comprehensive Study to Low Rank Compensation. *arXiv preprint arXiv:2303.08302* (2023). arXiv:2303.08302 [cs.LG] https://arxiv.org/abs/2303.08302

[107] 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)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[108] Jeffrey Yu, Kartik Prabhu, Yonatan Urman, Robert M. Radway, Eric Han, and Priyanka Raina. 2024. 8-bit Transformer Inference and Fine-tuning for Edge Accelerators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA,) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 5–21. doi:10.1145/3620666.3651368

[109] Lingfan Yu, Jinkun Lin, and Jinyang Li. 2025. Stateful Large Language Model Serving with Pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 144–158. doi:10.1145/3689031.3696086

[110] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1049–1062. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[111] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. 2020. PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 1051, 12 pages.

[112] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, Cheng Li, Ping Luo, and Heming Cui. 2022. vPipe: A Virtualized Acceleration System for Achieving Efficient and Scalable Pipeline Parallel DNN Training. *IEEE Transactions on Parallel and Distributed Systems* 33, 3 (2022), 489–506. doi:10.1109/TPDS.2021.3094364

[113] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *arXiv preprint arXiv:2304.11277* (2023).

[114] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[115] Donglin Zhuang, Zhen Zheng, Haojun Xia, Xiafei Qiu, Junjie Bai, Wei Lin, and Shuaiwen Leon Song. 2024. MonoNN: enabling a new monolithic optimization space for neural network inference tasks on modern GPU-centric architectures. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 53, 17 pages.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

1 Overview of Contributions and Artifacts

1.1 Paper's Main Contributions

C_1 We introduce LLM fingerprints, a compact proxy that captures the essence of the model, enabling lightweight profiling.

C_2 We are the first to formulate an analytical model that accurately captures the effects of parallelism techniques on inference latency for LLMs. Combining both fingerprints and analytical models, `MaverIQ` reduces profiling cost by 7-15× while reducing estimation error by 1.3-1.7×.

C_3 We are the first to show that we can unevenly distribute LLM layers across GPUs to utilize fragmented resources without harming inference latency. This technique reduces operational cost by up to 2×.

C_4 We build `MaverIQ` atop TensorRT-LLM and show that under strict accuracy requirements, `MaverIQ` reduces latency by 28-45% for the user while reducing operational cost by 3.8-8.3× for the provider. Under lower accuracy requirements, `MaverIQ` can exploit compression techniques to further reduce both user cost and latency by about 72%.

1.2 Computational Artifacts

A_1 https://doi.org/10.5281/zenodo.16945750

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2	Figures 1, 6, 7, 9
A_1	C_3	Figure 3
A_1	C_4	Table 1 Figures 5, 10, 11, 12, 13, 14

2 Artifact Identification

2.1 Computational Artifact A_1

Relation To Contributions

This artifact provides the complete implementation of `MaverIQ` alongside the baseline systems used for comparative analysis in C_4. `MaverIQ` is built atop TensorRT-LLM, which we have extended to support unbalanced pipeline parallelism. The code is organized into multiple directories, each corresponding to a specific component of `MaverIQ`. Each directory includes a corresponding README file with detailed instructions to facilitate ease of use.

The *tools/profiling* directory contains the scripts used to generate the LLM fingerprints (C_1) and collect the metadata required by the profiler. The *tools/estimator* directory includes the analytical estimation models (C_2), which process the metadata to predict memory footprint and inference latency for arbitrary deployment configurations. This directory also includes the implementation of a gradient descent algorithm, which is used to determine the constant parameters of the analytical models.

The *tools/measurement* directory contains the scripts utilized in the characterization experiments, including C_3.

Finally, the *tools/controller* directory includes the runtime evaluation scripts responsible for executing `MaverIQ`, while the *baselines* directory contains the instructions and code necessary to build and evaluate the baseline systems used in C_4.

Expected Results

Overall, `MaverIQ` is designed to interpret high-level user intents and automatically translate them into optimal deployment configurations. It then deploys and serves LLMs accordingly. Compared to the baselines, `MaverIQ` consistently reduces latency and/or cost based on user-defined objectives, while also enhancing system throughput and lowering operational costs. These improvements are largely attributable to `MaverIQ`'s ability to accurately map user intents to LLM configurations and its load-aware deployment strategy, which effectively utilizes fragmented GPU memory.

Expected Reproduction Time (in Minutes)

Building `MaverIQ` requires approximately 5-10 minutes, whereas downloading the necessary model weights locally may take a few hours, depending on available internet bandwidth. Comprehensive build and usage instructions are provided in the following sections. Prior to running each experiment, it is necessary to generate the corresponding computational graphs, as required by the TensorRT-LLM framework. The graph generation process for each full LLM typically requires 2-35 minutes, depending on the selected model and configuration. Although each script includes predefined commands to automate graph generation, it is essential—particularly for the evaluation in C_4—that users pre-generate the computational graphs before executing the experiments. This ensures that runtime performance measurements are not inflated by graph construction overheads. In our main evaluation, the execution time for each experiment, excluding graph generation, ranges approximately from 30 to 40 minutes.

Artifact Setup (incl. Inputs)

Hardware: All experiments are conducted on a server equipped with 8 × NVIDIA RTX A6000 GPUs, each with 48GB of memory. GPU pairs are interconnected via NVLink, while remaining connections utilize PCIe. The server is powered by an AMD EPYC 7763 processor with 64 cores.

Software: Our experiments are performed on Ubuntu 22.04 with NVIDIA Driver version 535.171.04 and CUDA 12.2. The software environment includes Python-3.10.12, PyTorch-2.1.2, and a customized version of TensorRT-LLM-0.9.0.dev2024022000. It is essential to set Python 3.10 as the default Python to build `MaverIQ`.

Datasets / Inputs: We evaluate `MaverIQ` using six state-of-the art LLMs: Falcon-7B, Falcon-40B, GPT-J-6B, Llama-2-7B, Llama-2-13B, and Llama-2-70B—all available through the HuggingFace library. We utilize Azure's LLM inference traces, which are provided in A_1 under the *datasets* directory. Query inputs are derived from the ShareGPT dataset and are automatically processed through our scripts.

Installation and Deployment: We build `MaverIQ` atop TensorRT-LLM-0.9.0.dev2024022000, which we have modified to support unbalanced pipeline parallelism. The instructions to compile the source code are detailed below:

- To avoid package conflicts and crashes during the initial installation, we strongly suggest using a Docker container for the installation of `MaverIQ`:

```
# Instal nvidia-container-toolkit:
sudo apt-get install -y nvidia-container-toolkit

# Build the Docker container:
docker build -t <name_of_image> MaverIQ/environment

# Initialize Docker container:
docker run --rm --runtime=nvidia --gpus all
  --shm-size=10.24gb --entrypoint /bin/bash -it
  <name_of_image>

# Create and activate a virtual environment inside
  the container:
virtualenv -p /usr/bin/python3 /workspace/my_env
source /workspace/my_env/bin/activate
```

- Install TensorRT-LLM and required packages:

```
apt-get update && apt-get -y install python3.10
  python3-pip openmpi-bin libopenmpi-dev
pip3 install MaverIQ/environment/
  nvidia_ammo-0.7.4.post1-py3-none-any.whl
pip3 install tensorrt_llm==0.9.0.dev2024022000 -U
  --pre --extra-index-url https://pypi.nvidia.com
pip3 install -r
  MaverIQ/TensorRT-LLM/requirements_MaverIQ.txt
```

- Ensure CUDA is installed and update it:

```
python -m pip uninstall -y cuda || true
python -m pip install --no-cache-dir
  "cuda-python>=12,<13"
```

- Allow OpenMPI to run as root (if required):

```
export OMPI_ALLOW_RUN_AS_ROOT=1
export OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1
```

- Find the *location* of TensorRT-LLM's installation:

```
pip3 show tensorrt_llm
```

- Overwrite installation files with the custom version of TensorRT-LLM:

```
cp -r MaverIQ/TensorRT-LLM/tensorrt_llm/ <location>
```

Artifact Execution

To execute the experiments, the first step involves collecting profiling metadata and determining the constant parameters of the analytical models using the gradient descent algorithm. Once this step is completed, users can proceed to run the evaluation scripts located in the *tools/controller* directory. For baseline comparison, the configuration data must first be generated by executing the scripts in the *baselines* directory, followed by running the corresponding evaluation scripts under *tools/controller*. However, for convenience, all experiments reported in the paper include pre-defined configurations embedded within the scripts. This allows users to bypass the baseline configuration step and directly reproduce the results. Additionally, we provide comprehensive bash scripts that include all commands used in our evaluation experiments, enabling straightforward replication of our results.

Artifact Analysis (incl. Outputs)

We provide a plotting script in A_1, located in the *tools/artifacts-plotting* directory, which post-processes all collected experimental results. All output data is organized into structured subdirectories within the *outputs* folder, categorized by experiment. Once the results have been collected, users can generate all the figures presented in the paper by executing the plotting script.

Artifact Evaluation (AE)

2.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

We provide detailed instructions for installing the required packages and tools for the end-to-end evaluation, including `MaverIQ` and the *baselines*. Additionally, we include guidelines for downloading the models used in our experimental setup.

`MaverIQ` Installation. As described in the AD Artifact Setup, `MaverIQ` is built atop TensorRT-LLM-0.9.0.dev2024022000, which we have modified to support unbalanced pipeline parallelism. The instructions to compile the source code are detailed below:

- To avoid package conflicts and crashes during the initial installation, we strongly suggest using a Docker container for the installation of `MaverIQ`:

```
# Instal nvidia-container-toolkit:
sudo apt-get install -y nvidia-container-toolkit

# Build the Docker container:
docker build -t <name_of_image> MaverIQ/environment

# Initialize Docker container:
docker run --rm --runtime=nvidia --gpus all
  --shm-size=10.24gb --entrypoint /bin/bash -it
  <name_of_image>

# Create and activate a virtual environment inside
  the container:
virtualenv -p /usr/bin/python3 /workspace/my_env
source /workspace/my_env/bin/activate
```

- Install TensorRT-LLM and required packages:

```
apt-get update && apt-get -y install python3.10
  python3-pip openmpi-bin libopenmpi-dev
pip3 install MaverIQ/environment/
  nvidia_ammo-0.7.4.post1-py3-none-any.whl
pip3 install tensorrt_llm==0.9.0.dev2024022000 -U
  --pre --extra-index-url https://pypi.nvidia.com
pip3 install -r
  MaverIQ/TensorRT-LLM/requirements_MaverIQ.txt
```

- Ensure CUDA is installed and update it:

```
python -m pip uninstall -y cuda || true
python -m pip install --no-cache-dir
  "cuda-python>=12,<13"
```

- Allow OpenMPI to run as root (if required):

```
export OMPI_ALLOW_RUN_AS_ROOT=1
export OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1
```

- Find the *location* of TensorRT-LLM's installation:

```
pip3 show tensorrt_llm
```

- Overwrite installation files with the custom version of TensorRT-LLM:

```
cp -r MaverIQ/TensorRT-LLM/tensorrt_llm/ <location>
```

Baseline Installation. In our end-to-end evaluation, we used three state-of-the art baselines: (1) *Accelerate*, (2) *AlpaServe*, and (3) *Vidur*. *Accelerate* is provided as a Python package through the HuggingFace library; its installation is included in `MaverIQ`'s installation guide. *AlpaServe* and *Vidur* are provided as separate packages and have to be manually installed. We advise users to install and execute *AlpaServe* and *Vidur* using a Docker container, so the installation of MaverIQ is not affected. An example Dockerfile can be found in *baselines/Vidur/Dockerfile*. Detailed instructions about their installation and execution can be found within their respective *README.md* files in the *baselines* directory. Nevertheless, we provide the instructions for their environmental setup:

- ***Accelerate***. The required packages for *Accelerate* can be installed with the following command:

```
pip install matplotlib pandas datasets torch pynvml
  transformers bitsandbytes
```

- ***AlpaServe***. To set up the environment, users must execute from *baselines/AlpaServe/mms* the following commands:

```
pip install -e .
pip install ray pandas datasets parameterized
```

- ***Vidur***. For the *Vidur* installation, we detail the commands as follows:

```
#Step 1. Build the Docker container (inside
  MaverIQ/baselines/Vidur):
docker build -t vidur_baseline .

#Step 2. Initialize Docker container (inside
  MaverIQ):
docker run --rm --runtime=nvidia --gpus all
  --shm-size=10.24gb --entrypoint /bin/bash -it -v
  ./models/:/workspace/models -v
  ./baselines/Vidur/:/workspace/Vidur
  vidur_baseline

#Step 3. Create and activate virtual environment
  (inside the container):
which python3
virtualenv -p /usr/bin/python3
  /workspace/Vidur/Vidur_virtualenv
source
  /workspace/Vidur/Vidur_virtualenv/bin/activate

#Step 4. Two Docker containers must be created,
  one with NVIDIA-SMI support and one without.
  Make sure that you commit the containers
  appropriately with their respective names. To
  install the NVIDIA drivers (only for the
  container with NVIDIA-SMI support), execute the
  following commands:
apt-get update
apt-get install -y nvidia-utils-535
```

```
#Step 5. Clone Sarathi-Serve and Vidur repos
   (inside /workspace/Vidur):
cd /workspace/Vidur
git clone -b vidur
   https://github.com/microsoft/sarathi-serve.git
git clone https://github.com/microsoft/vidur.git

#Step 6. Install Sarathi-Serve (inside
   /workspace/Vidur/sarathi-serve):
cd /workspace/Vidur/sarathi-serve
pip install -e . --extra-index-url
   https://flashinfer.ai/whl/cu121/torch2.3/
pip uninstall flashinfer -y
pip install flashinfer==0.0.5 --extra-index-url
   https://flashinfer.ai/whl/cu121/torch2.3/

#Step 7. Install Vidur (inside
   /workspace/Vidur/vidur):
cd /workspace/Vidur/vidur
python -m pip install -r requirements.txt
python -m pip install -e .
```

Once the experiments using Vidur have been completed, users can exit the Docker container safely.

Model Download. For our experiments we used six state-of-the art LLMs: Falcon-7B, Falcon-40B, GPT-J-6B, Llama-2-7B, Llama-2-13B, and Llama-2-70B—all available through the HuggingFace library. Users need to locally download their weights and store them in the *models* directory. The general commands to download a model's weights from HuggingFace are:

```
apt-get update && apt-get install -y git git-lfs
git lfs install
git clone <url_to_model_weights> <local_path>
```

Specifically for the Falcon-7B, Falcon-40B, and GPT-J-6B models the commands to download the weights locally are as follows:

```
# falcon-7b-instruct
git clone
   https://huggingface.co/tiiuae/falcon-7b-instruct
   MaverIQ/models/falcon/falcon-7b

# falcon-40b-instruct
git clone
   https://huggingface.co/tiiuae/falcon-40b-instruct
   MaverIQ/models/falcon/falcon-40b

# gptj-6b
git clone https://huggingface.co/EleutherAI/gpt-j-6b
   MaverIQ/models/gptj/gptj-6b
```

The Llama-2 models are governed by the Meta license. Therefore, users must request access for the weights through HuggingFace. Specifically, for each of the three models, users must access the following links and submit their request to gain access. The links for the Llama-2 models used in our experiments are the following:

- Llama-2-7b: https://huggingface.co/meta-llama/Llama-2-7b-hf
- Llama-2-13b: https://huggingface.co/meta-llama/Llama-2-13b-hf
- Llama-2-70b: https://huggingface.co/meta-llama/Llama-2-70b-hf

Once access has been granted, users must create an access token (e.g., "HF_TOKEN") through HuggingFace by navigating to "*Profile → Settings → Access Tokens → Create new Access Token*". Afterwards, the model's weight can be downloaded locally through the following script. We note that this script refers to Llama-2-13B, but similar instructions can be used for the other models (i.e., Llama-2-7B and Llama-2-70B) by modifying the *repo_id* and *local_dir*.

```
from huggingface_hub import snapshot_download

snapshot_download(
    repo_id="meta-llama/Llama-2-13b-hf",
    token="HF_TOKEN",
    local_dir="MaverIQ/models/llama-2/llama-2-13b",
    local_dir_use_symlinks=False
)
```

Artifact Execution

We tested and evaluated our implementation on a server equipped with 8 × NVIDIA RTX A6000 GPUs (48GB each) with NVLink for each GPU pair and PCIe for the remaining connections. The server uses an AMD EPYC 7763 processor with 64 cores. We used Ubuntu 22.04 with NVIDIA Driver version 535.171.04 and CUDA 12.2. The software environment includes Python-3.10.12, PyTorch-2.1.2, and a customized version of TensorRT-LLM-0.9.0.dev2024022000.

All the required scripts to execute the experiments and reproduce the results are located in the *tools* directory. In this section, we provide a high-level description of the experimental workflow, however detailed instructions can be found in the *README.md* file under the *tools* directory. We note that our experimental setup included 8 working GPUs, therefore all the scripts have been encoded for this number. If the scripts are executed in a different environment, users must change appropriately the *MAX_WORKERS* and *MAX_WORKERS_NUM* variables across all scripts from 8 to the number of working GPUs that their setup has.

1. Measurement Study, Characterization & Profiling. We provide files used for the measurement study, characterization, and profiling experiments used for this work. Those files are located in the *tools/measurement* directory. Each script collects data for a single experiment (see *tools/README.md for more details*) and all results are stored in the *outputs* directory. Executing those file is straightforward, as they are Python scripts:

```
python3 <script_name.py>
```

2. Profiling & Estimation. We provide the scripts used for the profiling step of `MaverIQ`. Those files are located in *tools/profiling*. To perform the profiling of an individual model, users need to execute the following command:

```
python3 profiler.py --model_name <model_name>
```

Furthermore, we provide the scripts used for the fingerprint-based estimation step of `MaverIQ`. Those files are located in *tools/estimator*. Users do not need to execute them separately as those are invoked automatically through other scripts.

3. Runtime Controller. We provide all the files required to run and serve MaverIQ. Those files are located in *tools/controller*. To execute the end-to-end experiments we provide the *MaverIQ_evaluation.sh* bash script that contains the running commands that need to be executed for reproducing the results.

Users can also run directly MaverIQ's controller by executing:

```
python3 MS_controller.py
```

After initializing the controller, users can interact with it by using the following supported APIs:

- Registering a new model:

```
register <model_name>
```

- Deploying a model for inference:

```
<usr_id> deploy <model_name> <output_length>
  <cost_type> [OPTIONAL]<slo>
  [OPTIONAL]<use_only_float16>
  [OPTIONAL]<deployment_strategy>
  [OPTIONAL]<batch_size> [OPTIONAL]<accuracy>
```

- Serving a query:

```
<usr_id> inference [AUTO-FOR-USER-INPUT]<time>
  <input_text>
```

or

```
<usr_id> prior_inference
  [AUTO-FOR-USER-INPUT]<time> <input_text>
```

Using *inference* will put the new query at the bottom of the serving queue, while *prior_inference* will put it on the top. When using the *prior_inference* API, the queue must be populated by other requests; otherwise, after serving the specified request, the connection to this client will close.

- Removing the deployed model:

```
<usr_id> remove
```

- End Session:

```
Ctrl+C
```

An example usage for deploying and serving a single request using GPTJ-6B is as follows:

```
python3 MS_controller.py
  register gptj-6b
  1 deploy gptj-6b 20 min_cost
  1 inference Hello, World!
  1 remove
  < Ctrl+C >
```

The controller contains commands to automatically build the computational graphs required from TensorRT-LLM. However, to reduce overhead when executing the end-to-end experiments, we advise that users pre-build the graphs in advance. The easiest way to do so is by executing each script twice, once to ensure that the graphs are built and a second time to gather the results for post-execution analysis. Finally, all the results for the end-to-end evaluation can be found within the *outputs/evaluation* directory.

4. Baseline Execution. We provide all the required scripts to execute the baselines for the end-to-end evaluation. We detail the steps that need to be followed for each of the three baselines (i.e., *Accelerate*, *AlpaServe*, and *Vidur*) used in our experiments.

- **Accelerate.** The scripts for the end-to-end evaluation using *Accelerate* are located in the *baselines/Accelerate* directory. We provide the *accelerate_eval.sh* bash script that contains the running commands that need to be executed for reproducing the results. However, users can generate the scaled Azure trace and run individual experiments by executing the following command:

```
python3 accelerate_trace_replay.py --scale_factor
  <scale_factor> --trace [code, conversation]
  --model_list [large, regular, twentyone]
  --mapping_policy [balanced, sequential] --quant
  [16bit, 8bit, 4bit] --mode [trace, memory]
```

  The outputs of those experiments are located in the *outputs/evaluation* directory.
- **AlpaServe.** The files required to run and serve *Alpaserve* and *AlpaServe** are located in *tools/controller*. We provide the *AlpaServe_evaluation.sh* bash script, which contains all the commands executed for the end-to-end experiments using AlpaServe. We note that for the baseline evaluation, the deployment configurations of *AlpaServe* and *AlpaServe** have been integrated within the script *tools/controller/MS_trace_generator_baselines.py*. However, those refer to our 8-GPU setup. If users decide to use a different experimental setup, they need to collect the baseline deployment configurations following the instructions in *baselines* and modify the associated script. Finally, all the results can be found in the *outputs/evaluation* directory.
- **Vidur.** To execute the *Vidur* experiments for the end-to-end evaluation, we provide the modified script located in *baselines/Vidur/vidur_modified_scripts* alongside the executable scripts located in *baselines/Vidur*. Detailed instructions for collecting the required data can be found in the associated README file under *baselines/Vidur*. To collect the required profiling outputs for comparison, users must execute the following commands:

```
# Step I. Profiling using Vidur (inside
  /workspace/Vidur)
# Collect MLP and ATTN information for all models
  and GPU utilization metrics (in the Docker
  container with NVIDIA-SMI support):
python vidur_profiling_cost.py

# Collect communication overheads (in the Docker
  container without the NVIDIA-SMI support):
python vidur_profiling_cost_GPU.py

# Step II. Accuracy estimation using Vidur (inside
  /workspace/Vidur/vidur)
python /workspace/Vidur/vidur_accuracy.py | tee
  /workspace/Vidur/vidur_accuracy_log.txt
```

  The results of this experiment can be found in the *outputs/evaluation/profiling* directory.

Artifact Analysis (incl. Outputs)

After executing all the experiments and collecting the associated results, users can utilize the Jupyter Notebook *tools/artifacts-plotting/Plotting_Functions.ipynb* to generate the figures that are presented in the paper. Although the figures presented in the paper refer to our experimental setup, similar trends should be visible regardless of the underlying hardware at use. More specifically, Figures 2,3,4 refer to the *Characterization Study* and should show the effect of parallelism, unbalanced pipeline parallelism, and quantization on performance metrics. Parallelism configurations create a diverse tradeoff space across performance metrics, unbalanced pipeline parallelism does not affect latency and memory consumption across different layer partition schemes, and quantization impacts serving latency in a distinct way for each LLM. Figures 7 & 9 refer to `MaverIQ`'s profiling methodology, which is more accurate and cost-efficient compared to baselines. Finally, Figures 10-14, show the end-to-end results, where `MaverIQ` outperforms the baselines across multiple traces, rate scales, and workloads.