HAPE: Hardware-Aware LLM Pruning For Efficient On-Device Inference Optimization

WENQIAN ZHAO, The Chinese University of Hong Kong, Hong Kong, Hong Kong
LANCHENG ZOU, The Chinese University of Hong Kong, Hong Kong, Hong Kong
ZIXIAO WANG, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
XUFENG YAO, Computer Science and Engineering, CUHK, Hong Kong, Hong Kong
BEI YU, The Chinese University of Hong Kong Department of Computer Science and Engineering, Hong Kong, Hong Kong

Over the past few years, large language models (LLMs) have demonstrated remarkable performance and versatility across a variety of complex tasks. However, their deployment has been challenged by their substantial model size and computational requirements. Pruning is a effective approach to make the model parameters sparse, thereby acquire inference acceleration. While not everyone requires training or fine-tuning large models, the diverse range of applications necessitates the deployment of LLMs on different devices. Model pruning and compression have emerged as areas of deep research interest to address these challenges. In consideration of versatility and practicality, we have designed a hardware-aware pruning process for general-purpose hardware/edge devices to enable efficient deployment and inference of LLMs. Instead of considering sparse ratio alone, we are motivated to design a pruning framework that incorporates genuine inference speed-up sensitivity from each pruning structure. Moreover, our framework breaks the layer-by-layer pruning setting and fuse several layers into one pruning stage to allow cross-layer optimization. Apart from that, we hold pragmatism by conducting compilation optimization during pruning. This step is critical because most sparsity patterns barely show distinct speed acceleration with corresponding dataflow and memory optimization. Our process operates within a post-training framework, obviating the need for additional training and thereby reducing resource requirements, while ensuring diverse inference speed and accuracy requirements on hardware.

CCS Concepts: • **Computing methodologies** → *Natural language processing*; • **Computer systems organization** → *Embedded hardware*; • **General and reference** → **Performance;**

Additional Key Words and Phrases: LLM compression, Hardware-Aware Compression, On-device inference optimization

This work is partially supported by The Research Grants Council of Hong Kong SAR (No. RFS2425-4S02 and No. CUHK14210723).
Authors' Contact Information: Wenqian Zhao, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: wqzhao@cse.cuhk.edu.hk; Lancheng Zou, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: lanchengzou@gmail.com; Zixiao Wang, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, None, Hong Kong; e-mail: zxwang22@cse.cuhk.edu.hk; Xufeng Yao, Computer Science and Engineering, CUHK, Hong Kong, Hong Kong; e-mail: yxf12fdu@gmail.com; Bei Yu (corresponding author), The Chinese University of Hong Kong Department of Computer Science and Engineering, Hong Kong, Hong Kong; e-mail: byu@cse.cuhk.edu.hk.

ACM 1084-4309/2025/07-ART61
https://doi.org/10.1145/3744244

ACM Reference Format:
Wenqian Zhao, Lancheng Zou, Zixiao Wang, Xufeng Yao, and Bei Yu. 2025. HAPE: Hardware-Aware LLM Pruning For Efficient On-Device Inference Optimization. *ACM Trans. Des. Autom. Electron. Syst.* 30, 4, Article 61 (July 2025), 18 pages. https://doi.org/10.1145/3744244

1 Introduction

Over the past few years, **large language models (LLMs)** [1, 2] have demonstrated remarkable performance and versatility across a variety of complex tasks. However, their deployment has been challenged by their enormous model size and computational requirements. This has led to an increasing focus on model pruning and compression research, to bridge the gap between the potential of LLMs and the practical challenges associated with their deployment.

Model pruning, a technique that simplifies the model parameters to achieve inference acceleration, has shown promise in model compression. Structured pruning, which eliminates entire neurons or filters, offers better hardware efficiency but often at the cost of model performance. Balancing these tradeoffs is a critical area of research. Apart from that, conventional pruning algorithms usually require fine-tuning the model, which shows better performance but is barely practical for LLM. The training/tuning stage of LLM is extremely expensive in terms of dataset, computation power, hardware and time cost.

To conquer this challenge, we propose a post-training LLM pruning framework: HAPE, which is hardware-aware, specifically designed for general-purpose hardware and user-end devices. The overall design flow of HAPE is in Figure 1. In order to avoid the extremely expensive tuning cost, we redesign the pruning strategy that does not require fine-tuning on the original LLM model. Our implementation requires one single CPU device (or any platform) and only a few hours' time budget for a real deployment scenario.

We hold the key insight that the model pruning process should not be decoupled from the backend-level implementation. We integrate structure pruning with on-device optimization at each pruning stage, as visualized in Figure 1. To compensate for the performance drop from regular sparsity pattern limitation, We assign dynamic sparsity ratios within the transformer block and gather all pruning candidates within each block as global optimization space. This approach enhances the efficiency of the pruning process and mitigates some of the limitations associated with structured pruning. We dig deeper at this step by fusing inter-dependent neurons for the chained effect: During the forward stage, the output of consecutive zeros remains zeros. We recursively fuse the neurons to simplify the connection and impart dependency information into the importance estimation.

Another potential problem is hardware information missing in the pruning metrics. Traditional post-training approaches focus on achieving high sparse ratios regarding accuracy loss. However, a higher sparsity ratio does not guarantee faster inference. Some sparsity patterns do fit computation flow on certain hardware. Our framework considers the genuine inference speed-up sensitivity from each pruning structure and accuracy loss. This unique perspective enables us to design a pruning framework more attuned to the performance requirements of different hardware platforms in real scenarios.

In addition, our approach pragmatically incorporates on-device sparse compilation optimization during the pruning process to fit the target hardware device. In each transformer block, most calculations, such as multi-head attention, MLP layers, and projection layers, are batched matrix multiplication. Given pruned sparse matrices (usually large), we analyze the necessity to optimize the sparse computation flow with computation graph extraction, operator fusion, and dataflow optimization (loop reordering, tiling et al.) to generate high-performance sparse kernels for real

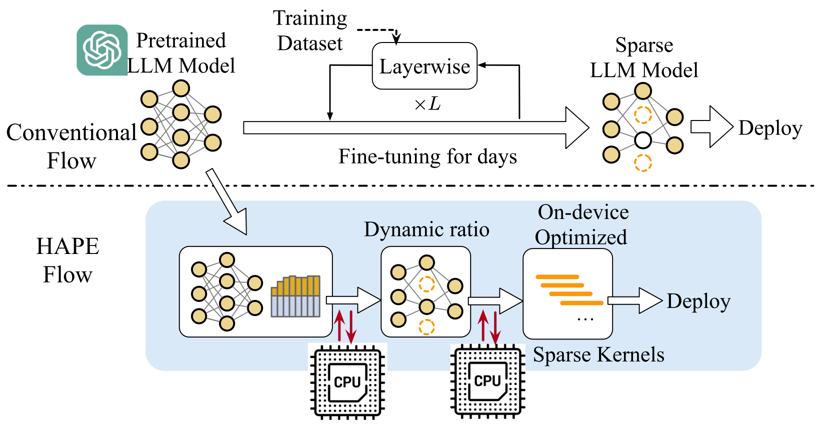


Fig. 1. HAPE post-training pruning flow. In comparison with conventional layer-wise pruning flow, HAPE do not require LLM tuning and is hardware-aware at each step.

speed-up on the device. This step is crucial, as most sparsity patterns do not yield significant speed acceleration without corresponding memory and dataflow optimization. By integrating this optimization into the pruning process, we ensure that the resulting models are not only smaller and faster but also more compatible with the hardware platforms. The contributions of this article are listed as follows:

— A hardware-aware LLM pruning framework, realizing structure pruning on single CPU (or other hardware) with no costly fine-tuning during pruning.
— We integrate genuine latency sensitivity into pruning importance instead of bare sparsity ratio alone.
— We apply dynamic sparsity ratios with each transformer block to reach optimal performance under different pruning ratios.
— We brought on-device sparse dataflow compilation into the progressive pruning to reach extreme latency-accuracy optimality.

The rest of this article is organized as follows. Section 2 is the background for LLM model compression and resource-driven pruning and optimization. Section 3 comprises our complete workflow and description of each component. Section 4 is our corresponding experiment, followed by conclusion in Section 5.

2 Preliminaries

2.1 Challenges of LLM Application

Increased along with the impressive performance of language models nowadays, the size of current LLMs challenges the development and deployment of hardware. In the earlier period, where Bert series [3] with model size ranging from 29 million to 334 million is regarded as "large" language model. On the other hand, the latest open-sourced Meta's Llama-3 has already reached 70 billion parameters, and X.AI's Grok-1 even comprises 314 billion parameters. Let alone pre-training, as we are only concerned about inference or light fine-tuning for application. The bulkiness of the current mainstream Language model brings obstacles with overhead of loading and storing. The storage itself is a huge cost not only for main memory or **High Bandwidth Memory** (**HBM**) on GPU, but also for the hard-disk to keep copies of each model when being stored or loaded for each down-stream task. In addition, it also requires practical experience to keep individual LLM models to load and offload memory if necessary, which is rather inefficient for bandwidth limit.

As for the computation challenge, the mainstream LLMs are Transformer-based [4] decoder architecture, of which the inference process is auto-regressive generation with token-by-token fashion. As the sequence length increases, the computation cost increases quadratically. Within each transformer block relies the key attention operation where each sequence is encoded as key $\mathbf{K}$, query $\mathbf{Q}$ and value $\mathbf{V}$ vectors. All these vectors $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ have size of $n \times$ `dim_hidden` where `dim_hidden` denotes embedding dimension, which is 4096 for LLama family model. The attention operation is

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{n}}\right)\mathbf{V}, \tag{1}$$

where the computation of each matrix multiplication scales with complexity $\mathcal{O}(n \times n)$. As we know, the current Llama already supports 8K context length here, leading the operation cost grows gigantically with it. While KV Cache has been proposed to store the existing $\mathbf{K}, \mathbf{V}$ vectors, avoiding repetitive computation of these large matrix multiplications for each new generated token, the scale itself is still computation intensive. Although some approaches such as `vLLM` [5] with PagedAttention or FLashAttention [6] are employed to partition each sequence's KV cache into fixed-size blocks and recompute the blocks without loading very large matrices. Such balancing and compute and memory does not thoroughly solve the problem.

2.2 Compression of LLM

Language models [1, 2] have gained a broad reputation for incredible performance and massive deployment cost, thereby necessitating a reduction in parameter size and latency. Previous studies aimed at compressing language models usually include: network pruning [7, 8], knowledge distillation [9, 10], quantization [11, 12], and other methodologies, such as recent appearing dynamic token reduction [13] or early exit [14].

Those approaches to compress the computation of LLM can be categorized into two kinds: (1) Compression of each element-wise operation, for example, quantization. (2) Compression of model parameter scale, such as distillation or pruning, resulting with less parameters and multiplication required. As a matter of fact, quantization is a popular area as well for deep learning deployment, which reduces the FP32 operation into FP16 or lower INT8 or INT4, which are well supported by conventional hardwares for their calculation simplicity. One of the primary challenges in LLM quantization is the systematic numeric outliers with large magnitude in a few specific channels of activations when scaling up LLM size [15].

On the contrary, pruning serves as a complementary technique to quantization, focusing on the post-training phase by removing specific portions of a model's weights without compromising its performance. It is crucial to differentiate between structured and unstructured pruning approaches. Structured pruning involves replacing dense segments of a model with smaller yet still dense components. In contrast, unstructured pruning results in weights of value zero, which have no impact on the network's behavior and can theoretically be eliminated. Structural pruning, characterized by the removal of an entire filter from the neural network, renders structured sparsity pattern of LLM model. Several techniques exist for structure removal, including l1-dependent pruning [16], first-order importance estimation [17], Hessian-based estimation [7].

2.3 Hardware/Resource-Constrained Pruning

As the sizes of models continue to expand, there is an explosive demand for efficient compression of LLMs, which is independent of the original training data. Concerning efficient compression, studies such as [18] optimize pruning with a linear least squares problem formulation. Research such as [8, 19] have put forth the layer-wise pruning strategies. Given the constraint of the training data's accessibility and training cost, data-free pruning techniques have been developed, introducing several strategies to prune the model with neurons' similarity evaluation. Moreover, methods have been proposed distillation-based methods without relying on the model's training data. However, these methods can be excessively time-consuming as they involve synthesizing samples by back-propagating the pre-trained language models.

As a matter of fact, both pruning granularity and backend itself contribute to the final speed-up or accuracy optimization. In general, unstructured pruning [20, 21] with element-wise zero-out values can achieve extreme compression ratio, but its irregular data format requires specific encoding and decoding and may not easily achieve significant acceleration on general devices or even common domain specific accelerators [22]. Structured pruning [23, 24], on the other hand, possesses filter-wise sparsity which ends up still with dense weight matrices but may sacrifice precision with thorough channel/layer elimination. Even as some new approaches such as N:M [25], $1 \times N$ [26] pruning in the middle of these granularities are proposed, it is not guaranteed to solve the problem. For example, that N:M pattern can only achieve significant acceleration on specific GPUs with NVIDIA Tensor Cores [27]. Still, it is not easy to achieve both extreme speed-up and accuracy, and the adaptation to general processor devices such as CPU are more challenging.

3 Framework

The section is the illustration of the main framework of HAPE. Firstly, we introduce our algorithm-level cross-layer pruning strategy with neuron fusion technique in Section 3.1. Secondly, we re-design the importance assignment in LLM structure pruning with on-device latency sensitivity for practicality in Section 3.2. Thirdly, we discuss the necessity of joint compilation flow in pruning for sparse LLM model in Section 3.3. Last but not least, we wrap the complete HAPE pruning flow in Section 3.4.

3.1 Cross-Layer Pruning with Grouping

In our framework, we carefully select the sparse granularity. Firstly, we abandon the conventional layer-by-layer pruning paradigm and compose a series of consecutive layers together as a pruning block. Our reconstruction pruning takes a transformer block as a pruning group, given that the mainstream model architecture of current LLM is a stack of transformer blocks. Let us take Llama-2-7B [2] as an example, which is composed of 32 transformer blocks, as shown in Table 1. Among all the layers listed, we will apply pruning operations on all the linear layers and multi-head attention layers. The reason is twofold: (1) They are the dominating layers, costing most model parameters and computation; (2) They are in general matrix multiplication form, which is friendly for sparse optimization. As shown in Figure 2, we will take all prunable layers within a transformer block and extract all pruning units as a composition of candidates. For linear layers, each unit is a neuron whose parameters represent a column in the linear transformation matrix. For attention layers, each unit is an attention head.

We group all the pruning candidates together and rank them in descending order according to their estimated importance regarding on-device speed and accuracy sensitivity (described in Section 3.2). Given a series of all layers in the model: $L = \{l_1, l_2, \ldots, l_n\}$ with an expected pruning ratio $\alpha \in [0, 1)$, a classical way is to prune each layer l_i at a time. Conventional approaches rank these candidates within each layer and conduct the ranking in descending order and, cutting off the tails in one layer l_i at a time, as visualized in the upper part of Figure 2. The resulting sparse model is $L_{\text{prune}} = \{l'_1, l'_2, \ldots, l'_n\}$ where each pruned layer $||l'_i||_0 \leq \alpha ||l_i||_0$ has fixed sparse ratio threshold.

Table 1. Details of All Layers within a Transformer Block

Name	Layer Type	Dimension	Prune
Q,K,V projection	Linear	4096×4096	✓
O projection	Linear	4096×4096	✓
Multi-head attention	Matrix Mul.	32 heads	✓
SoftMax	Softmax	4096/32	×
MLP gate/up	Linear	11008×4096	✓
MLP down	Linear	4096×11008	✓
SwiGLU	Activation	4096	×
Input/Post LayerNorm	Normalize	4096	×

The dimension is from Llama-2-7B [2] model.

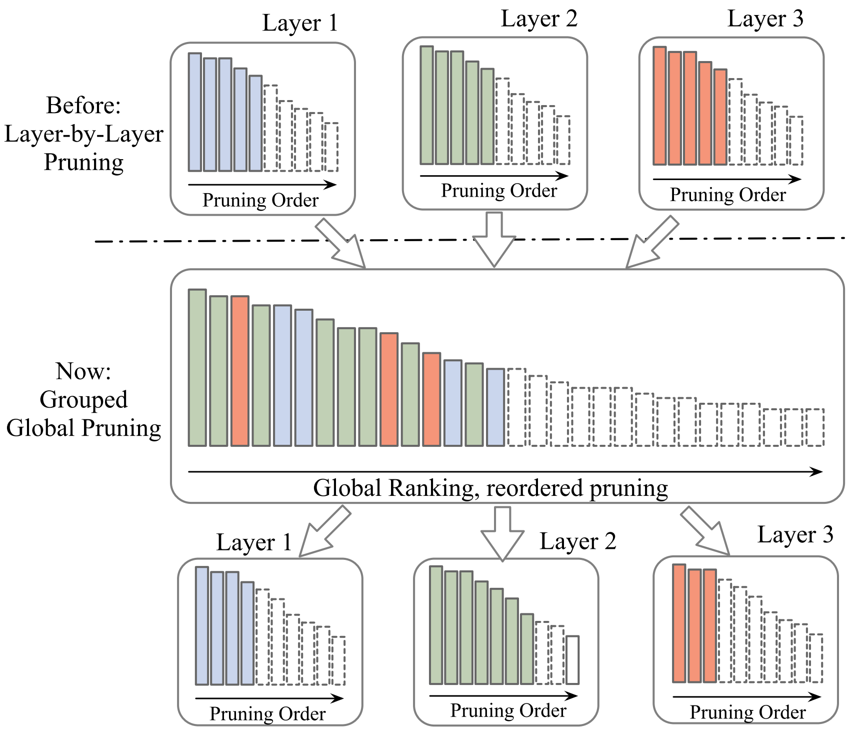


Fig. 2. Cross-layer pruning within transformer block.

Although the layer-by-layer style strikes a balance for each layer to be compressed to a fixed sparse ratio, our grouping strategy leaves more flexibility by allowing each layer to be pruned less or more. At the same time, we avoid tackling all layers in the model as a whole group. The post-training structure pruning requires the ranking of neurons to be pruned. Such aggressive step may over-prune some layers and jeopardize performance by cutting critical connections. Our approach takes the original model L into a series of blocks $\{B_1, B_2, \ldots, B_m\}$. In our practice, the block number is 32, which is the number of transformer blocks in the Llama-7B [2] decoder model. Instead of fixing a sparsity ratio, we assign dynamic sparsity ratio $\alpha_1, \alpha_2, \ldots, \alpha_k$ for each layer in the block. As long as the overall sparsity ratio of a block is below the required mode sparsity threshold α, such flexibility allows each layer to be pruned to a different ratio. The formulation of this problem is shown in Equation (2).

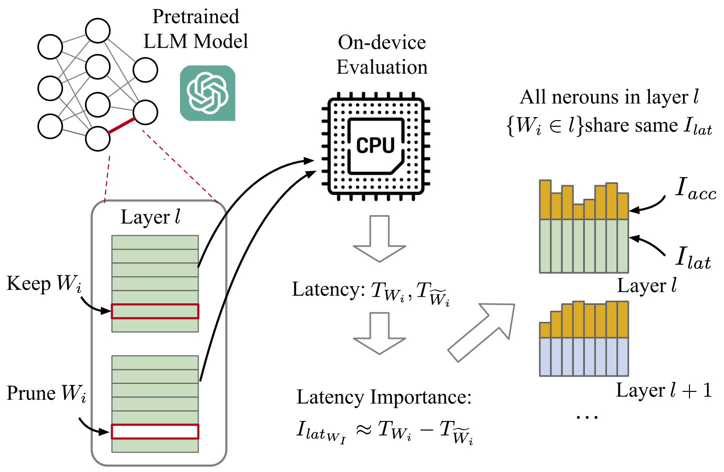


Fig. 3. Visualization of latency importance.

$$
\begin{aligned}
&L_{\text{prune}} = \{B_1, B_2, \ldots, B_m\}, \quad \text{where} \quad B_i = \{l_1, l_2, \ldots, l_k\},\\
&\text{s.t.} \quad \text{For each } B_i \in L_{\text{prune}}, \quad \frac{\sum_{l_j \in B_i} \alpha_j \|l_j\|_0}{\sum_{l_j \in B_i} \|l_j\|_0} \le \alpha, \qquad (2)\\
&\{\alpha_1, \alpha_2, \ldots, \alpha_k\} \longrightarrow \text{sparsity ratio of } l_1, l_2, \ldots, l_k \in B_i.
\end{aligned}
$$

Within each block, the pruning unit/candidates are assigned an importance score (Section 3.2), which is an indicator of their influence on the latency/accuracy performance of the model.

Neuron Fusion. During the group ranking stage, we also need to consider connection dependency. Given two consecutive units W_i and W_j within the same block B, it is meaningless to keep W_j while pruning W_i if W_j is dependent on W_i. We fuse them into one compound unit W_i^{new} and prune neither or both of them at the same time. Here, the dependency simply denotes that: (1) all input connections of W_j are from the output of W_i or (2) all output connections of W_j are input of W_i.

The neuron fusion technique is shown as

$$
\begin{aligned}
&W_i^{\text{new}} \leftarrow \text{Fuse}(W_i, W_j) \quad \text{for} \quad W_i, W_j \in B,\\
&\text{if} \quad In(W_j) \subseteq Out(W_i) \text{ or } Out(W_j) \subseteq In(W_i).
\end{aligned} \qquad (3)
$$

The insight of this dependency is that once a previous neuron is pruned, the corresponding output is zeros. If all input connections of the next neuron is a subset of the pruned, we might as well prune the next neuron and vice versa. The output of multiplication of zero is zero, and such fusion can enhance the inference efficiency by reducing unnecessary computation on zeros from vacant layers. This step is rather efficient when we apply computation graph and dataflow optimization for sparse pattern in Section 3.3. The dependency fusion calibrates on the input/output dimension, which corresponds to the following joint sparse compilation step: fusion of the two consecutive sparse operations into one kernel at the computation graph slimming. Only in this way can we witness a significant speed-up from sparse inference.

Pruning Scope. We restrict neuron fusion to within individual transformer blocks for two primary reasons. First, this limitation helps manage the computational complexity of the algorithm. Expanding the search space to include all dimensions across all layers in the network would create a significantly larger pool for comparison and selection, making the optimization process more challenging. Since our method relies on ranking rather than an exhaustive exploration of the design space, and given the varying information density across transformer blocks, identifying a globally optimal solution becomes unreliable. Additionally, evaluating all potential combinations would demand an impractical amount of computational time.

Second, conducting pruning within each transformer block promotes a balanced pruning ratio across the network. This ensures that the hardware resource demands during inference are distributed more evenly among the blocks, minimizing the likelihood of excessive pressure on specific parts of the inference platform.

3.2 Hardware Aware Importance Estimation

Given limited data and no training cost, prioritizing the neurons to be pruned is required before pruning. Unlike the mainstream structure pruning approaches that only consider accuracy sensitivity and sparsity ratio, we need to consider the real improvement brought by pruning: How much faster is the model if we prune out this channel? To tackle this problem, our work introduces hardware latency sensitivity with an on-board pseudo-gradient to re-evaluate the importance of each pruning candidate. The importance assignment in our framework includes both accuracy loss importance and latency importance.

For the latency importance I_{lat}, we consider how much genuine speed-up is accomplished when pruning out the ith structure W_i in a group $G = \{W_i\}_{i=1}^{M}$, where M is the number of candidates or structure. We chose the metric expected pseudo-gradient $\mathbb{E}(\nabla_{W_i} T)$, where T is the average per-token latency of the original model on the device. Such gradient denotes the affection on inference latency from W_i:

$$\begin{aligned} I_{lat_{W_i}} &= \mathbb{E}(\nabla_{W_i} T) = \mathbb{E}\left(\frac{d}{dW_i} T\right) \approx \frac{1}{N}\sum_{i=1}^{N} \frac{d}{dW_i} T(x_i), \\ &\approx \frac{1}{N}\sum_{i=1}^{N} \frac{1}{dW_i}(T_{W_i}(x_i) - T_{\widetilde{W_i}}(x_i)), \\ &\propto \frac{1}{N}\sum_{i=1}^{N}(T_{W_i}(x_i) - T_{\widetilde{W_i}}(x_i)), \end{aligned} \tag{4}$$

where N is the size of sample set $\mathcal{S} = \{x_i, y_i\}_{i=1}^{N}$ and $T_{W_i}, T_{\widetilde{W_i}}$ denotes the inference latency before/after pruning out W_i. As visualized in Figure 3, we collect the on-board inference latency difference by pruning out some channel within each group/layer and use that as estimation as latency importance I_T. One thing to notice is that, within each linear/attention layer l, all the neurons/heads structure $\{W_i | W_i \in l\}$ are homogeneous in terms of speed, thereby do not necessarily need repetitive experiments on latency collection. Only linear $\mathcal{O}(L)$ time complexity latency evaluation is needed where L denotes the number of layers. On the other hand, the accuracy importance I_{acc} is estimated with the loss deviation $|\Delta\mathcal{L}|$ on the sample set $\mathcal{S}$, which indicates how much more loss the pruning of each structure will cause. The objective is to determine which structure W_i shows the least affection on the model output result. Equation (5) shows the formulation of importance $I_{acc_{W_i}}$ on each component.

$$\begin{aligned} I_{acc_{W_i}} &= |\Delta\mathcal{L}(\mathcal{S})| = \left|\mathcal{L}_{W_i}(\mathcal{S}) - \mathcal{L}_{\widetilde{W_i}}(\mathcal{S})\right|, \\ &\approx \left|\Delta W_i^T \nabla_{W_i}\mathcal{L}(\mathcal{S}) + \frac{1}{2}\Delta W_i^T \nabla^2_{W_i}\mathcal{L}(\mathcal{S})\Delta W_i\right|, \\ &= \left|\underbrace{W_i^T \frac{\partial \mathcal{L}^\top(\mathcal{S})}{\partial W_i}}_{\neq 0} + \underbrace{\frac{1}{2} W_i^\top H_{W_i} W_i}_{\text{expensive}}\right|. \end{aligned} \tag{5}$$

The second line of Equation (5) was derived by second-order Taylor expansion with the latter terms ignored. We replace ΔW_i with W_i because $\Delta W_i = W_i$. Here, the only deviation is the removal of W_i. H_{W_i} is the Hessian matrix, which denotes the second-order derivative as accuracy dependency. The computation complexity of Hessian is $\mathcal{O}(\|W_i\|^2)$, which is much more expensive than first term. Some assume that the second term is a diagonal matrix and use diagonal value as accuracy importance [28]. In their scenarios, the first-order derivative is usually unusable because they assume the original model is well pre-trained and the first-order derivative for all structures W_i are 0. On the other hand, some propose to avoid using Hessian Matrix for its computation cost is high [29] by claiming the first-order gradient is not zero under different tuning/test dataset. In our case, our sample set $\mathcal{S}$ is deliberately sampled from another calibration dataset instead of the original training set, so this $\partial \mathcal{L}^\top / \partial W_i \not\approx 0$. In this way, we can follow [29] to utilize the first-order derivative information by directly using the gradient information from back-propagation as accuracy information. We can also avoid the effort of deriving following higher-order derivative, which is rather expensive. Therefore, our accuracy importance is more efficient.

Our hardware-aware importance I assigned to each structure W_i is designed to estimate both accuracy and latency regarding both objectives:

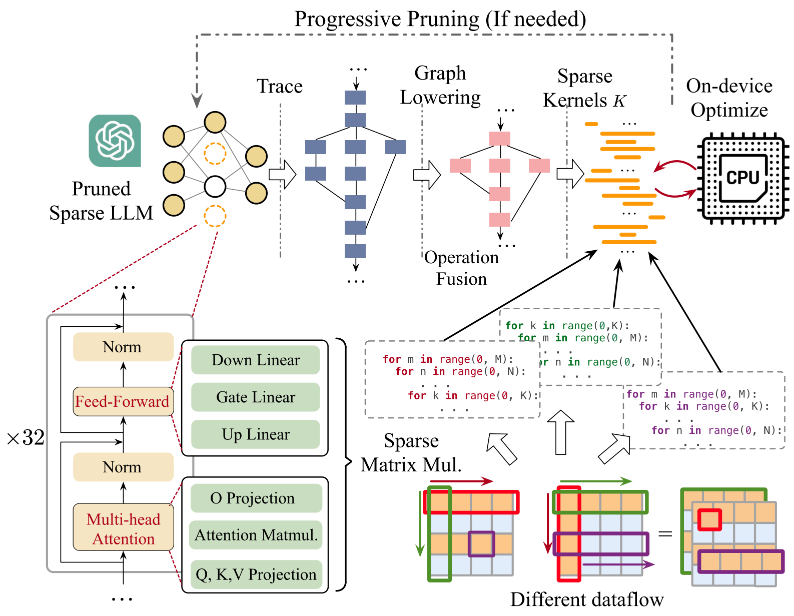
$$I_{W_i} = \beta_1 I_{lat_{W_i}} + \beta_2 I_{acc_{W_i}}, \tag{6}$$

therefore, is composed of both accuracy importance and latency importance, where β_1 and β_2 are weights for each objective. For the selection of β_1 and β_2, it's important to emphasize that accuracy is prioritized over latency, which is intuitive. For instance, if we set $\beta_1 = 1$ and $\beta_2 = 0$, the score $I_{W_i} = I_{latWi}$, meaning only the most computation-intensive components would be pruned, leading to severe accuracy collapse. Conversely, setting $\beta_1 = 0$ and $\beta_2 = 1$ makes $I_{W_i} = I_{accWi}$, resulting in traditional Taylor/gradient-based pruning with guaranteed accuracy but potentially suboptimal latency. In our experiments, we avoid sacrificing significant accuracy for latency improvement, especially when critical components are pruned. Thus, we set $\beta_1 = 0.05$ and $\beta_2 = 0.95$. Additionally, since the value ranges of I_{latWi} and I_{accWi} differ, we normalize them to the same range (0,1) before combining into one score.

3.3 Joint On-device Sparse Compilation Flow

Sparsification is not the final stage of our objective. We need to guarantee the on-device performance satisfies the latency/accuracy requirement on the local devices. We propose joint sparse compilation at the local platform along with the HAPE pruning strategy for utter performance optimization at deployment. We apply a full-stack compilation flow into our pruning flow, including: (1) computation graph tracing; (2) graph lowering and operation fusion; (3) sparse kernel dataflow optimization.

Original pruning objectives simplify the problem and ignore the hardware detailed implementation of each sparse operation. However, such a step is rather critical. A different dataflow/loop-level implementation for the same operation, even with the same sparse pattern, can perform significantly various inference latency. As we have analyzed in Table 1, the majority of computation layers within each transformer block are general matrix multiplication (large scale). Figure 4 is a visual example of hardware dataflow schemes and data layout influence: For a single sparse general matrix multiplication, the Inner-product scheme requires reading the first matrix by row and the second matrix by column, where the best encoding scheme for two input matrices is **compressed sparse row** (**CSR**) and **compressed sparse column** (**CSC**). The encoding scheme determines the data layout on memory, where a consistent visit on continuous words on memory can reduce the cache-miss rate and increase memory read/write efficiency. A negative example is that the inner-product scheme reads the first matrix by row while the data layout on memory is column-based.



```
for k in range(0,K):
  for m in range(0, M):
    . . .
      for n in range(0, N):
        . . .
```

```
for m in range(0, M):
  for n in range(0, N):
    . . .
      for k in range(0, K):
        . . .
```

```
for m in range(0, M):
  for k in range(0, K):
    . . .
      for n in range(0, N):
        . . .
```

Fig. 4. Visualization of joint compilation pruning. The upper part is an overview of all stages of post-pruning sparse compilation flow. The lower part is a visual demonstration of pruned candidates with different dataflow.

In this case, each element-wise read will jump to a different column and cause a cache miss, which will result in a very slow multiplication. On the other hand, the Outer-product scheme reads the first matrix by column and the second matrix by row. Our HAPE flow is aware of this low-level feature with post-compilation feedback.

Apart from that, pruning should also consider the hardware memory and computation constraint, which is another deterministic factor on latency. We also use the sparse matrix multiplication example in Figure 4, where the "red" flow denotes inner-product and "green" flow denotes outer-product, the "purple" is the row-based product (in between). Assuming the input matrices are A, B with size $d_1 \times d_2$ and $d_2 \times d_3$. If we focus on computation cost, the outer-product is much more efficient because it only read A and B for once. The inner-product instead read each row of A for d_3 times and each column of B for d_1 times, resulting in repetitive computation. However, on the other hand, the outer-product requires large memory size/bandwidth for it to update the entire output at each round. Transforming such as loop-tiling is required to handle large matrix tile by tile for less computation cost and less burden on large-scale "reduce" and less memory/cache pressure. On-device optimization on these kernel-level parameters such as tile size or loop order, can help balance the hardware resources with optimized sparse kernel.

Another critical factor in the HAPE flow to explore the pruned inference potential is graph-level optimization. As shown in the upper part in Figure 4, when the computation graph is traced, we can fuse the consecutive sparse operations based on predefined rules so that multiple operations are merged into single kernels such that unnecessary data read/write between layers can be saved. Such operator fusion holds two advantages: (1) The computation flow is simplified, and redundant calculation is saved. (2) Cross-layer internal data can skip being saved and loaded during the execution of different layers. Moreover, this step aligns with the inter-dependent neuron-fusion step in Section 3.1, where inter-dependent neurons with consecutive order can be fused into a single unit and pruned as a single unit.

ALGORITHM 1: Complete HAPE Pruning Flow

```
1: Input: sample set $\mathcal{S}$, initial sparsity ratio $\alpha_0$, target latency $\Psi_{lat}$, accuracy threshold $\Phi_{acc}$, pre-trained LLM $\mathbf{M}$, sparsity ratio gap $\lambda$;
2: $t \leftarrow 0, \alpha_0 \leftarrow 0.2$;
3: $I_{lat}, I_{acc} \leftarrow \mathbf{M}$.forward($\mathcal{S}$);   //On-device.        ▷ Equation (4)
4: $\mathbf{M} \leftarrow$ Dependency Fusion($\mathbf{M}$);
5: $I_{W_i} \leftarrow$ for all $W_i \in \mathbf{M}$;        ▷ Equation (6)
6: $\mathbf{M}_{sparse} \leftarrow$ Cross-layer Prune($\mathbf{M}, \alpha_0$);
7: Computation graph $G \leftarrow$ Trace($\mathbf{M}_{sparse}$)
8: Optimized Sparse Kernels $\mathbf{K} \leftarrow$ Compile($G$)   // On-device.
9: Latency $T_0$, accuracy $P_0 \leftarrow \mathbf{K}$.forward($\mathcal{S}$)
10: if $T_0 \leq \Psi_{lat}$ and $P_0 \geq \Phi_{acc}$ then
11:     Return $\mathbf{K}$; // Deploy successfully. Finish.
12: end if
13: while $P_t \geq \Phi_{acc}$ do   // Need further pruning.
14:     $t \leftarrow t+1, \alpha_t \leftarrow \alpha_{t-1} * 2$;
15:     $\mathbf{M}_{sparse} \leftarrow$ Cross-layer Prune($\mathbf{M}, \alpha_t$));
16:     $G \leftarrow$ Trace($\mathbf{M}_{sparse}$);
17:     $\mathbf{K} \leftarrow$ Compile($G$);  // On-device.
18:     $T_t, P_t \leftarrow \mathbf{K}$.forward($\mathcal{S}$);
19:     if $T_t \leq \Psi_{lat}$ then
20:         while $\alpha_t - \alpha_{t-1} \geq \lambda$; do   // Mitigate over-pruning.
21:             $t \leftarrow t+1$;
22:             $\alpha_t \leftarrow$ binary_search($\alpha_{t-1}, \alpha_{t-2}$);
23:             $\mathbf{K} \leftarrow$ Compile(Trace(Prune($\mathbf{M}, \alpha_t$)));
24:         end while
25:         Return $\mathbf{K}$; // Deploy successfully. Finish.
26:     end if
27: end while
28: Report: Infeasible latency/accuracy requirement on given hardware platform.
```

3.4 Complete HAPE Progressive Pruning Flow

Our complete pruning flow with joint on-device compilation is illustrated in Algorithm 1. In a real deployment scenario, given a local hardware platform and deployment inference latency requirement Ψ_{lat}, we start with on-device inference with one batch of N calibration data to derive the accuracy sensitivity and latency importance. In practice, we only use 10 calibration samples for this step. Once we achieve the block-wise grouping and ranking, we start with an initial sparsity ratio. Just in case if the latency requirement is not met meanwhile accuracy is still beyond threshold Φ_{acc}, we can progressively increase the sparsity ratio to fit the deployment requirement. At each round, we increase the sparsity ratio ×2. To avoid the over-excessive pruning, once the target latency Ψ_{lat} is met at round t, we apply a binary search between α_t and α_{t-1} to mitigate the increasing step size. Within each step, we evaluate the on-device accuracy and compile the pruned sparse model with all graph-level optimization and operation-level dataflow optimization to retrieve the genuine performance feedback. Given this progressive pruning flow, we can fully explore the sparse inference capability on certain devices. Even if the model cannot fulfill the latency/accuracy target, we can report the Pereto-optimal performance boundary with fully-optimized inference time for different sparsity ratios.

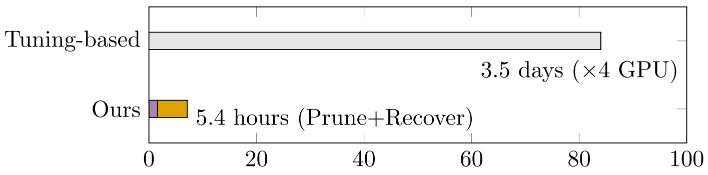


Fig. 5. Pruning time cost comparison with conventional transformer compression flow [9], whose down-tasking requires fine-tuning on original LLM model.

4 Experimental Results

4.1 Implementation Details

We utilize half-precision Llama-2-7B-hf [2] from META's official huggingface repository as the benchmark LLM model. The CPUdevice for deployment is Intel(R) Xeon(R) Silver 4210R CPU@2.40GHz. The GPU device for deployment as comparison is Nvidia H800 with 80 GB HBM size. The pruning algorithm was implemented in PyTorch, including model forward/backward and Autograd to collect the gradient of each pruning unit for importance estimation. We use TorchDynamo and Triton [30] for computation graph tracing and sparse kernel implementation, as well as the optimization on the hardware device. For the sample set to collect importance, we use sample size $N = 10$ with each item as a randomly generated sequence with a length of 128. For fair comparison, we prune the Llama-2-7B-hf [2] from the 4-th decoder block to the 30-th decoder block, which is exactly the same setting as LLM-pruner.

Dataset and Evaluation. Our pruning stage does not require a training set. For evaluation, we follow the same setting as Llama [2] to perform zero-shot task classification on seven common sense reasoning datasets to test the model performance on accuracy (%): BoolQ [31], PIQA [32], HellaSwag [33], WinoGrande [34], ARC-easy [35], ARC-challenge [35], and OpenbookQA [36]. We implement evaluation code using the same *lm-evaluation-harness* library [37], which is commonly used for language model performance evaluation. To evaluate the down-tasking ability, we use the same dataset as LLM-Pruner: zero-shot perplexity (PPL) analysis on WikiText2 [38] and PTB [39]. These two benchmarks use perplexity (PPL) as metrics. Higher perplexity (PPL) denotes worse accuracy performance.

4.2 Pruning Efficiency

Figure 5 is the pruning time comparison with the conventional method TinyBert [9], which fine-tunes the original model parameters for following compression and down-tasking by distillation. However, due to the gigantic size of current LLM models, it costs more than 3.5 days to tune Llama-2-7B [2] on 4 GPUs. In comparison, our pruning strategy only requires 1.6 hours on a single CPU device. The pruning stage only takes 10 sample inputs with a sequence length of 128. Even if we apply the same LoRA-based recovering stage in LLM to repair the accuracy drop from post-training pruning, the overall time consumption is only 5.4 hours, which is still 94% faster than TinyBert [9].

In addition to time cost removal, HAPE does not require a large server cluster or a considerable amount of data. We prune the model simply on a CPU device Intel(R) Xeon(R) Silver 4210R CPU@2.40GHz. Such an advantage significantly reduce the difficulty of the down-tasking and deployment of LLM in real scenarios.

4.3 Performance Analysis

Model Accuracy. First, we compare with the state-of-the-art LLM-Pruner [29] on the common Llama benchmark [2]: zero-shot task classification evaluation. We evaluate the same seven

Table 2. Performance Comparison on Accuracy (%) and Inference Latency (s) at Different Sparsity Ratio on Llama-2 (7B) on CPU Device

Pruning Ratio	Method	granularity	BoolQ	PIQA	HellaSwag	WinoGrande	ARC-e	ARC-c	OBQA	Avg. Acc.	Avg. time
Ratio = 0.0%	Original	–	71.10	78.35	56.70	67.01	69.28	39.85	31.80	59.15	66.8
ratio = 20%	LLM-Pruner [29]	Layer-wise	55.60	61.32	30.26	51.54	39.18	20.56	18.80	39.61	51.4
	HAPE	Transformer	53.49	76.77	51.18	64.48	66.54	37.20	28.20	**53.98**	**42.4**
ratio = 40%	LLM-Pruner [29]	Layer-wise	58.72	64.58	35.95	52.01	43.90	23.12	19.40	42.52	42.6
	HAPE	Transformer	51.83	75.73	48.26	60.62	57.91	31.57	27.00	**50.41**	**36.4**
ratio = 60%	LLM-Pruner [29]	Layer-wise	40.34	54.57	27.24	50.83	30.51	20.14	15.20	34.11	33.4
	HAPE	Transformer	59.08	69.04	38.19	52.01	47.69	25.77	22.00	**44.82**	**30.0**

Table 3. Performance Comparison on Accuracy (%) and Inference Latency (s) at Different Sparsity Ratio on Llama-3.1 (8B) on GPU Device

Pruning Ratio	Method	granularity	BoolQ	PIQA	HellaSwag	WinoGrande	ARC-e	ARC-c	OBQA	Avg. Acc.	Avg. time
Ratio = 0.0%	Original	–	83.21	82.54	79.80	75.14	83.08	56.23	46.40	72.34	0.1894
ratio = 20%	LLM-Pruner [29]	Layer-wise	76.73	76.66	69.64	67.56	72.90	44.03	40.00	63.93	0.3448
	HAPE	Transformer	76.54	79.11	75.38	68.98	77.23	48.55	44.20	**67.14**	**0.2196**
ratio = 40%	LLM-Pruner [29]	Layer-wise	64.34	66.70	47.91	56.04	56.57	30.20	30.00	50.25	0.3410
	HAPE	Transformer	63.24	75.84	61.53	58.80	64.52	37.12	35.40	**56.64**	**0.3018**
ratio = 60%	LLM-Pruner [29]	Layer-wise	61.01	58.43	30.55	50.28	34.97	22.78	25.40	40.49	0.3426
	HAPE	Transformer	47.80	70.84	50.38	57.22	55.05	30.89	30.80	**49.00**	**0.16334**

benchmark datasets as Llama [2]. We list the performance under different sparsity ratios: 20%, 40%, and 60% to show that our strategy has a solid performance advantage. In Table 2, our HAPE pruning strategy applies dynamic ratios within each transformer block and a fixed overall pruning ratio for each block. In contrast, we set the baseline method's pruning granularity at each layer to show the performance improvement from the flexible pruning ratio assignment. At this evaluation stage, we add the same LoRA-based 2-epoch recovery tuning on the pruned model as LLM-Pruner [29] described for fair comparison. Table 2 has shown our pruning strategy performs 16%, 19% and 36% better performance at three ratios. Interestingly, we notice the performance of a 60% pruning ratio on BoolQ surpasses that of a 40% pruning ratio. Such inconsistency is due to the dataset itself as we notice inconsistency in Table 3 such that our model shows accuracy degradation. We use the lm-evaluation-harness library [37] for evaluation, which is the public evaluation benchmark to make sure the evaluation process is solid and consistent.

In order to prove the advantage of our method, we also evaluate the performance on Llama-3.1 (8B) on Nvidia H800 GPU. The result is shown in Table 3. The result shows that HAPE can also achieve better performance on Llama-3.1 (8B) on Nvidia H800 GPU. The performance improvement is 5%, 12%, and 21% at three ratios. We also set an ablative study on the down-tasking ability with dynamic ratio assignment within transformer block in Figure 6. We prune the model from 15% all the way to 60% and notice that the dynamic ratios can reduce the perplexity by over 10^2 times. It is shown that HAPE holds better down-tasking ability as the pruning ratio increases with dynamic ratio, avoiding exponentially increased PPL.

Inference Latency. We also compare the inference latency. HAPE's on-device sparse compilation helps optimize inference speed. We conduct speed evaluation on sparsity ratio: 20%, 40% and 60% as well. In order to emulate the transformer inference on natural language query with a series of tokens, we start with a sequence with a length of 128 tokens and collect the time for predicting the next token for fair comparison. We skip the first ten inferences to avoid cold-start influence, and conduct 50 inferences five times, and calculate the average time in Table 2. We achieve 21.2%, 17.1%, and 11.3% faster inference with during-pruning sparse compilation optimization on the device. However, if we look at the inference speed in Table 3 vertically, the inference speed trend varies

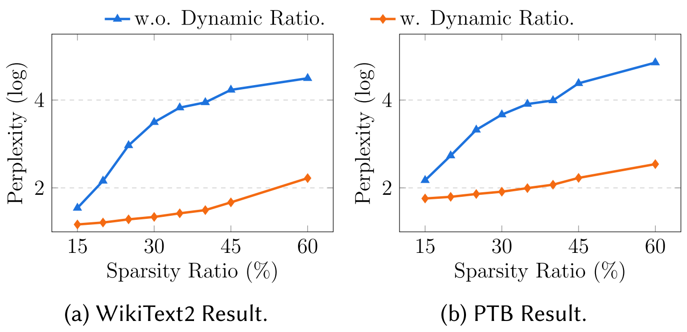


Fig. 6. Evaluation on down-tasking ability with/without cross-layer dynamic ratio. Benchmark WikiText2 and PTB with metric of logarithmic perplexity (PPL) with base 10.

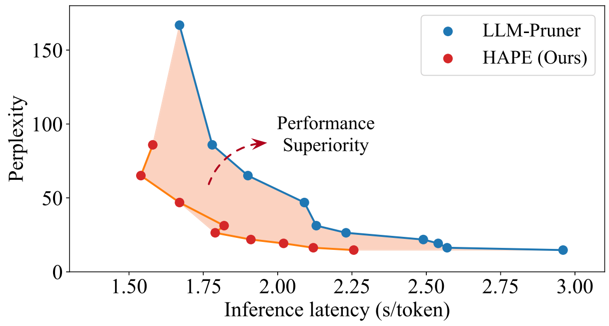


Fig. 7. Optimal frontiers Visualization. HAPE shows both superior accuracy and speed.

from the CPU device. The inference speed on Nvidia H800 GPU does not necessarily increase with increased pruning ratio. This is because the GPU has a different architecture and memory hierarchy. More importantly, the sparsity pattern instead of the sparsity ratio is the dominating factor after compilation into sparse kernels. We can tell the same result in both Table 3 (on Llama-3.1-8B) and Figure 8 (on Llama-2-7b).

Optimal Pareto Frontier. Last but not least, we evaluate the complete progressive pruning flow of HAPE and compare it with the complete flow of LLM-Pruner and plot the optimal frontier regarding on-device accuracy and inference speed in Figure 7. At this stage, we conduct pruning from all reasonable sparsity ratios ranging from 15% to 60%. As an example, we plot PPL on Wikitext2 and speed on next-token prediction time with sequence length 128. In Figure 7, the lower curve connecting "red" dots is the optimal Pareto frontier of HAPE, depicting the best accuracy-speed performance tradeoff that HAPE can reach. The upper curve connecting blue dots is the optimal Pareto frontier of the baseline LLM-Pruner. The "pink" area bounded by the two curves symbolizes our performance superiority. Apart from that, the two curves have no cross, indicating that we dominate the superiority at all pruning ratios.

Hardware Applicability. We also evaluate the hardware applicability of HAPE on a range of hardware backends. We conduct the same optimal frontier evaluation on Nvidia H800 GPU in Figure 8. The evaluation is conducted on the same Llama-2-7B model with the same sequence length of 128. The result shows that HAPE can also achieve the same performance superiority on Nvidia H800 GPU with 80 GB HBM memory. The optimal frontier is similar to the CPU device,

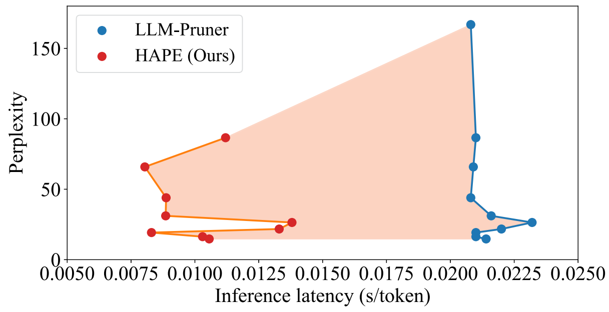


Fig. 8. Hardware applicability: Optimal frontier visualization of nvidia H800 GPU.

Table 4. Performance Analysis of Model Parameter Numeric Influence on Llama-2 (7B) and Llama-3 (8B), their Different Embedding Part is Unchanged in Pruning Process

Ratio	Model	B.Q	PIQA	H.S.	W.G.	A.-e	A.-c	OBQA
0%	Llama-2 (7B)	71.10	78.35	56.70	67.01	69.28	39.85	31.80
	Llama-3 (8B)	83.00	80.63	61.36	74.66	83.29	55.72	35.80
20%	Llama-2 (7B)	53.49	76.77	51.18	64.48	66.54	37.20	28.20
	Llama-3 (8B)	76.67	77.53	53.13	68.19	71.51	41.04	28.20
40%	Llama-2 (7B)	51.83	75.73	48.26	60.62	57.91	31.57	27.00
	Llama-3 (8B)	71.83	76.50	50.88	65.59	67.34	36.35	28.20
60%	Llama-2 (7B)	59.08	69.04	38.19	52.01	47.69	25.77	22.00
	Llama-3 (8B)	61.19	71.65	42.33	60.06	58.21	29.01	24.80

which indicates that HAPE is hardware-agnostic and can be applied to a wide range of hardware backends. Interestingly, the optimal frontier on Nvidia H800 GPU is different from the CPU device, whose inference speed does not necessarily increase with increased pruning ratio. This is because the GPU has a different architecture and memory hierarchy compared to the CPU device. The optimal frontier on Nvidia H800 GPU shows that the inference speed can be improved by 10% with a 60% pruning ratio, which is different from the CPU device. This indicates that HAPE can achieve different performance superiority on different hardware backends.

Model Parameters Influence. We also validate the influence of the numerical difference of model parameters. We conduct same pruning process on LLama-3-8B [40] and evaluate the model performance before and after the same LORA-tuning. The core idea comes from the fact that Llama-2-7B [2] and Llama-3-8B [40] share nearly similar architecture: both with 32 transformer layers the same hidden dimension of 4,096. A trial difference is the projection dimension goes to 14,336 from 11,008, which is not the dominating factor. The main difference is the vocabulary size, which results in embedding dimension of 128,256 for Llama-3 and 32,000 for Llama-2, however, this embedding stage is invariant in our framework and will not adjust during pruning, therefore, the pruning candidates are similar. From Table 4 we discover that Llama-3-8B performs somewhat better than Llama-2-7B under same pruning ratio: 20%, 40%, 60% with10.17% $\uparrow$, 12.4% $\uparrow$, 10.67% $\uparrow$ improvement from model parameters influence after LORA-tuning.

On the other hand, we evaluate the zero-shot performance without LORA-tuning. The direct performance on WikiText2 and PTB shows that Llama-3-8B shows much higher robustness as pruning ratio increases. Figure 9 indicates that as the pruning ratio increases from 0% to 60%,

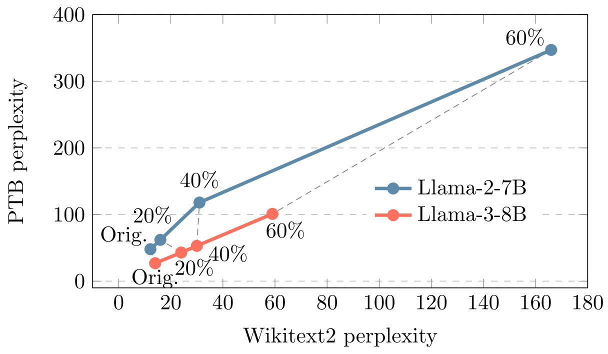


Fig. 9. Comparison of robustness to pruning. Under original model or 20% pruning, Llama-3-8B may not perform better than Llama-2-7B. As pruning ratio increases, Llama-3-8B shows robustness with less perplexity increase.

Llama-2-7B gives perplexity from 12.18 to 166.86 on WikiText2 and 48.37 to 347.7 on PTB. On the other hand, Llama-3-8B gives 14.14 on WikiText2, which is even worse than Llama-2-7B, however, reaches a much lower 59.86 at 60% pruning ratio, and the same trend for PTB. Therefore, We can tell the model parameter numeric also matters to the pruning robustness.

5 Conclusion

We hold the key insight that LLM model pruning shall not be decoupled from hardware backend information when being deployed. Therefore, we propose HAPE: a hardware-aware LLM pruning strategy for efficient LLM compression and deployment on general-purpose hardware. This framework is a post-training pruning flow that involves backend information and on-device optimization at each stage. We avoid the time-consuming and expensive fine-tuning on the original LLM model. As a matter of fact, we successfully conducted the pruning process with a single CPU device and a tiny time budget. Not only did we accumulate on-device latency sensitivity for importance assignment, but we also brought in on-device compilation to retrieve genuine inference feedback during pruning. We also apply dynamic pruning ratios within each transformer block for flexibility. We accomplish the optimality in both performance and efficiency with our on-device hardware-aware optimization.

References

[1] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30, 1 (2020), 681–694.

[2] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

[3] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL'19)*.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Annual Conference on Neural Information Processing Systems* 30, 1 (2017), 5998–6008.

[5] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'23)*.

[6] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*. 2023.

[7] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. 2022. The optimal BERT surgeon: Scalable and accurate second-order pruning for large language models. In *Findings of the Association for Computational Linguistics: EMNLP*. 4163–4181.

[8] Wenqian Zhao, Qi Sun, Yang Bai, Wenbo Li, Haisheng Zheng, Bei Yu, and Martin DF Wong. 2021. A high-performance accelerator for super-resolution processing on embedded GPU. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'21)*. IEEE, 1–9.

[9] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP*. 4163–4174.

[10] Xufeng Yao, Fanbin Lu, Yuechen Zhang, Xinyun Zhang, Wenqian Zhao, and Bei Yu. 2024. Progressively knowledge distillation via re-parameterizing diffusion reverse process. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'24)*.

[11] Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jin Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. 2021. BinaryBERT: Pushing the limit of BERT quantization. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL'21)*. 4334–4348.

[12] Zehua Pei, Xufeng Yao, Wenqian Zhao, and Bei Yu. 2023. Quantization via distillation and contrastive learning. *IEEE Transactions on Neural Networks and Learning Systems* 35, 1 (2023), 17164–17176.

[13] Deming Ye, Yankai Lin, Yufei Huang, and Maosong Sun. 2021. TR-BERT: Dynamic token reduction for accelerating BERT inference. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL'21)*. 5798–5809.

[14] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic early exiting for accelerating BERT inference. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL'20)*. 2246–2251.

[15] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *CoRR* abs/2208.07339. 2022.

[16] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. *Annual Conference on Neural Information Processing Systems* 28, 1 (2015), 1135–1143.

[17] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. 2020. Dynabert: Dynamic bert with adaptive width and depth. *Annual Conference on Neural Information Processing Systems* 33, 1 (2020), 9782–9793.

[18] Woosuk Kwon, Sehoon Kim, Michael W. Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. 2022. A fast post-training pruning framework for transformers. *Annual Conference on Neural Information Processing Systems* 35, 1 (2022), 24101–24116.

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

[20] Se Jung Kwon, Dongsoo Lee, Byeongwook Kim, Parichay Kapoor, Baeseong Park, and Gu-Yeon Wei. 2020. Structured compression by weight encryption for unstructured pruning and quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'20)*. 1909–1918.

[21] Xizi Chen, Jingyang Zhu, Jingbo Jiang, and Chi-Ying Tsui. 2020. Tight compression: Compressing CNN model tightly through unstructured pruning and simulated annealing based permutation. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC'20)*. IEEE, 1–6.

[22] Tianyang Yu, Bi Wu, Ke Chen, Chenggang Yan, and Weiqiang Liu. 2022. Data stream oriented fine-grained sparse CNN accelerator with efficient unstructured pruning strategy. In *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI'22)*. 243–248.

[23] Mingbao Lin, Rongrong Ji, Yuxin Zhang, Baochang Zhang, Yongjian Wu, and Yonghong Tian. 2020. Channel pruning via automatic structure search. *arXiv preprint arXiv:2001.08565*.

[24] Zi Wang, Chengcheng Li, and Xiangyang Wang. 2021. Convolutional neural network pruning with structural redundancy reduction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'21)*. 14913–14922.

[25] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. Learning n: m fine-grained structured sparse neural networks from scratch. *arXiv preprint arXiv:2102.04010*.

[26] Mingbao Lin, Yuxin Zhang, Yuchao Li, Bohong Chen, Fei Chao, Mengdi Wang, Shen Li, Yonghong Tian, and Rongrong Ji. 2022. 1xn pattern for pruning convolutional neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 4 (2022), 3999–4008.

[27] Yuezhou Hu, Kang Zhao, Weiyu Huang, Jianfei Chen, and Jun Zhu. 2024. Accelerating transformer pre-training with 2: 4 sparsity. *arXiv preprint arXiv:2404.01847*.

[28] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. 2020. Up or down? Adaptive rounding for post-training quantization. In *Proceedings of the International Conference on Machine Learning*. PMLR, 7197–7206.

[29] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. *Advances in Neural Information Processing Systems* 36 (2023), 21702–21720.

[30] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[31] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2924–2936.

[32] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 7432–7439.

[33] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a machine really finish your sentence?. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4791–4800.

[34] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2020. WinoGrande: An adversarial winograd schema challenge at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 8732–8740.

[35] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

[36] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? A new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 2381–2391.

[37] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, et al. 2021. A Framework for Few-shot Language Model Evaluation. (Sept. 2021). DOI: http://dx.doi.org/10.5281/zenodo.5371628

[38] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2022. Pointer sentinel mixture models. In *Proceedings of the International Conference on Learning Representations*.

[39] Mitch Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics* 19, 2 (1993), 313–330.

[40] Meta. 2024. Meta Llama 3. (2024). Retrieved April 5, 2024 from https://github.com/meta-llama/llama3

Received 8 July 2024; revised 18 February 2025; accepted 9 April 2025