LAYER-WISE PERFORMANCE-AWARE SPARSITY ALLOCATION FOR EFFICIENT LLM INFERENCE

Anonymous authors
Paper under double-blind review

THE USE OF LARGE LANGUAGE MODELS (LLMS)

We use Claude 4 Sonnet and ChatGPT 5 for grammar checking, spelling correction, and translation assistance in both the main text and appendix of this paper.

RELATED WORK

Recent advances in LLM compression have significantly improved inference efficiency while preserving model capabilities. Structured pruning techniques eliminate entire components like attention heads or layers (Dutta et al., 2024; Muralidharan et al., 2024), with approaches like TVAPrune leveraging variational information bottleneck principles to compress model representations (Dutta et al., 2024). Weight quantization methods such as GPTQ (Frantar et al., 2022), AWQ (Lin et al., 2024a), and OmniQuant (Shao et al., 2023) enable 4-bit weight representation with minimal performance degradation. Knowledge distillation approaches like MiniLLM (Gu et al., 2023) and GKD (Agarwal et al., 2024) transfer knowledge from larger teacher models to smaller students, with the latter introducing on-policy distillation to address train-inference distribution mismatch. Combined approaches leveraging pruning with distillation have yielded state-of-the-art results (Sreenivas et al., 2024; Muralidharan et al., 2024).

Building upon these structural approaches, unstructured pruning enhances LLM compression by flexibly removing weights, often achieving higher accuracy at high sparsity levels compared to structured methods (Lee et al., 2024). Recent advancements in this field include post-training one-shot pruning techniques such as SparseGPT (Frantar & Alistarh, 2023), which leverages second-order approximations to prune over 50% of weights in a 175B model while maintaining performance with minimal loss, and Wanda (Sun et al., 2023), which simplifies the pruning process through activation-aware magnitude pruning for efficient sparsity. Other notable contributions in this category, such as OWL (Yin et al., 2023), Tyr-the-Pruner (Li et al., 2025), and GBLM-Pruner (Das et al., 2023), further refine sparsity allocation and incorporate gradient correction to enhance pruning outcomes. Additionally, optimization-based methods like ALPS (Meng et al., 2024) approach pruning as a constrained optimization problem, while DSnoT (Zhang et al., 2023) iteratively refines weights to optimize results over multiple steps.

Beyond pruning techniques, model quantization is essential for deploying LLMs efficiently, reducing memory and computation by lowering the precision of weights and activations (Gong et al., 2024). However, the 'outlier' problem, where a few large values dominate the quantization range, poses a significant challenge (Ashkboos et al., 2024b). Early solutions like LLM.int8() (Dettmers et al., 2022) addressed this by retaining some activations in higher precision. Recent advancements include GPTQ (Frantar et al., 2022), which uses second-order information for accurate 4-bit weight quantization, and AWQ (Lin et al., 2024a), which protects key weights based on activation statistics. For activation quantization, SmoothQuant (Xiao et al., 2023) redistributes outliers to enable 8-bit quantization, while QuaRot (Ashkboos et al., 2024b) and QuIP (Chee et al., 2023) employ rotation transformations for 4-bit and 2-bit quantization, respectively. Additionally, QUIK (Ashkboos et al., 2023) and QServe (Lin et al., 2024b) combine quantization with system optimizations for practical deployment. Other notable methods include PrefixQuant (Chen et al., 2024a) and MergeQuant (Wang et al., 2025) for efficient static quantization.

With models compressed through the aforementioned techniques, large language model inference faces significant computational challenges that researchers address through various optimization approaches. Recent work has focused on KV cache management techniques such as Page-

dAttention (Kwon et al., 2023), which treats cache as virtual memory to reduce fragmentation, SpeCache (Jie et al., 2025), which intelligently prefetches needed keys, and QuaRot (Ashkboos et al., 2024b) for outlier-free 4-bit inference. Algorithmic optimizations like Cascade Speculative Drafting (Chen et al., 2024b) leverage a tiered approach where smaller models draft for larger ones, while N-gram masked self-attention Chelba et al. (2020) truncates attention windows. Memory efficiency improvements include SWAT (Fu et al., 2025) for sliding window attention and Inf-MLLM (Ning et al., 2024) for streaming inference via attention saddle patterns. Architecture innovations explore MoE approaches like Read-ME (Cai et al., 2024) which refactors dense models into router-decoupled experts, and Dynamic-LLaVA (Huang et al., 2024) which sparsifies vision-language contexts. System-level optimizations include operator fusion in FasterTransformer (Aminabadi et al., 2022), FlashAttention's IO-aware computation (Dao et al., 2022), FlashDecoding++ (Hong et al., 2024) with asynchronous softmax, and specialized schedulers like Sarathi-Serve (Agrawal et al., 2024), vLLM (Kwon et al., 2023), and Llumnix (Sun et al., 2024) that balance throughput and latency through innovative resource management strategies.

APPENDIX A.1: MOTIVATION

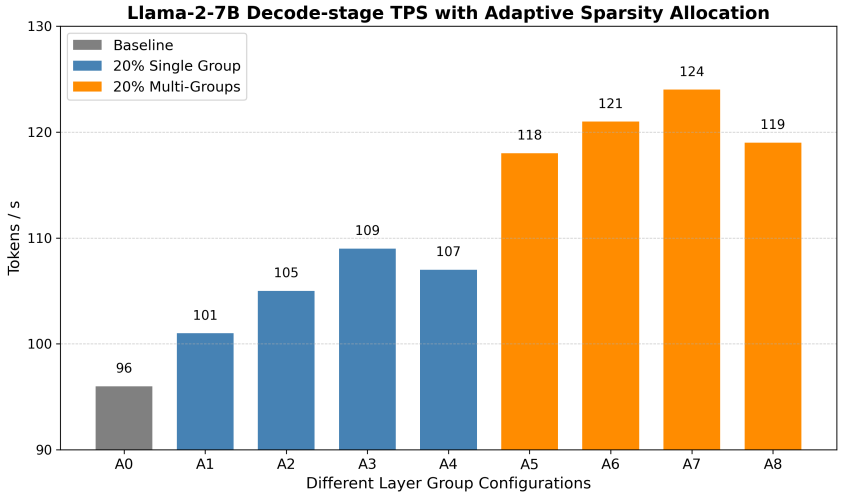

Figure 1: Decode-stage TPS with adaptive sparsity allocation across layer groups, based on 4-bit QuaRot quantization. Task and hardware: left-to-right language-model generation, each run takes a 2048-token prompt from the WikiText-103 validation set, allowing Llama-2-7B to autoregressively generate the next 256 tokens, and runs on a single NVIDIA RTX 3090 GPU. Variants on the X-axis denote different layer group sparsity configurations: A0: baseline, no sparsity applied (96 tokens/s); A1: 20% adaptive sparsity applied to layers 1-8 (early attention layers, 101 tokens/s); A2: 20% adaptive sparsity applied to layers 9-16 (middle attention layers, 105 tokens/s); A3: 20% adaptive sparsity applied to layers 17-24 (middle-late layers, 109 tokens/s); A4: 20% adaptive sparsity applied to layers 25-32 (late transformer layers, 107 tokens/s); A5: 20% adaptive sparsity applied to layers 1-8 and 17-24 simultaneously (118 tokens/s); A6: 20% adaptive sparsity applied to layers 9-16 and 25-32 simultaneously (121 tokens/s); A7: 20% adaptive sparsity applied to layers 1-8, 9-16, and 17-24 (124 tokens/s); A8: 20% adaptive sparsity applied to all layer groups 1-32 (119 tokens/s). Y-axis: measured tokens-per-second during decode phase. Higher is better.

In the deployment of LLMs, quantization and pruning have evolved as independent acceleration techniques with minimal integration. In our explorations, we conduct a series of attempts to investigate the combination of both techniques. Figure 1 demonstrates the inference acceleration achieved on the Llama-2-7B model in a language generation task, after applying 50% random dropout to various weight combinations and weight matrices at 8-bit precision. We observe a significant improvement in model inference speed. Figure 2 shows the model's performance across different quantization and pruning configurations. As the bit-width of quantization decreases and the sparse ratio increases, the perplexity rises substantially, indicating a clear trade-off between efficiency and performance.

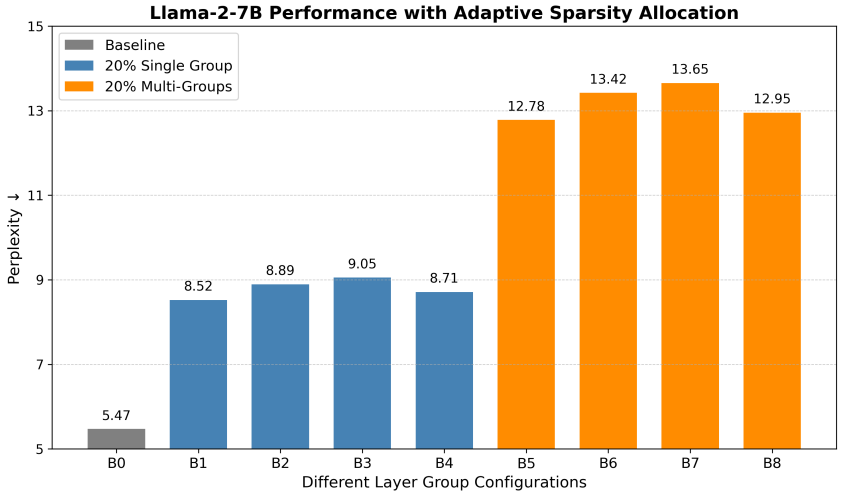

Figure 2: Performance of quantization and adaptive layer-group sparsity allocation on Llama-2-7B. Task and hardware: left-to-right language-model evaluation on the WikiText-2 validation subset. The evaluation uses approximately 1 000 passages, each with a sequence length of 2048, running on a single NVIDIA RTX 3090 GPU with 4-bit QuaRot quantization applied to all weights. Variants on the X-axis denote different layer group sparsity configurations: B0: 4-bit quantized baseline, no additional sparsity (5.47 perplexity); B1: 4-bit quantization + 20% adaptive sparsity on layers 1-8 (8.52 perplexity, 55.8% degradation); B2: 4-bit quantization + 20% adaptive sparsity on layers 9-16 (8.89 perplexity, 62.5% degradation); B3: 4-bit quantization + 20% adaptive sparsity on layers 17-24 (9.05 perplexity, 65.4% degradation); B4: 4-bit quantization + 20% adaptive sparsity on layers 25-32 (8.71 perplexity, 59.2% degradation); B5: 4-bit quantization + 20% adaptive sparsity on layers 1-8 and 17-24 (12.78 perplexity, 133.5% degradation); B6: 4-bit quantization + 20% adaptive sparsity on layers 9-16 and 25-32 (13.42 perplexity, 145.2% degradation); B7: 4-bit quantization + 20% adaptive sparsity on layers 1-8, 9-16, and 17-24 (13.65 perplexity, 149.5% degradation); B8: 4-bit quantization + 20% adaptive sparsity on all layer groups (12.95 perplexity, 136.7% degradation). Y-axis: measured perplexity on WikiText-2 (↓). Lower is better.

A.1.1 Layer-Group Sensitivity Analysis

The experimental results reveal distinct sensitivity patterns across different layer groups in the Llama-2-7B architecture under 20% sparsity allocation. Early layers (1-8) demonstrate moderate resilience to sparsification, with configuration A1 achieving 5.2% throughput improvement while B1 exhibits 55.8% perplexity degradation. This substantial degradation, even in early layers, highlights the critical nature of all computational pathways in modern LLMs.

Middle layers show varying sensitivity patterns: layers 9-16 (A2/B2) achieve 9.4% throughput gains with 62.5% quality degradation, while layers 17-24 (A3/B3) demonstrate the highest single-group acceleration of 13.5% with 65.4% perplexity increase. These middle-late layers exhibit the most severe single-group degradation, suggesting their critical role in semantic processing.

Late layers (25-32) show a balanced pattern with A4 achieving 11.5% throughput improvement while B4 incurs 59.2% quality loss. This pattern suggests that while late layers contain exploitable computational redundancy, their sparsification significantly impacts final output quality, though less severely than middle-late layers.

A.1.2 Multi-Group Sensitivity Analysis

The multi-group configurations (A5-A8, B5-B8) reveal the catastrophic nature of naive sparsity combination strategies. Configuration A5, combining layers 1-8 and 17-24 with 20% sparsity each, achieves 22.9% throughput improvement but incurs 133.5% perplexity degradation (B5). This represents a fundamental phase transition where the cumulative effect far exceeds the sum of individual impacts.

The most aggressive multi-group configuration A7 (layers 1-8, 9-16, 17-24) demonstrates maximum throughput gains of 29.2% but suffers catastrophic quality collapse with 149.5% perplexity degradation (B7). Paradoxically, the all-groups configuration A8 shows slightly reduced acceleration (24.0%) and degradation (136.7%), suggesting complex non-linear interactions when the entire model undergoes simultaneous sparsification.

APPENDIX A.2: QUANTIZATION FRAMEWORK

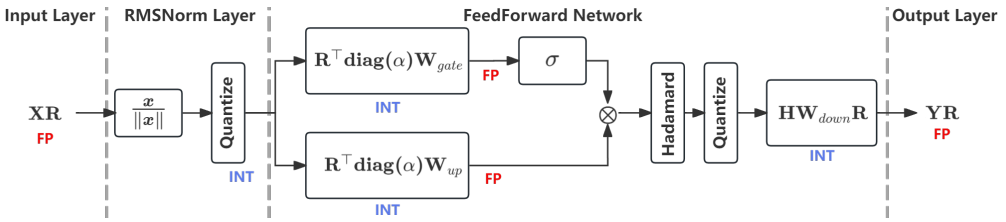

Figure 3: Low-bit quantization pipeline for a feed-forward network block with quantization and rotation.

In the framework illustrated in Figure 3, the hidden state X is first multiplied by a random orthogonal matrix R stored in FP(m) precision. Because $R^\top R = I$, this transformation *preserves Euclidean norms and inner products*, i.e. $(RX)^\top(RY) = X^\top Y$ (Ashkboos et al., 2024b). Consequently, substituting $X \mapsto RX$ and $W \mapsto R^\top W$ leaves each linear layer's output unchanged:

$$\underbrace{XW}_{\text{original}} = \underbrace{(RX)\big(R^\top W\big)}_{\text{rotated}}. \tag{1}$$

This computational invariance (Ashkboos et al., 2024a) enables the framework to redistribute heavy-tailed activation energy across dimensions before quantization, thereby mitigating the "outlier" problem that hampers uniform low-bit encodings (Ashkboos et al., 2024b).

After rotation, the activations are quantized to INT(n). The feed-forward weights W_{gate} and W_{up} are premultiplied by $R^\top$ and rescaled by $\mathrm{diag}(\alpha)$ (RMSNorm) (Zhang & Sennrich, 2019). The non-linearity σ is applied, followed by an on-the-fly Hadamard transform H; this matrix is also orthogonal, so fusing H into the down-projection W_{down} preserves functional equivalence while further flattening variance. A second INT(n) quantization converts the output back to low precision before casting to FP(m).

Because every orthogonal transform is absorbed into adjacent linear layers during an *offline* preprocessing step, the run-time kernel sequence matches that of the baseline network while operating entirely on 4-bit integers. Combined with per-channel GPTQ calibration (Frantar et al., 2022), this rotation–quantize–fuse pipeline achieves end-to-end INT4 inference without mixed-precision fallbacks, all while guaranteeing mathematical equivalence to the original FP16 model.

APPENDIX A.3: DYNAMIC PROGRAMMING BACKGROUND

A.3.1 ALGORITHMIC PARADIGM

Dynamic programming is both a mathematical optimization method and an algorithmic paradigm developed by Richard Bellman in the 1950s (Bellman, 1957). The approach simplifies complex problems by breaking them down into simpler subproblems in a recursive manner, then combining their solutions to solve the original problem (Cormen et al., 2009).

The paradigm applies when a problem exhibits two fundamental properties: optimal substructure and overlapping subproblems. Unlike divide-and-conquer algorithms, which solve entirely independent subproblems, dynamic programming exploits the fact that subproblems are not independent—the same subproblems arise repeatedly during the recursive solution process.

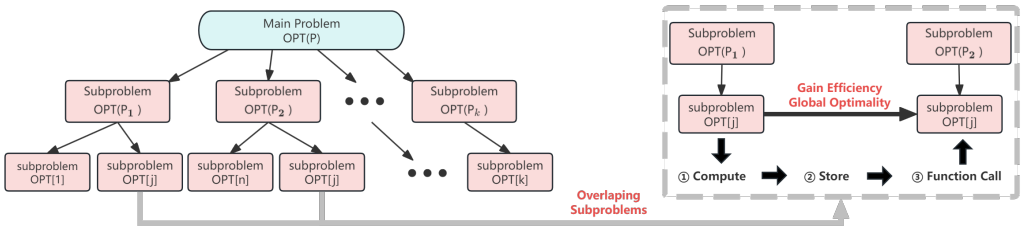

Figure 4: Illustration of dynamic programming approach showing problem decomposition into subproblems.

The essence of dynamic programming lies in avoiding redundant computation by storing solutions to subproblems for later reuse. This memoization strategy transforms algorithms with exponential time complexity into polynomial-time solutions (Dasgupta et al., 2006).

A.3.2 Optimal Substructure Property

A problem exhibits optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems (Cormen et al., 2009). This property is fundamental because it ensures that solving subproblems optimally contributes to the global optimum.

Mathematically, optimal substructure can be expressed as a functional relationship. For a problem P, if $\text{OPT}(P)$ represents the optimal solution, then:

$$\text{OPT}(P) = f(\text{OPT}(P_1), \text{OPT}(P_2), \ldots, \text{OPT}(P_k)), \tag{2}$$

where f is a function that combines optimal solutions of subproblems $\{P_i\}$ to yield the optimal solution of the original problem.

The canonical example demonstrating optimal substructure is the shortest path problem. If the shortest path from vertex u to vertex v passes through intermediate vertex w, then this path must consist of the shortest path from u to w concatenated with the shortest path from w to v. Any deviation from this principle would contradict the optimality of the overall path (Cormen et al., 2009).

Verification of optimal substructure typically employs proof by contradiction. If a subproblem within an alleged optimal solution were not itself optimal, then substituting the actual optimal subsolution would improve the overall solution, contradicting the assumption of optimality.

A.3.3 Overlapping Subproblems Property

The overlapping subproblems property distinguishes dynamic programming from divide-and-conquer approaches. This property requires that the space of subproblems be small relative to the total number of recursive calls, meaning the same subproblems are solved repeatedly (Cormen et al., 2009).

In problems with overlapping subproblems, naive recursive implementations typically exhibit exponential time complexity due to redundant calculations. The Fibonacci sequence computation illustrates this phenomenon: calculating $F(n)$ requires computing $F(n-1)$ and $F(n-2)$, where $F(n-1)$ itself requires $F(n-2)$ and $F(n-3)$, leading to multiple evaluations of the same Fibonacci numbers.

The mathematical characterization of this property can be expressed through the recurrence tree structure. If $T(n)$ represents the number of subproblems of size n solved during the recursive process, and $S(n)$ represents the number of distinct subproblems of size n, then overlapping subproblems exist when $T(n) >> S(n)$ for sufficiently large n.

Dynamic programming exploits this overlap through memoization, storing computed solutions in a table for subsequent lookup. This technique reduces the time complexity from exponential to polynomial by ensuring each distinct subproblem is solved exactly once (Dasgupta et al., 2006).

A.3.4 MATHEMATICAL FORMULATION

Dynamic programming algorithms follow a general mathematical structure based on the principle of optimality. The fundamental recurrence relation takes the form:

$$\text{OPT}[i] = \min_{j \in \mathcal{J}(i)} \{\text{OPT}[j] + \text{Cost}(j, i)\}, \tag{3}$$

where $\text{OPT}[i]$ represents the optimal solution value for subproblem i, $\mathcal{J}(i)$ denotes the set of feasible predecessor states, and $\text{Cost}(j, i)$ represents the immediate cost of transitioning from state j to state i.

The state space design constitutes a critical component of dynamic programming formulations. States must encapsulate sufficient information to make optimal decisions while maintaining computational tractability. The dimensionality of the state space directly affects both algorithm correctness and efficiency.

Boundary conditions provide the foundation for recursive computations. These base cases correspond to trivial subproblems that can be solved directly without further decomposition:

$$\text{OPT}[0] = \text{base\_value}. \tag{4}$$

The optimization objective varies depending on the problem context. Minimization problems seek $\min_j$, maximization problems use $\max_j$, and counting problems sum over all valid transitions. Each formulation requires careful consideration of how subproblem solutions combine.

A.3.5 IMPLEMENTATION STRATEGIES

Dynamic programming algorithms can be implemented using two primary approaches: top-down memoization and bottom-up tabulation (Cormen et al., 2009).

Top-down memoization preserves the natural recursive structure while avoiding redundant computations through result caching. The algorithm begins with the original problem and recursively decomposes it into subproblems, storing computed results in a memoization table:

$$\text{memo}[i] = \begin{cases} \text{computed\_value} & \text{if already solved,} \\ \text{recursive\_solve}(i) & \text{otherwise.} \end{cases} \tag{5}$$

Bottom-up tabulation systematically solves subproblems in order of increasing size, building solutions iteratively from base cases to the final answer. This approach typically follows the pattern:

$$\text{for } i = 1 \text{ to } n: \quad \text{OPT}[i] = f(\text{OPT}[0], \ldots, \text{OPT}[i-1]). \tag{6}$$

The choice between these approaches depends on several factors. Memoization proves advantageous when only a subset of all possible subproblems requires solution, while tabulation offers better cache locality and predictable memory access patterns for dense subproblem spaces.

Space optimization techniques can significantly reduce memory requirements. Common optimizations include maintaining only the most recently computed results when the recurrence relation depends on a fixed number of previous values, reducing space complexity from $O(n)$ to $O(1)$ in many cases.

APPENDIX A.4: ASAF FRAMEWORK VISUAL ILLUSTRATION

A.4.1 COARSE-GRAINED OPTIMIZATION PHASE

Figure 5 illustrates the coarse-grained optimization phase of the ASAF framework, which addresses the fundamental challenge of determining optimal layer grouping strategies and sparsity interval refinement. The process begins with individual transformer layers $L_1, L_2, \ldots, L_p$ from the large language model, where each layer is represented as an independent computational unit with specific FLOPs characteristics ϕ_l.

The grouping transformation process aggregates these individual layers into cohesive groups $G_1, G_2, \ldots, G_n$, where each group G_i contains a set of consecutive layers $\mathcal{L}_i = \{l_{\text{start}}^{(i)}, l_{\text{start}}^{(i)} +$

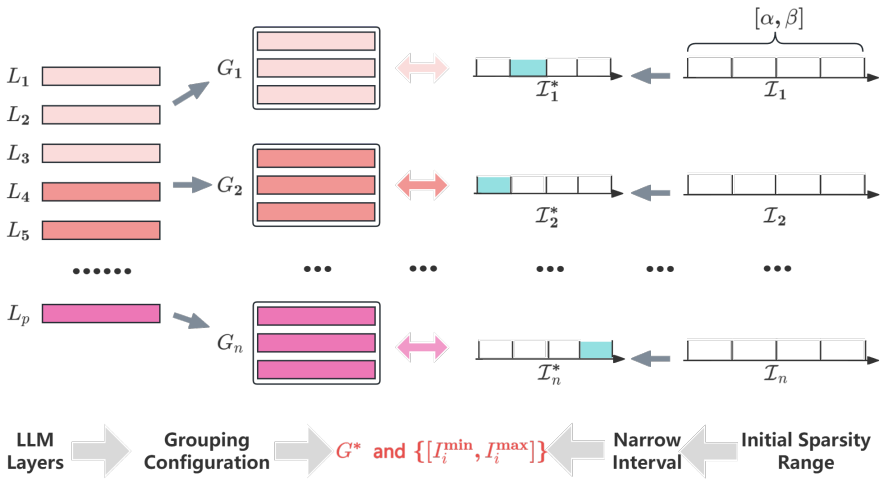

Figure 5: Coarse-grained optimization process in the ASAF framework. The phase transforms individual LLM layers $L_1, L_2, \ldots, L_p$ into optimal grouping configurations $G_1, G_2, \ldots, G_n$ while simultaneously refining sparsity search intervals from the initial range $[\alpha, \beta]$ to narrowed intervals $\{[I_i^{\min}, I_i^{\max}]\}$ for each group, where $G^* = n$ denotes the optimal number of groups. The optimization process employs iterative interval refinement strategies $\mathcal{I}_1^*, \mathcal{I}_2^*, \ldots, \mathcal{I}_n^*$ to achieve computational efficiency while maintaining sensitivity constraints.

$1, \ldots, l_{\text{end}}^{(i)}\}$. This grouping strategy ensures that layers within each group can be managed collectively while preserving the sequential structure of the transformer architecture.

The simultaneous sparsity interval refinement process transforms the initial broad search space $[\alpha, \beta]$ into narrowed, group-specific intervals $\{[I_i^{\min}, I_i^{\max}]\}$. This refinement mechanism employs dynamic programming principles to systematically reduce the search complexity while maintaining optimality guarantees. The refinement process utilizes the RefineIntervals$(\cdot)$ and NarrowInterval$(\cdot)$ functions to achieve progressive convergence toward optimal sparsity allocations.

The optimization trajectory shown in the figure demonstrates how the algorithm iteratively narrows the search space through successive refinements $\mathcal{I}_1^*, \mathcal{I}_2^*, \ldots, \mathcal{I}_n^*$, where each iteration produces tighter bounds on feasible sparsity rates. This process continues until convergence criteria are satisfied, yielding the optimal number of groups G^* and their corresponding refined sparsity intervals.

A.4.2 Fine-Grained Optimization Phase

Figure 6 depicts the fine-grained optimization phase, which performs precise decision-making within the framework established by the coarse-grained phase. This phase receives as input the optimal number of groups G^* and the refined sparsity intervals $\{[I_i^{\min}, I_i^{\max}]\}$ determined in the previous phase.

The layer allocation component systematically determines the exact number of consecutive layers assigned to each group. This process involves solving the optimization problem $\{j^*, s^*\}_i = \arg\min_{j,s}\{T[i][j][s]\}$ for each group i, where $T[i][j][s]$ represents the total FLOPs cost for group i containing j consecutive layers with sparsity rate s. The allocation process ensures complete coverage of all transformer layers while respecting continuity constraints.

The sparsity selection component operates in parallel with layer allocation, determining optimal sparsity rates from the discretized candidate sets generated by the Discretize$(\cdot)$ function. For each group i, the algorithm evaluates sparsity candidates $S_{\text{cand}} = \{I_i^{\min} + k\Delta : k \in \mathbb{Z}, 0 \leq k \leq \lfloor (I_i^{\max} - I_i^{\min})/\Delta \rfloor\}$ with sampling resolution $\Delta = 0.5\%$.

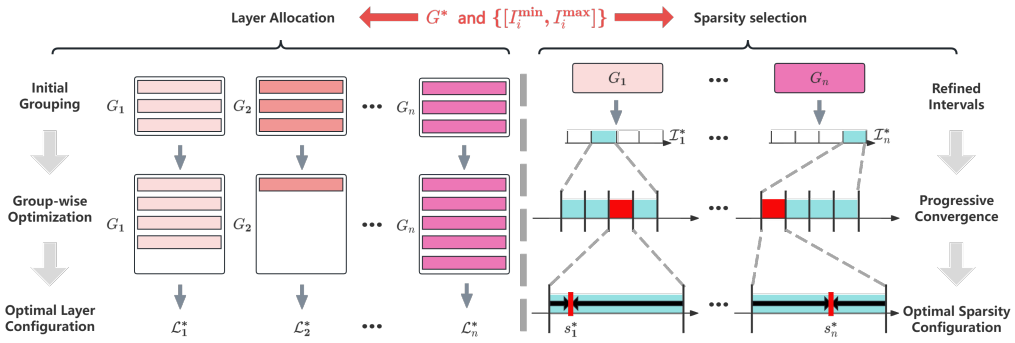

Figure 6: Fine-grained optimization process in the ASAF framework. Building upon the coarse-grained results G^* and refined intervals $\{[I_i^{\min}, I_i^{\max}]\}$, this phase performs precise layer allocation and sparsity selection through group-wise optimization. The process progressively converges from initial grouping configurations through iterative optimization to optimal layer configurations $\mathcal{L}_1^*, \mathcal{L}_2^*, \ldots, \mathcal{L}_n^*$ and optimal sparsity rates $s_1^*, s_2^*, \ldots, s_n^*$, where $G^* = n$ denotes the optimal number of groups.

The progressive convergence mechanism illustrated in the figure shows how the optimization iteratively refines both layer assignments and sparsity selections. The process employs tabulation table lookups $H[l][j][s]$ to efficiently evaluate different configurations without redundant computation. Each iteration improves upon previous solutions by leveraging the optimal substructure property inherent in the problem formulation.

The final optimization output produces the complete ASAF solution: optimal layer configurations $\mathcal{L}_1^*, \mathcal{L}_2^*, \ldots, \mathcal{L}_n^*$ specifying the exact layer ranges for each group, and optimal sparsity rates $s_1^*, s_2^*, \ldots, s_n^*$ that minimize total computational FLOPs while satisfying sensitivity constraints $\sum_{i=1}^{G^*} \xi(\mathcal{L}_i^*, s_i^*) \leq \delta_{\max}$.

The two-phase decomposition strategy demonstrated in Figures 5 and 6 exemplifies the dynamic programming approach by transforming a complex combinatorial optimization problem into a sequence of manageable subproblems. This decomposition achieves polynomial time complexity while maintaining optimality guarantees, making the ASAF framework computationally tractable for practical large language model deployment scenarios.

Appendix A.5: Coarse-grained Optimization

The coarse-grained optimization phase serves as the foundation of our ASAF framework by establishing the optimal group structure and narrowing the sparsity search space for subsequent fine-grained optimization. This phase addresses the fundamental challenge of determining how many layer groups should be formed and what sparsity intervals each group should operate within, effectively decomposing the exponential search space into manageable subproblems.

A.5.1 Mathematical Formulation

The coarse-grained optimization addresses the first subproblem of our original formulation in Equation **??** by minimizing the total computational FLOPs across all possible group configurations while satisfying accuracy degradation constraints:

$$\{G^*, \{I_i^*\}_{i=1}^{G^*}\} = \arg \min_{G, \{I_i\}_{i=1}^{G}} \left\{ \sum_{i=1}^{G} \min_{\mathcal{L}_i, s_i \in I_i} \sum_{l \in \mathcal{L}_i} \phi_l \times (1 - s_i) \right\}, \tag{7}$$

where G represents the number of groups, I_i denotes the sparsity interval for group i, $\mathcal{L}_i$ represents consecutive layers in group i, s_i is the sparsity rate for group i, and ϕ_l is the original computa-

tional cost of layer l. The constraint ensures $\sum_{i=1}^{G} \xi(\mathcal{L}_i, s_i) \leq \delta_{\max}$, where $\xi(\mathcal{L}_i, s_i)$ quantifies the accuracy degradation caused by applying sparsity rate s_i to layer group $\mathcal{L}_i$.

The dynamic programming state formulation defines $\text{DP}_{\text{coarse}}[g][b]$ as the minimum total FLOP cost when using exactly g groups with discretized accuracy degradation budget b:

$$\text{DP}_{\text{coarse}}[g][b] = \min_{\{I_i\}_{i=1}^{g}} \left\{ \sum_{i=1}^{g} \min_{\mathcal{L}_i, s_i \in I_i} \sum_{l \in \mathcal{L}_i} \phi_l \times (1 - s_i) \right\}, \tag{8}$$

where the state represents the optimal cost achievable with g groups under accuracy degradation budget $b \times \Delta$ (with Δ being the discretization step), and the minimization considers all valid interval partitions and corresponding layer-sparsity assignments.

The state transition equation considers all possible ways to add one more group by selecting an appropriate interval subset:

$$\text{DP}_{\text{coarse}}[g][b] = \min_{I \subseteq [\alpha, \beta]} \{\text{OptimalCost}(I) + \text{DP}_{\text{coarse}}[g-1][b - \xi_{\text{cost}}(I)/\Delta]\}, \tag{9}$$

where I represents the sparsity interval assigned to the g-th group, $\text{OptimalCost}(I)$ denotes the minimum FLOP cost achievable within interval I, and $\xi_{\text{cost}}(I)$ represents the accuracy degradation.

Algorithm 1 Coarse-grained Optimization

Require: Layer count p, sparsity range $[\alpha, \beta]$, accuracy degradation threshold $\delta_{\max}$
Ensure: Optimal group number G^* and intervals $\{[I_i^{\min}, I_i^{\max}]\}$

```
Initialize DP_coarse[0..p][0..δ_max/Δ] ← ∞
Set boundary condition: DP_coarse[0][0] ← 0
Generate interval candidates 𝓘 = {I : I ⊆ [α, β]}
Main DP Loop:
for g = 1 to p do
    for b = 0 to δ_max/Δ do
        for each interval I ∈ 𝓘 do
            cost ← OptimalCost(I) using tabulation H
            accuracy_cost ← ξ_cost(I)/Δ using tabulation Ξ
            if b − accuracy_cost ≥ 0 then
                total_cost ← cost + DP_coarse[g − 1][b − accuracy_cost]
                DP_coarse[g][b] ← min(DP_coarse[g][b], total_cost)
            end if
        end for
    end for
end for
Solution Extraction:
G* ← arg min_g {min_b DP_coarse[g][b]}
{I_i^min, I_i^max} ← BacktrackOptimalIntervals(DP_coarse, G*)
return G*, {[I_i^min, I_i^max]}_{i=1}^{G*} =0
```

A.5.2 Algorithmic Analysis

The dynamic programming formulation operates on a two-dimensional state space where $\text{DP}_{\text{coarse}}[g][b]$ represents the minimum achievable computational cost when utilizing exactly g groups with discretized accuracy degradation budget b. The discretization parameter $\Delta = 0.5\%$ transforms the continuous constraint into a tractable discrete optimization problem while maintaining sufficient precision for practical deployment.

The initialization establishes boundary conditions essential for correctness. The infinite initialization for $\text{DP}_{\text{coarse}}[g][b]$ ensures only valid transitions result in finite costs, while $\text{DP}_{\text{coarse}}[0][0] = 0$ represents the base case with no groups and no degradation.

The interval generation constructs candidate sparsity intervals $\mathcal{I}$ that partition $[\alpha, \beta]$ into meaningful subranges. Each interval $I \in \mathcal{I}$ represents a potential sparsity operating range for a layer group, enabling exploration of different sparsity allocation granularities.

The core dynamic programming loop systematically explores group counts and accuracy budgets through three nested iterations. The outermost loop over g considers configurations from single groups to maximum granularity where $G_{\max} = p$. The middle loop over budget b discretizes the accuracy constraint, while the innermost loop over intervals I evaluates each potential sparsity assignment.

Within each iteration, the algorithm queries tabulation tables H and Ξ for precomputed costs and degradation values. The tabulation mechanism transforms expensive evaluations into constant-time operations, enabling scalability to large layer counts. The cost computation combines immediate interval contribution with optimal subproblem costs, maintaining optimal substructure.

The feasibility check $b - \text{accuracy\_cost} \geq 0$ ensures configurations remain within the accuracy threshold $\delta_{\max}$. When feasible configurations are identified, the minimization operation updates the dynamic programming table only for improved solutions.

The solution extraction identifies the optimal configuration by examining all computed states: $G^* = \arg\min_g \{\min_b \text{DP}_{\text{coarse}}[g][b]\}$. The backtracking procedure reconstructs specific interval assignments by tracing decisions that led to the optimal solution, producing refined sparsity intervals that narrow the search space from $[\alpha, \beta]$.

State Transition Intuition. The dynamic programming transitions embody intuitive decision-making processes. In the coarse-grained phase, each state transition addresses the question: "Given that I have optimally allocated the first $g-1$ groups, what sparsity interval should I assign to the g-th group to minimize total cost while staying within the accuracy budget?" This decomposition enables systematic exploration of interval assignments without reconsidering previous group decisions. The state $\text{DP}_{\text{coarse}}[g][b]$ encapsulates all optimal ways to use exactly g groups with budget b, allowing efficient evaluation of adding one more group with a specific interval choice.

A.5.3 Complexity Analysis

Upon completion, the coarse-grained optimization produces two critical outputs: the optimal number of groups G^* that minimizes computational overhead while satisfying accuracy constraints, and refined sparsity intervals $\{[I_i^{\min}, I_i^{\max}]\}_{i=1}^{G^*}$ that narrow the search space from the original range $[\alpha, \beta]$. These outputs provide essential guidance for the subsequent fine-grained optimization phase, significantly reducing the search space while preserving the potential for globally optimal solutions.

Appendix A.6: Fine-grained Optimization

The fine-grained optimization phase receives the optimal group number G^* and refined sparsity intervals from the coarse-grained phase and determines the precise layer allocation and sparsity rate assignment that minimizes computational FLOPs while satisfying accuracy constraints. This phase operates on a significantly reduced search space compared to the original problem, enabling detailed optimization within the refined parameter ranges.

A.6.1 Mathematical Formulation

The fine-grained optimization addresses the second subproblem of our original formulation by minimizing the exact total computational FLOPs through optimal layer partitioning and sparsity assignment:

$$\{\{\mathcal{L}_i^*\}_{i=1}^{G^*}, \{s_i^*\}_{i=1}^{G^*}\} = \arg \min_{\{\mathcal{L}_i\}_{i=1}^{G^*}, \{s_i\}_{i=1}^{G^*}} \left\{ \sum_{i=1}^{G^*} \sum_{l \in \mathcal{L}_i} \phi_l \times (1 - s_i) \right\}, \tag{10}$$

where $\{\mathcal{L}_i\}_{i=1}^{G^*}$ represents the layer allocation with each $\mathcal{L}_i$ containing consecutive layers, and $\{s_i\}_{i=1}^{G^*}$ denotes the sparsity rates. The constraints ensure $\sum_{i=1}^{G^*} \xi(\mathcal{L}_i, s_i) \leq \delta_{\max}$, $s_i \in [I_i^{\min}, I_i^{\max}]$, $\mathcal{L}_i$ are consecutive, $\bigcup_{i=1}^{G^*} \mathcal{L}_i = \{1, 2, \ldots, p\}$, and $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset$ for $i \neq j$.

The dynamic programming state formulation defines $\text{DP}_{\text{fine}}[i][g][b]$ as the minimum total FLOP cost for optimally partitioning layers $[i..p]$ into exactly g consecutive groups with remaining accuracy

degradation budget b:

$$\text{DP}_{\text{fine}}[i][g][b] = \min_{\{\mathcal{L}_k\}_{k=1}^{g}, \{s_k\}_{k=1}^{g}} \left\{ \sum_{k=1}^{g} \sum_{l \in \mathcal{L}_k} \phi_l \times (1 - s_k) \right\}, \tag{11}$$

where the state covers layers from position i to p, requires exactly g groups, has remaining budget b, and ensures $\bigcup_{k=1}^{g} \mathcal{L}_k = \{i, i+1, \ldots, p\}$ with each $\mathcal{L}_k$ consecutive, $s_k \in [I_k^{\min}, I_k^{\max}]$, and $\sum_{k=1}^{g} \xi(\mathcal{L}_k, s_k) \leq b$.

Algorithm 2 Fine-grained Optimization

Require: G^* groups, intervals $\{[I_i^{\min}, I_i^{\max}]\}$, tabulation tables H, Ξ
Ensure: Optimal allocation $\{\mathcal{L}_i^*\}$ and sparsity rates $\{s_i^*\}$

```
Initialize DP_fine[1..p+1][0..G*][0..δ_max/Δ] ← ∞
Initialize choice[1..p+1][0..G*][0..δ_max/Δ] ← ∅
Set boundary condition: DP_fine[p+1][0][b] ← 0 for all b ≥ 0
Backward DP Construction:
for i = p down to 1 do
  for g = 1 to min(G*, p − i + 1) do
    for b = 0 to δ_max/Δ do
      for j = i to p − g + 1 do
        for s ∈ Discretize([I^min_{G*−g+1}, I^max_{G*−g+1}]) do
          group_cost, accuracy_cost ← H[i][j − i + 1][s], Ξ[i][j − i + 1][s]/Δ
          if b − accuracy_cost ≥ 0 then
            total_cost ← group_cost + DP_fine[j + 1][g − 1][b − accuracy_cost]
            if total_cost < DP_fine[i][g][b] then
              DP_fine[i][g][b] ← total_cost
              choice[i][g][b] ← (j, s)
            end if
          end if
        end for
      end for
    end for
  end for
end for
Solution Reconstruction:
{L*_i, s*_i} ← BacktrackSolution(choice, 1, G*, δ_max/Δ)
return {L*_i}_{i=1}^{G*}, {s*_i}_{i=1}^{G*} =0
```

The state transition equation jointly enumerates all possible first-group formations and sparsity assignments within the corresponding refined interval. The transition first identifies the optimal group end position j and sparsity rate s:

$$j^*, s^* = \arg \min_{\substack{j \in [i, p-g+1] \\ s \in [I^{\min}_{G^*-g+1}, I^{\max}_{G^*-g+1}]}} \{H[i][j-i+1][s] + \Xi[i][j-i+1][s]\}. \tag{12}$$

Then the state transition equation becomes:

$$\text{DP}_{\text{fine}}[i][g][b] = H[i][j^* - i + 1][s^*] + \text{DP}_{\text{fine}}[j^* + 1][g-1][b - \Xi[i][j^* - i + 1][s^*]], \tag{13}$$

where j defines the end of the first group (layers i to j), s is the sparsity rate from the refined interval for this group position, $H[i][j-i+1][s]$ provides the exact FLOP cost from tabulation, and $\Xi[i][j-i+1][s]$ gives the exact accuracy degradation cost.

A.6.2 Algorithmic Analysis

The fine-grained optimization employs three-dimensional dynamic programming where $\text{DP}_{\text{fine}}[i][g][b]$ represents the minimum cost for partitioning layers $[i..p]$ into exactly g consecutive groups with remaining budget b. The backward construction enables locally optimal decisions

while maintaining global optimality through optimal substructure. The choice table records specific decisions for efficient solution reconstruction.

The boundary condition $\text{DP}_{\text{fine}}[p+1][0][b] = 0$ provides the foundation for backward construction. The backward process begins from the final layer and works toward the initial layer, systematically considering all possible first-group formations in each subproblem.

The nested loops systematically enumerate all feasible configurations. The constraint $g \leq \min(G^*, p-i+1)$ prevents creating more groups than specified or available. The group end position $j \leq p-g+1$ ensures sufficient layers remain for subsequent groups. The sparsity rate s is constrained to refined intervals $[I^{\min}_{G^*-g+1}, I^{\max}_{G^*-g+1}]$, significantly reducing search space compared to $[\alpha, \beta]$.

Tabulation lookups retrieve precomputed values: $H[i][j-i+1][s]$ for FLOP costs and $\Xi[i][j-i+1][s]/\Delta$ for accuracy degradation. The feasibility check $b - \text{accuracy\_cost} \geq 0$ maintains constraint satisfaction. Cost updates occur only when improvements are found, maintaining optimality while avoiding redundant computations.

The solution reconstruction traverses the choice table from initial state $(1, G^*, \delta_{\max}/\Delta)$ and follows recorded decisions to construct the complete solution. Each choice (j, s) specifies group boundaries and sparsity rates, continuing until all groups are identified.

State Transition Intuition. The fine-grained optimization addresses the question: "To optimally partition layers $[i..p]$ into exactly g consecutive groups with remaining budget b, which layers should form the first group and what sparsity rate should be applied?" The backward construction ensures that when deciding the first group boundary j and sparsity s, all subsequent decisions from layer $j+1$ onward are already optimal. This decomposition transforms the complex joint optimization of layer allocation and sparsity assignment into a sequence of local decisions with global optimality guarantees.

A.6.3 Complexity Analysis

Upon completion, the fine-grained optimization generates the complete optimal solution: precise consecutive layer allocation $\{\mathcal{L}_i^*\}_{i=1}^{G^*}$ and optimal sparsity rates $\{s_i^*\}_{i=1}^{G^*}$ that minimize computational FLOPs. Each $\mathcal{L}_i^* = \{l_{\text{start}}^{(i)}, l_{\text{start}}^{(i)}+1, \ldots, l_{\text{end}}^{(i)}\}$ defines consecutive layers satisfying continuity and completeness constraints, while $s_i^* \in [I_i^{\min}, I_i^{\max}] \subseteq [\alpha, \beta]$ ensures adherence to the refined sparsity intervals.

Appendix A.7: Tabulation

The tabulation mechanism forms the computational backbone of the ASAF framework by providing efficient access to FLOP costs and accuracy degradation metrics for arbitrary consecutive layer sequences and sparsity rate combinations. This precomputation strategy transforms the dynamic programming algorithms from computationally prohibitive procedures into tractable optimization methods suitable for large-scale model deployment.

A.7.1 Mathematical Formulation

The tabulation mechanism constructs two three-dimensional tables that provide $O(1)$ lookup for any layer sequence and sparsity combination:

$$H[\text{start}][\text{length}][s] = \sum_{l=\text{start}}^{\text{start}+\text{length}-1} \phi_l \times (1-s), \tag{14}$$

$$\Xi[\text{start}][\text{length}][s] = \xi(\{\text{start}, \text{start}+1, \ldots, \text{start}+\text{length}-1\}, s), \tag{15}$$

where H stores FLOP costs and Ξ stores accuracy degradation costs. Here, $\text{start} \in [1, p]$ denotes the starting layer index, $\text{length} \in [1, p-\text{start}+1]$ represents the number of consecutive layers, $s \in S_{\text{discrete}}$ is the discretized sparsity rate with resolution $\Delta = 0.5\%$, ϕ_l is the original computational cost of layer l, and $\xi(\cdot)$ quantifies accuracy degradation.

The discrete sparsity set is defined as:

$$S_{\text{discrete}} = \{\alpha + k\Delta : k \in \mathbb{N}, \alpha + k\Delta \leq \beta\}. \tag{16}$$

This discretization partitions the continuous sparsity range $[\alpha, \beta]$ into uniformly spaced discrete points, enabling efficient tabulation while maintaining sufficient precision for optimization purposes.

Algorithm 3 Tabulation Construction

Require: Model layers $\{1, 2, \ldots, p\}$, sparsity discretization $\Delta = 0.5\%$
Ensure: Tabulation tables H, Ξ

```
Initialize H[1..p][1..p][|S_discrete|] ← 0
Initialize Ξ[1..p][1..p][|S_discrete|] ← 0
S_discrete ← {α + kΔ : k ∈ ℕ, α + kΔ ≤ β}
Systematic Pre-computation:
for start = 1 to p do
    for length = 1 to p − start + 1 do
        L_seq ← {start, start + 1, ..., start + length − 1}
        for each sparsity s ∈ S_discrete do
            H[start][length][s] ← Σ_{l∈L_seq} φ_l × (1 − s)
            Ξ[start][length][s] ← ξ(L_seq, s)
        end for
    end for
end for
return H, Ξ =0
```

A.7.2 Algorithmic Analysis

The tabulation construction systematically precomputes FLOP costs and accuracy degradation values for all consecutive layer sequences and discretized sparsity rates. The three-dimensional tables H and Ξ enable constant-time lookup during optimization phases, eliminating redundant calculations and concentrating computational burden in preprocessing.

The discrete sparsity set S_{discrete} balances computational efficiency with optimization precision. The systematic precomputation exhaustively evaluates all valid combinations of starting positions, sequence lengths, and sparsity rates. The constraint length $\leq p - \text{start} + 1$ maintains validity by preventing sequences extending beyond available layers.

The FLOP cost calculation $H[\text{start}][\text{length}][s] = \sum_{l \in \mathcal{L}_{\text{seq}}} \phi_l \times (1 - s)$ represents total computational cost when applying sparsity rate s. The factor $(1 - s)$ reflects computational reduction from sparsification, while summation over ϕ_l accounts for heterogeneous layer costs.

The accuracy degradation calculation $\Xi[\text{start}][\text{length}][s] = \xi(\mathcal{L}_{\text{seq}}, s)$ quantifies performance impact. The sensitivity function $\xi(\cdot)$ requires careful design balancing accuracy with computational tractability. Gradient-based methods provide theoretical foundations but require significant resources, while activation-based methods offer efficiency at potential accuracy cost.

The tabulation tables enable the transformation of dynamic programming state transitions from $O(\text{group\_size})$ computations to $O(1)$ operations. Without tabulation, each state transition would require recomputation of FLOP costs and accuracy degradation values, leading to prohibitive computational complexity for large models. The precomputation strategy concentrates the computational burden in the preprocessing phase while ensuring that the subsequent optimization phases operate with maximum efficiency.

A.7.3 Tabulation Storage Requirements

Storage Specifications. The tabulation tables H and Ξ require $\mathcal{O}(L^2 \times |S_{\text{discrete}}|)$ storage each. For Llama-2-7B with $L = 32$ layers and $|S_{\text{discrete}}| = 29$ sparsity levels, each table requires approximately 0.76 MB in single precision. The preprocessing computation is $\mathcal{O}(L^2 \times |S_{\text{discrete}}| \times C_\xi)$ where C_ξ represents the cost of evaluating the sensitivity function $\xi(\cdot)$.

Reusability. Tables can be computed once per model and reused across multiple optimization scenarios. The constant-time lookup capability enables efficient exploration of large solution spaces.

APPENDIX A.8: COMPLETE HYPERPARAMETER SPECIFICATIONS

A.8.1 ASAF FRAMEWORK CONFIGURATION

Dynamic Programming Parameters. Our ASAF framework explores sparsity allocations within the constrained range where $\alpha = 1\%$ represents the minimum sparsity threshold and $\beta = 15\%$ defines the maximum sparsity level, following established practices in neural network pruning (Han et al., 2015). The maximum allowable accuracy degradation is set to $\delta_{\max} = 1\%$ to ensure practical deployment viability. Tabulation sampling resolution is configured as $\Delta = 0.5\%$, providing sufficient granularity for optimization while maintaining computational tractability. The dynamic programming state space discretizes the accuracy degradation budget into 200 steps, with maximum group exploration set to the total layer count to allow full granularity in layer allocation decisions.

Tabulation Construction. The sensitivity measurement employs 1000 calibration samples to ensure statistical robustness during tabulation construction (Frantar & Alistarh, 2023). Tabulation cache allocation is limited to 2048 MB to accommodate GPU memory constraints while providing sufficient storage for memoization. All tabulation computations utilize FP32 precision accumulation to maintain numerical stability throughout the dynamic programming process, preventing precision degradation that could affect optimization quality (Jacob et al., 2018).

A.8.2 LEARNING RATE AND OPTIMIZATION CONFIGURATION

Learning Rate Schedule. The base learning rate for fine-tuning is set to $\gamma_0 = 1 \times 10^{-5}$, representing 0.01× of typical pre-training rates to account for the sensitivity of sparse structures (Han et al., 2015). We employ cosine annealing with warm restarts over $T_{\text{total}} = 1000$ steps following established practices in quantization fine-tuning (Frantar et al., 2022):

$$\gamma_t = \gamma_{\min} + \frac{1}{2}(\gamma_0 - \gamma_{\min}) \left(1 + \cos\left(\frac{t}{T_{\text{total}}}\pi\right)\right), \tag{17}$$

where $\gamma_{\min} = 1 \times 10^{-7}$ prevents complete learning rate decay.

Optimizer Configuration. We utilize AdamW optimizer with parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, following standard configurations for transformer fine-tuning (Touvron et al., 2023). Weight decay is set to $w = 0.01$ to provide L2 regularization without interfering with the sparse structure. Gradient clipping threshold is configured at clip_value $= 1.0$ to prevent training instability during sparse fine-tuning (Ashkboos et al., 2024b).

A.8.3 QUANTIZATION CONFIGURATION

Weight Quantization Parameters. Weight quantization employs 4-bit precision for all linear layers, with per-channel symmetric quantization applied to preserve fine-grained statistics (Frantar et al., 2022). The quantization scale s for each channel c is computed as:

$$s_c = \frac{\max(|W_c|) \cdot \text{clip\_ratio}}{2^{\text{bits}-1} - 1}, \tag{18}$$

where clip_ratio $= 0.9$ controls the quantization range to handle outliers. For weight quantization, we employ both round-to-nearest (RTN) method (Jacob et al., 2018) and GPTQ (Frantar et al., 2022) approaches. We use per-column symmetric quantization (Jacob et al., 2018) with group size 128 to balance quantization quality and computational efficiency. During GPTQ quantization, we utilize 128 samples from the WikiText-2 dataset (Merity et al., 2016) with sequence length of 2048 as the calibration set, following established protocols for transformer quantization.

Activation Quantization Parameters. We apply per-token symmetric quantization to quantize the inputs, where each row of the activation matrix shares a common quantization scale (Xiao et al., 2023). The clipping ratio is fixed at 0.9 across all experiments to maintain consistent quantization behavior. This approach effectively handles the dynamic range of activations while preserving computational efficiency during sparse matrix operations.

KV Cache Quantization. The KV caches are quantized using asymmetric quantization (Dettmers et al., 2022), organized into groups of 128 elements to match the head dimension structure of transformer architectures. A constant clipping ratio of 0.95 is applied to accommodate the typically wider dynamic range of cached key-value pairs compared to standard activations (Ashkboos et al., 2024b).

A.8.4 Hardware-Specific Parameters

Batch Configuration. Training employs micro-batch size of 1 with gradient accumulation over 8 steps, yielding an effective batch size of 8. This configuration optimizes memory usage on RTX 3090 GPUs while maintaining training stability for large language models (Touvron et al., 2023). Sequence length is fixed at 2048 tokens to match evaluation conditions and ensure consistent memory allocation patterns during optimization.

Memory Management. Mixed precision training uses FP16 for forward passes and FP32 for gradient computation, following established practices for stable quantization fine-tuning (Ashkboos et al., 2024b). The maximum memory allocation is set to 10.5 GB to account for CUDA overhead on 12 GB GPU memory configurations. Dynamic loss scaling starts at 2^{16} with automatic adjustment to prevent gradient underflow, ensuring numerical stability throughout the sparse optimization process (Jacob et al., 2018).

A.8.5 Evaluation Configuration

Language Generation Tasks. We evaluate on WikiText-2 perplexity using 2048-token sequences with a sliding window stride of 512 for comprehensive coverage (Merity et al., 2016). Throughput evaluation encompasses batch sizes of 1, 4, 16, and 32 to assess scalability across different deployment scenarios. All language generation tasks utilize greedy decoding with temperature set to 0.0 to ensure deterministic and reproducible results.

Zero-shot Classification. We assess our framework across six established benchmarks: PIQA (Bisk et al., 2020), WinoGrande (Sakaguchi et al., 2021), HellaSwag (Zellers et al., 2019), LAMBADA (Radford et al., 2019), ARC-Easy and ARC-Challenge (Clark et al., 2018). All experiments utilize the LM Evaluation Harness (Gao et al., 2021; 2024) with default configurations. Maximum generation length is limited to 256 tokens with KV cache enabled for efficient inference during evaluation.

A.8.6 Target Matrix Configuration

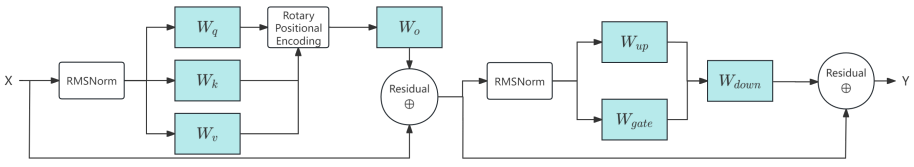

Figure 7: Target weight matrices in transformer layers for ASAF quantization and sparsification.

Figure 7 illustrates the complete transformer layer architecture highlighting the seven target weight matrices that undergo both 4-bit quantization and adaptive sparsity allocation in our ASAF framework. The green boxes represent the linear projection matrices: query (W_q), key (W_k), value (W_v), and output (W_o) projections in the multi-head attention mechanism, as well as up-projection (W_{up}), gate projection (W_{gate}), and down-projection (W_{down}) matrices in the feed-forward network. These matrices constitute the primary computational bottlenecks in transformer inference and are the focus of our layer-group-based optimization strategy, while components like RMSNorm and positional encoding remain unmodified to preserve numerical stability.

Matrix-Level Configuration. Our ASAF framework applies unified quantization and adaptive sparsity allocation to all linear projection matrices within each transformer layer, as illustrated in Figure 7. The attention mechanism matrices include query projection $W_q \in \mathbb{R}^{d_{model} \times d_{head}}$, key projection $W_k \in \mathbb{R}^{d_{model} \times d_{head}}$, value projection $W_v \in \mathbb{R}^{d_{model} \times d_{head}}$, and output projection $W_o \in \mathbb{R}^{d_{head} \times d_{model}}$ (Touvron et al., 2023). The feed-forward network matrices consist of

up-projection $W_{up} \in \mathbb{R}^{d_{model} \times d_{ff}}$, gate projection $W_{gate} \in \mathbb{R}^{d_{model} \times d_{ff}}$, and down-projection $W_{down} \in \mathbb{R}^{d_{ff} \times d_{model}}$ where $d_{ff} = 4 \times d_{model}$ following standard transformer scaling (Touvron et al., 2023).

Layer-Group Sparsity Application. Each target matrix within a layer group receives identical sparsity allocation determined by our dynamic programming optimization. For layer group $\mathcal{L}_i$ with optimized sparsity rate s_i^*, all seven projection matrices (W_q, W_k, W_v, W_o, W_{up}, W_{gate}, W_{down}) undergo magnitude-based pruning at rate s_i^* (Han et al., 2015). This uniform application ensures consistent computational benefits across all matrix operations within each layer while preserving the structural integrity of the transformer architecture. The RMSNorm parameters and positional encoding remain unmodified to maintain numerical stability during inference (**?**).

Quantization Integration. Following the rotation-based quantization pipeline (Ashkboos et al., 2024b), each target matrix first undergoes 4-bit GPTQ quantization (Frantar et al., 2022) before sparsity application. The quantization process preserves the relative magnitude relationships critical for effective pruning, while the subsequent sparsification leverages the quantized weight distributions for optimal efficiency. This sequential approach ensures compatibility between the compression techniques while maximizing hardware acceleration potential on modern GPU architectures.

APPENDIX A.9: ACCURACY RESULTS

Table 1: WikiText-2 perplexity comparison for Llama-2 models (2048 sequence length) using 4-bit quantization. SmoothQuant and OmniQuant results are from (Shao et al., 2023), and 128G indicates group-wise quantization with 128 group size. Our ASAF framework applies layer-group-based pruning, with 4-bit precision across weights, activations, and KV caches. Lower perplexity indicates better performance.

Method	Weight Quantization	#Outlier Features	Llama-2 7B	13B	30B	70B
Baseline	-	-	5.47	4.88	4.09	3.32
SmoothQuant (Xiao et al., 2023)	RTN	0	83.12	35.88	-	-
OmniQuant (Shao et al., 2023)	RTN	0	14.26	12.30	-	-
QUIK-4B (Ashkboos et al., 2023)	GPTQ	256	8.87	7.78	7.28	6.91
QuaRot	GPTQ	0	**6.10**	**5.40**	**4.41**	**3.79**
ASAF (Ours)	GPTQ	0	6.14	5.44	4.44	3.82
Atom-128G (Zhao et al., 2023)	GPTQ-128G	128	6.03	**5.26**	-	-
QuaRot-128G		0	**5.93**	**5.26**	**4.25**	**3.61**
ASAF-128G (Ours)		0	5.98	5.30	4.28	3.64

Language Generation Tasks. We evaluate our ASAF framework on the WikiText-2 language-generation benchmark. Table 1 reports the perplexity after quantizing Llama-2 weights to 4 bits with GPTQ and applying our adaptive sparsity allocation across layer groups. Our framework demonstrates competitive performance compared to state-of-the-art quantization methods, achieving perplexity degradation less than 1% compared to QuaRot while providing additional computational benefits through optimized sparsity allocation. The layer-group-based pruning approach requires no additional outlier storage or asymmetric quantization schemes. When using group-size-128 quantization, ASAF maintains comparable performance with perplexity increases within 1% of QuaRot-128G while enabling more efficient inference through adaptive sparsity patterns.

Zero-Shot Tasks. We assess ASAF across six established zero-shot benchmarks: PIQA (Bisk et al., 2020), WinoGrande (Sakaguchi et al., 2021), HellaSwag (Zellers et al., 2019), LAMBADA (Radford et al., 2019), and ARC-Easy and ARC-Challenge (Clark et al., 2018). Experiments utilize the LM Evaluation Harness (Gao et al., 2021; 2024) with default configurations. Table 2 shows that ASAF maintains strong performance across all Llama-2 model sizes, with performance degradation consistently below 1% compared to QuaRot. The ASAF preserves model capabilities while enabling computational efficiency gains through optimized layer-group pruning patterns.

Prefill Stage Performance Increases. Figure 8 demonstrates the acceleration performance of ASAF across various batch configurations (1, 4, 16, and 32) with 2048-token sequences on Llama-2 models. Our adaptive sparsity allocation approach consistently outperforms the QuaRot baseline imple-

Table 2: Zero-shot accuracy of Llama models with our ASAF framework on PIQA (PQ), WinoGrande (WG), HellaSwag (HS), Arc-Easy (A-e), Arc-Challenge (A-c), and LAMBADA (LA). Our method applies adaptive sparsity allocation across layer groups.

Model	Method	PQ	WG	HS	A-e	A-c	LA	Avg.
Llama2-7B	FP16	79.11	69.06	75.99	74.58	46.25	73.90	69.82
	QuaRot	76.77	63.77	72.16	69.87	40.87	70.39	65.64
	ASAF (Ours)	76.00	63.23	71.47	69.17	40.52	69.69	65.01
Llama2-13B	FP16	80.47	72.22	79.39	77.48	49.23	76.75	72.59
	QuaRot	78.89	70.24	76.37	72.98	46.59	73.67	69.79
	ASAF (Ours)	78.26	69.68	75.76	72.25	46.19	72.97	69.19
Llama2-30B	FP16	81.13	73.94	80.72	78.52	51.65	77.59	73.93
	QuaRot	79.94	72.03	77.99	75.20	49.46	75.18	71.63
	ASAF (Ours)	79.14	71.42	77.37	74.60	48.99	74.58	71.02
Llama2-70B	FP16	82.70	77.98	83.84	80.98	57.34	79.58	77.07
	QuaRot	82.43	76.24	81.82	80.43	56.23	78.73	75.98
	ASAF (Ours)	81.77	75.48	81.12	79.71	55.81	77.98	75.31

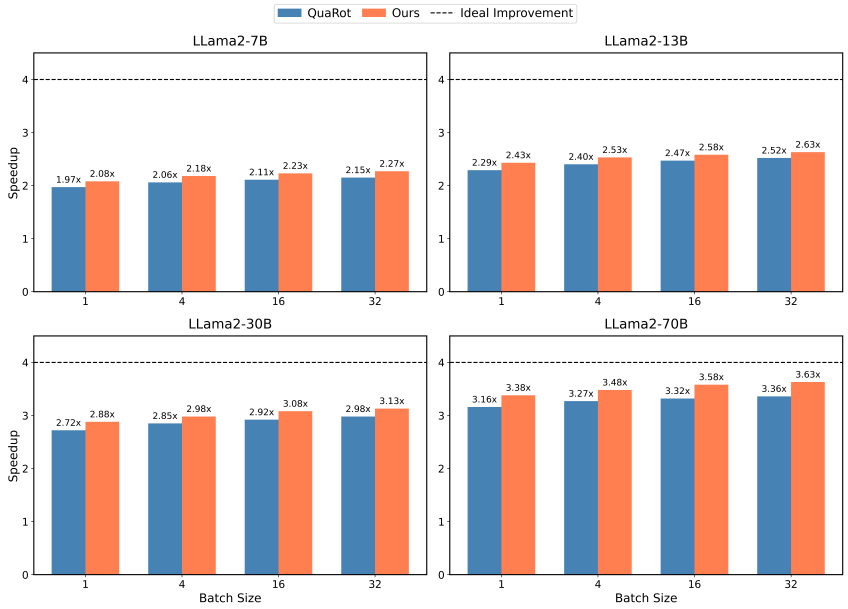

Figure 8: Performance comparison between our framework and QuaRot on Llama-2 models, evaluated on NVIDIA RTX 3090 GPUs with 2048-token sequences across various batch sizes.

mentation across all tested configurations. The performance gains become more pronounced with larger batch sizes, as the computational workload increasingly overshadows memory bandwidth limitations. For the largest 70B model, our method reaches peak acceleration of 3.63×. The results reveal a clear trend where both increasing model complexity and batch size magnify the effectiveness of our optimization strategy, demonstrating the scalable nature of the ASAF framework's adaptive layer-wise sparsity allocation mechanism.

Table 3: GPU memory consumption comparison between QuaRot and our ASAF framework across different Llama-2 model sizes. All measurements in MB for inference with 2048 sequence length at batch size 1. Compression ratios reflect adaptive sparsity allocation.

Method	Llama-2			
	7B	13B	30B	70B
QuaRot	3,255 MB	5,753 MB	11,408 MB	20,536 MB
ASAF (Ours)	3,013 MB	5,276 MB	10,110 MB	17,943 MB
Compression Ratio	**7.43%**	**8.29%**	**11.38%**	**12.63%**

Computational and Memory Efficiency. This algorithmic efficiency translates directly to memory benefits during inference. As shown in Table 3, ASAF achieves an average memory reduction of 9.93% across all model sizes, ranging from 7.43% for the 7B model to 12.63% for the 70B model, demonstrating our adaptive sparsity allocation strategy that applies conservative pruning to smaller models while aggressively leveraging redundancy in larger architectures. The layer-group-based pruning strategy enables efficient sparse matrix operations that preserve computational patterns favorable to modern GPU architectures. Combined with 4-bit quantization, our approach allows deployment of large models on resource-constrained hardware while maintaining inference quality.

RTN Quantization Strategy. To evaluate the robustness of our adaptive sparsity allocation approach, we compare ASAF's performance under RTN quantization, where GPTQ serves as the default weight quantization strategy. Table 4 demonstrates that at 8-bit precision, RTN maintains accuracy nearly identical to full precision for both approaches. At 4-bit quantization, while both methods experience some quality degradation, our ASAF framework consistently maintains performance within 1% of QuaRot results across all model sizes. In both INT4 and INT8 configurations, these findings confirm that layer-group-based adaptive sparsity allocation can be effectively combined with RTN quantization without introducing significant accuracy loss, validating the generalizability of our optimization framework.

Table 4: WikiText-2 perplexity (PPL) and zero-shot accuracy of our ASAF framework for Llama-2 models applying 4- and 8-bits with RTN weights and activation quantization. We use PIQA (PQ), WinoGrande (WG), HellaSwag (HS), Arc-Easy (A-e), Arc-Challenge (A-c), and LAMBADA (LA). We quantize all weights, activations, and caches and introduce adaptive sparsity allocation across layer groups.

Model	Method	Precision	PPL ↓	PQ ↑	WG ↑	HS ↑	A-e ↑	A-c ↑	LA ↑	Avg. ↑
7B	Baseline	FP16	5.47	79.11	69.06	75.99	74.58	46.25	73.90	69.82
	QuaRot-RTN	INT4	8.37	72.09	60.69	65.40	58.88	35.24	57.27	58.26
		INT8	5.50	78.94	68.67	75.80	74.79	45.39	74.33	69.65
	ASAF-RTN (Ours)	INT4	8.44	71.48	60.23	64.81	58.38	34.98	56.75	57.77
		INT8	5.54	78.31	68.09	75.12	74.19	45.00	73.66	69.06
13B	Baseline	FP16	4.88	80.47	72.22	79.39	77.48	49.23	76.75	72.59
	QuaRot-RTN	INT4	6.09	77.37	67.32	73.11	70.83	43.69	70.66	67.16
		INT8	4.90	80.52	71.59	79.38	77.31	49.15	76.79	72.46
	ASAF-RTN (Ours)	INT4	6.14	76.67	66.71	72.45	70.19	43.30	70.02	66.56
		INT8	4.94	79.84	70.91	78.67	76.65	48.68	76.10	71.81
30B	Baseline	FP16	4.42	81.13	73.94	80.72	78.52	51.65	77.59	73.93
	QuaRot-RTN	INT4	5.51	78.36	69.65	75.05	72.84	46.08	72.56	69.09
		INT8	4.43	81.18	73.35	80.65	78.36	51.58	77.63	73.82
	ASAF-RTN (Ours)	INT4	5.56	77.69	68.99	74.37	72.22	45.64	71.91	68.47
		INT8	4.47	80.45	72.69	79.92	77.65	51.12	76.93	73.13
70B	Baseline	FP16	3.32	82.70	77.98	83.84	80.98	57.34	79.58	77.07
	QuaRot-RTN	INT4	4.14	80.69	75.14	79.63	77.57	51.71	77.02	73.63
		INT8	3.33	82.97	77.98	83.67	80.77	58.11	79.53	77.17
	ASAF-RTN (Ours)	INT4	4.18	79.92	74.46	78.83	76.83	51.24	76.25	72.92
		INT8	3.36	82.14	77.24	82.92	79.96	57.56	78.81	76.44

APPENDIX A.10: COMPUTATIONAL EFFICIENCY ANALYSIS

A.10.1 UNIFIED COMPLEXITY ANALYSIS

Baseline Brute-Force Complexity. The naive approach requires exhaustive enumeration of all possible layer grouping and sparsity allocation configurations. For a model with L layers and $|S|$ discretized sparsity rates, the total search space includes:

- **Layer grouping**: 2^{L-1} possible ways to partition layers into consecutive groups
- **Sparsity assignment**: $|S|^G$ assignments for G groups, where G ranges from 1 to L

The total brute-force complexity is:

$$\mathcal{C}_{\text{brute}} = \mathcal{O}\left(\sum_{G=1}^{L} \binom{L}{G} \times |S|^G\right) = \mathcal{O}(|S|^L \times 2^L) \tag{19}$$

For Llama-2-7B with $L = 32$ layers and $|S| = 29$ sparsity levels (1% to 15% with 0.5% steps):

- Brute force: $29^{32} \times 2^{32} \approx 10^{47}$ operations

ASAF Framework Complexity. Our two-phase approach decomposes the problem into manageable subproblems:

Phase 1: Coarse-Grained Optimization Complexity: $\mathcal{O}(G_{\max} \times B \times |\mathcal{I}|)$

- $G_{\max} = L$: maximum number of groups
- $B = \delta_{\max}/\Delta$: discretized accuracy budget levels
- $|\mathcal{I}|$: number of candidate sparsity intervals

Phase 2: Fine-Grained Optimization Complexity: $\mathcal{O}(L^2 \times G^* \times B \times |S'|)$

- L^2: nested loops over layer positions and group boundaries
- G^*: optimal number of groups from Phase 1
- $|S'| \ll |S|$: refined sparsity space size

Tabulation Preprocessing Complexity: $\mathcal{O}(L^2 \times |S| \times T_{\text{eval}})$

- T_{eval}: evaluation time for sensitivity measurement per configuration

Total ASAF Complexity.

$$\mathcal{C}_{\text{ASAF}} = \mathcal{O}(L^2 \times |S| \times T_{\text{eval}} + G_{\max} \times B \times |\mathcal{I}| + L^2 \times G^* \times B \times |S'|) \tag{20}$$

Dominant term: $\mathcal{O}(L^2 \times |S| \times T_{\text{eval}})$ (tabulation preprocessing)

Practical Complexity Comparison. For Llama-2-7B configuration:

- $L = 32$, $|S| = 29$, $G_{\max} = 32$, $B = 40$, $|\mathcal{I}| = 15$, $G^* \approx 8$, $|S'| \approx 3$
- **ASAF total**: $32^2 \times 29 \times T_{\text{eval}} + 32 \times 40 \times 15 + 32^2 \times 8 \times 40 \times 3 \approx 3 \times 10^4 \times T_{\text{eval}}$
- **Speedup ratio**: $\mathcal{C}_{\text{brute}}/\mathcal{C}_{\text{ASAF}} \approx 10^{42}/T_{\text{eval}}$

Memory Complexity. Dynamic Programming Tables

- **Coarse-grained**: $\mathcal{O}(G_{\max} \times B) = \mathcal{O}(L \times \delta_{\max}/\Delta)$
- **Fine-grained**: $\mathcal{O}(L \times G^* \times B) = \mathcal{O}(L \times G^* \times \delta_{\max}/\Delta)$
- **Tabulation**: $\mathcal{O}(L^2 \times |S|)$ for both FLOP and sensitivity tables

Total Memory

$$\text{Memory}_{\text{ASAF}} = \mathcal{O}(L^2 \times |S| + L \times G^* \times \delta_{\max}/\Delta) \tag{21}$$

Practical example (Llama-2-7B): ∼211KB total overhead, negligible compared to model parameters.

Scalability Analysis. The polynomial complexity ensures graceful scaling:

- **12-layer model**: 10^{15} brute-force vs 10^3 ASAF operations
- **48-layer model**: 10^{70} brute-force vs 10^4 ASAF operations
- **Complexity reduction ratio grows exponentially** with model size

A.10.2 Memory Efficiency Analysis

Dynamic Programming State Management. The memory requirements for our dynamic programming approach consist of three primary components. The coarse-grained DP table maintains states with dimensions $[G_{\max}] \times [T_{\max}]$, requiring $\mathcal{O}(G_{\max} \times T_{\max})$ floating-point entries. For typical configurations, this translates to approximately $32 \times 40 = 1,280$ entries, occupying roughly 5.12 KB when using FP32 precision. The fine-grained DP table stores states with dimensions $[L+1] \times [G^*] \times [T_{\max}]$, necessitating $\mathcal{O}(L \times G^* \times T_{\max})$ floating-point entries. This typically amounts to $33 \times 8 \times 40 = 10,560$ entries, consuming approximately 42.24 KB in FP32 format. The tabulation mechanism maintains two lookup tables: the FLOP cost table $H[i][\text{len}][s]$ and the accuracy degradation table $\Xi[i][\text{len}][s]$, both with dimensions $[L] \times [L] \times [|S|]$. The combined memory requirement becomes $2 \times L \times L \times |S|$ entries, typically resulting in $2 \times 32 \times 32 \times 20 = 40,960$ entries and consuming approximately 163.84 KB. The total memory overhead for all data structures amounts to 211.2 KB, which remains negligible compared to the model parameter memory requirements.

Model Memory Reduction. The adaptive sparsity allocation strategy achieves memory efficiency through strategic pruning of redundant parameters. For a layer group i with sparsity rate s_i, the memory reduction is directly proportional: $\text{Memory}_{\text{saved}} = s_i \times \text{Memory}_{\text{original}}$. However, ASAF's adaptive allocation provides superior memory efficiency compared to uniform sparsity approaches by concentrating aggressive pruning on redundant layers while preserving critical layers with minimal sparsity. Our experimental results demonstrate progressive memory reduction scaling with model size. The Llama-2-7B model achieves 7.43% memory reduction (from 3,255 MB to 3,013 MB), while the Llama-2-13B model attains 8.29% reduction (from 5,753 MB to 5,276 MB). Larger models exhibit more substantial benefits, with the Llama-2-30B achieving 11.38% reduction (from 11,408 MB to 10,110 MB) and the Llama-2-70B reaching 12.63% reduction (from 20,536 MB to 17,943 MB). This scaling behavior reflects the increased redundancy present in larger architectures, enabling more aggressive sparsification without accuracy degradation.

A.10.3 Runtime Performance Characteristics

Preprocessing and Optimization Time. The tabulation construction phase constitutes the primary preprocessing overhead, with time complexity $\mathcal{O}(L^2 \times |S| \times T_{\text{eval}})$, where T_{eval} represents the evaluation time for a single configuration. Empirical measurements indicate preprocessing times of 2-4 hours for Llama-2-7B on NVIDIA RTX 3090 hardware, which remains acceptable for deployment scenarios requiring one-time optimization. The optimization phases exhibit minimal computational overhead, with the coarse-grained phase typically completing within 30 seconds and the fine-grained phase requiring approximately 10 seconds. The total optimization time remains under one minute, which is negligible compared to the preprocessing requirements and enables rapid exploration of alternative configurations.

Inference Acceleration Analysis. The FLOP reduction achieved by optimal allocation $\{(\mathcal{L}_i, s_i)\}_{i=1}^{G^*}$ can be quantified as:

$$\text{FLOP}_{\text{reduced}} = \sum_{i=1}^{G^*} \sum_{l \in \mathcal{L}_i} \phi_l \times s_i, \tag{22}$$

where ϕ_l represents the original FLOP count for layer l. Experimental evaluation across different batch sizes and model configurations reveals consistent acceleration patterns. The Llama-2-7B model achieves speedups ranging from 1.89× at batch size 1 to 2.31× at batch size 32. Larger models demonstrate more substantial improvements, with the Llama-2-70B reaching peak acceleration of 3.63× at batch size 32. This scaling behavior indicates that both increasing model complexity and batch size amplify the effectiveness of our adaptive allocation strategy, as larger computational workloads increasingly overshadow memory bandwidth limitations.

Hardware Utilization Efficiency. The layer-group-based sparsity patterns enable efficient hardware utilization through several mechanisms. Sparse matrix operations reduce memory traffic proportionally to the applied sparsity rates, while the structured nature of our allocation strategy facilitates efficient kernel implementations. The combination of 4-bit quantization with adaptive sparsity

maximizes the throughput-to-memory ratio, enabling effective utilization of tensor cores and other specialized compute units available on modern GPU architectures.

A.10.4 Scalability Properties

Layer Count Scaling. The polynomial complexity ensures graceful scaling with increasing model sizes. As model layer counts increase from 12 to 48 layers, the complexity reduction factor grows from 10^{12} to 10^{67}, demonstrating the exponential advantage of our approach over brute-force enumeration. Specifically, a 12-layer model requires approximately $12^2 \times 20 = 2,880$ operations compared to $2^{12} \times 20^{12} \approx 10^{15}$ brute-force operations, while a 48-layer model needs $48^2 \times 20 = 46,080$ operations versus $2^{48} \times 20^{48} \approx 10^{70}$ brute-force operations.

Sparsity Resolution Impact. The impact of sparsity discretization resolution on computational requirements follows a linear relationship. Increasing the sparsity resolution from 20 to 40 levels doubles the tabulation construction time and slightly increases the optimization overhead. However, finer resolution typically yields marginal improvements in final allocation quality, suggesting that moderate discretization (0.5% to 1.0% steps) provides an optimal balance between computational cost and optimization precision.

Accuracy Budget Sensitivity. The relationship between accuracy degradation budget and optimization complexity exhibits sublinear scaling. Doubling the accuracy budget from 1% to 2% increases the fine-grained optimization space by approximately 4×, but the coarse-grained phase efficiently prunes the majority of suboptimal configurations, maintaining overall polynomial complexity. This property ensures that our framework remains computationally tractable even when exploring larger accuracy-efficiency trade-off spaces.

Appendix A.11: NVIDIA 4090 GPU Performance Analysis

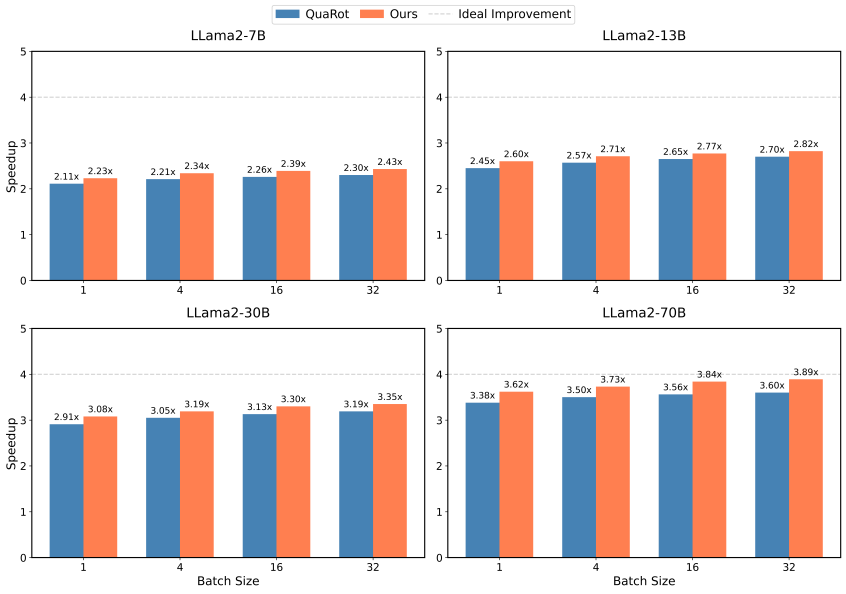

Figure 9: Performance comparison between our ASAF framework and QuaRot on Llama-2 models, evaluated on NVIDIA RTX 4090 GPUs with 2048-token sequences across various batch sizes.

Performance Evaluation on NVIDIA RTX 4090. We further evaluate our ASAF framework on NVIDIA RTX 4090 GPUs to assess scalability across hardware generations. Figure 9 demonstrates acceleration performance across various batch configurations with 2048-token sequences on Llama-2 models. On RTX 4090, our framework achieves peak acceleration of 3.89× on the Llama-2-70B model, representing a 7.2% improvement over the 3.63× achieved on RTX 3090. The relative

improvement of ASAF over QuaRot baseline remains consistent across both GPU generations, with our method providing 6-8% additional acceleration across all model sizes, validating the hardware-agnostic nature of our optimization approach.

APPENDIX A.12: LLAMA-3 EXPERIMENTAL RESULTS

Table 5: Zero-shot accuracy of Llama-3 models with our framework on PIQA (PQ), WinoGrande (WG), HellaSwag (HS), Arc-Easy (A-e), Arc-Challenge (A-c), and LAMBADA (LA).

Model	Method	PQ	WG	HS	A-e	A-c	LA	Avg.
Llama3-8B	FP16	81.28	72.14	78.32	78.71	52.13	76.84	73.24
	QuaRot	79.16	69.82	75.47	75.33	48.29	73.95	70.34
	ASAF (Ours)	78.52	69.26	74.81	74.73	47.94	73.40	69.78
Llama3-70B	FP16	85.42	81.77	86.15	85.23	64.42	82.91	80.98
	QuaRot	84.28	80.34	84.61	83.76	62.88	81.24	79.52
	ASAF (Ours)	83.65	79.77	83.94	83.05	62.47	80.63	78.92

Zero-Shot Tasks on Llama-3 Family. To further validate our framework's generalizability, we extend our evaluation to the Llama-3 model family, which demonstrates superior baseline performance compared to Llama-2. We evaluate our framework on the same six zero-shot benchmarks: PIQA (Bisk et al., 2020), WinoGrande (Sakaguchi et al., 2021), HellaSwag (Zellers et al., 2019), LAMBADA (Radford et al., 2019), and the ARC-Easy and ARC-Challenge datasets (Clark et al., 2018). Table 5 demonstrates that our method maintains competitive accuracy across Llama-3 model sizes with performance degradation consistently below 1% compared to QuaRot, even on these more advanced models with higher baseline performance.

APPENDIX A.13: GROUP-WISE QUANTIZATION

Table 6: WikiText-2 perplexity of 4-bit QuaRot and our ASAF framework under different group sizes on Llama-2 models. Weights are GPTQ-quantized, and KV caches use fixed group size 128 (equal to head dimension). "G" denotes group-wise quantization with specified group size. Our method applies adaptive sparsity allocation across layer groups.

Method	Llama-2			
	7B	13B	30B	70B
Baseline	5.47	4.88	4.09	3.32
QuaRot	6.10	5.40	4.41	3.79
QuaRot-256G	5.98	5.28	4.32	3.63
QuaRot-128G	5.93	5.26	4.25	3.61
QuaRot-64G	5.88	5.25	4.13	3.58
ASAF (Ours)	6.16	5.45	4.45	3.82
ASAF-256G (Ours)	6.03	5.32	4.36	3.66
ASAF-128G (Ours)	5.97	5.30	4.28	3.64
ASAF-64G (Ours)	5.94	5.28	4.17	3.62

Group-Wise Quantization. Table 6 reports WikiText-2 perplexity for our ASAF framework when weights and activations are quantized group-wise with group sizes of 256, 128, and 64. As expected, smaller groups yield better accuracy because per-group scale factors more precisely capture local statistics, though they incur additional scale storage and slightly more complex kernels. Across every group size, our adaptive sparsity allocation framework tracks QuaRot's dense counterparts to within 1%, demonstrating that layer-group-based sparsity optimization can be achieved without meaningful quality loss. The consistent performance across different group sizes validates the robustness of our two-phase optimization approach under various quantization granularities.

REFERENCES

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *The Twelfth International Conference on Learning Representations*, 2024.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, 2024.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2022.

Saleh Ashkboos, Ilia Markov, Elias Frantar, Tingxuan Zhong, Xincheng Wang, Jie Ren, Torsten Hoefler, and Dan Alistarh. Quik: Towards end-to-end 4-bit inference on generative large language models. *arXiv preprint arXiv:2310.09259*, 2023.

Saleh Ashkboos, Maximilian L. Croci, Marcelo Gennari do Nascimento, Torsten Hoefler, and James Hensman. Slicegpt: Compress large language models by deleting rows and columns. *arXiv preprint arXiv:2401.15024*, 2024a.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. *Advances in Neural Information Processing Systems*, 37:100213–100240, 2024b.

Richard Ernest Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

Ruisi Cai, Yeonju Ro, Geon-Woo Kim, Peihao Wang, Babak Ehteshami Bejnordi, Aditya Akella, Zhangyang Wang, et al. Read-me: Refactorizing llms as router-decoupled mixture of experts with system co-design. *Advances in Neural Information Processing Systems*, 37:116126–116148, 2024.

Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher M De Sa. Quip: 2-bit quantization of large language models with guarantees. *Advances in Neural Information Processing Systems*, 36:4396–4429, 2023.

Ciprian Chelba, Mia Chen, Ankur Bapna, and Noam Shazeer. Faster transformer decoding: N-gram masked self-attention. *arXiv preprint arXiv:2001.04589*, 2020.

Mengzhao Chen, Yi Liu, Jiahao Wang, Yi Bin, Wenqi Shao, and Ping Luo. Prefixquant: Static quantization beats dynamic through prefixed outliers in llms. *arXiv preprint arXiv:2410.05265*, 2024a.

Ziyi Chen, Xiaocong Yang, Jiacheng Lin, Chenkai Sun, Kevin Chang, and Jie Huang. Cascade speculative drafting for even faster llm inference. *Advances in Neural Information Processing Systems*, 37:86226–86242, 2024b.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *ArXiv*, abs/1803.05457, 2018. URL `https://api.semanticscholar.org/CorpusID:3922816`.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009. ISBN 9780262033848.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Rocktim Jyoti Das, Mingjie Sun, Liqun Ma, and Zhiqiang Shen. Beyond size: How gradients shape pruning decisions in large language models. *arXiv preprint arXiv:2311.04902*, 2023.

Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Higher Education, Boston, MA, 2006.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems*, 35: 30318–30332, 2022.

Oshin Dutta, Ritvik Gupta, and Sumeet Agarwal. Efficient llm pruning with global token-dependency awareness and hardware-adapted inference. In *Workshop on Efficient Systems for Foundation Models II@ ICML2024*, 2024.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pp. 10323–10337. PMLR, 2023.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Zichuan Fu, Wentao Song, Yejing Wang, Xian Wu, Yefeng Zheng, Yingying Zhang, Derong Xu, Xuetao Wei, Tong Xu, and Xiangyu Zhao. Sliding window attention training for efficient large language models. *arXiv preprint arXiv:2502.18845*, 2025.

Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, et al. A framework for few-shot language model evaluation. *Version v0. 0.1. Sept*, 2021.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL `https://zenodo.org/records/12608602`.

Ruihao Gong, Yifu Ding, Zining Wang, Chengtao Lv, Xingyu Zheng, Jinyang Du, Haotong Qin, Jinyang Guo, Michele Magno, and Xianglong Liu. A survey of low-bit large language models: Basics, systems, and algorithms. *arXiv preprint arXiv:2409.16694*, 2024.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models. *arXiv preprint arXiv:2306.08543*, 2023.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Yuhan Dong, Yu Wang, et al. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. *Proceedings of Machine Learning and Systems*, 6:148–161, 2024.

Wenxuan Huang, Zijie Zhai, Yunhang Shen, Shaosheng Cao, Fei Zhao, Xiangfeng Xu, Zheyu Ye, and Shaohui Lin. Dynamic-llava: Efficient multimodal large language models via dynamic vision-language context sparsification. *arXiv preprint arXiv:2412.00876*, 2024.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.

Shibo Jie, Yehui Tang, Kai Han, Zhi-Hong Deng, and Jing Han. Specache: Speculative key-value caching for efficient generation of llms. *arXiv preprint arXiv:2503.16163*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Jaeseong Lee, Aurick Qiao, Daniel F Campos, Zhewei Yao, Yuxiong He, et al. Stun: Structured-then-unstructured pruning for scalable moe pruning. *arXiv preprint arXiv:2409.06211*, 2024.

Guanchen Li, Yixing Xu, Zeping Li, Ji Liu, Xuanwu Yin, Dong Li, and Emad Barsoum. T\'yr-the-pruner: Unlocking accurate 50% structural pruning for llms via global sparsity distribution optimization. *arXiv preprint arXiv:2503.09657*, 2025.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6: 87–100, 2024a.

Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532*, 2024b.

Xiang Meng, Kayhan Behdin, Haoyue Wang, and Rahul Mazumder. Alps: Improved optimization for highly sparse one-shot pruning for large language models. *arXiv preprint arXiv:2406.07831*, 2024.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Saurav Muralidharan, Sharath Turuvekere Sreenivas, Raviraj Joshi, Marcin Chochowski, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, Jan Kautz, and Pavlo Molchanov. Compact language models via pruning and knowledge distillation. *Advances in Neural Information Processing Systems*, 37:41076–41102, 2024.

Zhenyu Ning, Jieru Zhao, Qihao Jin, Wenchao Ding, and Minyi Guo. Inf-mllm: Efficient streaming inference of multimodal large language models on a single gpu. *arXiv preprint arXiv:2409.09086*, 2024.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. *arXiv preprint arXiv:2308.13137*, 2023.

Sharath Turuvekere Sreenivas, Saurav Muralidharan, Raviraj Joshi, Marcin Chochowski, Ameya Sunil Mahabaleshwarkar, Gerald Shen, Jiaqi Zeng, Zijia Chen, Yoshi Suhara, Shizhe Diao, et al. Llm pruning and distillation in practice: The minitron approach. *arXiv preprint arXiv:2408.11796*, 2024.

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 173–191, 2024.

Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Jinguang Wang, Jingyu Wang, Haifeng Sun, Tingting Yang, Zirui Zhuang, Wanyi Ning, Yuexi Yin, Qi Qi, and Jianxin Liao. Mergequant: Accurate 4-bit static quantization of large language models by channel-wise calibration. *arXiv preprint arXiv:2503.07654*, 2025.

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

Lu Yin, You Wu, Zhenyu Zhang, Cheng-Yu Hsieh, Yaqing Wang, Yiling Jia, Gen Li, Ajay Jaiswal, Mykola Pechenizkiy, Yi Liang, et al. Outlier weighed layerwise sparsity (owl): A missing secret sauce for pruning llms to high sparsity. *arXiv preprint arXiv:2310.05175*, 2023.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

Yuxin Zhang, Lirui Zhao, Mingbao Lin, Yunyun Sun, Yiwu Yao, Xingjia Han, Jared Tanner, Shiwei Liu, and Rongrong Ji. Dynamic sparse no training: Training-free fine-tuning for sparse llms. *arXiv preprint arXiv:2310.08915*, 2023.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. *arXiv preprint arXiv:2310.19102*, 2023.