Hybrid JIT–CUDA Graph Optimization for Low-Latency Large Language Model Inference

Divakar Kumar Yadav
Department of Computer Science
University of Wisconsin-Milwaukee
Milwaukee, WI, USA
dkyadav@uwm.edu

Tian Zhao
Department of Computer Science
University of Wisconsin-Milwaukee
Milwaukee, WI, USA
tzhao@uwm.edu

***Abstract*—Large Language Models (LLMs) have achieved strong performance across natural language and multimodal tasks, yet their practical deployment remains constrained by inference latency and kernel launch overhead, particularly in interactive, short-sequence settings. This paper presents a hybrid runtime framework that combines Just-In-Time (JIT) compilation with CUDA Graph execution to reduce launch overhead while preserving runtime flexibility during autoregressive decoding. The framework partitions transformer inference into static components executed via CUDA Graph replay and dynamic components handled through JIT-compiled kernels, enabling asynchronous graph capture and reuse across decoding steps.**

We evaluate the proposed approach on LLaMA-2 7B using single-GPU, batch-size-one inference across prompt lengths from 10 to 500 tokens. Experimental results show that the hybrid runtime reduces Time-to-First-Token (TTFT) by up to 66.0% and achieves lower P99 latency compared with TensorRT–LLM in this regime. These results indicate that hybrid JIT–CUDA Graph execution can effectively reduce inference latency and variance for short-sequence LLM workloads, making it a practical optimization strategy for latency-sensitive AI applications.

***Index Terms*—Large Language Model, CUDA Graph, JIT Compilation, Inference Optimization, GPU Acceleration, Low Latency**

I. Introduction

Large Language Models (LLMs) such as GPT-4 [1], Claude [2], Gemini [3], and Grok [4] have rapidly evolved into core inference engines for conversational, coding, and multimodal AI systems. Despite their strong model capabilities, the real-time deployment of LLMs remains constrained by inference latency, particularly in interactive settings where users generate and consume responses incrementally. Prior work has shown that autoregressive decoding incurs substantial overhead from repeated GPU kernel launches, synchronization events, and host–device coordination [5]–[8]. These costs accumulate across decoding steps and become especially visible in latency-sensitive inference scenarios.

To mitigate inference overhead, modern LLM runtimes employ a combination of compiler- and kernel-level optimizations, including operator fusion [9]–[11], quantization [12], [13], and specialized attention kernels such as FlashAttention [14], [15]. Production-oriented frameworks such as TensorRT–LLM [16] and FasterTransformer [17] combine these techniques to improve throughput and reduce memory overhead. Nevertheless, even highly optimized pipelines continue to incur non-trivial latency from repeated kernel dispatch and dynamic tensor management during autoregressive decoding, particularly for small batch sizes and interactive use cases.

A. Inference Length

Interactive LLM applications, including conversational agents, code assistants, and decision-support systems, are typically dominated by short-to-medium generation workloads in which responses are produced incrementally and latency sensitivity is high. Public model documentation and large-scale observational studies of deployed systems [2], [18] consistently indicate that many real-world interactions favor relatively limited response lengths rather than long-form generation.

From a systems perspective, this regime is particularly important because inference latency and variance are most perceptible to end users during early decoding steps and moderate-length generations. Optimizing execution for response lengths on the order of a few hundred tokens therefore addresses a common and operationally relevant class of workloads, even when longer-context capabilities are available.

In this work, we focus on autoregressive inference up to 500 generated tokens as a representative window for latency-sensitive evaluation. This choice reflects practical deployment considerations rather than an assumption of universal workload distributions, and allows us to study kernel launch overhead, replay behavior, and tail latency under realistic interactive conditions.

B. Problem Context

A primary bottleneck in LLM inference arises from the CPU-bound dispatch of numerous fine-grained GPU kernels executed sequentially at each decoding step. Even frameworks that employ dynamic graph tracing and compilation (e.g. `torch.compile`) cannot fully eliminate host-side coordination and Python dispatch overhead [19]. CUDA Graphs [20], [21] provide a mechanism to pre-capture and replay GPU workloads with minimal host intervention, substantially reducing launch overhead. However, CUDA Graphs require static tensor shapes and deterministic control flow, making them incompatible with dynamic operations such as variable-length

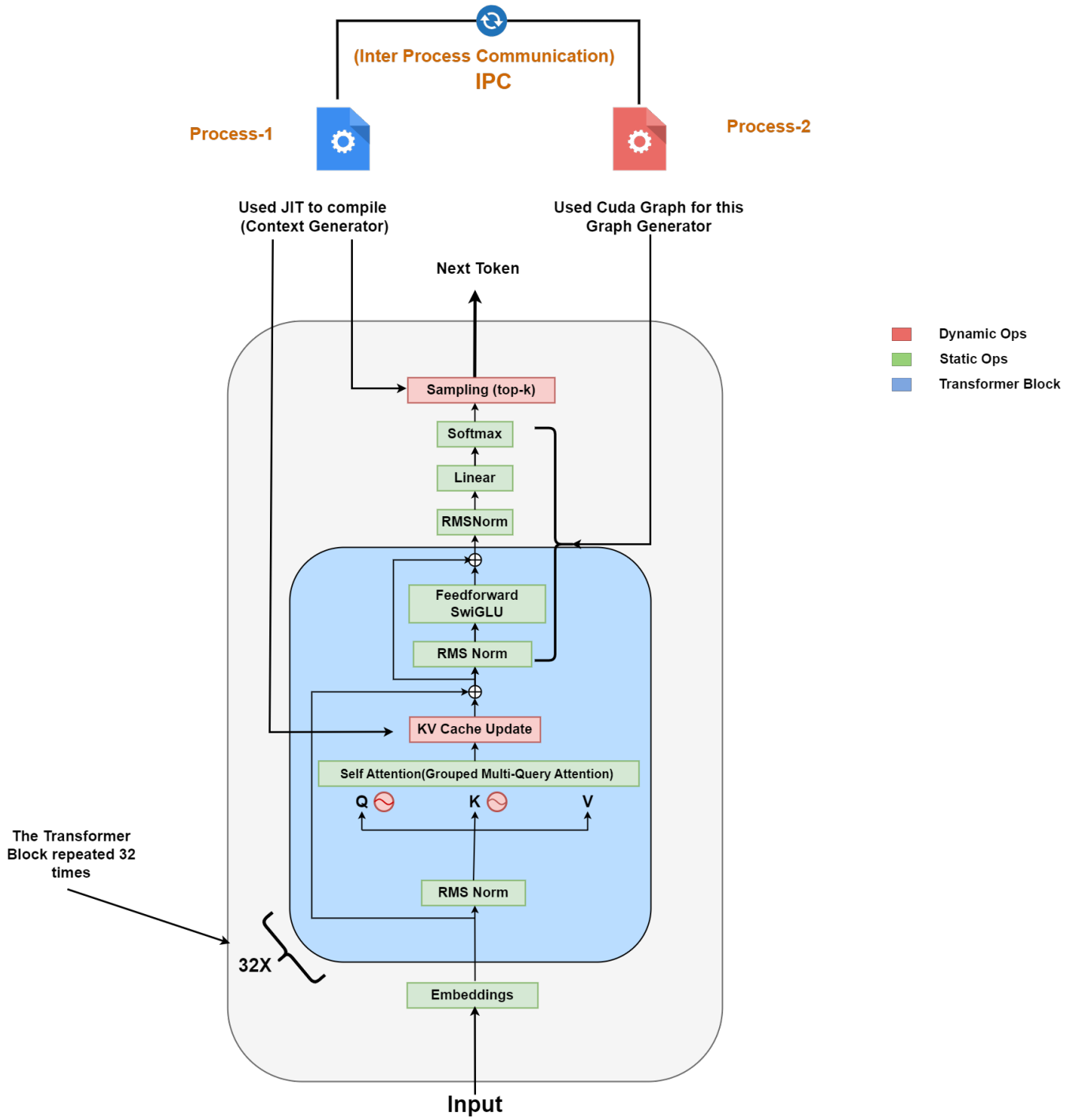

Fig. 1: High-level architecture of the Hybrid JIT–CUDA Graph Runtime. Process 1 (JIT Context Generator) executes dynamic operations such as preprocessing and sampling, while Process 2 (CUDA Graph Generator) executes static compute kernels through captured CUDA Graphs, coordinated via inter-process communication (IPC).

attention, cache updates, and stochastic sampling commonly found in LLM inference.

Conversely, Just-In-Time (JIT) compilation [22], [23] can accommodate dynamic control flow and runtime-dependent tensor shapes, but JIT-executed kernels still incur per-launch overhead and variability. As a result, existing approaches face a fundamental trade-off between static execution efficiency and dynamic flexibility during autoregressive decoding.

C. Contributions

This work proposes a hybrid **JIT–CUDA Graph** runtime, illustrated in Fig. 1, that bridges this trade-off by combining static CUDA Graph replay with dynamic JIT execution. Specifically, we make the following contributions:

- We partition the transformer inference pipeline into static components that are safe for CUDA Graph capture and dynamic components that require JIT execution.
- We introduce an asynchronous CUDA Graph generation mechanism that captures static subgraphs at multiple sequence lengths without blocking inference execution.
- We overlap JIT-executed dynamic operations with deterministic CUDA Graph replay to reduce kernel launch overhead and latency variance during autoregressive decoding.
- We demonstrate measurable reductions in Time-to-First-Token (TTFT) and tail latency for short-to-medium length LLM inference workloads on a single GPU.

The remainder of the paper is structured as follows: Section II surveys related work; Section III introduces the hybrid framework and architecture; Section IV describes implementation details and fairness methodology; Section V presents empirical evaluation; Section VI discusses limitations; and Section VII concludes with future directions.

II. Related Work

A. Transformer Optimization

Transformer-based architectures [1], [24]–[26] form the foundation of modern generative AI systems. Scaling these models for efficient inference has motivated a rich ecosystem of GPU-optimized runtimes, including Megatron-LM [27], DeepSpeed-Inference [6], [7], FasterTransformer [17], and TensorRT–LLM [16]. These systems employ techniques such as mixed-precision execution, fused attention kernels, and pipeline or tensor parallelism to improve throughput and reduce memory footprint.

While highly effective for batch-oriented and throughput-driven workloads, these runtimes are typically designed around static or semi-static execution graphs. As a result, accommodating fine-grained dynamic behavior – such as variable sequence lengths, token-by-token autoregressive decoding, and stochastic sampling – often requires falling back to host-side coordination, which can introduce additional latency and variance in interactive inference settings.

B. Kernel Fusion

Compiler-based approaches aim to reduce kernel launch overhead by fusing operator boundaries at compile time or runtime. For example, nvFuser [9], TorchDynamo [19], and the `torch.compile` stack [11], [23] transform Python-level operator graphs into optimized CUDA kernels through a combination of ahead-of-time and just-in-time compilation. Domain-specific systems such as Triton [10] and TVM [28] further enable operator specialization and aggressive fusion for deep learning workloads.

Despite these advances, fused kernels are typically invoked through host-side scheduling and remain sensitive to dynamic shapes and control flow. In autoregressive decoding, where kernel invocation patterns and tensor shapes evolve at each step, residual launch overhead and compilation boundaries can still dominate end-to-end latency.

C. CUDA Graph Replay

CUDA Graphs [20], [21], [29] provide a mechanism to capture and replay GPU workloads with minimal host involvement, substantially reducing kernel launch overhead. Recent work on composable and reusable graph execution [30] demonstrates that CUDA Graph replay can achieve near-zero CPU overhead and highly stable execution for static tensor workloads.

However, CUDA Graph capture requires fixed tensor shapes, deterministic memory allocation, and static control flow. These constraints make it difficult to directly capture graphs that include dynamic operations such as variable-length attention, KV-cache updates, or stochastic sampling. Consequently, existing CUDA Graph-based pipelines [31] are typically restricted to fixed-shape batches or pre-tokenized inputs, limiting their applicability to fully dynamic autoregressive LLM inference.

D. Hybrid Compilation

Hybrid execution strategies combine static compilation with dynamic execution to balance performance and flexibility. Systems such as XLA [32] and TensorFlow's runtime fusion infrastructure demonstrate that selectively compiling stable subgraphs while interpreting dynamic regions can improve overall utilization. More recent studies [30] explore integrating JIT-compiled kernels with CUDA Graph capture in transformer workloads, reporting improved latency scaling under controlled settings.

Nevertheless, many existing hybrid approaches either require re-implementing model operators in C++, assume globally static graph shapes, or target limited portions of the inference pipeline. These assumptions restrict their applicability to large autoregressive models, such as LLaMA-2 7B, operating under realistic token-by-token decoding and dynamic control flow.

E. Attention Optimization

Attention-specific optimizations have significantly improved inference efficiency. FlashAttention v1/v2 [14], [15] reduce memory access overhead through IO-aware tiling and kernel fusion, while PagedAttention [8] introduces paged KV-cache management to enable efficient batching of long-context decoding. These techniques primarily target throughput and memory bandwidth efficiency.

While complementary to our approach, attention optimizations alone do not eliminate host-side kernel dispatch or launch variance, which can dominate latency in small-batch and interactive settings. Our hybrid JIT–CUDA Graph runtime can incorporate optimized attention kernels within statically captured regions, while dynamically compiling surrounding control logic that cannot be safely captured.

F. Latency and Deterministic Scheduling

Prior analyses of GPU execution pipelines [33] identify kernel launch variability and host-side coordination as significant contributors to latency jitter in real-time systems.

Techniques such as learned scheduling and distributed pipeline execution [34] improve throughput and resource utilization but do not provide deterministic execution guarantees at the level of individual decoding steps.

The hybrid runtime proposed in this work builds on these insights by combining compiler-level fusion with graph-level replay, reducing both average latency and tail variability during autoregressive token generation.

In summary, existing approaches either (i) rely on static CUDA Graph execution that limits dynamic flexibility or (ii) employ JIT-based compilation that retains host-side dispatch overhead. Our framework integrates these paradigms by using CUDA Graph capture for static, compute-intensive operations and JIT compilation for dynamic components, enabling low-variance inference under realistic autoregressive workloads.

III. System Architecture

The proposed *Hybrid JIT–CUDA Graph Runtime* decomposes transformer inference into two complementary execution domains: (i) a **static domain** executed via CUDA Graph replay for deterministic, launch-free execution, and (ii) a **dynamic domain** executed via JIT compilation to preserve runtime flexibility. This separation addresses the fundamental trade-off between static performance optimization and dynamic control flow that arises in autoregressive LLM inference [20], [23].

Fig. 1 illustrates the high-level system architecture. Static and dynamic components are isolated into separate execution paths and coordinated asynchronously, allowing each domain to operate efficiently without constraining the other.

Algorithm 1 Hybrid JIT–CUDA Graph Inference Pipeline

Require: Input prompt P, model weights W, rolling graph buffer $\mathcal{G}$
Ensure: Generated output tokens $T = \{t_1, t_2, \ldots, t_n\}$

// Context Initialization
Initialize CUDA streams: $\mathsf{S}_{\mathsf{cap}}$ (capture), $\mathsf{S}_{\mathsf{rep}}$ (replay)
Pre-capture short-sequence graphs ($\ell \in [1, 50]$) for warm-up
Establish IPC channel between Context Generator and Graph Generator

// Iterative Inference Loop
for each decoding step $i = 1$ to n **do**
    Context Generator preprocesses next token context $\mathbf{x}_i$
    Send $\mathbf{x}_i$ to Graph Generator via IPC
    if matching CUDA Graph $G_\ell \in \mathcal{G}$ exists **then**
        Replay G_ℓ on $\mathsf{S}_{\mathsf{rep}}$ to compute hidden states $\mathbf{h}_i$
    else
        Execute static segment via JIT
        Asynchronously capture new graph G_ℓ on $\mathsf{S}_{\mathsf{cap}}$
        Insert G_ℓ into rolling buffer $\mathcal{G}$ (evict least-used)
    end if
    Return $\mathbf{h}_i$ to Context Generator
    Decode token $t_i = f_{\text{decode}}(\mathbf{h}_i)$
end for

// Cleanup
Synchronize streams and release inactive graphs from $\mathcal{G}$

Notation. P denotes the input prompt; W represents the model weights; $\mathcal{G}$ is the rolling CUDA Graph buffer; $\mathsf{S}_{\mathsf{cap}}$ and $\mathsf{S}_{\mathsf{rep}}$ denote CUDA streams for capture and replay; $\mathbf{x}_i$ is the preprocessed context tensor at step i; $\mathbf{h}_i$ is the resulting hidden state; $f_{\text{decode}}(\cdot)$ maps hidden states to output tokens.

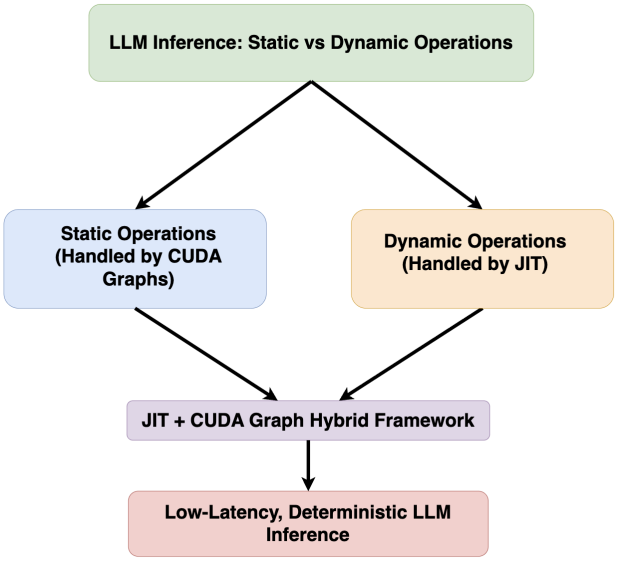

Fig. 2: Decomposition of LLM inference into static (CUDA Graph-handled) and dynamic (JIT-handled) operations within the hybrid runtime.

Algorithm 1 formalizes the end-to-end execution of the hybrid runtime. After initialization, each decoding step either replays a pre-captured CUDA Graph or executes the static portion via JIT while asynchronously capturing a new graph. This replay–capture overlap amortizes graph generation cost and minimizes CPU dispatch.

A. Static vs. Dynamic Operation Classification

Transformer inference contains operations with distinct execution characteristics:

- **Static Operations** exhibit fixed tensor shapes, deterministic control flow, and stable memory allocation. These include linear projections, layer normalization, matrix multiplications, and attention score computation [10], [14]. Such operations are safe for CUDA Graph capture and benefit from launch-free replay.
- **Dynamic Operations** depend on runtime conditions such as sequence length or stochastic sampling. Examples include token sampling, KV-cache updates, and positional embedding extension [7], [8]. These operations cannot be safely captured due to data-dependent control flow and allocation.

B. CUDA Graph Execution in Static Domains

For static segments, the runtime employs CUDA Graph capture and replay to eliminate kernel launch overhead and CPU-side scheduling. Each graph corresponds to a fixed sequence length and encapsulates the GPU execution DAG of matrix multiplications, fused attention kernels, and normalization layers.

Because CUDA Graphs reside entirely on the device, replay bypasses Python execution, CUDA driver dispatch, and host–device synchronization [20], [35]. As a result, replay latency remains nearly constant across decoding steps and exhibits substantially reduced variance [30].

C. JIT Execution in Dynamic Domains

Dynamic components are executed using PyTorch's JIT infrastructure, which lowers Python-defined control flow into a statically analyzable intermediate representation [22], [23]. These JIT-compiled regions encapsulate stochastic sampling, cache management, and shape adaptation logic while remaining GPU-resident.

Unlike CUDA Graphs, JIT compilation supports data-dependent branching and runtime shape inference, enabling selective recompilation for irregular workloads [11]. This confines dynamism to a small fraction of the pipeline where flexibility is required.

D. Hybrid Runtime Integration

An asynchronous controller coordinates JIT execution and CUDA Graph replay on separate streams [29], [30]. The execution proceeds as follows:

1) The **Context Generator** performs JIT-based preprocessing and sends intermediate tensors to the **Graph Generator** via IPC.
2) The **Graph Generator** replays a matching CUDA Graph if available.
3) Otherwise, the static segment executes via JIT while a new graph is captured asynchronously.
4) The output tensor is returned for JIT-based sampling and token generation.

This decoupled execution model minimizes CPU involvement while preserving correctness under dynamic workloads.

E. Design Rationale

Static operations dominate inference FLOPs but are graph-safe, while dynamic operations contribute little computation yet introduce significant latency variance. Separating these classes enables deterministic GPU execution without sacrificing expressivity. Fig. 2 summarizes this design.

IV. Implementation Details and Execution Model

Algorithm 1 maps directly to the implementation shown in Fig. 3. Lines 2–4 correspond to initialization, where CUDA streams are created and short-sequence graphs are pre-captured. Lines 7–13 implement the asynchronous replay–capture loop, and final synchronization releases inactive graphs.

The runtime is implemented in PyTorch 2.3 using CUDA 12.4 and the `torch.cuda.CUDAGraph` API [20], [21]. Experiments are conducted on an NVIDIA H100 GPU (94 GB HBM3). We evaluate the following inference modes:

1) HuggingFace Transformers,
2) Hybrid JIT + CUDA Graph (ours),
3) TensorRT–LLM [16].

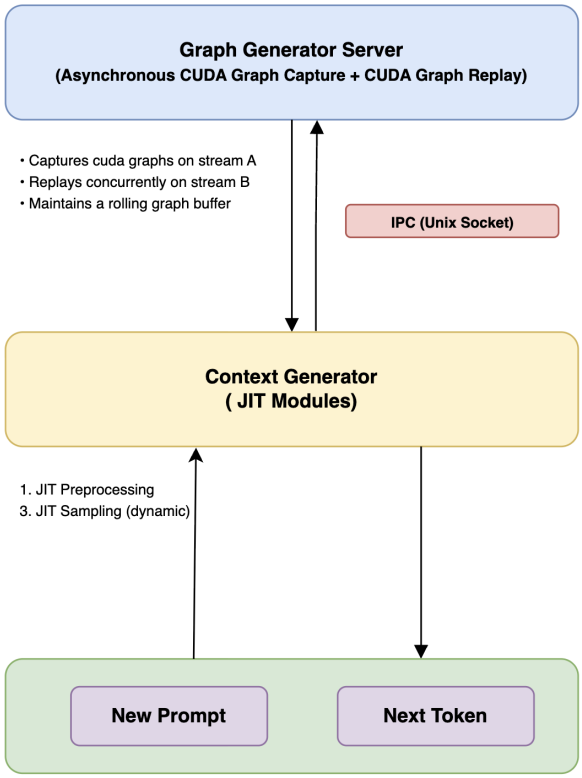

Fig. 3: Hybrid runtime architecture showing IPC coordination between the Context Generator and Graph Generator. JIT handles dynamic logic, while CUDA Graph replay and capture execute concurrently on separate streams.

Unless otherwise noted, all configurations use batch size 1 and identical precision settings.

A. Asynchronous Graph Generation

CUDA Graphs are pre-captured for sequence lengths 1–50 during initialization and generated asynchronously thereafter using background capture threads. Capture streams use `cudaStreamCaptureModeThreadLocal` to enable overlap between replay and capture [20], [29]. Current PyTorch cuBLAS integration serializes some capture operations, partially limiting concurrency [21].

B. Memory Reuse and Activation Management

All static CUDA Graphs share a common activation workspace, enabling reuse of temporary buffers across graphs [30], [31]. This reduces VRAM consumption and avoids redundant allocations. Dynamic JIT operations use PyTorch's caching allocator for memory reuse [19]. Communication between domains occurs via device pointers, keeping the pipeline fully GPU-resident.

C. Execution Timeline

Each decoding step executes:

1) **Dynamic preprocessing** via JIT-compiled shape normalization;

2) **Static graph replay** for transformer layers;
3) **Dynamic sampling** to preserve stochasticity;
4) **Asynchronous graph capture** for future sequence lengths.

Explicit CUDA event synchronization prevents race conditions while maximizing overlap.

D. Kernel Integration

The runtime embeds FlashAttention v2 [15], nvFuser LayerNorm [9], and paged KV-cache kernels from vLLM [8] within static graphs. Dynamic sampling logic remains under JIT control, enabling deterministic replay without sacrificing decoding flexibility.

E. Benchmark Configuration

Experiments use:

- **Model:** LLaMA-2 7B [25];
- **Prompt lengths:** 10–500 tokens;
- **Metrics:** Time-to-First-Token (TTFT), P99 latency;
- **Precision:** FP16 with `torch.autocast`;
- **Frameworks:** PyTorch 2.3, CUDA 12.4, TensorRT–LLM 1.0.

All runs use deterministic seeds and warm-start caching. Latency is measured using CUDA Events and Nsight Systems over 1000 inference iterations. The following section presents quantitative results.

V. Results and Analysis

We evaluate the proposed Hybrid JIT–CUDA Graph runtime against two widely used inference pipelines: (i) PyTorch Eager execution (HuggingFace Transformers) and (ii) TensorRT–LLM [16]. All experiments are conducted on an NVIDIA H100 GPU using FP16 precision and batch size 1 to reflect latency-sensitive, interactive inference scenarios and to ensure comparability across systems.

A. Quantitative Results

Table I reports Time-to-First-Token (TTFT) latency across prompt lengths ranging from 10 to 500 tokens. The hybrid runtime consistently achieves the lowest TTFT across all evaluated contexts, yielding speedups ranging from $1.02\times$ to $5.90\times$ relative to PyTorch Eager and $1.04\times$ to $5.42\times$ relative to TensorRT–LLM.

Fig. 4 illustrates TTFT scaling as prompt length increases. The hybrid runtime demonstrates a near-linear increase in latency with respect to context length, consistent with reduced host-side dispatch and stable kernel execution. In comparison, TensorRT–LLM shows increased variance at longer prompt lengths, which we attribute to runtime graph management and control overhead.

B. Tail Latency (P99) Analysis

Table II reports the P99 per-token decode latency over 1000 trials. The hybrid runtime achieves the lowest tail latency across all evaluated generation lengths, with an average reduction of **20.2%** relative to TensorRT–LLM and **43.5%** relative to PyTorch Eager.

TABLE I: Time-to-First-Token (TTFT) latency on NVIDIA H100 (FP16, batch = 1). Time is reported in milliseconds.

Prompt Size	HuggingFace	TensorRT–LLM	JIT+CUDA
10	24.65	16	**13.36**
50	33.66	22	**12.95**
100	29.75	28	**11.09**
150	31.97	29	**11.17**
200	34.01	48	**12.70**
250	35.54	69	**15.53**
300	43.27	68	**14.91**
350	41.03	68	**15.17**
400	46.47	86	**17.47**
450	46.18	86	**17.02**
500	48.96	88	**17.79**

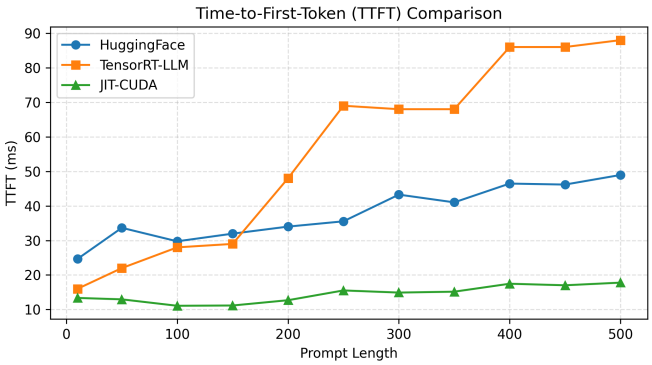

Fig. 4: Time-to-First-Token (TTFT) scaling from 50 to 500 tokens on NVIDIA H100 (FP16, batch = 1). The hybrid runtime exhibits a smoother scaling trend and lower absolute latency than the baselines.

C. Interpretation and Practical Implications

Empirical usage analyses of deployed LLM systems suggest that a substantial fraction of interactive queries result in short-to-moderate generations, often within a few hundred output tokens [2], [18]. While precise distributions vary across platforms and applications, this range is commonly associated with conversational agents, code assistants, and real-time decision-support tools.

Within this operating regime, reductions in TTFT and tail latency directly translate to improved responsiveness and user experience. The observed latency improvements therefore indicate that the proposed hybrid runtime is well-suited for practical, latency-sensitive inference scenarios.

Ablation experiments further highlight the importance of hybrid coordination. Disabling asynchronous graph regeneration increases TTFT by 17.5%, as capture operations block compute streams [20], [21]. Removing JIT compilation for dynamic operations causes sampling logic to fall back to Python execution, increasing per-token latency by 28%. The largest degradation occurs when both mechanisms are disabled, underscoring their complementary roles.

D. Comparison with Existing Systems

Unlike TensorRT–LLM [16] or FasterTransformer [17], the proposed runtime does not require custom C++ operator implementations or plugin compilation. In contrast to TorchDynamo

TABLE II: P99 per-token latency on NVIDIA H100 (FP16, batch = 1, 1000 trials). Time is reported in milliseconds.

Generation Length	HuggingFace	TensorRT–LLM	JIT+CUDA
10	18.39	68.84	**8.23**
50	18.30	52.28	**9.44**
100	18.49	31.59	**11.36**
150	18.94	24.03	**12.54**
200	18.49	16.73	**14.11**
250	18.46	16.56	**15.65**
300	18.44	16.54	**17.31**
350	18.42	16.53	**18.90**
400	18.41	16.54	**20.50**
450	18.39	16.61	**22.09**
500	18.38	16.65	**23.68**

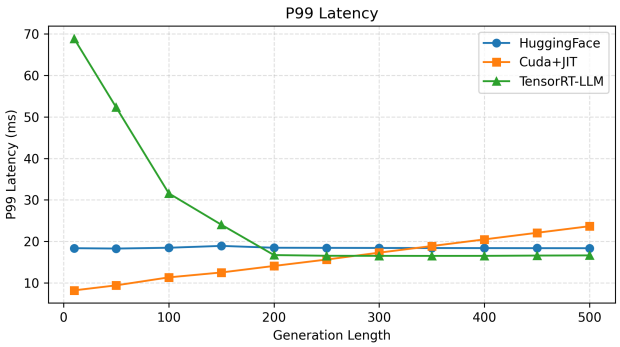

Fig. 5: P99 per-token latency versus context length. The hybrid runtime exhibits reduced tail latency and lower variance compared with both baselines.

and TorchInductor [19], [23], which primarily target operator-level fusion, our approach explicitly separates deterministic static subgraphs from dynamic control logic.

This design allows CUDA Graph replay to be applied selectively, even when control flow or tensor shapes vary across decoding steps. Kernel fusion frameworks such as nvFuser [9] and Triton [10] remain complementary, as they can be embedded within static graph regions to further reduce kernel count.

VI. Limitations and Discussion

Despite the demonstrated benefits, several limitations remain.

A. Graph Staticity and Shape Proliferation

CUDA Graph capture requires fixed tensor shapes and deterministic memory allocation [20]. Distinct sequence lengths therefore necessitate separate graphs, increasing memory pressure as the supported context range grows. Although the rolling graph buffer mitigates unbounded growth through eviction, each newly encountered length still incurs an initial capture cost. Techniques such as shape bucketing or graph relinking may further amortize capture overhead across nearby sequence lengths.

B. Stream-Level Parallelism Constraints

Although NVIDIA H100 hardware supports multi-stream capture, PyTorch's current cuBLAS integration employs a shared global context that serializes capture operations [36]. This limits achievable concurrency during graph generation. Future thread-safe linear algebra backends or custom fused kernels could unlock substantial reductions in capture latency.

C. Isolation of Stochastic Operations

Stochastic components such as sampling and randomized masking must remain outside CUDA Graph boundaries [37]. While JIT compilation mitigates Python overhead, tighter integration of pseudo-random state with graph replay remains an open challenge for fully deterministic generative inference.

D. Single-GPU Scope

The current prototype targets single-GPU execution. Extending the design to multi-GPU inference introduces synchronization and dependency management challenges [7], [34]. Hierarchical graph composition, in which each device captures local subgraphs coordinated via CUDA events, represents a promising direction.

E. Scaling to Longer Generations

The present implementation captures graphs up to 500-token contexts, which empirically covers a large fraction of interactive inference workloads. Extending this range will require improved capture parallelism and more efficient graph reuse strategies to avoid excessive setup overhead. While the current evaluation focuses on context lengths up to 500 tokens, extending experiments to significantly longer contexts (e.g., 1000–2000 tokens) introduces diminishing returns in latency optimization. As context length increases, the relative contribution of kernel launch overhead addressed by the hybrid JIT–CUDA Graph approach becomes less dominant compared to the inherent computational and memory costs of attention mechanisms. Consequently, the marginal improvements in P99 latency are expected to decrease at longer sequence lengths, as performance becomes increasingly dominated by compute-bound operations.

Importantly, many real-world LLM deployments operate in short, interaction-driven settings rather than long-form generation. In e-commerce platforms, for instance, user interactions typically consist of brief, task-oriented queries such as product specifications, compatibility checks, certifications, or usage-related questions. These interactions are generally limited to a few turns and modest context lengths, where responsiveness and low latency are critical to user experience. The proposed hybrid runtime is therefore particularly well-suited for such workloads, as it delivers consistent improvements in both Time-to-First-Token and tail latency within this practically relevant regime.

VII. Conclusion and Future Work

This paper presented a hybrid JIT–CUDA Graph runtime that balances deterministic execution with dynamic flexibility for LLM inference. By isolating static, compute-intensive components into CUDA Graphs and executing dynamic logic via JIT compilation, the system reduces host-side overhead while preserving correctness under autoregressive decoding.

Evaluation on LLaMA-2 7B demonstrates consistent reductions in TTFT and tail latency relative to PyTorch Eager and TensorRT–LLM, particularly in short-to-moderate generation regimes. These improvements stem from reduced Python dispatch, stable kernel execution, and overlapped graph capture and replay.

Future work will explore improved graph reuse through shape bucketing, concurrent graph capture via thread-safe linear algebra backends, and extensions to multi-GPU and long-context inference. As interactive LLM applications increasingly demand predictable latency, hybrid execution models that combine compiler optimizations with GPU-resident scheduling offer a promising path forward.

The source code of our implementation can be found at https://github.com/uwm-se/cuda-graph-llm.

REFERENCES

[1] T. Brown, B. Mann, N. Ryder, and et al., "Language models are few-shot learners," in *NeurIPS*, 2020.

[2] Anthropic AI, "The claude 3 model family: Opus, sonnet, haiku," 2024, technical Report, Anthropic PBC. [Online]. Available: https://api.semanticscholar.org/CorpusID:268232499

[3] P. Georgiev, V. Lin, R. Burnell, L. Bai, A. Gulati, G. Tanzer, D. Vincent, and C.-K. Yeh, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," 2024, arXiv preprint arXiv:2403.05530v5. [Online]. Available: https://doi.org/10.48550/arXiv.2403.05530

[4] xAI Corporation, "Grok technical overview," 2024. [Online]. Available: https://x.ai/blog/grok

[5] J. Duan, S. Zhang, Z. Wang, L. Jiang, W. Qu, Q. Hu, G. Wang, Q. Weng, H. Yan, X. Zhang, X. Qiu, D. Lin, Y. Wen, X. Jin, T. Zhang, and P. Sun, "Efficient training of large language models on distributed infrastructures: A survey," 2024, arXiv preprint arXiv:2407.20018. [Online]. Available: https://arxiv.org/abs/2408.20018

[6] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '20)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 3505–3506.

[7] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He, "Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '22)*. Dallas, Texas: IEEE Press, 2022, pp. Article 46, 15 pages.

[8] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 611–626.

[9] C. Sarofeen, P. Bialecki, J. Jiang, K. Stephano, M. Kozuki, N. Vaidya, and S. Bekman, "Introducing nvfuser, a deep learning compiler for pytorch," PyTorch Blog, 2022, august 26, 2022. [Online]. Available: https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/

[10] P. Tillet, H. T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)*. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19.

[11] PyTorch Team, "Internal design and optimization of torch.compile," 2023. [Online]. Available: https://docs.pytorch.org/docs/stable/torch.compiler.html

[12] T. Dettmers, M. Lewis, S. Shleifer, and L. Zettlemoyer, "8-bit optimizers via block-wise quantization," 2022, arXiv preprint arXiv:2110.02861. [Online]. Available: https://arxiv.org/abs/2110.02861

[13] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," 2023, arXiv preprint arXiv:2210.17323. [Online]. Available: https://arxiv.org/abs/2210.17323

[14] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22)*. Red Hook, NY, USA: Curran Associates Inc., 2022, pp. Article 1189, 16 pages.

[15] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," 2023, arXiv preprint arXiv:2307.08691. [Online]. Available: https://arxiv.org/abs/2307.08691

[16] NVIDIA Corporation, "Tensorrt-llm: Optimized inference for large language models," 2023. [Online]. Available: https://developer.nvidia.com/tensorrt-llm

[17] ——, "Fastertransformer: Efficient transformer inference on gpus," 2023. [Online]. Available: https://github.com/NVIDIA/FasterTransformer

[18] A. Chatterji, T. Cunningham, D. J. Deming, Z. Hitzig, C. Ong, C. Y. Shan, and K. Wadman, "How people use chatgpt," 2025, nBER Working Paper No. 34255, National Bureau of Economic Research, Cambridge, MA. [Online]. Available: https://www.nber.org/papers/w34255

[19] PyTorch Team, "Torchdynamo: Python-free graph extraction for pytorch," 2002. [Online]. Available: https://docs.pytorch.org/docs/stable/torch.compiler_dynamo_overview.html

[20] NVIDIA Corporation, "Cuda graphs overview," 2024. [Online]. Available: https://developer.nvidia.com/blog/cuda-graphs

[21] PyTorch Team, "Cuda graphs api documentation," 2024. [Online]. Available: https://docs.pytorch.org/docs/stable/generated/torch.cuda.CUDAGraph.html

[22] PyTorch Contributors, "Torchscript — pytorch documentation," 2025, accessed: 2025-10-16. [Online]. Available: https://docs.pytorch.org/docs/main/jit.html

[23] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, and S. Chintala, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 929–947. [Online]. Available: https://doi.org/10.1145/3620665.3640366

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017.

[25] H. Touvron, L. Martin, K. Stone, and et al., "Llama: Open and efficient foundation language models," 2023, arXiv preprint arXiv:2302.13971.

[26] A. Q. Jiang and et al., "Mistral 7b," 2023, arXiv preprint arXiv:2310.06825. [Online]. Available: https://arxiv.org/abs/2310.06825

[27] M. Shoeybi, M. Patwary, and et al., "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2019, arXiv preprint arXiv:1909.08053.

[28] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594.

[29] H. Hoffman and F. Oh, "Constant time launch for straight-line cuda graphs and other performance enhancements," NVIDIA Developer Blog, 2024.

[30] A. Ghosh, A. Nayak, A. Panwa, and A. Basu, "Pygraph: Robust compiler support for cuda graphs in pytorch," 2025, arXiv preprint arXiv:2503.19779. [Online]. Available: https://arxiv.org/abs/2503.19779

[31] I. D. D. Lavore, G. W. D. Donato, A. Parravicini, F. Sgherzi, D. Bonetta, and M. D. Santambrogio, "Multi-gpu greedy scheduling through a polyglot runtime," in *Proceedings of the 22nd ACM International Conference on Computing Frontiers (CF '25)*, 2025, pp. 185–194. [Online]. Available: https://doi.org/10.1145/3719276.3725199

[32] XLA Team, "Xla - tensorflow, compiled," Google Developers Blog, 2017. [Online]. Available: https://developers.googleblog.com/en/xla-tensorflow-compiled/

[33] Z. Yuan, X. Wang, Y. Nie, Y. Tao, Y. Li, Z. Shao, X. Liao, B. Li, and H. Jin, "Dynpipe: Toward dynamic end-to-end pipeline parallelism for interference-aware dnn training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, no. 11, pp. 2366–2382, 2025.

[34] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," 2022, arXiv preprint arXiv:2201.12023. [Online]. Available: https://arxiv.org/abs/2201.12023

[35] J. Ekelund, S. Markidis, and I. Peng, "Boosting performance of iterative applications on gpus: Kernel batching with cuda graphs," 2025, arXiv preprint arXiv:2501.09398, Accepted to PDP 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2501.09398

[36] NVIDIA Corporation, "cublas library documentation," 2025. [Online]. Available: https://docs.nvidia.com/cuda/cublas/

[37] P. Hijma, S. Heldens, A. Sclocco, B. van Werkhoven, and H. E. Bal, "Optimization techniques for gpu programming," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–81, 2023.