ReaLLM: A Trace-Driven Framework for Rapid Simulation of Large-Scale LLM Inference

Huwan Peng, Scott Davidson, C.-J. Richard Shi, Michael Taylor
University of Washington, Seattle, USA
{hwpeng, stdavids, cjshi, profmbt}@uw.edu

***Abstract*—As Large Language Models (LLMs) continue to scale, optimizing their deployment requires efficient hardware and system co-design. However, current LLM performance evaluation frameworks fail to capture both chip-level execution details and system-wide behavior, making it difficult to assess realistic performance bottlenecks. In this work, we introduce ReaLLM, a trace-driven simulation framework designed to bridge the gap between detailed accelerator design and large-scale inference evaluation. Unlike prior simulators, ReaLLM integrates kernel profiling derived from detailed microarchitectural simulations with a new trace-driven end-to-end system simulator, enabling precise evaluation of parallelism strategies, batching techniques, and scheduling policies. To address the high computational cost of exhaustive simulations, ReaLLM constructs a precomputed kernel library based on hypothesized scenarios, interpolating results to efficiently explore a vast design space of LLM inference systems. Our validation against real hardware demonstrates the framework's accuracy, achieving an average end-to-end latency prediction error of only 9.1% when simulating inference tasks running on 4 NVIDIA H100 GPUs. We further use ReaLLM to evaluate popular LLMs' end-to-end performance across traces from different applications and identify key system bottlenecks, showing that modern GPU-based LLM inference is increasingly compute-bound rather than memory-bandwidth bound at large scale. Additionally, we significantly reduce simulation time with our precomputed kernel library by a factor of 6× for full-simulations and 164× for workload SLO exploration. ReaLLM is open-source and available at https://github.com/bespoke-silicon-group/reallm.**

***Index Terms*—Large Language Models (LLM), Hardware Accelerator, Simulation Framework**

I. Introduction

Eight years have passed since the publication of the original transformer model [1], and since then Large Language Models (LLMs) have continued to have a profound impact on artificial intelligence, driving advancements in conversational AI [2], [3], code generation [4], and multimodal content creation [5], [6]. With the continued scaling of LLMs, performance gains have followed established scaling laws [7]. However, this scaling has been accompanied by an exponential growth in computational resource demands, raising concerns about scalability, cost-efficiency, and environmental sustainability. Consequently, optimizing LLM inference deployments through effective hardware and system co-design is increasingly essential.

Despite significant progress in LLM hardware accelerators, a substantial gap remains between theoretical peak performance and realized system-level efficiency. Traditional accelerator studies often focus on chip-level metrics such as FLOPS and DRAM bandwidth, neglecting crucial system-level factors that directly influence service-level objectives (SLOs) like time-to-first-token (TTFT) and time-between-tokens (TBT). Achieving high throughput and low latency in large-scale LLM inference requires the orchestration of parallelism strategies (data, tensor, pipeline, context, expert) [8]–[11], system-level optimizations (mixed continuous batching) [12], [13], efficient inter-device communication, and optimized chip-level kernel execution. Accurately bridging the gap between chip-level performance and system-level SLOs is essential for designing next-generation AI accelerators and scalable LLM-serving architectures.

LLM inference introduces complex interactions between hardware and system execution, making accurate system-level performance modeling challenging. Existing LLM hardware simulators typically focus either on chip-level simulations or system-level modeling, failing to capture key execution dynamics. Many approaches do not provide fine-grained kernel performance evaluations, instead relying on analytical models that estimate performance without simulating kernel execution at the hardware level. Additionally, full execution graph-based system simulation is missing from most frameworks, despite the fact that LLM inference spans multiple devices, requiring an integrated evaluation of parallelism, batching, scheduling, and communication.

Another challenge is the scalability of large-scale design space exploration. The number of devices, parallelism strategies, and system optimizations creates a massive combinatorial search space, making naive brute-force evaluations infeasible. Without an efficient methodology for evaluating design points, it is difficult to find optimal configurations for LLM hardware-software co-design.

To address these limitations, we introduce **ReaLLM**, a novel trace-driven simulation framework that uniquely integrates chip-level and system-level evaluations, enabling end-to-end LLM system performance modeling. *ReaLLM operates in three distinct stages.* The first stage, hypothesis-based kernel library construction, involves hypothesizing all feasible parallelism, batching, and scheduling strategies. This information is then used to generate a list of all unique kernels that will be executed such as matrix multiplication, layer normalization and softmax. Next, ReaLLM precomputes profiling data for each hypothesized kernel using an optimized chip-level kernel simulator. This simulator determines the best partitioning, loop blocking, and execution strategies for each kernel, storing

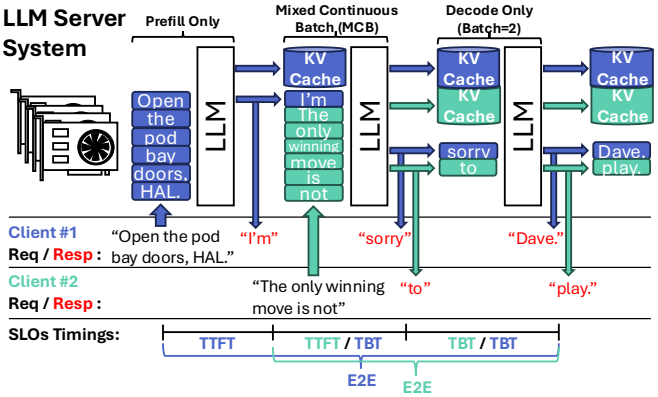

Fig. 1. High-level overview of a LLM inference serving system handling multiple client requests arriving at different times with labels for the 3 service level objectives (SLOs) for each client. Shown are 3 iterations through the LLM. First iteration is a prefill iteration for client 1. Second iteration uses mixed continuous batching (MCB) to combine the decode iteration for client 1 with a prefill iteration for client 2. The final iteration is a decode iteration for both clients forming a normal batched decode iteration.

results in a lookup table. Finally, ReaLLM generates traces that reflect real-world execution dynamics and simulates the execution graph at the system level. This flow allows rapid evaluation of service-level objectives across different system configurations, identifying the best performing design point for a given workload and hardware setup. ReaLLM's key innovations include:

- *Integrated Chip-System Modeling:* ReaLLM bridges the gap between chip-level kernel performance and system-level SLOs, providing a holistic view of LLM inference.
- *Precomputed Hypothesized Kernel Profiling:* ReaLLM significantly reduces simulation overhead by precomputing kernel performance with an optimized chip-level simulator, enabling rapid design exploration.
- *Trace-Driven System Simulation:* ReaLLM generates realistic execution traces that capture real-world execution dynamics, facilitating accurate system-level simulation and SLO evaluation.
- *Efficient Design Space Exploration:* ReaLLM's methodology enables efficient exploration of the vast hardware-software co-design space, identifying the best performing configurations for LLM serving.

II. Background and Related Work

A. Generative LLM Inference

Generative Large Language Models (LLMs) are constructed from stacked Transformer decoder layers [1], [14], [15], using *causal self-attention* to perform autoregressive generation. This process is typically segmented into two phases: *prefill* and *decode*. The prefill stage, which processes the entire input sequence to produce the initial output token, is often computationally intensive. Conversely, the decode stage generates subsequent tokens iteratively, leveraging cached key-value (KV) projections to minimize redundant computations, resulting in lower operational intensity.

To overcome low operational intensity, systems attempt to batch multiple users and processes them simultaneously.

TABLE I
Comparison of ReaLLM with Existing LLM Performance Evaluation Frameworks

Feature	GenZ [18]	Optimus [19]	LLMCompass [20]	ReaLLM
Micro Arch-Level	✗	✗	✓	✓
Non-Linear Kernel	✗	✗	✓	✓
Parallelism Exploration	✓	✓	✗	✓
Scheduling Exploration	✗	✗	✗	✓
Trace Generation	✗	✗	✗	✓
Trace Driven Simulation	✗	✗	✗	✓
SLO Analysis	✗	✗	✗	✓

Mixed continuous batching (MCB) [12] is a batching technique that concurrently manages ongoing decode operations and seamlessly integrates new prefill tasks into the processing pipeline, shown in Figure 1. To mitigate potential delays associated with long input prompts, the prefill phase can be partitioned into smaller, discrete chunks [13], thereby reducing latency impact on concurrent decode operations.

The memory and computational demands of LLM inference require distributed execution on multiple hardware devices. Common parallelism strategies include tensor, pipeline, sequence, and expert parallelism [8], [10], [16], [17], each presenting distinct performance trade-offs. For instance, tensor parallelism effectively reduces per-device memory requirements but introduces inter-device communication overhead.

Service Level Objectives (SLOs) are key metrics for evaluating the performance of LLM inference systems. The primary SLOs include time-to-first-token (TTFT) and time-between-tokens (TBT). TTFT measures the latency from when an input prompt is received to the generation of the first output token. TBT measures the latency between consecutive token generations. Together, TTFT and TBT contribute to the end-to-end latency (E2E) of a given request, annotated in Figure 1. These SLOs are critical for ensuring a responsive user experience in real-world LLM applications.

B. LLM Hardware Simulation and Challenges

Accurately modeling LLM inference requires both chip-level and system-level performance analysis. Existing simulation frameworks often focus on only one aspect, leading to incomplete performance evaluations.

Chip-level simulators like LLMCompass [20] offer detailed accelerator performance modeling but lack comprehensive system-level execution analysis. These frameworks focus on low-level kernel execution details, without capturing the entire end-to-end inference pipeline. Although LLMCompass models certain parallelism strategies, it primarily emphasizes individual kernel execution and does not capture the complexities of the complete system, limiting its applicability for real-world LLM serving scenarios.

Other works such as GenZ [18] and Optimus [19] provide system-level LLM inference simulation. These models approximate the impact of compute, memory, and interconnect bandwidth but suffer from some key limitations. They lack kernel-level accuracy, as both use simple linear models based on peak FLOPS, leading to imprecise latency predictions. Furthermore, they focus only on matrix multiplication kernels and ignore some other kernels such as layer normalization and softmax, which could have significant latency. They also fail

to model various widely used system-level optimizations such as mixed continuous batching [12].

ASTRA-sim 2.0 [21] is another more generalized machine learning system simulator. However, their work emphasizes complexities critical to large-scale training, such as sophisticated multi-dimensional network modeling for extensive inter-node communication and disaggregated memory. ReaLLM's focus is specifically in the LLM inference domain and challenges with LLM inference when deployed in different application environments. This includes a dedicated focus on the dynamic nature of inference requests and inference-specific SLOs, requiring advanced batching and scheduling strategies to handle varying input characteristics efficiently while providing a comprehensive framework to model these scenarios. We believe this enables a more efficient hardware and software co-design environment for exploring potential next-generation AI accelerators.

III. ReaLLM Simulation Framework

A major challenge in applying accurate kernel simulation to end-to-end LLM system modeling is slow simulation speed. For example, simulating a single inference pass with LLMCompass [20] can take several minutes, mainly due to the long simulation time of *Matmul* kernels. Accurate Matmul simulation requires exploring a vast mapping and scheduling space, including L2 and L1 tiling, loop ordering, and systolic array dataflows, etc. The number of possible mapping strategies for a single Matmul operation can reach into millions. While LLMCompass [20] applies heuristics to reduce this search space, simulating each Matmul still takes a minute.

This speed is impractical for system-level simulation, as it dramatically increases the number of required kernel evaluations. Table II highlights the order of magnitude of Matmul kernels that need to be simulated. An LLM contains approximately 10 distinct Matmul kernels. Considering variations in input request rates, batching strategies, and different parallelism configurations (data, tensor, pipeline, context, expert, etc.), the number of Matmul kernels grows exponentially. Furthermore, with dynamic prompt lengths during prefill and context lengths during decode, modern LLMs with context lengths up to 128K introduce over 10^5 variations. As a result, the total number of Matmul simulations required for a complete system evaluation can reach 10^9, which is computationally prohibitive given that each simulation takes minutes.

To overcome this challenge, ReaLLM adopts a three-phase simulation framework: *kernel library construction*, *kernel simulation* and *system simulation*, as shown in Figure 2. The *kernel library construction* phase, shown on the left side of Figure 2, takes as input an LLM model and hardware description. It systematically extracts all unique kernels based on batch sizes, parallelism configurations, and context lengths,

TABLE II
Order of Magnitude of Matmul Kernel Variations Given Different Input Factors.

In a LLM	Batch Sizes	Parallelisms	Context Lengths	Total
10	10	10^2	10^5	10^9

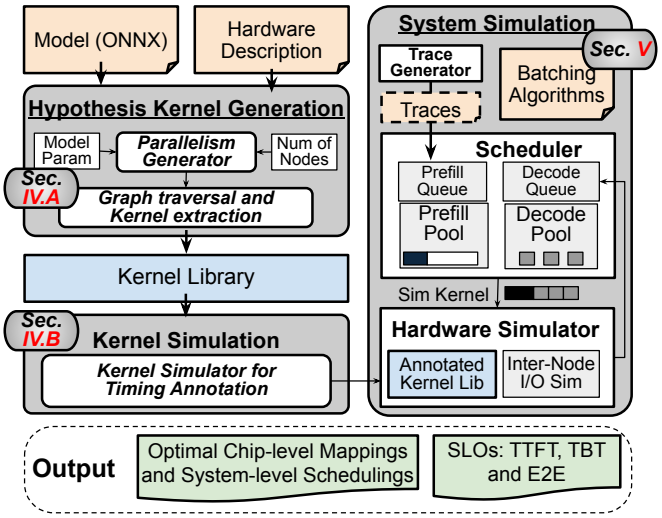

Fig. 2. Overview of the ReaLLM framework.

storing them in a kernel library. These kernels are then passed to the *kernel simulation* phase where each kernel is evaluated using a kernel-level simulator, such as LLMCompass, and the results are stored in a kernel performance table. To further reduce simulation overhead, continuous variables such as context length are interpolated between key simulation points rather than simulating every possible value.

The *system simulation* phase, shown on the right side of Figure 2, is driven by execution traces, which can be user-provided or generated by ReaLLM to mimic real-world workload dynamics. The built-in scheduler simulates batching and scheduling strategies such as continuous batching and mixed continuous batching. It then generates execution traces, using the kernel performance table to retrieve kernel latencies while also simulating inter-node communication overhead. The final output includes key system-level performance metrics such as time-to-first-token (TTFT), time-between-tokens (TBT), and end-to-end latency (E2E). Additionally, ReaLLM identifies the best performing chip-level kernel mappings and system-level scheduling strategies.

IV. Kernel Library Construction

The first phase of ReaLLM is the kernel library construction, where the framework identifies all unique kernels used in LLM serving (hypothesized kernel generation) and simulates their latencies for subsequent system-level end-to-end simulation (kernel simulation).

A. Hypothesis Driven Kernel Generation

To generate all possible kernel sizes, users must provide the target models and the number of devices in the system. Models can include one or multiple LLMs in ONNX format [22], a widely used graph-based representation for neural networks. Using ONNX as an input format enhances ReaLLM's compatibility and flexibility, supporting common operators such as *Matmul*, *Softmax*, *LayerNorm*, and *GELU*, etc.

The tool parses the ONNX graph, counts the occurrences of each kernel, and extracts their shapes. Kernel dimensions

TABLE III
MATMUL KERNEL SIZE FOR LLAMA-LIKE LLM (TOP) AND MULTI-LATENT ATTENTION (BOTTOM). D, T, C ARE THE SIZES OF DATA, TENSOR, AND CONTEXT PARALLELISM.

Matmul	$\mathbf{B}_1$	$\mathbf{B}_2$	**M**	**K**	**N**	**Collective**
q_proj	$\frac{batch}{D}$	1	$\frac{l_{in}}{C}$	d_m	$d_h \frac{n_h}{T}$	
kv_proj	$\frac{batch}{D}$	1	$\frac{l_{in}}{C}$	d_m	$d_h \frac{n_{kv}}{T}$	
q_k	$\frac{batch}{D}$	$\frac{n_{kv}}{T}$	$\frac{n_h}{n_{kv}} \frac{l_{in}}{C}$	d_h	l_{ctx}	SR($\frac{batch n_h l_{in} d_h}{DTC}$)
s_v	$\frac{batch}{D}$	$\frac{n_{kv}}{T}$	$\frac{n_h}{n_{kv}} \frac{l_{in}}{C}$	l_{ctx}	d_h	SR($\frac{batch n_h l_{in} l_{ctx}}{DTC}$)
o_proj	$\frac{batch}{D}$	1	$\frac{l_{in}}{C}$	$d_h \frac{n_h}{T}$	d_m	AR($\frac{batch l_{in} d_m}{DC}$)
mlp_gate	$\frac{batch}{D}$	1	$\frac{l_{in}}{C}$	d_m	$\frac{d_{ffn}}{T}$	
mlp_up	$\frac{batch}{D}$	1	$\frac{l_{in}}{C}$	$\frac{d_{ffn}}{T}$	d_m	
mlp_dn	$\frac{batch}{D}$	1	$\frac{l_{in}}{C}$	$\frac{d_{ffn}}{T}$	d_m	AR($\frac{batch l_{in} d_m}{DC}$)
q_k_1	$\frac{batch}{D}$	$\frac{n_h}{T}$	$\frac{l_{in}}{C}$	d_h	d_c	SR($\frac{batch n_h l_{in} d_h}{DTC}$)
q_k_2	$\frac{batch}{D}$	1	$\frac{n_h}{T} \frac{l_{in}}{C}$	d_c	l_{ctx}	SR $\frac{batch n_h l_{in} d_c}{DTC}$
q_k_pe	$\frac{batch}{D}$	1	$\frac{n_h}{T} \frac{l_{in}}{C}$	d_h^R	l_{ctx}	SR($\frac{batch n_h l_{in} d_h^R}{DTC}$)
s_v_1	$\frac{batch}{D}$	1	$\frac{n_h}{T} \frac{l_{in}}{C}$	l_{ctx}	d_c	SR($\frac{batch n_h l_{in} l_{ctx}}{DTC}$)
s_v_2	$\frac{batch}{D}$	$\frac{n_h}{T}$	$\frac{l_{in}}{C}$	d_c	d_h	SR($\frac{batch n_h l_{in} d_c}{DTC}$)

*AR=AllReduce, SR=SendRecv

depend on input factors such as batch size, input length (for prefill), and context length (for decode). ReaLLM includes a built-in shape inference engine that propagates these dimensions across different input configurations.

Since LLM systems often span multiple nodes, large kernels must be partitioned and distributed. The partitioned kernel size depends on the chosen parallelism strategy, which is an active area of research. ReaLLM supports commonly adopted parallelism strategies in state-of-the-art LLM systems, including data parallelism, tensor parallelism, pipeline parallelism, context parallelism, and expert parallelism. Given the model hyperparameters and system constraints, the *parallelism generator* generates all valid parallelism configurations. It ensures that the total number of nodes matches the product of all parallelism dimensions and verifies divisibility constraints for tensor, pipeline, and expert parallelism based on hyperparameters such as the number of heads, layers, and routed experts.

Given batch size, input/context lengths, and parallelism configurations, ReaLLM calculates and hypothesizes all possible sizes for each kernel. The top part of Table III lists Matmul kernels for a Llama-like [23] LLM, which uses group-query attention and gated linear units. Each Matmul operation is represented as $(B_1, B_2, M, K) \times (B_2, K, N) = (B_1, B_2, K, N)$. l_{in} denotes the input sequence length, which is the prompt length for prefill and 1 for decode. l_{ctx} denotes the context length, which is the prompt length for prefill and past context length for decode. All divisions in Table III use ceiling division to ensure the identification of the system's critical path. The table also lists collective operations required for certain parallelism strategies. Context parallelism requires SendRecv operations for `q_k` and `s_v` since each node must receive the complete query and scores. Tensor parallelism requires AllReduce operations for `o_proj` and `mlp_dn` to aggregate partial results. The bottom section of Table III presents Matmul kernels for multi-latent attention, which introduces additional smaller kernels. Low-rank adaptation is applied to key and value projections, compressing them into a lower-dimensional space d_c.

By iterating over all input factors, ReaLLM constructs a complete kernel library for further simulation. Pipeline and expert parallelism do not change kernel sizes but affect kernel execution times, which is accounted for in system simulation.

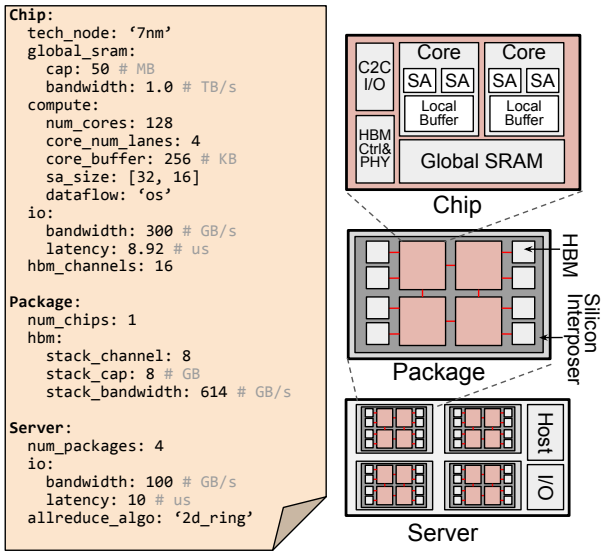

Fig. 3. Abstract hardware description example. ReaLLM supports flexible chip, package and server designs.

B. Kernel Latency Simulation

Once the kernel library is generated, the second step is latency simulation on the target hardware. The ReaLLM kernel simulator builds upon LLMCompass [20], an open-source hardware evaluation framework for LLMs. Several enhancements have been made to improve its suitability for modern LLM system simulation. ReaLLM extends support for additional attention mechanisms, including multi-query attention and multi-latent attention. Additional operators such as SiLU activation and element-wise multiplication for gated linear units have been added to improve compatibility with a wider range of LLM architectures. Furthermore, to accelerate Matmul simulation, multiprocessing is employed to evaluate multiple mappings in parallel.

Abstract Hardware Description. Figure 3 illustrates an example of ReaLLM's abstract hardware description in YAML format, following a structure similar to LLMCompass. A system consists of multiple devices connected through a device-to-device interconnect (e.g., NVLink, TPU Link). Each device contains multiple cores, a shared global buffer (L2), and off-chip memory. Each core has a local shared memory (L1) and compute units including vector units, systolic arrays, and registers (L0). This flexible template supports a wide range of ML accelerators, including GPUs (NVIDIA, AMD) and TPUs.

Large Simulation Space. In such hardware architectures, the two matrices involved in Matmul operations must traverse multiple memory hierarchies, from main memory to L2, L1, and finally L0 registers. Optimizing tiling sizes at each level is crucial to maximizing data reuse and minimizing memory access overhead. Additionally, factors such as loop ordering at L2 and L1, potential L2 double buffering, and systolic

array dataflow further expand the search space for optimal mappings. Due to these complexities, the number of possible mappings for a single Matmul operation can reach millions, significantly slowing down simulation. To improve efficiency, it is critical to reduce the number of Matmul kernels that require full simulation. As shown in Table II, the largest source of variation comes from context lengths l_{in} and l_{ctx}, introducing approximately 10^5 possibilities, which continue to grow as the maximum context length increases. To mitigate the impact of long prompts, most LLM systems adopt mixed continuous batching, which splits longer sequences into smaller chunks, typically ranging from 128 to 2048 tokens. In the decode phase, l_{in} is always 1, while l_{ctx} represents the past context length. Consequently, we focus our optimization efforts on Matmul kernels `q_k` and `s_v` in the prefill stage, where changes in l_{ctx} are the primary source of variation.

Matmul Kernel Interpolation. We observe that when only one dimension of a Matmul kernel changes, the relationship between latency and input size follows a predictable trend. Instead of simulating every possible configuration, we sample a subset of key points and interpolate intermediate values.

Figure 4 (left column) illustrates Matmul latencies as we sweep the single dimension N or K, with red dots marking sampled simulation points. The sampling points are logarithmically spaced to capture variations efficiently. As expected, the latency growth rate gradually increases before stabilizing into a linear trend. To determine the best interpolation strategy, we compared linear interpolation and third-degree polynomial interpolation. Figure 4 (right column) shows the relative error between interpolated and simulated values for randomly selected points. The linear interpolation method achieved an average error of 0.90% and 3.63%, significantly lower than the error of polynomial interpolation. Based on these results, we adopt linear interpolation for l_{ctx}, selecting key values in logarithmic steps and interpolating all intermediate points. This significantly reduces simulation overhead while maintaining high accuracy in kernel latency estimation, improving the efficiency of kernel library construction.

V. Trace Driven End-to-End System Simulation

A. Trace Generation

The right side of Figure 2 provides an overview of our trace-driven LLM system simulator. If a user does not provide a trace, ReaLLM includes a built-in trace generator that can synthesize traces for real workloads or based on predefined input-to-output ratios. To generate realistic traces that reflect the dynamics of coding and conversational tasks, two of the most common LLM applications, we utilize production traces from the Azure LLM Inference Dataset 2023 [24]. Figure 5 shows the distribution of context lengths (sum of input and output tokens) and input-to-output token ratios for coding and conversational tasks. Notably, conversation tasks tend to have a much lower input-to-output ratio compared to coding tasks. This indicates that conversation workloads typically involve shorter prompts but generate longer responses, resulting in a higher proportion of decode tasks.

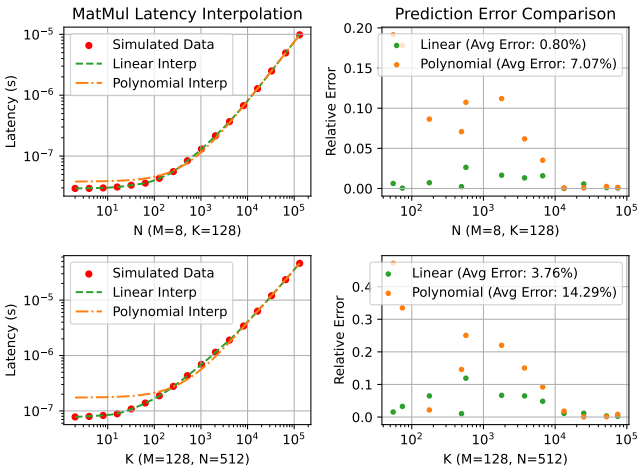

Fig. 4. Matmul latency interpolation comparison. ReaLLM adopts linear interpolation, which achieves lower error rates than polynomial interpolation.

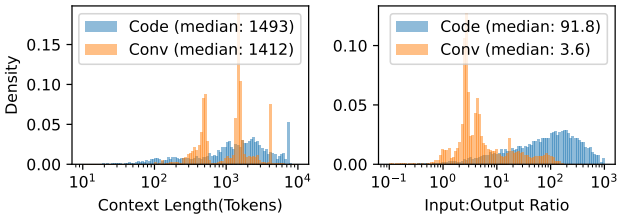

Fig. 5. Context length and input-output ratio of two traces taken from Azure LLM inference services [24]. Conversation-based tasks have a much lower input-output ratio.

B. Task Scheduler

To support various dynamic batching and scheduling strategies, we developed a task scheduler that processes traces and generates simulation tasks for the hardware model.

Initially, all incoming requests are placed in the prefill queue with annotated arrival times. The scheduler continuously monitors requests in both the prefill and decode queues, generating execution tasks for the hardware simulator. Depending on the selected batching and scheduling algorithms, the scheduler may group prefill and decode tasks into a single execution batch to optimize resource utilization. For example, in a prefill-prioritized continuous batching strategy, the scheduler prioritizes prefill tasks, ensuring they are fully processed before scheduling any decode tasks. In contrast, when chunked mixed continuous batching is adopted [13], long prefill tasks are split into smaller chunks and batched with decode tasks, improving system utilization and overall throughput.

In the simulation, each execution task is represented by an integer prefill length and an array of integers denoting the context lengths of all decode tasks. Once the hardware simulation completes, all associated requests are updated, and any unfinished requests are placed back into the decode queue for the next iteration.

TABLE IV
SUPPORTED ALLREDUCE ALGORITHMS.

Algorithms	Time
Ring AR	$2(p-1)\alpha + 2\frac{p-1}{p}N\beta$
2-D Ring AR [26]	$4(\sqrt{p}-1)\alpha + 2\frac{\sqrt{p}-1}{\sqrt{p}}N\beta$
Two Tree AR [27]	$4\log(p)\alpha + 2N\beta + 4\sqrt{2\log(p)\alpha N\beta}$
Two Tree BC [27]	$2\log(p)\alpha + N\beta + 2\sqrt{2\log(p)\alpha N\beta}$
Hierarchical AR [28]	$LocalAR + GlobalAR + LocalBC$

*AR=Allreduce, BC=Broadcast

C. Hardware Simulator

The hardware simulator processes these workloads by traversing the execution graph, retrieving kernel sizes, and querying the precomputed kernel library for latency values. If a kernel size is not found in the library, linear interpolation is applied, as discussed in the previous section. This approach ensures rapid execution, as no real-time simulation is performed.

A standard communication model is used to estimate I/O latency. The time required to transmit an N-byte message between any two nodes is modeled as $\alpha + N\beta$, where α represents the per-message latency (independent of message size), and β denotes the per-byte transmission time. Table IV lists the communication algorithms supported in ReaLLM, including ring-based, tree-based, and hierarchical allreduce algorithms. These methods have been widely adopted in large-scale TPU and GPU deep learning systems [25]. The table includes the time required to perform communication operation on N-byte tensors among p nodes.

D. Results Output

The system simulation records the arrival time and generation time of each output token for every request. From this data, ReaLLM measures Service Level Objectives (SLOs) for different Service Level Agreement (SLA) thresholds, including P50, P90, and P99 latencies, providing a comprehensive evaluation of system performance. ReaLLM measures three main metrics: time to first token (TTFT), time between tokens (TBT), and end-to-end (E2E) latency. In addition to SLO metrics, ReaLLM outputs the best performing chip-level kernel mappings and system-level scheduling strategies, including parallelism configurations and dynamic batching policies that achieve the best SLO performance.

VI. EVALUATION

A. Validation Against Real Hardware

To validate ReaLLM's accuracy, we compare the predicted kernel latencies and end-to-end request latencies against real measurements on NVIDIA A100 and H100 systems.

Kernel-Level Validation. To assess the accuracy of ReaLLM at the kernel level, we compare predicted versus measured latencies for key LLM inference operations on a NVIDIA A100 GPU. Figure 6 shows the latency of Matmul operations across different input sizes, demonstrating that ReaLLM's predictions align closely with real execution times. This high fidelity ensures that kernel-level estimations in

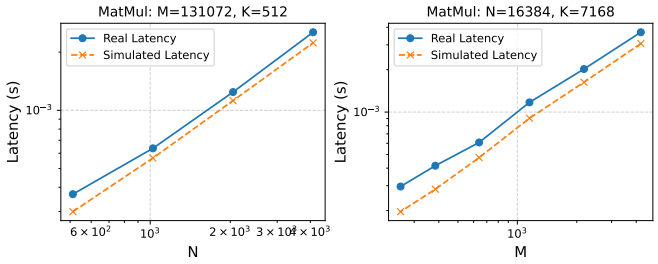

Fig. 6. Validation of kernel latency predictions on A100. Each subfigure compares real and simulated latencies for Matmul at different input sizes.

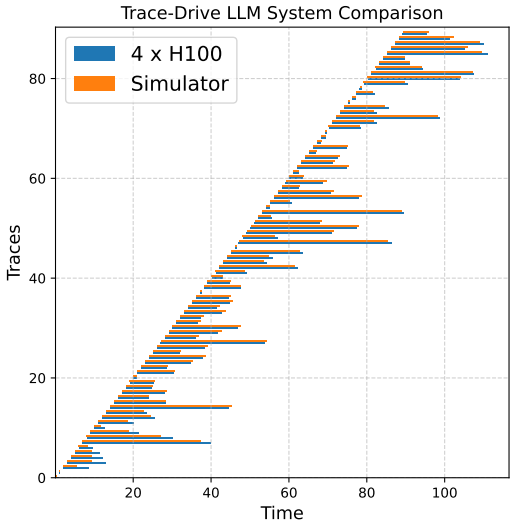

Fig. 7. Comparison of simulated and real end-to-end request latencies for LLaMA-70B inference on a four-H100 system.

ReaLLM provide precise performance insights, making it a reliable tool for evaluating large-scale LLM inference systems.

End-to-End Latency Validation. Beyond kernel-level validation, we assess end-to-end inference accuracy by comparing ReaLLM's simulated latencies for LLaMA-70B running on four H100 GPUs against real-world traces (Figure 7). The results indicate that ReaLLM predicts the end-to-end time (E2E) with an average error of 9.1% across 90 test traces. Notably, most of the early differences arise from transient system warm-up effects and variations in initial scheduling, while later traces have improved accuracy. This strong alignment with real hardware confirms the robustness and reliability of ReaLLM's system-level simulation. Furthermore, ReaLLM's trace-driven scheduling and dynamic batching models effectively adapt to fluctuating workloads, accurately reflecting real-world deployment scenarios. By incorporating execution-aware scheduling strategies, ReaLLM ensures that its predictions remain highly relevant for large-scale LLM inference studies.

B. Bottleneck Analysis and Performance Scaling

To identify performance bottlenecks and potential optimizations, we analyze two models: Llama3-70B [23] on an 8-node system and DeepSeek v3 [15] on a 32-node system. The baseline system models H100 style GPUs, while alternative configurations explore increased DRAM bandwidth,

TABLE V
SLOs FOR EVALUATION. E2E IS SET TO BE TTFT PLUS THE TIME TO GENERATE A NUMBER OF TOKENS WHILE MEETING TBT.

Workload	TTFT	TBT	E2E
Code	400 ms	50 ms	12.9 s (250 tokens generated)
Conversation	200 ms	50 ms	25.2 s (500 tokens generate)

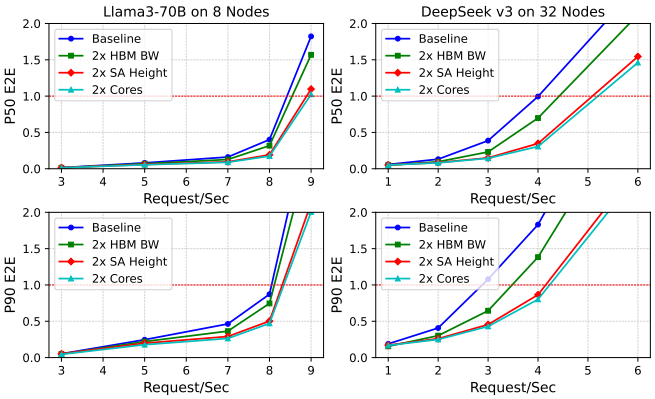

Fig. 8. Latency metrics across input loads of Llama3-70B on 8 nodes (left) and DeepSeek v3 on 32 nodes (right) systems with different architectures.

greater systolic array (tensor core) height, and additional compute cores (SMs). All configurations maintain consistent system-level settings, including node count, interconnect links, topology, and the dynamic batching strategy. We specifically leverage chunked mixed continuous batching with a prefill block size of 2048, which improves operational intensity for decode tasks. Llama3 employs tensor parallelism, whereas DeepSeek v3 uses expert parallelism. As input loads to an LLM system fluctuate over time, a crucial metric is whether the system can maintain SLOs under high request rates. To explore this, our trace generator produced traces for both coding and conversation applications at various input request rates, sampling from Figure 5.

Figure 8 presents P50 and P90 end-to-end latencies across different input loads for LLaMA3-70B (left) and DeepSeek v3 (right) inference. The x-axis represents input load, while the y-axis shows E2E latency normalized to the SLO thresholds in Table V. Results indicate that increasing tensor core height or core count significantly improves performance, whereas boosting HBM bandwidth only provides limited benefits. This suggests that modern LLM inference systems are increasingly compute-bound rather than memory-bandwidth-bound, largely due to the effectiveness of advanced batching techniques.

Additionally, we evaluate Llama3-70B on conversation workload in Figure 9. We observe that the conversation workload experiences an earlier latency increase as input request rates grow. This is because conversation-based tasks typically require generating more tokens per request, causing requests to remain in the system for longer durations. Consequently, conversation applications may require greater hardware resources compared to coding applications to maintain similar SLOs.

C. Impact of Mapping Strategies

ReaLLM enables comprehensive mapping exploration, including fine-grained chip-level kernel mapping and system-level batching strategies and parallelism configurations, all of which are crucial for optimizing large-scale LLM inference. Figure 10 presents the execution cycles for Matmul kernels across different mapping strategies, including loop blocking, loop ordering, and double buffering. Given that each Matmul operation can have millions of possible mappings, the selection of execution order in critical. The figure illustrates that choosing the best loop ordering strategy can reduce latency by an order of magnitude, emphasizing the significance of micro-architectural-level kernel simulation in performance optimization.

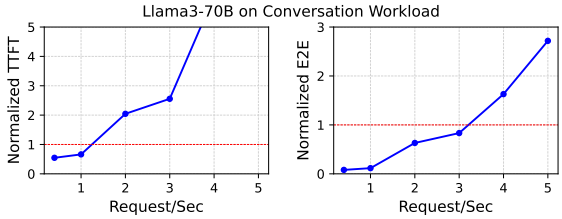

Fig. 9. TTFT and E2E across input loads of Llama3-70B for conversation applications on a 32-node system with different architectures.

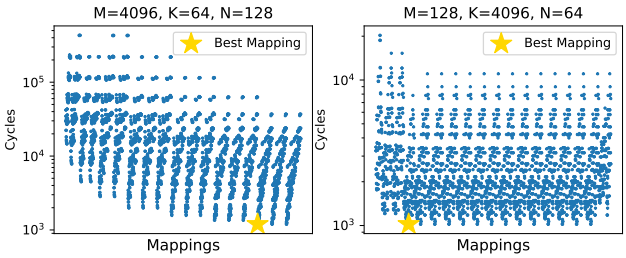

Fig. 10. Matmul kernel cycles on different mapping strategies.

D. Scalability and Efficiency Gains

We evaluate ReaLLM's impact on simulation efficiency by comparing its performance against a baseline approach that relies exclusively on a kernel simulator like LLMCompass. As shown in Figure 11, simulating traces with hundreds of requests and context lengths extending to thousands of tokens requires the baseline to perform approximately 10^4 Matmul simulations, resulting in an estimated runtime of 4,570 minutes. In contrast, ReaLLM drastically reduces this overhead by identifying 1,600 key kernels and precomputing their latencies in 729.6 minutes. Once the kernel library is constructed, trace-driven simulation takes only 27.9 minutes, leading to a 164× speedup in trace execution. Since kernel construction and precomputation is a one-time process when performing workload SLO explorations, this optimization significantly accelerates exploration while maintaining high fidelity in performance modeling.

By leveraging precomputed kernel latencies and an efficient trace-driven simulation methodology, ReaLLM transforms large-scale LLM system evaluation from an intractable computational problem into a practical and scalable process, enabling rapid architectural exploration and optimization.

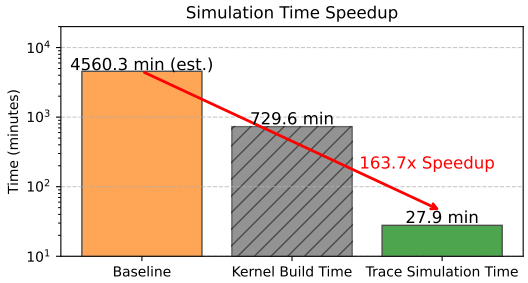

Fig. 11. ReaLLM achieves a 6× speedup in simulations with a 164× speedup in workload SLO exploration simulations compared to the baseline kernel simulator by leveraging precomputed kernel reuse.

VII. Conclusion

As LLMs continue to scale, achieving high-performance and cost-efficient inference remains a critical challenge. Traditional accelerator studies often neglect essential system-level execution dynamics, resulting in a gap between theoretical hardware capabilities and real-world inference efficiency. To address this, we introduced ReaLLM, a novel trace-driven simulation framework integrating detailed chip-level kernel modeling with comprehensive system-level evaluation. By leveraging precomputed kernel profiling and trace-driven scheduling, ReaLLM achieves a 6× speed up in simulation time with a further 164× reduction in workload SLO exploration time, facilitating rapid and accurate exploration of the extensive LLM hardware-software co-design space. Our results demonstrate that ReaLLM accurately captures real-world system behaviors, enabling architects to pinpoint system bottlenecks, optimize parallelism and batching strategies, and make informed hardware design decisions. As a practical tool for researchers, ReaLLM empowers the design of next-generation ASIC architectures specifically tailored for large-scale LLM inference.

Acknowledgments

This work was supported in part by ACE and CHIMES, two of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF Award 2118628. A special thanks to Shuaiwen Leon Song for his helpful discussions and insights.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *arXiv:1706.03762 [cs]*, 2017.

[2] OpenAI, "Introducing ChatGPT." https://openai.com/blog/chatgpt, 2022. Accessed May 2025.

[3] Google DeepMind, "Gemini." https://deepmind.google/technologies/gemini/, 2025. Accessed May 2025.

[4] GitHub, "GitHub Copilot Your AI Pair Programmer." https://github.com/features/copilot, 2023.

[5] OpenAI, "Sora." https://openai.com/sora/, 2025. Accessed May 2025.

[6] Suno, "Suno — AI Music." https://suno.com/home/, 2025. Accessed May 2025.

[7] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling Laws for Neural Language Models," *arXiv:2001.08361 [cs, stat]*, 2020.

[8] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv:1909.08053 [cs]*, 2020.

[9] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," *arXiv:1811.06965 [cs]*, 2019.

[10] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You, "Sequence Parallelism: Long Sequence Training from System Perspective," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023.

[11] W. Fedus, B. Zoph, and N. Shazeer, "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity," *arXiv:2101.03961 [cs]*, 2021.

[12] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko, and Y. He, "DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference," *arXiv:2401.08671 [cs]*, 2024.

[13] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2024.

[14] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," *arXiv:2005.14165 [cs]*, 2020.

[15] DeepSeek-AI, "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," *arXiv:2501.12948 [cs]*, 2025.

[16] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[17] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently Scaling Transformer Inference," *arXiv:2211.05102 [cs]*, 2022.

[18] A. Bambhaniya, R. Raj, G. Jeong, S. Kundu, S. Srinivasan, M. Elavazhagan, M. Kumar, and T. Krishna, "Demystifying Platform Requirements for Diverse LLM Inference Use Cases," *arXiv:2406.01698 [cs]*, 2024.

[19] J. Kundu, W. Guo, A. BanaGozar, U. De Alwis, S. Sengupta, P. Gupta, and A. Mallik, " Performance Modeling and Workload Analysis of Distributed Large Language Model Training and Inference ," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2024.

[20] H. Zhang, A. Ning, R. B. Prabhakar, and D. Wentzlaff, "LLMCompass: Enabling Efficient Hardware Design for Large Language Model Inference," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[21] W. Won, T. Heo, S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "ASTRA-sim2.0: Modeling Hierarchical Networks and Disaggregated Systems for Large-model Training at Scale," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

[22] J. Bai, F. Lu, K. Zhang, *et al.*, "Onnx: Open neural network exchange." https://github.com/onnx/onnx, 2019.

[23] LlamaTeam, "The Llama 3 Herd of Models," *arXiv:2407.21783 [cs]*, 2024.

[24] Microsoft, "Azure LLM Inference Trace 2023." https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md, 2024.

[25] S. Jeaugey, "Massively Scale Your Deep Learning Training with NCCL 2.4." https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4, 2019.

[26] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, "Image Classification at Supercomputer Scale," *arXiv:1811.06992 [cs]*, 2018.

[27] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," *Parallel Computing*, 2009.

[28] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi, and X. Chu, "Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes," *arXiv:1807.11205 [cs]*, 2018.