Review

Optimization of Large Models for Efficient Inference: Algorithm, Compiler, and System Co-Design

Shengyi Gao [1,*]

[1] Ningxia University, Yinchuan, Ningxia, China
* Correspondence: Shengyi Gao, Ningxia University, Yinchuan, Ningxia, China

Abstract: Large language models have achieved remarkable capabilities in natural language processing, yet their inference cost remains a significant challenge due to high computation, memory usage, and latency. This study presents a cross-layer co-optimization framework that integrates algorithmic, compiler, and system-level strategies to enhance inference efficiency. Algorithmic techniques, including structured pruning, sparsity, quantization, and dynamic inference, reduce computational workload and memory footprint. Compiler optimizations, such as operator fusion, graph rewriting, and layout specialization, translate algorithmic improvements into hardware-efficient execution. System-level strategies, encompassing parallel execution, memory management, and KV cache optimization, further improve resource utilization and reduce latency. The framework synergistically coordinates these layers, providing a theoretically grounded approach for reducing FLOPs, memory consumption, and inference latency. Its adaptability extends to cloud, edge, and interactive deployment scenarios, offering a unified methodology for efficient and scalable large-model inference. This work contributes a systematic and extensible pathway for accelerating model inference without relying on empirical performance measurements.

Keywords: large language models; inference optimization; algorithm-compiler-system co-design; computational efficiency; memory management; parallel execution

Received: 23 October 2025
Revised: 06 November 2025
Accepted: 17 November 2025
Published: 25 November 2025

1. Introduction

1.1. Research Background

Large-scale pretrained models (Large Language Models, LLMs) have achieved significant breakthroughs in natural language processing, multimodal understanding, and various human-AI interaction applications. However, as their parameter sizes continue to expand, the computational burden during the inference stage has increased dramatically. This growth results in substantial memory consumption, higher latency, and elevated energy costs, all of which directly restrict the practical deployment of LLMs in real-world systems [1].

At the same time, the demands for efficient inference have become increasingly diverse across different application scenarios. Cloud-based services emphasize high throughput and stable performance, while edge and mobile environments face stricter constraints on energy consumption and hardware resources. In interactive applications such as real-time translation, conversational agents, and intelligent customer service systems, low-latency responses are essential for ensuring user experience. Consequently, improving inference efficiency has become a central challenge for enabling the widespread adoption of large models.

1.2. Existing Challenges

Despite the rapid progress in LLM architecture and system support, achieving high-performance inference remains a difficult problem, mainly due to limitations at three different layers:

1) Algorithmic inefficiencies.

Redundant computation and the rapid expansion of the Key-Value (KV) cache lead to large memory overhead and limited throughput, especially in autoregressive decoding scenarios.

2) Compiler-level constraints.

Current compiler stacks often struggle to fully exploit hardware potential for LLM inference. Issues such as suboptimal operator fusion, insufficient support for low-bit quantization, and limited handling of dynamic shapes all hinder performance.

3) System-level bottlenecks.

System challenges, including memory scheduling, kernel dispatch overhead, and inefficient runtime architectures, further limit inference speed. These bottlenecks become more pronounced when serving multiple concurrent requests or deploying models on resource-limited devices.

Overall, the separation of optimization efforts across algorithms, compilers, and systems prevents the formation of a unified optimization pipeline, leading to suboptimal inference efficiency [2].

1.3. Research Objectives and Contributions

To address the limitations described above, this study investigates a cross-layer optimization paradigm that integrates techniques across the algorithm, compiler, and system levels. The main contributions of this paper are as follows:

1) A unified perspective on three-layer co-optimization.

The paper systematically analyzes the interactions between algorithmic techniques, compiler behaviors, and system runtime mechanisms, highlighting the necessity of coordinated design.

2) A collaborative framework for efficient inference.

A conceptual optimization framework is proposed to bridge the three layers, enabling information flow and performance feedback across them to achieve more effective inference acceleration.

3) A theoretical analysis of performance benefits.

Although no empirical experiments are conducted, the study provides a detailed theoretical evaluation showing how the proposed framework can reduce computation, memory usage, and latency in typical LLM inference scenarios.

2. Inference Computation Characteristics

2.1. Bottlenecks in Inference Computation

Large Transformer models face significant computational and memory challenges. The self-attention mechanism introduces quadratic complexity with respect to sequence length L, and feed-forward networks contribute additional computation proportional to the hidden dimension d. The per-layer computational cost can be expressed as

$$\mathrm{FLOP}_{\mathrm{Slayer}} = O(L^2 d + d^2),$$

highlighting that long sequences dramatically increase operations. Memory consumption is dominated by the KV cache, which grows linearly with sequence length and number of layers N:

$$\mathrm{Memory}_{\mathrm{KV}} = O(L \cdot d \cdot N).$$

These expressions reveal why reducing sequence length, hidden dimension, or sparsifying computation is critical to improving efficiency [3].

2.2. Transformer Block Critical Path

Within each Transformer block, the critical path consists of multi-head attention, residual connections, normalization layers, and feed-forward networks. From a computational perspective, attention dominates for long sequences, while feed-forward layers dominate for very wide hidden dimensions. Formally, the total operations per block can be estimated as

$$\mathrm{FLOP}_{\mathrm{Sblock}} = O(L^2 d + d^2),$$

which matches the per-layer cost previously described. Memory bottlenecks follow a similar pattern, as intermediate activations and cached keys and values require storage proportional to

$$\mathrm{Memory}_{\mathrm{block}} = O(L \cdot d) + O(d^2),$$

indicating that both computation and memory must be jointly considered when designing optimization strategies.

2.3. Theoretical Formulation and Computational Complexity

The computational characteristics of Transformer inference can be further understood through a theoretical analysis of its core operations. For a model with hidden size d, number of attention heads *h*, and sequence length *L*, the dominant cost arises from the self-attention mechanism. In autoregressive decoding, the query vector for the current token must attend to all previously generated tokens, resulting in a computational complexity of *O(Ld)* per layer for attention score computation and *O(Ld)* for the weighted aggregation of values. When extended across all layers, the total attention complexity becomes *O(LdN)*, where N denotes the number of layers. The feed-forward network contributes an additional cost of approximately $O(d^2)$ per token due to its two-layer structure with an intermediate expansion dimension. While this component is compute-intensive, its cost remains constant with respect to sequence length and therefore becomes relatively less dominant during long-context inference.

Memory complexity also plays a crucial role. The KV cache requires storing key and value vectors of dimension *d/h* for each head across every layer, producing a memory footprint proportional to *O(LdN)*. Accessing this cache during decoding introduces substantial bandwidth demand, as each new token triggers repeated retrieval of cached tensors across all layers. This repeated access often becomes the primary latency source on modern accelerators, where memory throughput limits overall performance more sharply than raw FLOPs. Together, these formulations reveal that the inference cost grows linearly with sequence length but multiplies across deep model stacks, underscoring why optimizing either computation or memory in isolation is insufficient. Instead, effective acceleration requires coordinated reduction of arithmetic operations, memory footprint, and data movement across the entire inference pipeline.

3. Algorithmic Optimization Methods

3.1. Pruning, Sparsification, Quantization, and Dynamic Inference

Algorithmic optimization has emerged as one of the most active research directions for accelerating large-model inference, and existing studies have proposed a variety of approaches to reduce computational cost without fundamentally altering the Transformer architecture. Among them, pruning constitutes an important line of work. By eliminating parameters or structural components that contribute minimally to model predictions, pruning reduces both the computational load and memory footprint. Structured pruning methods, such as removing attention heads or reducing intermediate dimensions of feed-forward networks, provide predictable latency benefits and are well aligned with hardware execution patterns. In contrast, unstructured pruning can achieve higher sparsity levels but often requires additional runtime support to fully exploit fine-grained sparsity. Despite these challenges, pruning consistently demonstrates that significant redundancy exists in large models, making it a practical pathway for inference acceleration.

Sparsification extends this idea by introducing explicit sparse patterns into matrix multiplications and attention operations. Research in this direction shows that enforcing block-sparse or head-sparse structures can substantially reduce multiply-accumulate operations, particularly in attention layers where computation scales with context length. However, sparsity must be carefully designed to ensure hardware-friendly execution; otherwise, the overhead of managing sparse indices can offset theoretical gains. Recent techniques explore learnable sparsity patterns or rely on routing-based architectures that selectively activate a subset of computation paths, thereby balancing accuracy and efficiency.

Quantization represents another widely used strategy to improve inference efficiency. By reducing the numerical precision of model weights and activations from floating-point formats to lower-bit representations, quantization directly decreases memory bandwidth requirements and accelerates matrix multiplication on specialized hardware. Studies have demonstrated that 8-bit and even 4-bit quantization can maintain high accuracy when combined with calibration or post-training correction techniques. More advanced approaches perform mixed-precision quantization, assigning different bit levels to different layers or tensor types according to their sensitivity. The key challenge lies in managing quantization-induced numerical instability during attention computation and normalization, but ongoing research continues to push the boundary of ultra-low-bit inference [3].

Dynamic inference methods further enhance efficiency by reducing unnecessary computation at runtime. Instead of relying on static model structures, dynamic approaches adjust computational paths based on input difficulty, model confidence, or context characteristics. Examples include early exiting mechanisms that stop computation when intermediate representations yield sufficiently confident predictions, as well as dynamic token or layer skipping strategies that exploit redundancy among successive tokens. For large language models in particular, several studies highlight that many tokens contribute limited new information to the attention mechanism, making dynamic sparsity or selective KV cache updates effective in reducing both computation and memory usage. These methods shift the focus from static compression to adaptive resource allocation, enabling inference to scale more efficiently with input complexity.

Overall, pruning, sparsification, quantization, and dynamic inference represent four complementary algorithmic directions that collectively illustrate the substantial redundancy present in large language models. When combined, these techniques can reduce arithmetic operations, shrink memory consumption, and mitigate latency, forming the algorithmic foundation for multi-layer inference optimization.

3.2. Mechanisms of Computational Reduction in Algorithmic Optimization

The effectiveness of algorithmic optimization techniques lies not only in their ability to compress model parameters but also in the specific computational mechanisms through which they reduce the arithmetic intensity and memory traffic of Transformer inference. Pruning achieves efficiency by directly eliminating redundant components from the model's computational graph. When entire attention heads, neurons, or feed-forward channels are removed, the corresponding matrix multiplications shrink proportionally, thereby reducing the dimensionality of intermediate tensors. This structural simplification shortens the critical computational path, lowers FLOPs, and decreases data movement, making inference both faster and more memory-efficient. Even when pruning is performed at a fine-grained level, the reduction in effective non-zero weights translates into fewer multiply-accumulate operations, provided that the underlying runtime system can exploit the resulting sparsity.

Sparsification reduces computation through deliberate introduction of structured or semi-structured zero patterns. By ensuring that large portions of query-key similarity matrices or feed-forward weight matrices do not require multiplication, sparsification lowers the number of active operations during attention and matrix transformation. Importantly, the computational savings arise when sparsity is aligned with hardware-friendly patterns such as block-level or head-level sparsity, enabling accelerators to skip entire blocks of operations without expensive index handling. In the context of long-context inference, sparse attention designs-such as local, blockwise, or dilated patterns-reduce the quadratic dependency on sequence length by limiting each token's receptive field, thereby transforming the theoretical complexity from O(L2) toward sub-quadratic or even linear forms.

Quantization reduces computation primarily by compressing numerical precision, which decreases both arithmetic cost and memory bandwidth requirements. Lower-bit integer operations require fewer hardware cycles than floating-point computation and allow more values to be packed into a single memory transaction, improving data throughput and cache utilization. At the algorithmic level, quantization shrinks the volume of data transferred between compute units and memory, thereby addressing the memory-bound nature of attention and feed-forward layers. Mixed-precision quantization further enhances efficiency by allocating low precision to computationally dominant but error-tolerant components, while preserving higher precision where numerical stability is critical. This selective reduction in precision yields substantial end-to-end latency benefits without proportionally degrading model accuracy [4].

Dynamic inference reduces unnecessary computation by adapting the computational workload to input-specific characteristics. Mechanisms such as early exiting shorten the effective depth of the model by skipping later layers when intermediate representations already produce confident predictions. Similarly, dynamic token skipping reduces the number of tokens that must be processed through the full attention mechanism, while adaptive KV cache updates prevent redundant recomputation of representations that contribute minimally to the model's output. These strategies exploit the observation that many decoding steps-especially those involving predictable or repetitive patterns-do not require full-capacity computation. As a result, dynamic inference converts what would otherwise be a fixed computational cost into a variable one, aligning resource consumption with task difficulty and significantly lowering average-case latency.

Collectively, these mechanisms demonstrate that algorithmic optimization reduces computation not merely by compressing models but by reshaping the distribution of computational work across time, layers, and tensor dimensions. This multi-faceted reduction lays the groundwork for deeper co-optimization with compiler and system layers, enabling further acceleration in practical deployment environments.

3.3. Integrated Algorithmic Strategies for End-to-End Inference Efficiency

While individual techniques such as pruning, sparsity, quantization, and dynamic inference each target specific inefficiencies within the model, recent research increasingly emphasizes the importance of combining these methods to achieve end-to-end performance gains. Integrated optimization approaches treat the model as a coupled system in which reductions in parameter count, activation footprint, and arithmetic precision reinforce one another rather than function as isolated enhancements. For example, structured pruning can reduce the dimensionality of matrix multiplications, which in turn lowers the sensitivity of subsequent quantization and enables more aggressive bit-width reduction without degrading accuracy. Similarly, dynamic inference mechanisms such as adaptive computation or early-exit schemes become more effective when applied to sparsified or quantized representations, because the reduced computational load per layer amplifies the benefits of conditional execution. These synergies reshape both the arithmetic and memory profiles of the model, allowing the inference path to be re-designed around lighter-weight kernels, smaller intermediate tensors, and lower memory traffic. As a result, integrated algorithmic strategies increasingly form the foundation of efficient LLM inference pipelines, enabling substantial speedups even before compiler and system-level optimizations are introduced [5].

4. Compiler Techniques for High-Performance Model Inference

4.1. Graph Optimization, Operator Fusion, and Layout Specialization

Compiler-level optimizations serve as a crucial bridge between algorithmic advances and hardware execution efficiency, and modern deep learning compilers increasingly rely on graph-level transformations to streamline the inference pipeline. Graph optimization restructures the computation graph by eliminating redundant operations, simplifying expression chains, and reordering independent kernels to expose greater parallelism while preserving semantic correctness. Within this optimized graph, operator fusion becomes a key mechanism for reducing overhead by merging sequential operations-such as linear projection, bias addition, and activation-into unified kernels that minimize intermediate memory writes and lower kernel-launch latency. In parallel, layout specialization adapts tensor memory layouts to the specific dataflow patterns of the underlying hardware, ensuring that frequently accessed dimensions are contiguous, aligned, and cache-friendly. This optimization is particularly critical for attention and feed-forward modules, where poor layouts can amplify memory bottlenecks despite high arithmetic throughput. Ultimately, the combination of graph rewriting, fusion strategies, and layout-aware transformations creates a more compact and hardware-efficient execution plan, allowing large models to better exploit available compute units and memory bandwidth.

4.2. Scheduling Optimization and Memory-Efficient Execution

Scheduling optimization within the compiler is essential for maximizing hardware utilization during LLM inference, as it determines the exact ordering, parallelization strategy, and resource allocation for each operation in the computation graph. Modern compilers employ techniques such as loop tiling, pipelining, and parallel thread mapping to ensure that compute units remain fully occupied while minimizing pipeline stalls caused by data dependencies. These scheduling strategies are tightly coupled with memory-efficient execution mechanisms, which aim to reduce the overall memory footprint and mitigate bandwidth bottlenecks. Key techniques include minimizing the lifetime of intermediate tensors through buffer reuse, strategically placing prefetching instructions to overlap computation with memory access, and splitting large kernels into cache-friendly tiles that reduce off-chip memory transactions. For attention-heavy workloads, compilers further apply KV cache-aware scheduling techniques to avoid repeated loading of large key-value tensors, often combining partial recomputation or paged access patterns to maintain locality. By jointly optimizing execution order, data movement, and memory residency, scheduling and memory-efficient compilation significantly shorten the critical path of inference, enabling large models to execute with lower latency and improved throughput across heterogeneous hardware platforms [6].

4.3. Co-Design of Algorithms and Compiler Optimization Pipelines

Effective inference acceleration increasingly depends on the co-design of model algorithms and compiler optimization pipelines, as algorithmic structures directly influence the compiler's ability to generate efficient executable kernels. Many recent model designs-such as linear attention variants, block-sparse architectures, and quantization-aware training methods-are explicitly crafted to expose patterns that compilers can exploit, including regular sparsity, predictable memory access, and reduced tensor dimensionality. In turn, the compiler aligns its optimization passes-such as fusion, layout transformation, and operator specialization-to leverage these algorithmic properties, enabling lower-precision kernels, sparsity-aware code generation, and mixed compute-memory scheduling strategies that would be less effective on unmodified architectures. This mutual reinforcement ensures that computational shortcuts introduced at the algorithmic level translate into real hardware gains rather than being obscured by unfavorable execution graphs or memory layouts. As co-design becomes standard practice, the boundary between model architecture and compiler stack grows increasingly blurred, leading to integrated frameworks in which models are designed with compilation constraints in mind and compilers incorporate model-specific heuristics to deliver optimal end-to-end inference performance.

5. System-Level Inference Optimization Techniques

5.1. Parallelism Strategies for Large-Scale Inference

System-level parallelism is essential for overcoming the computational and memory bottlenecks inherent in large-model inference, and modern serving frameworks rely on multiple forms of parallel execution to fully utilize heterogeneous hardware resources. Tensor parallelism partitions matrix operations across multiple devices, allowing large projection layers in attention and feed-forward modules to be computed concurrently while maintaining tightly synchronized communication. Pipeline parallelism divides the model into sequential segments distributed across devices, enabling different micro-batches or tokens to be processed simultaneously at different stages of the model and thereby increasing throughput under high concurrency. For autoregressive workloads, sequence parallelism further decomposes attention computation by distributing KV cache storage and attention heads across devices to reduce per-device memory pressure while preserving token-level dependency constraints. Recent systems additionally employ speculative decoding and multi-model parallel scheduling to overlap stages of generation, improving resource utilization during single-token decoding. Together, these parallelism paradigms form a layered execution strategy that balances compute distribution, memory scaling, and communication overhead, enabling large models to achieve low-latency, high-throughput inference across distributed hardware environments [7].

5.2. Memory Management and KV Cache Optimization

Efficient memory management is critical for sustaining high-performance inference in large language models, particularly when dealing with long-context sequences and limited device memory. The KV cache, which stores key and value tensors for all previously generated tokens, can quickly dominate memory usage if not carefully managed. Modern systems address this challenge through techniques such as buffer reuse, selective eviction, and memory paging, which reduce the peak memory footprint while preserving correctness in autoregressive decoding. Furthermore, memory-aware scheduling ensures that data movement overlaps with computation, minimizing idle cycles caused by memory access latency. For attention layers, partitioning the KV cache across devices or tiles allows each device to operate on a subset of the sequence while maintaining consistent global attention, thereby lowering per-device memory pressure. Additional strategies, such as compressed storage formats or low-precision representation of cached tensors, further reduce bandwidth requirements and improve cache locality. By combining these memory management and KV cache optimization techniques, systems can maintain high throughput and low latency, even when handling extremely large models and long input sequences, without exceeding hardware constraints.

5.3. Accelerator-Aware Scheduling and System-Level Co-Optimization

To fully exploit modern heterogeneous hardware for large-model inference, system-level scheduling must be tightly integrated with the underlying accelerator architecture. Accelerator-aware scheduling aligns computation, memory transfers, and kernel execution with the specific characteristics of GPUs, TPUs, or specialized inference accelerators, taking into account factors such as core utilization, on-chip memory hierarchy, and interconnect bandwidth. Techniques such as asynchronous kernel dispatch, overlapping computation with data movement, and dynamic load balancing allow the system to minimize idle cycles and ensure high throughput. Furthermore, co-optimization across multiple system layers-combining scheduling strategies with algorithmic adaptations like sparsity, quantization, and dynamic computation-enables the inference engine to adapt execution paths according to input characteristics and hardware capabilities. By coordinating workload partitioning, memory management, and parallel execution, accelerator-aware system scheduling reduces bottlenecks at both computation and communication levels, achieving lower latency and higher efficiency for large-scale model deployment in diverse runtime environments.

6. Collaborative Optimization Framework

6.1. Framework Design

The core of the proposed optimization framework lies in its integration of algorithm-level, compiler-level, and system-level strategies to enable efficient inference for large Transformer models. At the algorithm level, techniques such as pruning, sparsity enforcement, quantization, and dynamic inference mechanisms work together to reduce redundant computation and memory usage, decreasing arithmetic operations and intermediate data, which directly benefits memory-limited deployment scenarios such as edge devices. At the compiler level, graph optimizations, operator fusion, and layout adjustments translate these algorithmic improvements into hardware-efficient execution plans. Graph optimization removes unnecessary computations, operator fusion combines multiple kernels into single operations, and layout optimization aligns memory access patterns with hardware characteristics to improve cache utilization and reduce bandwidth overhead [8].

System-level strategies complement algorithm and compiler improvements by managing memory allocation, scheduling parallel execution, and optimizing data movement. Efficient memory scheduling prevents bottlenecks from KV cache access or intermediate tensor storage, while parallel execution distributes computation across cores or devices to maximize throughput. Data movement strategies reduce latency by aligning data transfer with computation requirements. By coordinating these three layers, the framework ensures that improvements are mutually reinforcing, resulting in significant gains in latency, throughput, and memory efficiency. The framework is modular and scalable, allowing independent adaptation of each layer to different model sizes or deployment scenarios, while fully realizing cross-layer benefits when all layers operate collaboratively. This integration distinguishes the proposed framework from single-layer optimization approaches and provides a solid foundation for further theoretical analysis.

6.2. Collaborative Strategy

The framework achieves efficiency through a tightly coordinated collaboration between algorithm-level, compiler-level, and system-level strategies. At the algorithm level, pruning removes redundant weights, sparsity enforcement reduces unnecessary computation, and quantization lowers numerical precision without significantly impacting accuracy. Dynamic inference strategies, such as early token skipping and adaptive computation, further reduce the workload for tokens that require less processing. These algorithmic improvements not only decrease the number of arithmetic operations and memory footprint but also provide the foundation for more effective compiler optimizations. For instance, reduced operations enable the compiler to generate streamlined computation graphs, apply aggressive operator fusion, and reorganize memory layouts for better alignment with hardware characteristics. By explicitly connecting algorithmic improvements with compiler capabilities, the framework ensures that optimizations at one level magnify benefits at other levels, rather than remaining isolated.

At the system level, memory management, scheduling, and data movement strategies complement algorithm and compiler enhancements to fully realize performance gains. Memory allocation policies reduce bottlenecks caused by KV cache growth or intermediate tensor storage, while parallel execution distributes workload across multiple cores or devices to maximize throughput. Data movement optimizations minimize latency by aligning transfers with computation requirements, ensuring that hardware resources are efficiently utilized. The framework also considers deployment-specific adaptations: in cloud environments, throughput is prioritized through parallel execution; on edge devices, memory usage is minimized via layout optimization and quantization; in interactive applications, latency is reduced by leveraging dynamic inference and efficient caching. Through this comprehensive cross-layer collaboration, the framework achieves cumulative efficiency gains, demonstrating a clear advantage over approaches that optimize only a single layer of the inference pipeline.

6.3. Mechanism Analysis and Performance Trends

The collaborative optimization framework produces several interrelated benefits across computation, memory, and latency. At the algorithm level, pruning and sparsity reduce the number of operations, while quantization decreases the size of intermediate data, directly lowering both arithmetic workload and memory footprint. Dynamic inference further reduces unnecessary computation for less critical tokens. Compiler-level optimizations translate these algorithmic gains into efficient execution: operator fusion minimizes kernel launch overhead, graph simplification reduces dependency chains, and layout adjustments improve cache utilization and memory access patterns. System-level strategies complement these improvements by efficiently scheduling memory usage, parallelizing computations, and optimizing data movement. Together, these mechanisms ensure that improvements at one level amplify benefits at the others, resulting in a cumulative reduction in latency, throughput bottlenecks, and overall resource consumption [9].

The framework also demonstrates adaptability across diverse deployment scenarios. In cloud environments, parallel execution and optimized scheduling increase throughput for batch inference tasks. On edge devices, memory-efficient layouts and quantized computations minimize storage and bandwidth demands. For interactive applications, dynamic inference and efficient caching reduce response time and provide low-latency experiences. Although no empirical data is presented, theoretical analysis and computational trends indicate clear improvements: the total FLOPs and memory access requirements decrease, latency is reduced, and the framework can efficiently handle both long-sequence and high-dimensional inference tasks. This analysis highlights the practical potential of cross-layer collaboration and provides a strong foundation for future evaluation and enhancement of large model inference optimization strategies.

7. Theoretical Feasibility Analysis

7.1. Computational Complexity Analysis

The computational characteristics of large Transformer models reveal that the dominant costs arise from both the self-attention mechanism and feed-forward networks. Algorithm-level optimizations, such as pruning and sparsity, reduce the number of arithmetic operations, while quantization decreases memory usage by representing data in lower precision. Dynamic inference mechanisms further lower workload by skipping unnecessary computations for less critical tokens. These reductions collectively decrease the overall FLOPs and memory footprint, illustrating that targeted algorithmic strategies can theoretically improve efficiency without affecting model accuracy. Compiler-level improvements amplify these benefits: operator fusion and graph simplification reduce execution overhead, while layout optimization ensures that memory access patterns match hardware capabilities, further minimizing effective computation time. System-level strategies, including memory scheduling and parallel execution, complement these improvements by preventing bottlenecks and optimizing resource utilization across multiple cores or devices.

From a theoretical perspective, the cumulative effect of cross-layer optimizations is clear: FLOPs and memory access requirements decrease, bandwidth pressure is reduced, and latency is improved, particularly in scenarios with long sequences or high-dimensional hidden states. Even without empirical measurements, trend analysis indicates that reducing redundant computation and optimizing memory hierarchies can produce substantial efficiency gains. Moreover, the framework allows for scalable adjustments, meaning that models of different sizes or architectures can benefit similarly from these strategies. This theoretical complexity analysis highlights the importance of coordinated algorithm-compiler-system strategies and sets a solid foundation for comparing the framework against existing approaches and deployment scenarios.

7.2. Comparison with Mainstream Approaches

Mainstream optimization approaches for large Transformer inference typically focus on a single layer, such as algorithmic pruning, compiler-level operator fusion, or system-level parallelization. While these methods offer measurable improvements within their respective layers, they often fail to exploit the synergistic potential of cross-layer collaboration. For instance, pruning alone may reduce computation but cannot fully leverage memory layout optimizations at the compiler level, and compiler fusion without algorithmic simplification cannot eliminate unnecessary arithmetic operations. Consequently, single-layer optimizations may achieve partial gains but leave significant efficiency improvements untapped.

In contrast, the proposed collaborative framework integrates algorithm, compiler, and system-level strategies, producing cumulative benefits that surpass any single-layer method. Algorithmic reductions in FLOPs and memory footprint enable the compiler to generate highly efficient execution graphs, while system-level scheduling and memory management further enhance performance. This cross-layer coordination ensures that optimizations reinforce one another, theoretically resulting in lower latency, higher throughput, and reduced resource consumption across various deployment scenarios. By structurally combining these layers, the framework addresses limitations inherent in mainstream approaches and provides a more complete and adaptable solution for efficient large model inference.

7.3. Adaptability Across Different Application Scenarios

The proposed collaborative optimization framework demonstrates theoretical adaptability across a variety of deployment scenarios. In cloud environments, where throughput is a primary concern, the combination of algorithmic pruning, compiler-level graph simplification, and system-level parallelization enables the model to handle large batches efficiently while minimizing computational bottlenecks. Memory-efficient layouts and operator fusion reduce data movement overhead, allowing high-throughput inference without exceeding hardware resource limits. This ensures that cloud servers can maintain responsiveness even under heavy workloads, highlighting the framework's scalability in large-scale processing environments.

For edge devices and interactive applications, memory footprint and latency are critical constraints. Quantization and sparsity reduce intermediate storage requirements, while dynamic inference mechanisms skip redundant computations for less significant tokens. Compiler optimizations align memory layouts with hardware characteristics, further improving cache utilization and reducing bandwidth demand. System-level strategies, including intelligent scheduling and efficient data transfer, minimize latency and provide smooth real-time responses. These adaptations show that the framework can be effectively tuned for both high-performance cloud deployments and resource-constrained edge environments, demonstrating its versatility and robust theoretical feasibility across diverse application scenarios.

8. Conclusion and Future Directions

8.1. Research Summary

This study has developed a comprehensive cross-layer co-optimization framework that integrates algorithmic, compiler, and system-level strategies to improve the efficiency of large-model inference. At the algorithmic level, structured pruning, sparsity, quantization, and dynamic inference reduce computational workload and memory footprint, exposing patterns that compilers can exploit. Compiler-level optimizations, including operator fusion, graph rewriting, layout specialization, and scheduling, translate these algorithmic reductions into hardware-efficient execution, minimizing redundant operations and memory traffic. At the system level, parallel execution, memory management, and KV cache optimization further improve resource utilization, ensuring high throughput and low latency across heterogeneous hardware. By coordinating these layers in a synergistic manner, the framework addresses the limitations of single-layer optimization approaches, providing a theoretically grounded path for reducing FLOPs, memory consumption, and inference latency. Overall, the research demonstrates that a holistic, cross-layer perspective can significantly enhance the practical efficiency of large-scale models while maintaining model functionality and generality. This unified approach not only clarifies the interactions between algorithm, compiler, and system optimizations but also provides a structured methodology for designing future high-performance inference pipelines.

8.2. Limitations and Future Directions

Despite the theoretical advantages of the proposed framework, several limitations remain and point to directions for future research. Extending the framework to multi-modal inference represents a key opportunity, as different modalities introduce heterogeneous computation patterns and memory requirements that may challenge existing scheduling and compiler strategies. Another promising direction is edge-cloud collaborative inference, which would leverage the complementary strengths of local and cloud resources to optimize latency, throughput, and memory usage simultaneously. Additionally, automated joint search of algorithmic and compiler configurations could identify optimal combinations of pruning, quantization, kernel fusion, and scheduling, tailored to specific models and deployment environments, further enhancing performance. Future studies could also explore dynamic adaptation mechanisms, where the system automatically adjusts computation strategies based on workload characteristics or hardware constraints. Taken together, these directions suggest that the framework presented here can serve as a foundation for a broad class of efficient, adaptable, and scalable inference solutions, bridging theoretical analysis and practical deployment in diverse application scenarios.

References

1. Z. Zhou, X. Ning, K. Hong, T. Fu, J. Xu, S. Li, and Y. Wang, "A survey on efficient inference for large language models," *arXiv preprint arXiv:2404.14294*, 2024. doi: 10.48550/arXiv.2404.14294.
2. R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, and Y. He, "Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale," In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, November, 2022, pp. 1-15. doi: 10.1109/SC41404.2022.00051.
3. S. Park, S. Jeon, C. Lee, S. Jeon, B. S. Kim, and J. Lee, "A survey on inference engines for large language models: Perspectives on optimization and efficiency," *arXiv preprint arXiv:2505.01658*, 2025. doi: 10.48550/arXiv.2505.01658.

4. Y. Liu, J. Wu, Y. He, R. Gong, J. Xia, L. Li, and K. Li, "Efficient inference for large reasoning models: A survey," *arXiv preprint arXiv:2503.23077*, 2025. doi: 10.48550/arXiv.2503.23077.
5. W. Wang, W. Chen, Y. Luo, Y. Long, Z. Lin, L. Zhang, and X. He, "Model compression and efficient inference for large language models: A survey," *arXiv preprint arXiv:2402.09748*, 2024.
6. C. Guo, F. Cheng, Z. Du, J. Kiessling, J. Ku, S. Li, and Y. Chen, "A survey: Collaborative hardware and software design in the era of large language models," *IEEE Circuits and Systems Magazine*, vol. 25, no. 1, pp. 35-57, 2025. doi: 10.1109/mcas.2024.3476008.
7. X. Zhang, J. Liu, Z. Xiong, Y. Huang, G. Xie, and R. Zhang, "Edge intelligence optimization for large language model inference with batching and quantization," In *2024 IEEE Wireless Communications and Networking Conference (WCNC)*, April, 2024, pp. 1-6. doi: 10.1109/wcnc57260.2024.10571127.
8. J. Li, J. Xu, S. Huang, Y. Chen, W. Li, J. Liu, and G. Dai, "Large language model inference acceleration: A comprehensive hardware perspective," *arXiv preprint arXiv:2410.04466*, 2024. doi: 10.48550/arXiv.2410.04466.
9. J. Liu, P. Tang, W. Wang, Y. Ren, X. Hou, P. A. Heng, and C. Li, "A survey on inference optimization techniques for mixture of experts models," *arXiv preprint arXiv:2412.14219*, 2024. doi: 10.48550/arXiv.2412.14219.