Toward Efficient Machine Learning Systems with Sampling and Compression

Chien-Yu Lin

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington
2025

Reading Committee:
Luis Ceze, Chair
Baris Kasikci
Arvind Krishnamurthy

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Toward Efficient Machine Learning Systems with
Sampling and Compression

Chien-Yu Lin

Chair of the Supervisory Committee:

Professor Luis Ceze
Computer Science and Engineering

The rapid growth of machine learning - in terms of model size, dataset volume, and task complexity - has created significant computational and memory efficiency challenges. This thesis addresses these challenges by developing sampling and compression techniques across a variety of machine learning applications. Specifically, we introduce sampling strategies for efficient graph neural networks (*CacheSample*), accelerated 3D image rendering (*FastSR-NeRF*), and optimized retrieval-augmented generation systems (*TeleRAG*). We further propose novel compression methods targeting convolutional neural networks (*SPIN*), large language models (*Atom*), and their key-value caches (*Palu*). Collectively, these techniques substantially reduce computational and memory requirements while preserving model accuracy, facilitating the scalability and accessibility of machine learning systems. Finally, I present my vision for future efficiency innovations to ensure continued scalability and robustness as machine learning models continue to grow in complexity.

Acknowledgements

When I started my PhD in 2018, I had heard it would be hard—but I never imagined it would take nearly seven years and bring so many challenges. Along the way, I've been incredibly fortunate to have the support of many people. Without them, I would not have made it to the finish line.

First and foremost, I want to sincerely thank my advisor, Luis Ceze. Thank you for your guidance and support throughout my PhD, and for taking me on when I decided to focus more on machine learning systems after my first year. That decision opened the door to some of the most exciting research I've ever done. Thank you for believing in me—even during the times when I was stuck. Your support meant a lot and helped me keep going.

I'm also grateful to Baris Kasikci. Though I'm not officially your student, thank you for your generosity in mentoring me and offering invaluable feedback in our discussions and paper writing.

Thank you to Arvind Krishnamurthy for being on my committee from the start. I really appreciated the opportunity to help design the ML systems course in Fall 2024—it was a fun and formative teaching experience. I also thank Luke Zettlemoyer and Arka Majumdar for bringing valuable perspectives from different domains to my committee, and Stephanie Wang for your mentorship in navigating collaborations and helping organize our research group.

To my mentors and collaborators at Apple—Carlo, Anish, Thomas, Qichen, Karren, Anurag, Sachin, and Max—thank you for working with me and devoting time to our joint projects. Those experiences gave me confidence that I can contribute meaningfully to impactful research.

Thanks to all my collaborators in the Sampl Lab. A special shout-out to Zihao—your ability to tackle complex problems continues to amaze me. Working with you expanded my perspective and inspired me to take on harder challenges. To the early Sampl Lab crew—Liang Luo, Luis Vega, Thierry Moreau,

Megan, and Eddie—thank you for making school life fun and welcoming. And to the post-COVID lab members—Yile, Dedong, Liangyu, Kan, and Rohan, Zechou, Haoran and more—I've really appreciated your friendliness, whether collaborating or just chatting about life.

I'm also thankful for the interns I've had the pleasure of working closely with: Yilong Zhao, Chi-Chih Chang, and Yiyu Liu. It's been a privilege to work alongside you and to see you grow during and beyond the internship.

To my office mates in CSE 394—especially Champ, Valentina, and Ensol—thank you for making the office a place I actually enjoyed coming to every day.

To the CSE Bible study group—Ensol, Aaron, Robie, Nick, Christine, and Andrew—those early morning sessions were a highlight of my first two years. I'm grateful for the joy, depth, and friendship we shared.

I want to thank the Bethel Fellowship community and mentors—Daniel, Ellen, Danny, Judy and many more. I'm so glad I was part of this fellowship through the core of my PhD. Your prayers, support, and friendship sustained me through many ups and downs.

To my close friends, Yoyee and Calvin—I truly couldn't have made it through life in Seattle without you. You made it so much more colorful. And to my adventure crew—Yoyee, Meredith, Tian Tian, Mandy, So, David, George, and more—thank you for all the unforgettable memories from our many crazy outdoor escapades.

Last but not least, I want to thank my parents. Thank you for your unconditional love and for raising me. Thank you for supporting my decision to pursue a PhD and for always being there through every situation. None of this would have been possible without you. Sorry for making you wait so long—I'll make sure the wait was worth it.

DEDICATION

To my sister. I will miss you always.

Contents

List of Figures

List of Tables

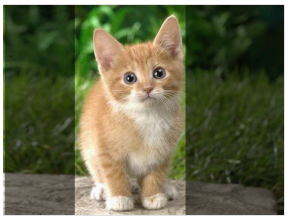
Chapter 1

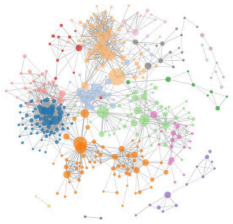
Introduction

Machine learning (ML) models have become ubiquitous across a wide range of domains, enabling the processing and understanding of diverse data modalities, including images [103], text [66], and graphs [148]. More recently, advances in multimodal learning have extended these capabilities to handle combinations of input types—such as image-text pairs [219]—and to interact dynamically with external systems through tools like information retrieval [184] and function calling [211].

A defining trend in modern ML is the rapid and exponential growth in model scale [87], encompassing both the number of parameters and the size of training datasets. For instance, early deep learning models such as AlexNet [152] or ResNet [103] were trained on datasets like ImageNet [63], which contains approximately 1.2 million labeled images. In contrast, recent large language models (LLMs) such as GPT-3 [37] and LLaMA [251] are trained on corpora exceeding 15 trillion tokens. On the model size dimension, transformer-based architectures have expanded from BERT's 110 million parameters in 2018 [66] to over 175 billion in GPT-3 [37], and even surpass 1 trillion parameters in models like Switch Transformer [75].

This scaling has not only improved performance but also shifted the paradigm of ML tasks. Earlier models, including convolutional neural networks (CNNs) and recurrent neural networks (RNNs), were primarily designed for discriminative tasks such as image classification and sentiment analysis. In contrast, modern generative models—powered by LLMs and diffusion models—can synthesize coherent text, produce photorealistic images, and even generate high-fidelity video content. These capabilities reflect a shift toward models that must reason about structure, context, and intent, exhibiting a more general form of intelligence.





Q: Is Seattle rainy everyday?
Wikipedia: Seattle
The Seattle area is the cloudiest.…
In an average year, … **precipitation falls on 150 days**, more than nearly all U.S. cities east of.……

(a) Redundant patches in an image. **(b)** Redundant edges might exist in a large-scale graph. **(c)** Redundant text in the context that are irrelevant to a question.

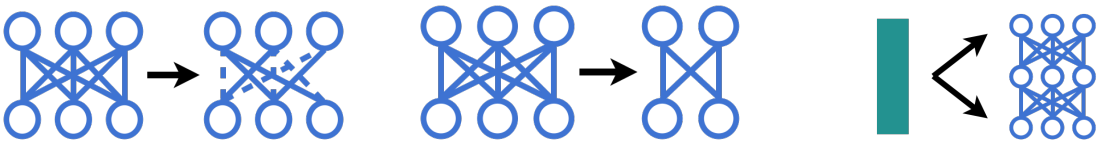
Figure 1.1: Illustration of redundancy in image, graph structure data, and text.

However, as model sizes and data requirements continue to grow, so do the computational and memory demands for training and inference. This introduces significant challenges in terms of energy efficiency, hardware accessibility, and deployment scalability. To ensure that the benefits of modern ML systems remain broadly accessible, it is essential to develop more efficient algorithms and systems that can reduce cost and latency while maintaining or enhancing model performance. Democratizing access to these powerful models requires innovations that span across algorithm design, system optimization, and hardware utilization.

Despite their large scale, redundancy is a prevalent and exploitable characteristic in both data and models. Redundancy may naturally occur in datasets for training or arise from model over-parameterization. For instance, a substantial portion of an image background often carries little information about the object itself; CNNs maintain performance even with extensive pruning [99], and LLMs can maintain accuracy when compressed to very low bit-width (4-bit) [313]. Leveraging such redundancy offers potential for enhancing computational efficiency by reducing unnecessary calculations or saving memory.

Among the techniques for leveraging redundancy in machine learning, sampling and compression emerge as particularly effective approaches. Sampling identifies a subset of informative data or computations, allowing the system to skip redundant or less informative processing while maintaining overall accuracy. Compression, on the other hand, identifies and eliminates redundancy within model weights or activations, thereby reducing memory usage, lowering data transfer overhead, and decreasing computational cost. Figure 1.1 illustrates examples of sampling techniques applied to image, graph, and text domains, while Figure 1.2 presents commonly used compression methods in deep learning.

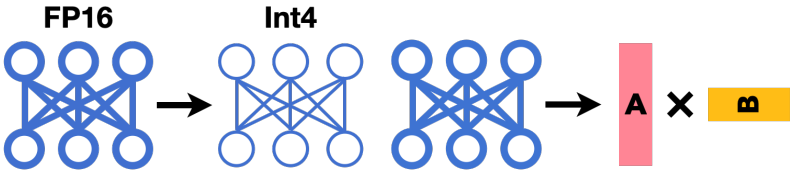
However, effectively exploiting redundancy in machine learning workloads remains challenging without carefully designed algorithms and system support. o Naive approaches—such as randomly dropping data



(a) Pruning: removing redundant weights.

(b) Distillation: training small models from a larger model.

(c) Weight sharing: reusing weights for multiple layers in the neural network.



(d) Quantization: using less bit-width to represent each model weight.

(e) Low-rank decomposition: factorizing a large weight matrix into two small matrices.

Figure 1.2: Illustration of common use compression technique for deep learning models.

during sampling or aggressively compressing sensitive model weights—can significantly degrade model accuracy. On the efficiency side, models with unstructured sparsity often fail to deliver practical speedups due to poor hardware utilization, as modern hardware architectures are often not well-suited for irregular computation patterns. These challenges underscore the need for hardware-friendly algorithms and tightly co-optimized system implementations. In this thesis, I aim to enhance the efficiency of machine learning models across various data types and modalities by developing advanced sampling and compression techniques. My approach emphasizes algorithm-system co-design, enabling practical performance gains while retaining model quality.

For sampling techniques, I have explored applications in Graph Neural Networks (GNNs), 3D neural rendering (NeRF), and retrieval-augmented generation (RAG). Specifically, I introduced CacheSample (Chapter 2), an in-kernel edge sampling method accelerating GNN inference by up to 20× with less than a 1% accuracy drop. In 3D neural rendering, I developed FastSR-NeRF (Chapter 3), which employs a CNN-based super-resolution model and introduces a novel random patch sampling algorithm that significantly improves training efficiency. For retrieval-augmented generation (RAG), my work, TeleRAG (Chapter 4), exploits the structure of the commonly used inverted file index (IVF) and modular design of RAG models to create an efficient sampling algorithm and system design, significantly accelerating GPU-based retrieval while maintaining minimal memory usage.

Regarding compression, I explored weight sharing, quantization, and low-rank decomposition techniques, applying them to CNN models and LLMs. In CNNs, I studied different weight-sharing strategies to compress model sizes and proposed SPIN (Chapter 5), an empirical study framework with a novel distillation-based method that achieves efficient compression without accuracy loss. For LLMs, my work on Atom (Chapter 6) introduced a 4-bit quantization framework, enabling significant reductions in model size and activating size, thereby enhancing serving throughput on GPUs. Additionally, with Palu (Chapter 7), I advanced low-rank projection by redesigning the projection mechanism to dramatically reduce overheads and integrate seamlessly with quantization. This method outperforms existing state-of-the-art quantization approaches in both compression rate and speed.

In summary, this thesis presents an extensive exploration into sampling and compression techniques through algorithm and system co-design, significantly improving the efficiency of diverse machine learning models. Future directions include extending this research to multi-modal systems, exploring novel architectures beyond transformers, and addressing increasingly complex model interactions.

Part I

Efficient Machine Learning Systems with Sampling

Chapter 2

Accelerating Graph Neural Networks Inference with Edge Sampling

Remarks on Chapter Material

The content of this chapter is adapted from the following tech report:

- Chien-Yu Lin, Liang Luo, Luis Ceze, "Accelerating SpMM Kernel with Cache-First Edge Sampling for Graph Neural Networks", ArXiv 2021.

2.1 Introduction

GNNs are a class of powerful deep learning (DL) models that can extract high-level embeddings from graph-structured data. Because the graph structure can represent many different types of information, GNN has a wide-range of applications, including network data mining [148; 96], program synthesis [20], physics modeling [31] and medical decision making [323]. As a result, this model is attracting significant attention from both academia and industry.

Unfortunately, GNNs are notoriously inefficient to run because of their sparse operations [112; 114]. As Figure 2.1 shows, a typical GNN layer uses a multi-layer perceptron (MLP) to extract dense node features and aggregate the features according to the edges of the graph. In practice, MLP feature extraction uses the general matrix multiplication (GEMM) operation, and feature aggregation uses the sparse-dense matrix

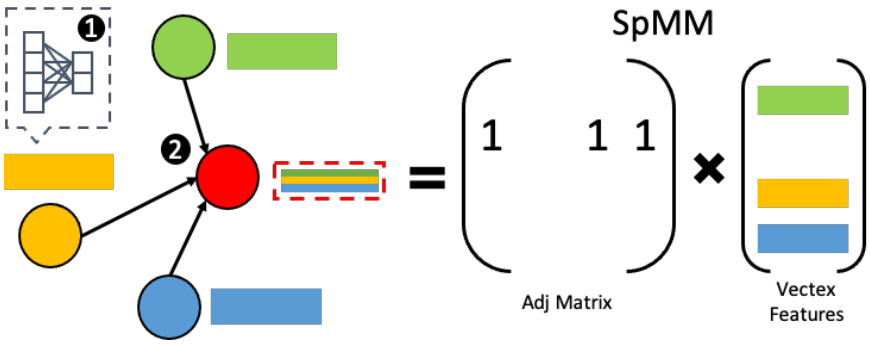


Figure 2.1: Basic operations of a typical GNN layer.

multiplication (SpMM) or an SpMM-like operation. Since GNNs usually run on GPUs, where the hardware architecture and memory system are optimized for dense matrix operations, the GPU time spent on the dense GEMM operation is relatively short, hence the performance bottleneck usually lies in the sparse SpMM operation. In our analysis, for a standard GCN model [148], the SpMM kernel can consume up to 95% of the end-to-end inference time.

To make SpMM run faster on GPUs, researchers have explored different SpMM kernel optimizations, including leveraging GPU's Instruction-Level Parallelism (ILP) [285], Thread-Level Parallelism (TLP) [285], and cache memory [285; 114]; improving work balance; and using customized sparse formats [207; 90]. However, these methods still fall short of speeding GNN inference performance on GPUs as previous literatures only show limited speedup (10% to 30%) [285; 114; 106] over the standard *cuSPARSE* kernel or is optimized for problems with lower sparsity, e.g., pruned neural networks [80]. It is thus critical to broader solutions than kernel optimizations alone to meet this goal.

Independently, edge sampling techniques [225; 102; 314] proposed by researchers allow the training of possibly deeper and more accurate GNN models by solving the over-smoothing and over-fitting problems [165; 284; 150]. Since edge sampling reduces the number of connections in the graph, which in turn reduces SpMM computation time, it could potentially be used to speed up the SpMM kernel. However, existing methods requires preprocessing the graph on the CPU [225] or using a complex model to sample the edges [314]. The resulting added overheads eventually eclipse the benefit of reduced computational complexity.

To more efficiently leverage edge sampling for GNN acceleration, we introduce *CacheSample*, a sampling algorithm and codesigned SpMM kernel to speed GNN inference performance. Key *CacheSample*

concepts include: (1) an *in-kernel sampling mechanism*, which eliminates preprocessing overheads and leverages a GPU's parallel computing power to accelerate the sampling process, (2) a *cache-first edge sampling design* that fits the entire sampled graph on a GPU's shared memory by using the shared memory size as the sampling target, reducing compute load while improving cache locality, (3) *coalesced global memory access* to efficiently fetch the sampled graph data and load shared memory. (4) and two *lightweight sampling strategies* to use in conjunction with the kernel to achieve optimal speedup with minimal accuracy. The two strategies offer different trade-offs between model accuracy and speedup and are adaptable to the dataset specifics. Using these novel optimizations and designs, *CacheSample* significantly outperforms state-of-the-art SpMM implementations with negligible accuracy loss for GNNs' workloads.

To verify the effectiveness of our design, we integrated *CacheSample* into DGL, a popular graph learning library that provides state-of-the-art GNN runtime performance. We designed *CacheSample* to work with the standard CSR (Compressed Sparse Row) format to avoid format transformation overhead, which lets us seamlessly swap DGL's backend SpMM kernel with *CacheSample* with no modification to user-level code. While the proposed techniques in *CacheSample* are sufficiently general to benefit both GNN training and inference, we focus here on inference only. We conducted comprehensive experiments using two popular GNN models, GCN and GraphSage, on four representative graph benchmarks. Our evaluation results for inference time show that *CacheSample* outperformed *cuSPARSE* by up to 4.35x with no accuracy loss and 45.3x with less than a 1% accuracy loss.

In sum, our main contributions include:

- The design and implementation of *CacheSample*, a cache-first edge sampling mechanism and SpMM kernel codesign, to speed up GNN inference.
- A novel in-kernel edge sampling mechanism that accelerates the sampling process and eliminates the preprocessing overhead.
- Two lightweight sampling strategies to achieve optimal speedup with minimal accuracy loss across different datasets.
- Standard CSR format support and the integration with a popular GNN framework, DGL, without touching user-level code.

- Comprehensive evaluations on representative GNN models and datasets to verify *CacheSample*'s significantly improved inference efficiency.
- To the best of our knowledge, *CacheSample* is the first work that uses edge sammpling to accelerate GNN inference.

2.2 Graph Structure Data and Sparse Computation

In this section, we now briefly describe the GNN, the Compressed Sparse Row (CSR) format, GPU preliminaries, relevant SpMM optimization on GPUs, and the current state of the GNN computational bottleneck.

Graph Structure Data

GNNs target feature-rich graphs with have high-dimensional, dense features associated with nodes or edges. A typical GNN has a multi-layer structure. Within a layer, it uses an MLP to extract high-level features from the input features, aggregate the extracted features based on the graph structure and apply an activation function like ReLU on the output features (see Figure 2.1). Activated features become the input features for the next layer. The final layer's output features are then be used for downstream prediction tasks, such as node classification or edge prediction.

GNNs differ from basic neural networks mainly due to feature aggregation. Among the different feature aggregation methods, *Sum* is the most commonly used feature aggregator. In *Sum* aggregation, a node's new features are the sum of its neighborhood nodes' features. Equation 2.1 shows the formalized computation of a GNN layer using *Sum* aggregation. $H^{(l)}$ and $W^{(l)}$ are node features and MLP weights for layer l, A is the adjacency (adj) matrix, $\sigma(\cdot)$ is the activation function, and the output of the equation becomes the next layer's node features ($H^{(l+1)}$).

$$H^{(l+1)} = \sigma(AH^{(l)}W^{(l)}) \tag{2.1}$$

Here, the computation of $H^{(l)}W^{(l)}$ is essentially a GEMM since both H and W are dense matrices. The multiplication of A and $H^{(l)}W^{(l)}$ is an SpMM operation because A is a sparse matrix and the result of $H^{(l)}W^{(l)}$ is a dense matrix. Since *Sum* is an effective aggregator and adopted by many GNN models

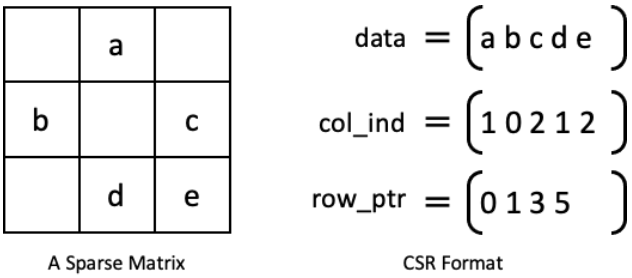


Figure 2.2: The CSR sparse format.

[148; 96; 45; 54; 284; 115; 304; 164; 193], in this work we focus on accelerating the GNNs using *Sum* aggregation.

Compressed-Sparse Row (CSR) Format

Sparse formats efficiently store sparse data; in the context of GNN, the sparse adjacency matrix is stored in a sparse format, which skips all zero values to compress the matrix.

There are several common sparse formats. Here, we introduce the most standard one, CSR (see Figure 2.2). CSR has three arrays to represent a 2D sparse matrix. The first, the *data* array, keeps all non-zero values following the row-major order. For an unweighted graph's adjacency matrix, non-zero values are all ones, which is a common case in GNNs' workloads. The second, *col_ind* (column indices), keeps column indices of each non-zero value and uses the same order as the *data* array. The third, *row_ptr* (row pointers) array, records the accumulated number of non-zero (NNZ) values for each row.

Due to its high efficiency and simplicity, CSR has been adopted by virtually all of the platforms that support sparse operations. Further, most graph matrices come in the CSR format. Therefore, we design *CacheSample* to directly work with this format. By supporting CSR, *CacheSample* can serve as a drop-in replacement for any existing sparse system without the need to transform formats and incurring resulting large overheads.

GPU Preliminaries

We next briefly introduce the GPU hardware architecture and execution model, using NVIDIA's terminology since this work is implemented in CUDA.

Table 2.1: Statistics of the evaluated graph datasets.

DATASET	# NODE	# EDGE	AVG. DEGREE	# CLASS
PUBMED	19.7K	88.6K	4.5	3
ARXIV	169K	2.3M	13.7	40
PROTEINS	132K	79.1M	597	112
REDDIT	233K	115M	493	41

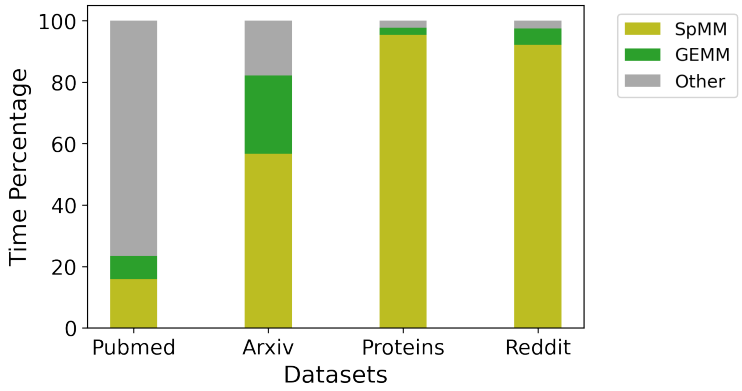


Figure 2.3: End-to-end compute time breakdown of the GCN model on different datasets.

In terms of hardware architecture, a GPU is composed of an array of streaming multiprocessors (SMs). Each SM consists of a few blocks of 32 CUDA cores, where a CUDA core is the smallest compute unit on the GPU. It also has a piece of on-chip, software-managed shared memory (L1 cache) and a pool of registers. Beyond the SM level, there is an L2 cache and a large off-chip global memory shared by all SMs.

For the execution model, a GPU kernel is typically executed by thousands of parallel threads, grouped into thread blocks; each thread block is mapped to an SM. A thread block is further divided into warps, where each warp consists 32 threads. A warp of threads is mapped to a block of 32 CUDA cores and executes the same instruction simultaneously; thus, this execution model is called Single Instruction Multiple Threads (SIMT). The warps in a thread block are streamed into the SM by a hardware scheduler. When a warp is idle, e.g., waiting for memory requests, the warp is context switched out in favor of other warps and is continued later. Therefore, given sufficient warps in a thread block, it's possible to hide the memory latency with this execution model. Furthermore, a GPU tries to combine memory requests from a warp into as few global memory transactions as possible.

Performance Bottleneck of GNNs

To illustrate the performance bottleneck of GNNs, Figure 6.3 breaks down the computation time of GCN [148], a popular GNN model, for four different graph datasets. The GCN model is implemented in DGL, which uses the cuBLAS and cuSPARSE libraries for the GEMM and SpMM operations, respectively. The experiments are conducted on an AWS p3.2xlarge instance with a NVIDIA's V100 GPU. The statistics of the evaluated datasets and the model setup are shown on Table 2.1 and Table 2.3. The detailed descriptions of the datasets and models are in Section 2.5.

As Figure 6.3 shows, despite the use of the highly optimized cuSPARSE library, the performance bottleneck of GNNs still lies in the SpMM kernel. For the smaller datasets, such as Pubmed and Arxiv, the SpMM kernel takes 16% and 56% of the end-to-end compute time, respectively. For the larger datasets, such as Proteins and Reddit, the SpMM kernel consumes an unacceptale 95.4% of the total inference time.

These dramatic results are due to GPUs being optimized for dense and regular operations. SpMM's sparse and irregular features make it difficult to achieve high throughput on a GPU. As the scale of the graphs involved in GNNs' workloads becomes progressively larger [111], the compute bottleneck on the SpMM kernel becomes increasingly more severe and hinders the development of larger, deeper GNN models.

To relieve the compute bottleneck and further accelerate GNN inference, we propose a novel SpMM kernel that we codesigned with an edge sampling mechanism to achieve an order of magnitude speedup over existing SpMM kernels.

2.3 Edge Sampling for GNN's Inference

To demonstrate how we leverage edge sampling to accelerate SpMM kernel performance, in this section we briefly introduce edge sampling techniques in the context of GNNs, analyze its impacts on inference accuracy, and examine the limitations of existing methods.

Edge Sampling

Edge sampling is a technique to include only a subset of edges of the original graph while discarding the rest. The existing methods include randomly dropping edges [225; 102] or using a learned neural network

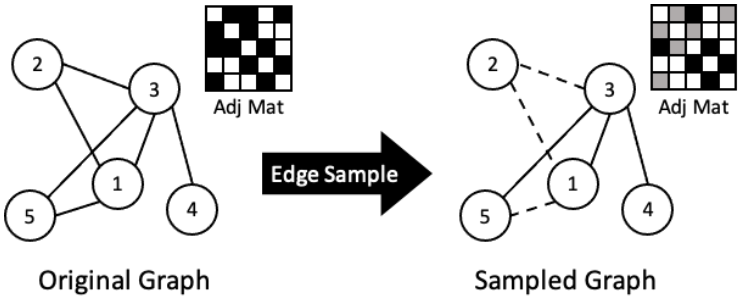


Figure 2.4: The edge sampling process. The NNZ of the adjacency matrix for the sampled graph is reduced compared to the original graph.

to determine which edges to abandon [314]. Figure 2.4 highlights the edge sampling process.

To date, edge sampling has been used to mitigate GNN's over-fitting and over-smoothing [165; 284; 150] training problems. For example, DropEdge [225] randomly samples a different subset of edges throughout training iterations with a fixed dropping rate. The results of DropEdge shows that such dynamic random edge sampling improves training accuracy for both shallow and deep GNN models.

However, rather than improving training accuracy, we for the first time use edge sampling to accelerate GNN's inference.

Inference Accuracy

Unlike previous edge sampling works for GNN training have training loops and backpropagation to adapt the loss of edges, model accuracy will likely diminish with the edge-sampled graph during inference, as some graph information is lost in the process of sampling. To understand whether edge sampling is applicable to GNN inference with an acceptable accuracy loss, we conducted an analysis of its effect on inference accuracy.

We mimiced DropEdge [225] by dropping a fixed rate of edges for a given graph, fed the sampled graph to a pre-trained GNN model to obtain test accuracy, and swept the drop rate from 20% to 80%. Figure 2.5 shows results with the GCN [148] model on four representative datasets.

On Figure 2.5, we observe that the inference accuracy of the GCN model generally declines as the drop rate increases. For small datasets such as Pubmed and Arxiv, accuracy remains close to the baseline when the edge drop rate falls below 40%. For large datasets such as Proteins and Reddit, accuracy is less sensitive

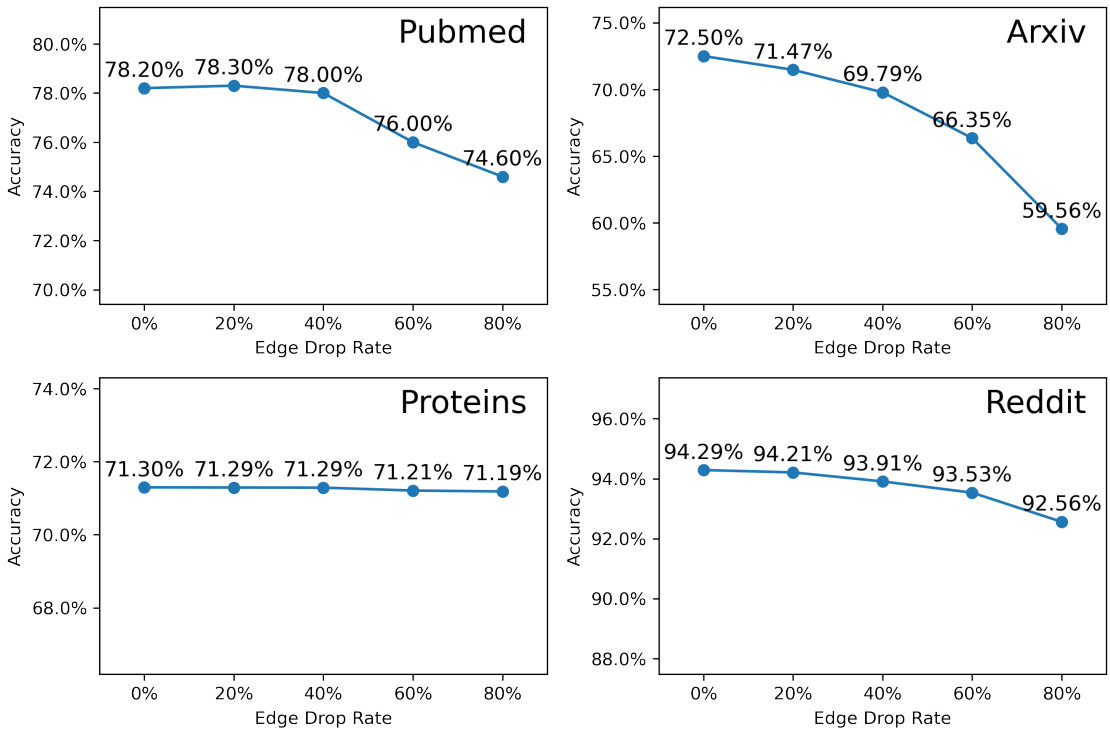


Figure 2.5: GCN's [148] inference accuracy when feeding sampled graphs with different edge dropping rates on different datasets. The dropped edges are randomly selected.

to larger drop rates compared to smaller datasets, and we observe only a small loss even when the edge drop rate reaches 80%.

This analysis shows that a pre-trained GNN model can resist the loss of edges, which provides an opportunity to trade off accuracy and latency. We leverage this fact in our design of *CacheSample*.

SpMM Time Reduction and Sampling Time

As shown in Figure 2.4, edge sampling reduces the NNZ of the sampled graph's adjacency matrix, which can diminish the runtime of the SpMM kernel since less computation is needed. However, it requires extra compute time to perform edge sampling. Therefore, an important question becomes whether edge sampling can bring a net end-to-end speedup.

To answer this question, we conducted another analysis to access how much a SpMM kernel can be accelerated using an edge-sampled graph and how much compute time it takes to perform edge sampling. We follow the same settings as Section 2.2 and use cuSPARSE's SpMM kernel to conduct this analysis.

We continued to mimic [225] for sampling methodology. We implemented edge sampling in NumPy, the standard way used in [225], and randomly dropped 20% of the edges for each graph. We chose a 20% drop rate here because the tested model maintained near baseline accuracy with this rate, as Figure 2.5 shows. Table 2.2 shows the comparison.

As shown in Table 2.2, the SpMM kernel time is generally faster with the sampled graph, with a speedup range from 1.05x to 1.16x. However, when we consider edge sampling time, the net inference time becomes significantly slower due to the huge number of computing cycles spent on the sampling itself. Using the *Reddit* dataset as an example, although the SpMM kernel runs 1.12x faster with the sampled graph, edge sampling consumes 6814 ms to run, which results in a 63.8x slower end-to-end inference time. Here, the used random edge sampling strategy is considered lightweight. If we use a more complicated sampling method, such as a neural network model [314], the gap between the sampling time and the SpMM kernel time reduction would be even larger. 6.10 Note in Table 2.2, we breakdown SpMM kernel time, edge sampling time, and the overall inference slowdown after combining the new SpMM kernel time and sampling overhead, for different datasets. The edge sampling is implemented in NumPy, and the drop rate is 20%. The time is measured in milliseconds.

As a result, applying existing edge sampling methods on existing SpMM kernels cannot yield a net acceleration to GNN inference, especially with the condition that the graph would be processed only once. We therefore, propose *CacheSample*, a novel cache-first edge sampling mechanism and SpMM kernel codesign that efficiently leverages edge sampling to significantly accelerate GNN inference.

Table 2.2: GCN's [148] inference time breakdown for the original and edge-sampled graphs.

Dataset	SpMM Time		SpMM	Sample	Overall
	Ori.	Sampled	Speedup	Time	Slowdown
Pubmed	0.68	0.65	1.05x	4.24	2.1x
Arxiv	14	13.1	1.07x	108.5	5.2x
Proteins	173.2	149.4	1.16x	13528	**75.7x**
Reddit	99.86	89.1	1.12x	6814	**63.8x**

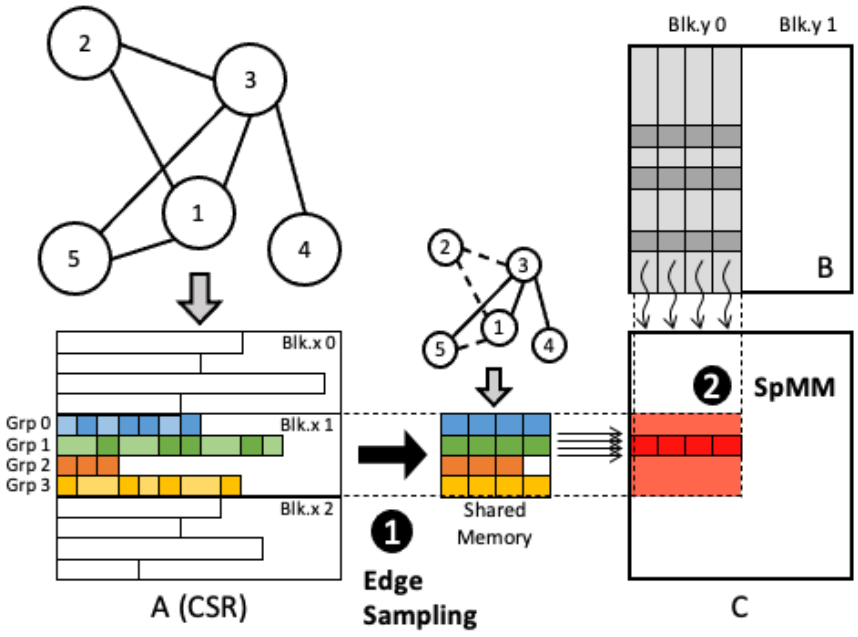


Figure 2.6: Overview of the *CacheSample* kernel.

2.4 CacheSample Kernel Design

We now provide a detailed description of *CacheSample* design, with the goal of leveraging edge sampling to speedup SpMM, the compute bottleneck of GNN inference, while minimizing edge sampling overhead.

Overview

Figure 2.6 presents an overview of *CacheSample*'s kernel architecture. Input to the kernel is a sparse matrix A in CSR format, which represent the graph structure. A dense matrix B contains the nodes' features. The output of the kernel is a dense matrix C, which contains the output nodes features. Dense matrices B and C are both in row-major format.

The main computations in *CacheSample* include *edge sampling*, which shrinks the sparse matrix A to fit on the GPU's shared memory, and a *SpMM operation*, compute the output features C based on the sampled graph. Algorithm depicted in 2.7 shows the pseudo code of *CacheSample*. We describe each design step in the following subsections.

```
__global__ void CacheSample-SpMM-Kernel(CsrMtx A, Mtx B
    , Mtx C, float sample_rate)
{
  extern __shared__ sh_mem[];
  int tid = threadIdx.x;
  int row, col, sm_off = get_offsets();
  float acc = 0.0;

  int lb = A.rowptr[row];
  int nnz = A.rowptr[row+1] - lb;
  //Calculate sampled nnz.
  int s_nnz = ceil(nnz*sample_rate);

  for(off=0; off<s_nnz; off+=blkDim.x){
    ptr = lb + off + tid;
    //A sampling hash function is called
    //to calculate the sampling index.
    smpl_ptr = lb + smpl_hash_func(ptr);
    //Load selected values to shared mem.
    shm[sm_off+tid] = A.colind[smpl_ptr];
    __syncthreads();

    for(i=0; i<blkDim.x; i++) {
      b_row = sh_mem[i];
      acc += B[b_row, col];
    }
    __syncthreads();
  }
  //Divide acc by s_nnz for the norm.
  C[row, col] = acc/s_nnz;
}
```

Figure 2.7: CUDA pseudo code of *CacheSample*.

Stage 1: Perform Cache-First Edge Sampling

To perform edge sampling in *CacheSample*, we first set a desired shared memory width, S. Then, the size of the shared memory each thread block can request is determined by $(\#\ of\ thread\ groups \times S)$. Note that this size is limited by the shared memory capacity of the GPU. The value S also serves as the sampling target for *CacheSample* to downsize the graph data. As shown in Figure 2.6, for each row of A, at most S non-zero values are fetched to shared memory. If S exceeds the NNZ of that row, then the whole row is

fetched to shared memory. In this way, the edge-sampled graph can be fitted to the shared memory within the scope of each thread block. We refer to this sampling mechanism as "cache-first edge sampling".

The cache-first edge sampling mechanism offers two main benefits: The first is the reduced computation time required by the SpMM operation in the next stage; the second is improved cache locality from keeping all the needed graph data on shared memory. This is not feasible for existing SpMM kernel implementations because they must ensure the completeness of the SpMM operation, and GPU's shared memory is usually not large enough to hold all graph data without sampling. However, our proposed cache-first edge sampling scheme exploits the fact that a GNN model can tolerate the loss of edges, and discards redundant edges for better cache locality and faster runtime.

To perform the proposed cache-first edge sampling efficiently, we assign a group of threads to sample a row of A in *CacheSample*. Here, each thread uses one of the lightweight sampling strategies (Section 2.4) to compute the sampling indices and loads the selected A's data and column indices into shared memory. We arrange 32 to 128 threads in each thread group. Depending on the size of each thread group, we arrange 4 to 8 thread groups in a thread block, forming a fixed 512 threads in each thread block. As the thread groups in the same thread block share the same shared memory space, each shared memory block holds 4 to 8 rows of A's sampled values. In the *SpMM* operation, the next stage of *CacheSample*, all values of A come from shared memory, and the kernel computes the SpMM only for the sampled values.

As noted in Section 2.3, edge sampling can cause substantial compute overhead that retards overall runtime. *CacheSample* therefore, parallelizes the sampling task into thousands of threads and significantly accelerates it. Such in-kernel sampling also eliminates the need to preprocess graph data on the CPU and lets user-level code remain untouched when using the *CacheSample* as the back-end SpMM kernel. In this way, although edge sampling must be re-computed each time the kernel is launched, our evaluation results (Section 4.5) show that *CacheSample* can still yield massive acceleration, which indicates that our parallelized edge sampling design is very efficient.

Stage 2: Compute SpMM Based on Sampled Results

Once edge sampling is done and shared memory is loaded, the second stage of *CacheSample* begins, which aims to compute the SpMM based on sampled results. Since *CacheSample* computes the multiplications

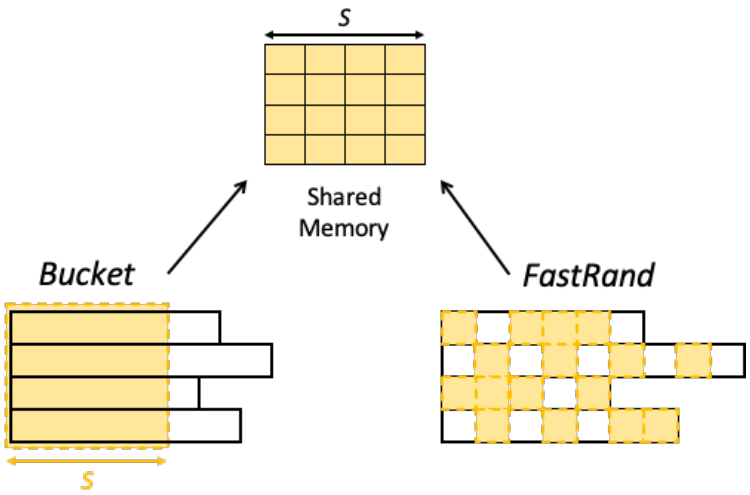


Figure 2.8: Overview of *CacheSample*'s *Bucket* and *FastRand* edge sampling strategies.

only for the sampled A's values, it approximates the SpMM.

Operations in this stage are straightforward, including reading A's data and column indices from shared memory, loading the corresponding B's data from global memory, calculating the product, and looping to repeat these process and accumulate multiplication results. Algorithm in Figure 2.7, lines 12 to 16, shows the pseudo code for this stage.

To perform these operations, *CacheSample* assigns to each thread the computation task for an element of C. This thread management resembles SpMM kernel optimizations in [285] and [114]. However, the major difference is that all needed A's values in *CacheSample* are loaded to shared memory prior to this stage. Therefore, the clean and dedicated loop here needs no synchronization or extra data loading, which means that *CacheSample* has fewer branch divergences and can achieve a higher ILP.

After the sampled SpMM is computed, each thread stores the multiplication result of an element of C from a local register to global memory, and *CacheSample* kernel operations are finished. The whole *CacheSample* process is implemented in roughly 50 lines of CUDA code.

Sampling Strategy

Section 2.4 stated that within the cache-first edge sampling mechanism, *CacheSample* could use different lightweight sampling strategies for each thread to compute exact sampling indices (see line 8 in Figure 2.7).

We now specify the design of the strategies.

A primary design consideration for *CacheSample*'s sampling is efficiency. Although we use parallel threads to accelerate edge sampling (Section 2.4), computation of the sampling itself must still be lightweight to maximize *CacheSample* performance. Based on this consideration, we designed two lightweight edge sampling strategies, *Bucket* and *FastRand*, with different speedup and accuracy impacts. Figure 2.8 highlights each strategy, which we describe below.

Bucket. Given the target shared memory width, S, the *Bucket* edge sampling strategy picks the first S non-zero data and column indices for each row of A. We call this strategy *Bucket* because it resembles the way a bucket carries items to fit a specific volume. *Bucket* is very fast since it needs only a boundary check to determine which indices to keep. However, we found that it could significantly sacrifice accuracy, especially on large graphs because it maintains a fixed range of edges for each node. See Section 4.5 for more information on *Bucket*'s accuracy.

FastRand. As noted in Section 2.3, GNN's inference accuracy can be well preserved using a random edge sampling scheme, especially for large graphs with numerous nodes and edges. Based on this observation, we designed the *FastRand* edge sampling strategy to perfor a pseudo-random sampling for *CacheSample*. To attain both good randomness and fast speed, *FastRand* uses an efficient hash function to calculate sample indices; Equation 2.2 shows the hash function.

As Equation 2.2 shows, to calculate the sample index ($sample\_idx$) for a given shared memory location ($shmem\_idx$), *FastRand* multiplies $shmem\_idx$ with a prime number P' and takes a modulo over the row_nnz, which is the NNZ of the targeted row. Then, *CacheSample* uses $sample\_idx$ as the index value to fetch A's data and column indices and store them to the shared memory at position $shmem\_idx$ (see line 9 in Figure 2.7). Note that in order to sample across the full range of non-zero values for each row of A, the prime number P' cannot be too small. In practice, we chose $P' = 577$ when implementing the *CacheSample* kernel. In this way, *FastRand* fulfills the pseudo-random edge sampling using only one multiplication and one modulo operation, which are sufficiently lightweight for each GPU thread to finish within a short latency. With this type of sampling, we expect the accuracy loss of *FastRand* to be less than *Bucket*. However, we expect *Bucket* to be faster since it needs less computation. See Section 4.5 for our detailed evaluation.

$$sample\_idx = (shmem\_idx \times P') \mod row\_nnz \tag{2.2}$$

Kernel Optimizations

In addition to cache-first edge sampling and the sampled SpMM, *CacheSample* uses other kernel optimizations to achieve optimal performance. We now describe these.

Thread management. Per Section 2.4, we use up to 128 threads to handle the sampling and data loading for each row of A in *CacheSample*. This is unique since a common thread management approach adopted by many existing SpMM kernel optimizations [285; 114; 80; 106] uses a warp (32) of threads to load a row of A's values to registers or shared memory. Previous works use this thread setting to provide an acceptable balance of parallelism and number of tasks per thread for SpMM workloads. However, unlike the previous SpMM kernels, which usually load only a partial row at one time, *CacheSample* must complete the sampling and data loading for an entire row in one stage. Because the targeted shared memory width, S, could be large (such as 512), using only 32 threads for a row requires multiple iterations to completely load the shared memory, which may slow down both sampling and data loading. Therefore, we use up to 128 threads to sample and load a row of A in *CacheSample* to improve performance.

Coalesced memory access. Coalesced memory access is an important optimization to reduce memory loading latency on GPU. In the second stage of *CacheSample*, we allowed a group of threads to load B's data in the same row but in neighboring columns at the same time. This coalesces memory access because the data requests from multiple threads can be combined and fulfilled by fewer global memory transactions. As a result, data loading for B in *CacheSample* is very efficient. This technique resembles those introduced in [285; 114].

Load balancing. It is natural for some nodes to have many more connections than other nodes in the graph, leading to a large variations of the NNZ from row to row, as Figure 2.6 shows. The irregular distribution can easily result in an unbalanced load for the SpMM kernel and degrade kernel performance, especially under a row-split work distribution. In *CacheSample*, the NNZ variation of each row is limited to the targeted shared memory width, S, after edge sampling. In this way, the work assigned to each thread is much better balanced than running on an unsampled graph, helping *CacheSample* achieve higher utilization

Table 2.3: Model parameters and baseline accuracy we achieve on each dataset for GCN and GraphSage.

MODEL	DATASET	#LAYER	#HIDDEN	TEST ACC
GCN	PUBMED	2	32	77.70%
	ARXIV	3	256	72.50%
	PROTEINS	3	256	71.30%
	REDDIT	2	128	94.29%
GRAPHSAGE	PUBMED	2	32	78.60%
	ARXIV	3	256	72.95%
	PROTEINS	3	256	77.28%
	REDDIT	2	128	96.22%

and improved performance.

2.5 Evaluation

We next describe system integration, experiment setup, and the comprehensive experiments we conducted to verify the effectiveness of *CacheSample*.

Integration with DGL

To maximize the applicability of *CacheSample* to GNN models, we integrated *CacheSample* into one of the most popular GNN frameworks, DGL [264], which provides a set of high-level Python APIs. The backend of DGL calls to the cuSPARSE's *cusparseSpMM()* kernel to perform the SpMM operation.

Since both *cusparseSpMM()* and *CacheSample* work on the standard CSR format, we simply replaced the *cusparseSpMM()* with our *CacheSample* kernel in the DGL backend for the integration. Because *CacheSample* needs no preprocessing on the graph data, the DGL user-level code remains unmodified during this process.

Experiment Setup

We now describe the setup we used to conduct our experiments.

Evaluated GNN models and datasets. We chose two classic GNN models, *GCN* [148] and *GraphSage* [96], and four popular node classification datasets to evaluate our *CacheSample* kernel. The four graph datasets were *Pubmed, Arxiv, Proteins and Reddit*. *Pubmed* is a citation network of medical academic

papers. *Arxiv* and *Proteins* were taken from the *Open Graph Benchmark* [111]. *Reddit* is a social network dataset collected from a popular online forum. These four datasets are of different scales in terms of the number of nodes and edge. For *GraphSage*, we used *mean* feature aggregation. Hence, experiment results using them can help us appreciate *CacheSample*'s performance for different graph scales. Table 2.3 lists the model parameters for each datasets, and Table 2.1 shows the dataset statistics.

Model setup. We first trained the GNN models on each dataset for ten times with the original DGL framework, and kept the ones with the highest test accuracy. Table 2.3 shows the best test accuracy we achieved for each model and dataset. After the models were trained, we used our modified DGL which calls to *CacheSample* and run the experiments on inference.

SpMM baselines. Our experiments compare the performance of *CacheSample* to the following CSR SpMM baselines.

- *cuSPARSE*: The default implementation in DGL. The main kernel is *cusparseSpMM()*, and the compute algorithm is set to *CUSPARSE_SPMM_CSR_ALG2* [201], as it achieves the best performance in our setup.
- *Merge-SpMM*: Proposed in [285]. It has a suite of different row-split and merge-based algorithms aimed at improving load balance. The code is available at [286]. To test this kernel, we ran it with all proposed algorithms and picked the highest performer in each case.
- *GE-SpMM*: Proposed in [114]. It is based on the row-split design of [285] but has an improved shared memory usage. We also ran all kernel setup provided in their open-source package [113] and picked the best performer.

Hardware environment. Our experiments were conducted on the following hardware environment.

- *AWS EC2 p3.2xlarge instance*. GPU model: NVIDIA V100 with 16GB global memory. CUDA version: 11.0. CPU: Intel Xeon E5-2686 v4 (8 virtual cores) with 61GB main memory.

Sampling Rate

As noted in Section 2.3, sampling rate is an important factor affecting inference accuracy. In *CacheSample*, the target shared memory width, S, determines the sampling rate. Since each graph has a different

Table 2.4: Sampling rates under different S values for different datasets in *CacheSample*.

DATASET	S=16	S=32	S=64	S=128	S=256	S=512
PUBMED	84.9%	95.8%	99.3%	99.9%	100.0%	100.0%
ARXIV	83.7%	96.8%	99.3%	99.8%	99.9%	100.0%
PROTEINS	2.6%	5.1%	9.9%	18.9%	34.3%	56.7%
REDDIT	3.1%	6.0%	11.3%	20.5%	34.8%	53.9%

distribution of nodes and edges, its sampling rate under each S value is different in *CacheSample*.

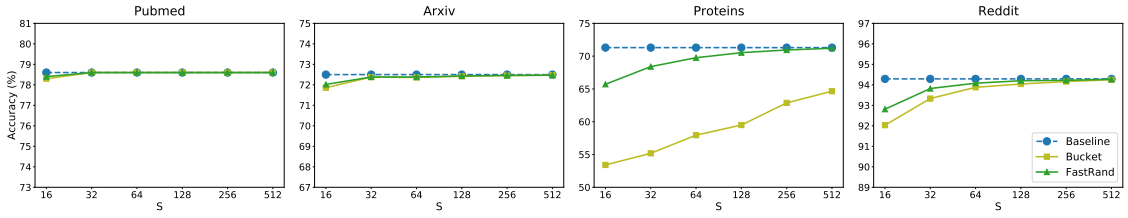
To obtain the sampling rates, we simulated the cache-first edge sampling mechanism in Python and calculated the equivalent sampling rates for each S. Table 2.4 shows the sampling rates for each dataset. Note that *Bucket* and *FastRand* have the same sampling rate for a given S since they fetch the same amount of A's values to shared memory.

Per Table 2.4, for small graphs such as *Pubmed* and *Arxiv*, a small S (16) already holds over 80% percent of the edges, and a larger S (above 128) actually holds over 99% of the edges. However, for large graphs such as *Proteins* and *Reddit*, the sampling rates are much lower compared to the small graphs under a fixed S. For example, when $S = 256$, the sampling rates are only around 34% for *Proteins* and *Reddit* but are 100% and 99.9% for *Pubmed* and *Arxiv*. Based on our sampling rate analysis, we expect that *CacheSample* will experience little to no accuracy loss for small graphs but a more sizable accuracy loss for large graphs, especially when S is small.

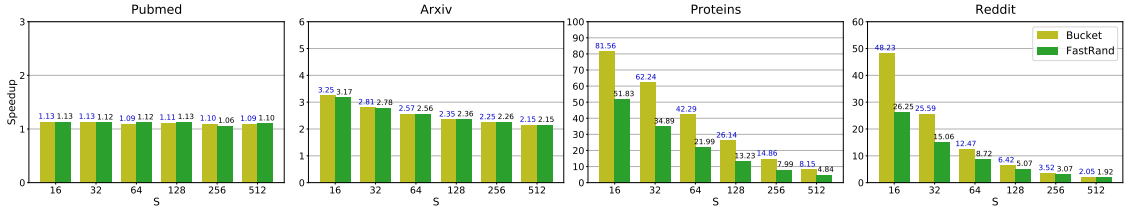
Performance Evaluation of *CacheSample*

We now present the inference accuracy and speedup of using *CacheSample* compared to cuSPARSE SpMM kernel.

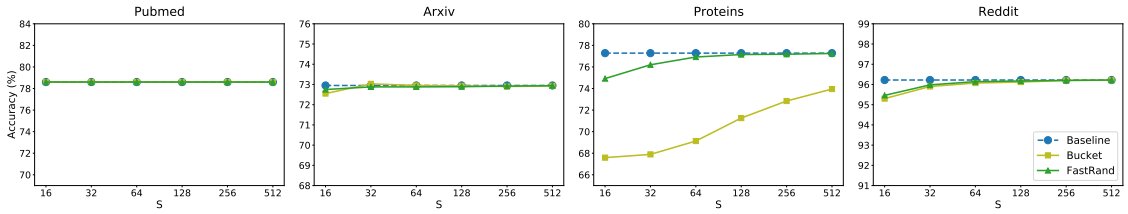
Inference accuracy and speedup under different S values. To conduct these experiments, we first ran *GCN* and *GraphSage* with the unmodified DGL, which uses cuSPARSE's SpMM kernel, to assess baseline inference accuracy and SpMM kernel time for each dataset. Here, we used *PyTorch Profiler* [208] to measure SpMM kernel time. Then, we ran the inference with our modified DGL that uses the *CacheSample* kernel, swept S from 16 to 512, and compared the test accuracy and SpMM performance. For SpMM performance, we compared the total SpMM kernel time of a complete inference and we took the average after running the inference 10 times. Figure 2.9 shows experimental results on the accuracy and SpMM speedup against



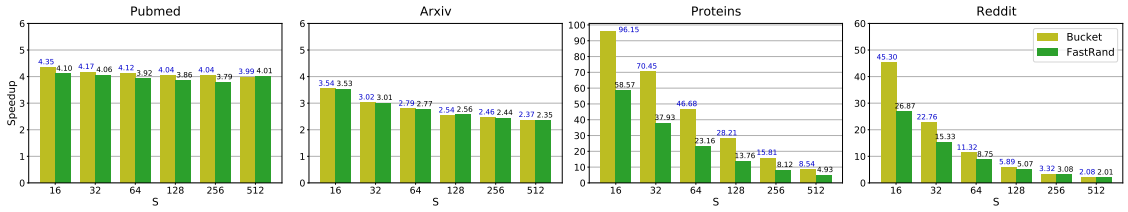
(a) GCN inference accuracy.



(b) SpMM kernel speedup for *GCN*.



(c) GraphSage inference accuracy.



(d) SpMM kernel speedup for *GraphSage*.

Figure 2.9: *CacheSample*'s accuracy and SpMM speedup compared to *cuSPARSE* under different S values.

cuSPARSE.

For accuracy on the small datasets, *Pubmed* and *Arxiv*, *CacheSample* generally had little to no accuracy loss across different S values for both *Bucket* and *FastRand*. For *GraphSage* on *Pubmed*, *CacheSample* had no accuracy loss even when S was as small as 16. This echoes the discussion in Section 2.5 since *CacheSample* can hold most edges with a small S for small graphs.

For SpMM speedup on small datasets, *CacheSample* achieved 1.1x to 4.35x speedup compared to *cuSPARSE*. For *Pubmed*, *CacheSample* achieved roughly 1.1x and 4x speedup for *GCN* and *GraphSage* respectively, across all S values. We saw a smaller speedup for *GCN* because the baseline SpMM kernel time was

already short (0.64ms). The steady speedup across different S values occurred because the sampling rates were at a similar level (Table 2.4). On *Arxiv*, *CacheSample*'s speedup slightly decreased when S went up. This is due to increased sampling overhead with a larger S, which diminished *CacheSample*'s performance. However, even with a large $S = 512$, *CacheSample* still achieved at least a 2.15x and 2.35x speedup for *GCN* and *GraphSage* respectively, on *Arxiv*.

Regarding accuracy on the datasets *Proteins* and *Reddit*, *CacheSample* experienced a larger loss of accuracy but still achieved near-baseline accuracy with a larger S value. Also, on these larger graphs, *FastRand* generally exhibited better accuracy than *Bucket*. For example, on *Proteins*, *Bucket* had a large accuracy loss even when $S = 512$. In our experiments, we found that S must be as large as 1792 to achieve less than a 1% accuracy loss when using *Bucket* on *Proteins*. However, for *FastRand*, it achieved near-baseline accuracy with $S = 128$ for *GCN* and $S = 64$ for *GraphSage*. *FastRand* also had much lower drop in accuracy than *Bucket* when S was smaller; this is because the graph degree of *Proteins* is large, and the *Bucket*'s method for picking the first continuous edges makes it lose many graph features. On the other hand, *FastRand* randomly sampled all edges, which preserved graph structure and yield much improved accuracy. For *Reddit*, we observed an accuracy trend similar to *Proteins*. However, since *Reddit* is an easier dataset that both *GCN* and *GraphSage* achieve high baseline accuracy (94.3% and 96.2%), *Bucket*'s accuracy loss on *Reddit* is not as pronounced as *Proteins*.

Regarding SpMM speedup on large datasets, *CacheSample* achieved a tremendous speedup agains *cuSPARSE* for both *Bucket* and *FastRand*, due to the low effective sample rates. On *Proteins*, although *CacheSample* with *Bucket* achieved up to 81.6x and 96.15x speedup for *GCN* and *GraphSage* respectively, it was less meaningful since the accuracy loss was large. However, with *FastRand*, *CacheSample* achieved a 13x speedup with only a negligible accuracy loss when $S = 128$ for both *GCN* and *GraphSage* on *Proteins*. On *Reddit*, both *Bucket* and *FastRand* achieved a meaningful speedup with only a small accuracy loss. When $S = 64$, *Bucket* achieved 12.5x and 11.3x and *FastRand* achieved 8.72x and 8.75x for *GCN* and *GraphSage* respectively, where both strategies showed negligible accuracy loss, a significant benefit due to *CacheSample*.

Speedup with less than 1% accuracy loss. Here, we set a 1% accuracy loss constraint and report the best *CacheSample* SpMM speedup, the corresponding S value, and inference accuracy for each sampling

Table 2.5: *CacheSample*'s best SpMM speedup against cuSPARSE with less than 1% accuracy loss.

Model	Dataset	S		Accuracy			Accuracy Loss		SpMM Time (ms)			SpMM Speedup	
		Bucket	*FastRand*	Baseline	*Bucket*	*FastRand*	*Bucket*	*FastRand*	Baseline	*Bucket*	*FastRand*	*Bucket*	*FastRand*
GCN	Pubmed	16	16	77.70%	77.30%	77.50%	0.40%	0.20%	0.64	0.56	0.56	1.13	1.13
	Arxiv	16	16	72.50%	71.86%	72.02%	0.64%	0.48%	12.39	3.81	3.91	3.25	3.17
	Proteins	1792	128	71.30%	70.72%	70.53%	0.58%	0.77%	172.92	71.93	13.07	2.40	**13.23**
	Reddit	32	32	94.29%	93.33%	93.82%	0.96%	0.47%	99.74	3.90	6.62	**25.59**	**15.06**
GraphSage	Pubmed	16	16	78.60%	78.60%	78.60%	**0.00%**	**0.00%**	3.56	0.82	0.87	4.35	4.10
	Arxiv	16	16	72.95%	72.55%	72.75%	0.40%	0.20%	18.83	5.32	5.33	3.54	3.53
	Proteins	2304	64	77.28%	76.47%	76.92%	0.81%	0.36%	237.50	140.02	10.25	1.70	**23.16**
	Reddit	16	16	96.22%	95.30%	95.46%	0.92%	0.76%	449.69	9.93	16.74	**45.30**	**26.87**

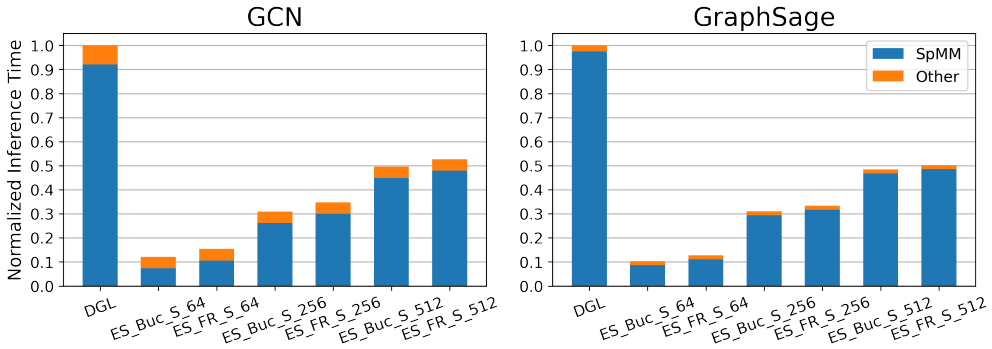


Figure 2.10: End-to-end inference time breakdown of *GCN* and *GraphSage* on *Reddit*.

strategy in Table 2.5; we still compare to cuSPARSE for speedup. We set this constraint because we assume that a 1% accuracy loss is an acceptable cost to trade off for a faster GNN inference time. In other words, if users can accept a minimally higher accuracy loss, then *CacheSample* can deliver a more significant speedup.

As Table 2.5 shows, for most of cases, $S = 16$ was sufficient for *CacheSample* to meet this constraint. For the small datasets, *CacheSample* achieved up to 3.25x for *GCN* and 4.35x for *GraphSage*, both using *Bucket*. For the large datasets, *CacheSample* generally offered an over 10x speedup against the strong cuSPARSE baseline, which is significant. The best SpMM speedup *CacheSample* achieved for *Proteins* and *Reddit* under this constraint were 23.16x with *FastRand* and 45.3x with *Bucket*. The only exception was that *CacheSample* needed a large S to meet this constraint when using *Bucket* on *Proteins*. However, *CacheSample* with *Bucket* still achieved a profitable 2.4x and 1.7x speedup for *GCN* and *GraphSage* respectively, with $S = 1792$ and $S = 2304$.

End-to-end inference time. Here we compare the end-to-end inference time of using *CacheSample* to the original DGL's results, which uses *cuSPARSE* for its SpMM operation. Figure 2.10 shows the end-

to-end inference time breakdown of running *GCN* and *GraphSage* on *Reddit* with the original DGL and with the *CacheSample* in different S values. As noted in Section 2.2: SpMM consumes a large portion of GNN inference time, and noted in Section 2.5: *CacheSample* significantly accelerates the SpMM operation, the end-to-end inference time was considerably reduced when using *CacheSample* as Figure 2.10 shows. *CacheSample* achieves about 10x faster end-to-end inference time for both models compared to the original DGL, when using a small S (64). When using a large S (512), *CacheSample* still achieves roughly 2x speedup.

SpMM Performance Compared to Open-Source Baselines

Besides *cuSPARSE*, we compared *CacheSample*'s SpMM performance to *Merge-SpMM*'s and *GE-SpMM*'s.

To conduct these evaluations, we did not modify the open-source packages of *Merge-SpMM* and *GE-SpMM*, and fed the SpMM kernels with the adjacency matrix extracted from DGL. This methodology prevented us from changing kernel behavior while re-implementing the kernels into DGL's backend. To get a similar comparison to the one we achieved in Section 2.5, we separately performed multiple SpMM operations that were needed for a complete inference, and combined the kernel times to get total SpMM time. We also did the same operations on *cuSPARSE* and *CacheSample* for a fair evaluation. The results we show are the average of 200 runs.

Figure 2.11 shows the SpMM time comparison of running GCN inference on each dataset. For *CacheSample*, we show the results of using $S = 32$ and $S = 128$ since these settings were generally optimal for *CacheSample* to achieve near-baseline accuracy for small and large graphs. As Figure 2.11 shows, *Merge-SpMM* and *GE-SpMM* only achieved a slightly faster kernel time than *cuSPARSE* on large graphs. However, *CacheSample* achieved the fastest SpMM kernel time across all datasets and excelled on the large graphs, consuming only 5% to 20% of *cuSPARSE*'s kernel time.

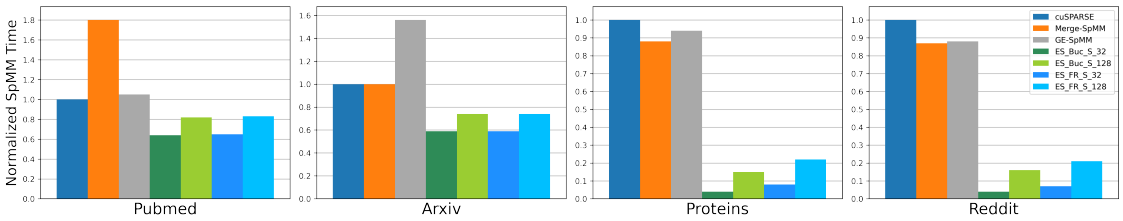


Figure 2.11: SpMM time of GCN's inference using different SpMM kernels.

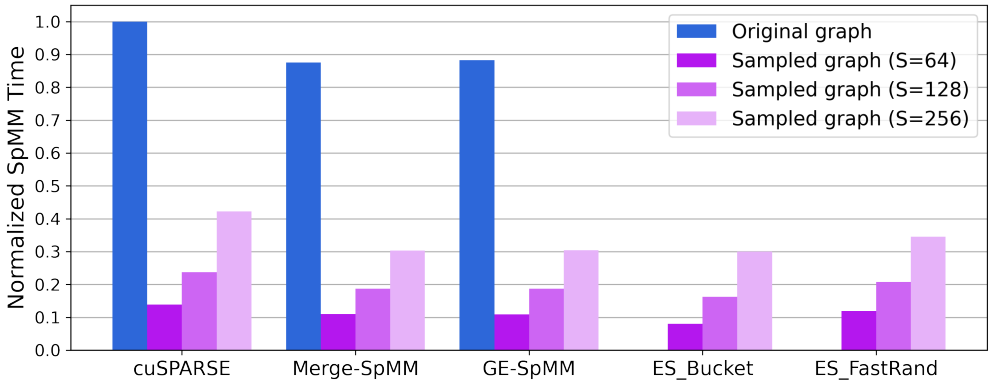


Figure 2.12: Comparison of SpMM time of GCN's inference on *Reddit* with the baseline SPMM kernels. The baseline kernels are evaluated with pre-sampled graphs.

Feeding Pre-Sampled Graphs to Baseline SpMM Kernels

To evaluate the source of *CacheSample*'s benefits, we reran an experiment (Section 2.3) that fed pre-sampled graphs to the SpMM kernels. Unlike Section 2.3 experiment doing a general random sampling, here we simulated the *FastRand* of *CacheSample* in Python for pre-sampling. Then, we fed the pre-sampled graph to *cuSPARSE*, *Merge-SpMM* and *GE-SpMM* and measured the SpMM kernel time. For *CacheSample*, we used the normal graph but set S to the corresponding value. Again, we summed SpMM kernel times for a complete inference and took the average of 200 runs for comparison.

Figure 2.12 shows the SpMM time comparison of running *GCN* on original and pre-sampled *Reddit* with $S = 128$ and 256. As the figure shows, with pre-sampled graphs, all baseline kernels had a dramatically faster kernel time. For *Merge-SpMM* and *GE-SpMM*, they actually achieved a similar to slightly better performance than *CacheSample* with pre-sampled graphs. *cuSPARSE* achieved a slightly slower SpMM time than *CacheSample*, but the results were close. These findings show that the major benefits of *CacheSample* come from the computation reduction enabled by edge sampling. However, additional optimizations (Section 2.4) helped *CacheSample* achieve a competitive performance with other highly optimized SpMM kernels when the input graph was already edge-sampled. Note that *CacheSample* needed to perform edge sampling in the kernel, which took extra overhead that other SpMM kernels did not have. *CacheSample*'s kernel time can be shortened if we remove time spent on sampling. We didn't show the results here because such in-kernel time breakdown is difficult to measure.

2.6 Discussion and Future Work

Comparison to DropEdge

DropEdge [225] uses a similar edge sampling scheme (random sampling) as we use in this work. However, DropEdge targets to improve the accuracy of training, and we focus on improving the speed of inference with minimal accuracy loss. Also, DropEdge performs the sampling in Python and they need to re-preform the sampling throughout every training iteration. Therefore, the sampling methodology in DropEdge only makes the runtime longer. In contrast, for CacheSample, we perform the sampling in the CUDA kernel with two novel efficient sampling strategies, so that we can use the edge sampling for acceleration.

Normalization of GNN

A GNN using sum aggregation, generally it normalizes node features by dividing them with the degree of each node. Practically, a GNN framework like DGL uses a separate operator to handle such normalization. However, this does not work for *CacheSample* since it reduces the number of edges internally in the SpMM kernel without modifying the graph managed by the framework. To make the normalization work, we must remove normalization from the DGL and compute it in the *CacheSample* kernel. Although gaining overhead to compute the normalization, *CacheSample* still achieves a massive speedup and considerable end-to-end acceleration for GNN inference, as Section 4.5 shows.

Accelerate GNN training

Although we focus on accelerating GNN inference in this work, *CacheSample* can be used to accelerate GNN training, which also involves many SpMM operations. However, with our current designs, *CacheSample* would experience lower accuracy when applied to GNN training. Unlike the inference scheme, users are willing to pay more compute time to improve accuracy during training. Therefore, applying *CacheSample* to accelerate GNN training does not make sense.

CacheSample reduces training accuracy because it samples the same subset of edges each time for a given graph, sampling strategy (*Bucket* or *FastRand*) and S value. As DropEdge [225] suggests, training accuracy can be maintained or improved by taking a different subset of edges throughout training iterations.

We expect *CacheSample* could accelerate GNN training without losing accuracy if we would efficiently implement a dynamic random sampling into the *CacheSample* kernel. We leave this to future work.

Edge Sampling vs. Node Sampling

Besides the edge sampling used in this work, there is another kind of sampling, *node sampling*, that is used in GNNs. Here we briefly introduce the node sampling, clarify the differences between edge sampling, and discuss *CacheSample*'s applicability to it.

Node sampling is first proposed by [96]. Unlike edge sampling which only takes subsets of edges, node sampling not only takes subsets of nodes but also discards unrelated edges for each subset. The main usage of existing node sampling methods [96; 45; 54; 304; 115] is to extract subgraphs and form the mini-batch training scheme for GNNs, allowing training of very large graphs that cannot be fitted into a single GPU's global memory.

Although the compute time of each node-sampled graph is much shorter because fewer nodes and edges are presented, it requires multiple iterations to run through the original graph. Therefore, the overall runtime to train or infer for a full graph is increased with node sampling since iterations over different subgraphs incur extra overhead [96; 45]; sometimes, node sampling could even cause high traffics between CPU's and GPU's memory [170; 192], which would significantly slowdown both training and inference. We expect *CacheSample* to further improve the inference speed for node-sampled graphs by applying edge sampling on top of it. However, the bottleneck will likely be the looping overhead or memory bandwidth contention and thus we expect limited speedup. We leave the exploration of this direction as future work.

2.7 Summary

This work proposed *CacheSample*, a cache-first edge sampling mechanism and SpMM kernel codesign. The *CacheSample* design leverages the fact that a GNN model can tolerate the loss of edges without losing much accuracy for inference. By strategically and efficiently sampling the edges and fitting all sampled edges on GPU's shared memory, *CacheSample* can significantly accelerate the SpMM operation with minimum accuracy loss compared to other state-of-the-art SpMM kernels. Our experimental results on representative GNN models and datasets show that *CacheSample* outperforms *cuSPARSE* by up to 4.35x with no accuracy

loss and by 45.3x with less than a 1% accuracy loss. To the best of our knowledge, ours is the first work to utilize edge sampling to accelerate GNN inference.

Remarks on Author Contributions

My major contributions are as follow:

- Led the project development and paper writing.

Chapter 3

Improving Neural Rendering Efficiency with Random Patch Sampling

Remarks on Chapter Material

The content of this chapter is adapted from the following paper. The work is done during my internship with Apple. I sincerely thank Anish Prabhu for the discussions and feedback during the early stage of the project.

- Chien-Yu Lin, Qichen Fu, Thomas Merth, Karren Yang, Anurag Ranjan, "FastSR-NeRF: Improving NeRF Efficiency on Consumer Devices with A Simple Super-Resolution Pipeline", In Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV), honored as Oral (Top 2.6%), 2024.

3.1 Introduction

Neural Radiance Field (NeRF) models [190] have become integral to many computer vision and computer graphics tasks, such as novel view synthesis [190; 30; 173], surface reconstruction [265; 290], camera pose estimation [293; 269] and 3D image generation [41; 214; 168]. Since the original NeRF model cannot render images efficiently, a large body of research [224; 173; 78; 44; 244; 194; 278] has been dedicated to address the rendering efficiency. Many of these works achieve impressive gains by decomposing and representing the 3D neural radiance field with explicit features [105; 78; 173; 50; 244; 44; 194]. However, these methods

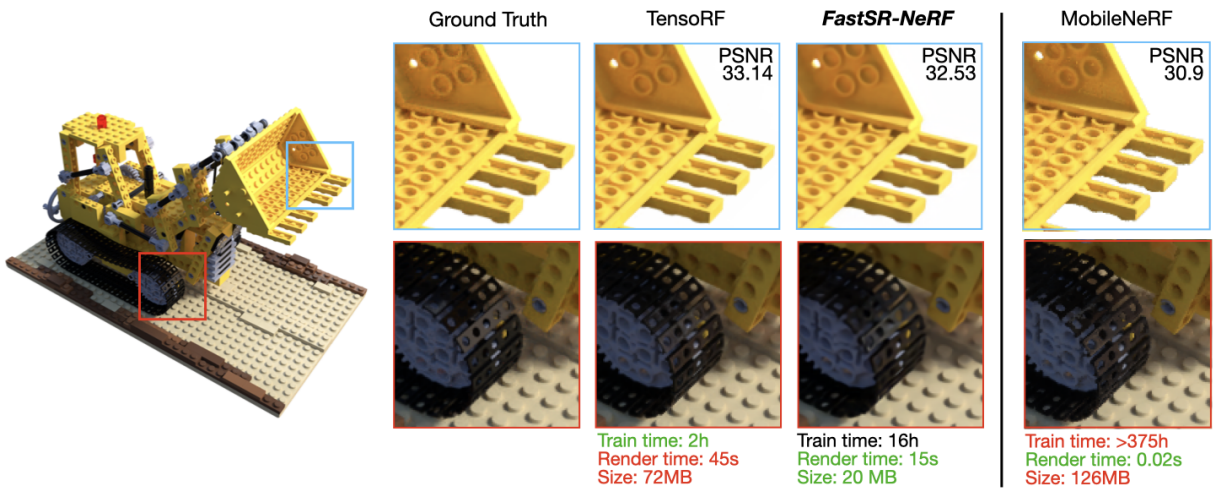


Figure 3.1: Comparison of TensoRF, FastSR-NeRF (ours), and MobileNeRF [50] on a consumer-grade MacBook Air M2 laptop.

often require extended training times and/or specialized architectures and kernel support on high-end GPUs. For example, MobileNeRF is capable of fast rendering on mobile devices [50], but uses 8 server-class GPUs for training [7], which translates to over 15 days (>375h) on a consumer-grade laptop (Figure 3.1, right). To improve the accessibility and personalized use of NeRFs, there is a need to explore efficient rendering techniques that can also be trained on consumer-grade devices.

In this paper, we introduce FastSR-NeRF, which demonstrates CNN-based super-resolution (SR) can be a simple, low-cost technique for improving the efficiency of NeRF models on consumer devices. The basic idea is to train a small NeRF model to generate lower-resolution scene features with 3D consistency, and a fast SR model to generate higher-resolution features. This combination reduces the number of pixels that need to be computed using the NeRF's slow volume rendering process, increasing rendering speed. While SR techniques for neural rendering have been proposed by previous works, these methods either (i) involve specialized SR modules that use extra input features such as high-resolution reference images [116]; (ii) employ expensive training procedures such as distillation [40]; or (iii) are trained on tens of thousands of images within a generative modeling framework [41]. None of these methods can be feasibly trained on low-power, consumer-grade platforms. Whether it is possible to achieve high-quality results with SR under a limited training budget remains an open question.

Here, we address this question by exploring a simple NeRF+SR pipeline that directly combines existing modules. We hypothesize that the spatial inductive bias of CNN-based SR is sufficient to generate high-quality outputs for low upscaling ratios, even without extra input features or complex training procedures. To improve synthesis quality, we propose only a lightweight augmentation technique called ***random patch sampling***: rather than extract patches from an image grid for training the SR module as done in existing works[116; 267], we extract patches from random positions to increase the diversity of image patches seen by the SR module. Experiments across three datasets show, somewhat surprisingly, our simple NeRF+SR pipeline with low training overhead can achieve comparable quality and greater rendering efficiency than existing complex NeRF+SR pipelines. To summarize, the key results of our study are as follows:

- **SR can be a nearly "free" technique for improving neural rendering efficiency.** Our comprehensive experiments across three datasets show that applying SR to a NeRF model at 2-4× upscaling ratios, without any complex training procedures or architectural modifications, can improve inference speeds by up to 35.7× on an NVIDIA V100 GPU and 12.8× on an M1 Pro chip, while maintaining peak signal-to-noise ratio (PSNR) in a 0.4-1.2 dB range. *Surprisingly, our simple pipeline can achieve comparable quality to recent and more complex SR techniques [261; 116], while being more efficient in training and inference.*

- **Random patch sampling is a crucial lightweight augmentation technique for NeRF+SR.** We propose random patch sampling, a lightweight augmentation technique. This augmentation improves the PSNR of the SR module by up to 0.89 dB for 2× upscaling and up to 1.44 dB for 4× upscaling compared to standard grid-based patch sampling, outperforming expensive distillation approaches [40] at a fraction of the time cost.

- **FastSR-NeRF is one of the few efficient methods that can be *trained* on a low-power device.** As shown in Figure 3.1, by utilizing a simple NeRF+SR pipeline, FastSR-NeRF can be trained on consumer devices such as a MacBook Air M2, whereas most other models and existing NeRF+SR pipelines fail to train with a meaningful time budget.

Overall, our analysis shows that SR can be a low-cost, plug-and-play strategy for improving the efficiency of neural rendering models under a limited training budget. Even a simple NeRF+SR pipeline can make neural rendering more efficient and accessible for those with low-power, consumer-grade hardware.

3.2 Neural Radiance Fields (NeRFs) and Super Resolution

Neural Radiance Fields

The NeRF model was first proposed in [190]. Given a position and view angle in a 3D scene, NeRF uses a large MLP network to map from the 5D input (3D coordinates plus 2D view angle) to an RGB and a density value. To render a 2D image, these MLP outputs are integrated along rays passing through each pixel using volume rendering. The MLP is optimized using gradient descent with respect to a photometric loss over a sparse set of scene-specific images. Due to its impressive results on static novel view synthesis, NeRF quickly propelled the state-of-the-art in many other fields, including 3D image generation [214; 168; 41], 3D scene editing [100] and landscape reconstruction [280]. However, the drawback of the original NeRF is that it takes a long time to render images due to the slow volume rendering process.

To address this issue, many works have since been proposed to improve NeRF's rendering efficiency. One line of work [85; 294; 105; 50] maps the learned radiance field to explicit representations such as octree-based [294] or voxel-based [105] data structures. These methods achieve faster rendering tine at the cost of larger memory and training time requirements. Another line of work [110; 74; 166; 194] focuses on improving the sampling algorithm to reduce overall computation, which yields a modest amount of acceleration. An emerging series of works divides the radiance field into explicit voxels [173; 224; 244], or some efficient representation such as matrix decomposition [44], hash table [194], and tri-plane [41; 229; 39]. As these models usually make use of a mix of explicit representations and MLP, they are referred as hybrid NeRFs. Typically, hybrid NeRF models can highly accelerate training time and rendering speed, but need a relatively large model size. Among these efficient NeRF methods, there is a trend to develop customized GPU kernels to further accelerate the specialized operations designed for each method. Although using customized kernels can bring a major speedup, it limits the ability to deploy the model on different classes of GPUs and consumer-grade devices [40]. Orthogonal to these works, we explore the application of SR modules for improving NeRF rendering efficiency under a limited training budget. To maximize flexibility for running on low-end devices, we consider Python implementations that do not use customized GPU kernels.

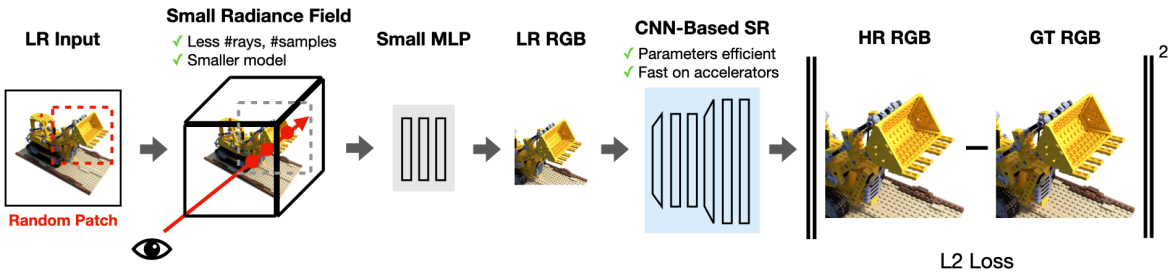


Figure 3.2: Overview of FastSR-NeRF.

Super Resolution with NeRFs

Super-resolution (SR) is a recent, still under-explored method for enhancing NeRF efficiency. EG3D [41] applies SR on top of volume rendering within a generative adversarial network for 3D faces. The SR module in their network is trained on tens of thousands of images, whereas we consider scene-specific optimization on much smaller datasets (*e.g.,* 20-200 images per scene). NeRF-SR [261] performs sub-pixel sampling to super-resolve outputs, but this requires more compute and thus a longer training time. MobileR2L [40] proposes a full CNN-based neural light field model and uses a SR model in its second stage, but their method is trained using an expensive distillation procedure. RefSR-NeRF [116] proposes a specialized SR module that uses high-resolution reference image as additional input, resulting in slower training and inference times. 4K-NeRF [267] synthesizes ultra high-resolution (4K) outputs using depth features as additional input and incorporates SR to achieve feasible inference times. Overall, these existing works all have high training overhead and are not meant to be optimized on lower-power consumer devices.

In our work, we approach SR techniques for neural rendering from a different perspective. Rather than develop a complex pipeline that pushes the limit of reconstruction quality on high-end GPUs, we ask what efficiency gains, if any, can be made from a simple NeRF+SR pipeline trained on consumer-grade hardware. Our experiments show that a simple NeRF+SR pipeline can achieve comparable quality to existing complex pipelines, while being lightweight enough to train on consumer-grade hardware.

3.3 Method of FastSR-NeRF

A Simple NeRF + SR Pipeline

As shown in Figure 3.2, our pipeline simply consists of a NeRF model concatenated with a CNN-based SR module. Given a ray $\mathbf{r} = \mathbf{o} + t\mathbf{d}$, where $\mathbf{o}$ and $\mathbf{d}$ are respectively the ray origin and direction, NeRF reconstructs the color $\widehat{C}(\mathbf{r})$ with volume rendering as follows:

$$\widehat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i \cdot (1 - \exp(-\sigma_i \delta_i)) \cdot c_i, \tag{3.1}$$

where N is the number of sampling points along the ray, δ_i is the distance between two point sampled at t_i and t_{i+1}, $T_i = \prod_{j=1}^{i-1} \exp(-\sigma_j \delta_j)$, and σ_i and c_i are the density and color respectively of a position $\mathbf{x}$ in the 3D scene. In the original NeRF model, σ_i and c_i are computed by MLP networks $\mathcal{F}_\sigma$ and $\mathcal{F}_c$ given position and viewing direction. In our pipeline, we use a hybrid NeRF model [44] to achieve state-of-the-art quality with improved training time and rendering speed. To compute the density and color, we fetch radiance features from a grid-based decomposition $\mathcal{G}_\sigma$ and $\mathcal{G}_c$, and then feed sampled features to MLP $\mathcal{F}_\sigma$ and $\mathcal{F}_c$:

$$\sigma_i = \mathcal{F}_\sigma(\mathcal{G}_\sigma(\mathbf{x})), c_i = \mathcal{F}_c(\mathcal{G}_c(\mathbf{x}), \mathbf{d}) \tag{3.2}$$

Due to the more powerful discrete features, the MLPs $\mathcal{F}_\sigma$ and $\mathcal{F}_c$ in the hybrid NeRF are smaller than the ones used in vanilla NeRF.

To train and render with the full NeRF+SR pipeline, we sample $\mathbf{r}$ from a patch of rays at low-resolution (LR) R^P_{LR}, perform volume rendering based on Equation 3.1, and upsample the output with SR module S to get the final high-resolution (HR) output H:

$$\forall \mathbf{r}_{\text{LR}} \in R^P_{\text{LR}}, H = \mathcal{S}(\widehat{C}(\mathbf{r}_{\text{LR}}; \mathcal{F}; \mathcal{G})) \tag{3.3}$$

Note that the sampling here covers a contiguous 2D patch, which differs from the stochastic sampling of rays used for training standard NeRFs. The NeRF+SR pipeline is optimized in an end-to-end manner with respect to the loss computed over the high-resolution image patch:

$$\mathcal{L}_{\text{MSE}} = \sum_{\mathbf{r}_{\text{HR}}, \mathbf{r}_{\text{LR}}} \left\| C(\mathbf{r}_{\text{HR}}) - S(\widehat{C}(\mathbf{r}_{\text{LR}}; \mathcal{F}; \mathcal{G})) \right\|_2^2 \tag{3.4}$$

where C is the ground-truth color and $\mathbf{r}_{\text{HR}}$ is the HR counterpart of $\mathbf{r}_{\text{LR}}$ in R^P_{HR}. In practice, we use bilinear interpolation to downsample R^P_{HR} and get R^P_{LR}.

Comments on Efficiency. For a high-resolution NeRF model, the number of rays computed by Equation 3.1 will be R_{HR}, and will also require N in the order of thousands to millions to reconstruct details. In contrast, in a NeRF + SR pipeline, the NeRF only needs output a low-resolution output, which reduces the number of rays to R_{LR}. N can also be reduced as there are fewer details in the lower-resolution image. In addition, we can reduce the grid and feature size of the hybrid NeRF to further improve NeRF efficiency, and still maintain the output quality at LR. As a result, we can greatly lower the computation overhead in Equation 3.1 and reduce the memory usage by letting NeRF output at LR.

The addition of the CNN-based SR module S does not present much of a computational bottleneck. With many years of progression on deep learning, the convolution operation is highly optimized and can efficiently run on modern commodity hardware such as GPUs [52], CPUs [6] and specialized accelerators [2; 8]. Furthermore, CNNs are parameter efficient by design [152; 104]. The memory savings from down-scaling the NeRF to LR are much more than the parameters overhead induced by the SR model, which reduces overall model size.

Random Patch Sampling

As discussed in Section 3.3, SR-based NeRF models need to be trained in a patch-style sampling instead of the tradition stochastic ray sampling. Previous works handle the patch sampling by dividing the rays of an image into equal-size patches following rigid grid lines which are determined by the given grid size. These ray patches are then shuffled and fed into the SR-based NeRF pipeline for supervision – we call this grid-based patch sampling. The problem of the grid-based patch sampling is that the sampling algorithm will cut the ray space strictly with a certain grid size. When the model gets trained on individual patches, there will be some variations that are never seen by the convolutional kernels as they sit between each grid boundaries.

To solve this issue, we propose **random patch sampling**. Instead of having a rigid grid lines and restricting sampling to the grid, we randomly sample a region in the ray space, and use that patch to train the model. In this way, we can still have a fixed patch size, but the content of each patch will be different regions of the input. After many iterations, the convolutional kernels in S will cover all of the patterns appeared in the training data and lead to a better training results.

In general, CNN-based models are prone to over-fitting and typically require large-scale datasets [64] to be trained. However, NeRF datasets usually only have tens to hundreds of training images, which is orders of magnitude smaller than a typical CNN pre-training dataset. Existing SR-based NeRF models tackle the overfitting issue by rendering extra data from a teacher model, or guiding training with additional features from high-resolution reference image or depth maps, but these significantly increase training overhead. Random patch sampling is a lightweight data augmentation technique that enables the convolutional kernels of the SR module to see more diversity in the training set. This crucial augmentation allows us to achieve high-quality results without the more complex architectures or training procedures of previous works. We provide ablation results of random patch sampling versus baselines in Section 3.4.

3.4 Evaluations of FastSR-NeRF

Datasets

We use the following three datasets for experiments.

NeRF Synthetic dataset. The NeRF Synthetic dataset was collected along with the origin NeRF [190] paper. It contains 8 different synthetic scenes with 360° degree views produced from the Blender [3] 3D computer graphics creation framework. Each scene in this dataset contains an object with complicated details. The object is placed in the middle of the 3D space and the backgound is white. For each scene, it has 100 training images and 200 testing images of the object from different views. The resolution of the collected images are 800×800.

NSVF Synthetic dataset. The NSVF Synthetic dastaset was released with the NSVF [173] paper. It has a similar setting as NeRF Synthetic dataset with gradually more complex geometry and lightening on the main object. The resolution is also at 800×800.

LLFF dataset. LLFF dataset was collected along with the LLFF [189] paper. The scenes in this dataset were captured in the real world with foward-facing angle. It also has a major object placed roughly in the middle of each scene. However, different from the NeRF Synthetic dataset, the scenes in LLFF dataset have complex background depending on the captured environment. The original resolution of the collected images are 4032×3024, and it also provide images at 4× and 8×lower resolution. Due to the practical usage, most of the NeRF works including us evaluate this dataset on 4x lower resolution at 1008×756. Each scene in the LLFF dataset has 20 to 40 images, and 7/8 are used for training and 1/8 are used for testing.

Dataset	Method	PSNR	
		SR-2×	SR-4×
NeRF-Synthetic	Grid-based	31.84	29.28
	Random	**32.53**	**30.47**
NSVF-Synthetic	Grid-based	34.34	30.45
	Random	**35.39**	**32.04**
LLFF	Grid-based	**26.2**	24.94
	Random	26.04	**25.41**

Table 3.1: PSNR comparison on using **random patch sampling** versus **grid-based patch sampling**.

Experiment Setup

We choose TensoRF [44] as our NeRF backbone as it achieves state-of-the-art results on both quality and efficiency, without requiring customized CUDA kernels, and therefore aligns with the goal of this paper. For our SR model, we chose EDSR [167] due to its accessible implementation and wide adoption in the computer vision community [5]. Although we choose TensoRF and EDSR as our NeRF and SR model, both of them can be replaced with other methods, as our pipeline is model agnostic. Since our SR module solely relies on the RGB output of the NeRF, we are able to leverage pretrained SR models. To train our pipeline, we first warm up the TensoRF model at LR using its default hyperparameters (inherited from the official implementation) for 5K iterations. After warming up, we plug a pretrained EDSR model with desired SR ratio into our pipeline and start training end-to-end using random patch sampling. For the training hyperparameters, we fix the learning rate at 0.0001, patch size at 256 and 128 for SR-2× and SR-4×. We use Adam optimizer [147] and train the pipeline for 150 epochs. For each iteration in a epoch, we only feed one patch to the pipeline. We run our experiments on a machine that is equipped with a single NVIDIA

V100 GPU with 16GB memory unless we specify the hardware platform.

Upsample Ratio	Upsample Method	Train Strategy	Train Time(m)	PSNR
2×	Bilinear	-	11	29.77
	EDSR	Pretrained	11	30.40
		Scratch	51	31.64
		FT-GridPatch	51	31.84
		FT-RandPatch	89	**32.53**
		Distillation	365	32.12
4×	Bilinear	-	3.5	26.67
	EDSR	Pretrained	3.5	27.62
		Scratch	19	29.03
		FT-GridPatch	19	29.28
		FT-RandPatch	30	**30.47**
		Distillation	166	29.94

Table 3.2: Training time and PSNR of different training strategies on NeRF Synthetic dataset.

Evaluation on Quality and Efficiency

Efficiency gains of utilizing SR. Here we evaluate the quality and efficiency gains of our simple NeRF+SR pipeline. We list peak signal-to-noise ratio (PSNR), structural similarity (SSIM) and perceptual similarity (LPIPS) [306] for quantitative quality measurements, and provide training time, rendering time and model size for efficiency evaluations. For LPIPS, we use VGG [238] as the backbone. The results can be found in Table 3.4.

As shown in Table 3.4, comparing to the backbone TensoRF [44] model, applying SR can generally maintain quality at the 2x ratio and enjoy efficiency benefits in rendering time and model size. For example, our pipeline with SR-2× only has a small 0.61dB, 1.13dB and 0.4dB PSNR drop and has near no loss on SSIM and LPIPS compared to the baseline model. Our pipeline at SR 2× even achieves a slight improvement on SSIM for NSVF-Synthetic. For efficiency, using 2x SR rate can improve rendering time by 4.5×, 4.6× and 7.5× and reducing model size by 3.6×, 2.8×, and 7.2× for NeRF-Synthetic, NSVF-Synthetic and LLFF respectively. For SR-4×, we observe a more notable quality loss to the baseline compared to SR-2×. However, it can still achieve qualified results such as over 30dB PSNR on synthetic datasets and just a small 1.19dB PSNR loss on LLFF dataset. At the mean time, with 4× SR rate, it can further improve the ren-

dering time speedup to 18.2×, 18.6× and 35.7×, achieve model size reduction at 5.5×, 6.2×, and 12.5× for NeRF-Synthetic, NSVF-Synthetic, and LLFF. Furthermore, the training time for SR-4× is down to 30 min for synthetic datasets and 1hr for LLFF as the model run and converge faster at this rate. For SR-8×, although the efficiency is further improved, our pipeline can not maintain a good quality at this upscaling rate.

Comparing to existing SR-based NeRFs. We compare our simple pipeline to three existing SR-based model, which are MobileR2L [40], NeRF-SR [261] and RefSR-NeRF [116]. Notice that MobileR2L is light field based model and is not based on radiance field. However, they still utilize SR to enhance rendering speed so we include it for comparison.

Comparing to them, our simple pipeline with only lightweight techniques in training achieves a very clear advantage on the training time. At SR-2×, our pipeline can be trained 23.3× and 19.2× faster than existing SR-based models on NeRF-Synthetic and LLFF, while achieving either on par or better quality.

Qualitative results. We show qualitative results and comparison on selected scenes from NeRF-Synthetic and LLFF datasets in Figure 3.3. As Figure 3.3 shows, with 2× upscaling rate, our pipeline can achieve on-par visual quality as the baseline TensoRF, while using bilinear interpolation at the same rate is not enough to get high fidelity results. Note in Figure 3.3, we compare the results of applying SR to the baseline NeRf model, TensoRF [44], and also list vanilla NeRF as a reference. For NeRF-Synthetic and LLFF, we also include the results of other SR-based models from their paper. The results are highlighted in red when there is a clear disadvantage of a method. ‡ means rendering time is for iPhone13, while other time is on GPUs.

Training on Consumer Devices.

We run our pipeline on a MacBook Pro with M1 Pro chip and a MacBook Air with M2 chip to evaluate efficiency on consumer platforms. The training time, rendering time and PSNR on NeRF-Synthetic for our pipeline at SR rate 2× and 4× are listed in Table 3.3. We also list MobileNeRF's [50] results for comparison.

As shown in Table 3.3, using SR can improve the rendering speed by up to 3.4× and 12.8× for 2× and 4× SR rate on the consumer-grade M-series CPUs. For MobileNeRF [50], although it can achieve a much faster rendering time from its specialized caching mechanism, it needs more than 15 days to be trained on the same device, which is difficult to make a meaningful NeRF application that run fully on a consumer-

grade platform. In contrast, our pipeline, although can not achieve real-time rendering, it still significantly accelerates the rendering process with a reasonable training time (less than 1 day). Note that our experiments are run on CPUs because the current Apple Metal Performance Shader (MPS) [1] support in PyTorch can not fully run the operators needed in NeRF and SR on the MPS device. We expect our training and rendering speed to be faster once PyTorch has a better MPS operators support. In Table 3.3, † means training time is approximated for training the vanilla NeRF on M-series CPUs, which is only the first training stage of MobileNeRF.

Method	PSNR	M1 Pro		M2	
		Train Time	Render Time	Train Time	Render Time
TensoRF	33.14	2h	54s	2.5h	45.4s
FastSR-NeRF (2×)	32.53	22.5h	15.9s	16h	15.2s
FastSR-NeRF (4×)	30.47	15.5h	4.2s	13.5h	4s
MobileNeRF	30.9	>375h†	0.016s	>375h†	0.017s

Table 3.3: PSNR and time profiling of running vanilla TensoRF, FastSR-NeRF (ours) and MobileNeRF [50] on a MacBook Pro laptop with M1 Pro chip.

The Importance of Random Patch Sampling

We evaluate the effectiveness of random patch sampling, which we discussed in Section 3.3. To evaluate, we first establish our model pipeline as we explained in Section 3.3. We then keep all the settings of the pipeline the same but use different patch sampling algorithm to train our model. Notice that we fix the patch size and training for the same number of iterations, so the number of patches seen by the model is the same for both sampling methods. For grid-based patch sampling, we randomize the order of the patches fed into the model to ensure training stability. We report the averaged PSNR for SR ratio 2× and 4× of training our model with these two patch sampling algorithms in Table 3.1.

As Table 3.1 shows, we observe that random patch sampling consistently leads to a higher PSNR than grid-based patch sampling on synthetic datasets, The highest improvement appear on NSVF-Synthetic when the SR rate is 4×, where random patch sampling records a 1.59 PSNR increase over grid-base patch sampling.

On real-world forward facing LLFF dataset, the PSNR enhancement is less significant than the synthetic

Method	NeRF-Synthetic					
	PSNR↑	**SSIM↑**	**LPIPS↓**	**Train Time↓**	**Render Time(s)↓**	**Model↓ Size(MB)**
NeRF [190]	31.01	0.947	0.081	∼35h	20	5
TensoRF [44]	**33.14**	0.963	**0.049**	18m	1.4	71.8
FastSR-NeRF(2×)	32.53	0.961	0.052	1.5h	0.309	20
FastSR-NeRF(4×)	30.47	0.944	0.075	30m	0.077	13
FastSR-NeRF(8×)	27.27	0.902	0.142	16m	0.030	8
MobileR2L [40]	31.34	**0.993**	0.051	>35h	0.026‡	8.3
NeRF-SR [261]	28.46	0.921	0.076	>35h	5.6	-

Method	NSVF-Synthetic					
	PSNR↑	**SSIM↑**	**LPIPS↓**	**Train Time↓**	**Render Time(s)↓**	**Model↓ Size(MB)**
NeRF [190]	30.81	0.952	-	∼35h	∼20	∼5
TensoRF [44]	**36.52**	0.959	**0.027**	15m	1.4	74
FastSR-NeRF(2×)	35.39	**0.979**	0.032	1.5h	0.302	26
FastSR-NeRF(4×)	32.04	0.958	0.059	30m	0.075	12
FastSR-NeRF(8×)	27.93	0.911	0.119	16m	0.030	9

Method	LLFF					
	PSNR↑	**SSIM↑**	**LPIPS↓**	**Train Time↓**	**Render Time(s)↓**	**Model↓ Size(MB)**
NeRF[190]	26.5	0.811	0.250	∼48h	33	5
TensoRF[44]	26.6	0.832	0.207	28m	5.9	188
FastSR-NeRF(2×)	26.20	0.822	0.241	2.5h	0.786	26
FastSR-NeRF(4×)	25.41	0.784	0.297	57m	0.165	15
FastSR-NeRF(8×)	21.30	0.584	0.475	15m	0.040	8
MobileR2L[40]	26.15	**0.966**	0.187	>48h	0.018‡	8.3
NeRF-SR[261]	**27.26**	0.842	**0.103**	>48h	39.1	-
RefSR-NeRF[116]	26.23	0.874	0.243	-	8.5	38

Table 3.4: Quantitative and efficiency results on NeRF-Synthetic, NSVF-Synthetic and LLFF datasets.

datasets. We observe a slight PSNR decrease (0.16dB) for SR-2× but a clear PSNR increase (0.47dB) for SR-4×. We hypothesize that this is because the real-world scenes typically contains greater complexity and finer-grained detail than the synthetic scenes. For example, in a synthetic scene, there is usually one major object and the space outside of the object is empty. Although, the object might has some difficult and fine-grained patterns, the model can focus on learning the patterns on the object. However, in a real world scene, although it usually still has a major object, the background is usually messy and contains a lot of small details. Therefore, using random patches on real world scenes such as LLFF dataset can not bring a

big difference in terms of the total number of patterns converged in the patches. As a result, random patch sampling shows a greater improvement on synthetic datasets but still has the ability to enhance PSNR on real world scenes dataset when the SR rate is higher, e.g. $4\times$.

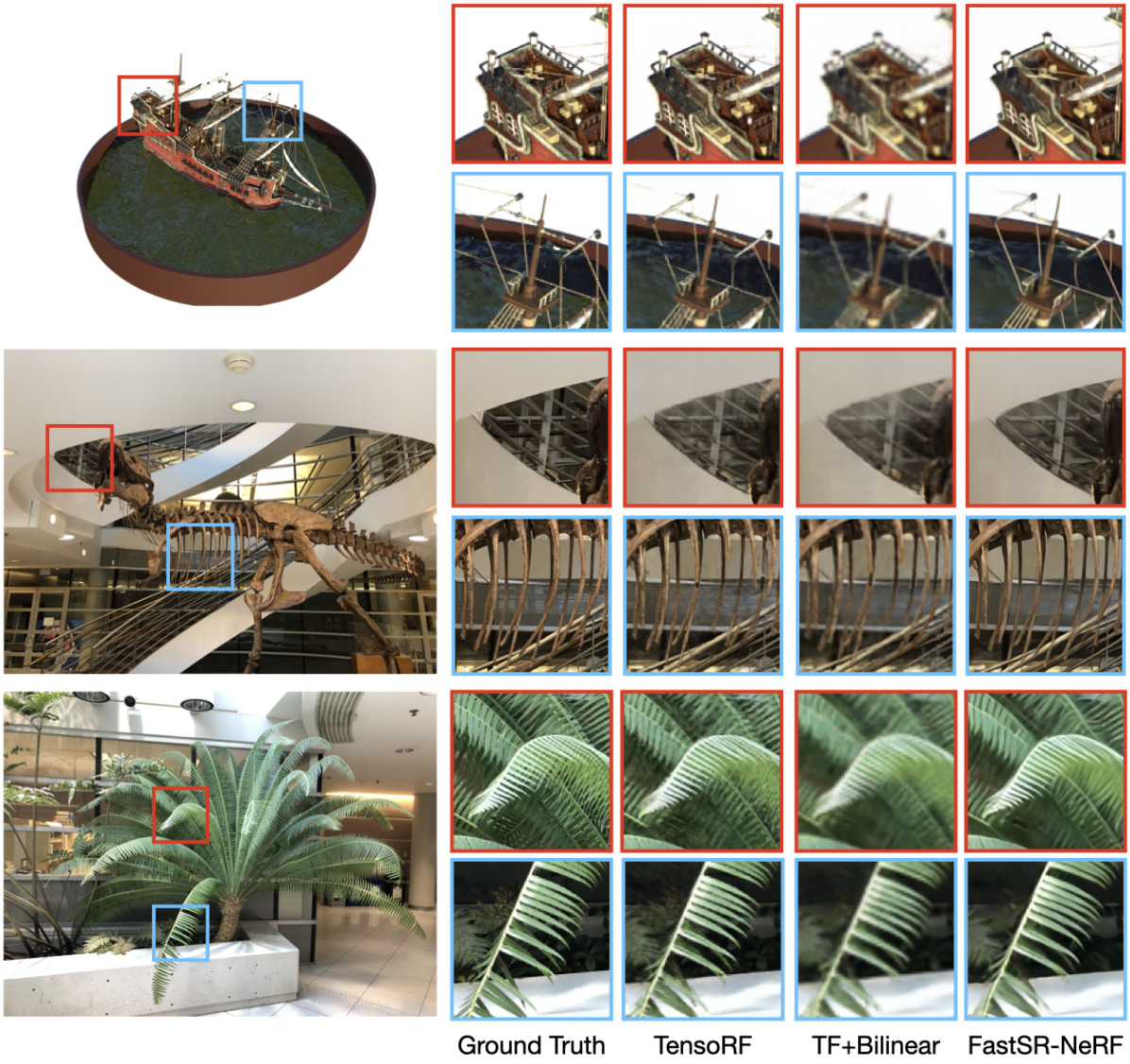


Figure 3.3: Qualitative results on a NeRF-Synthetic and LLFF.

Ablation Study on Training Strategies.

In Section 3.4, we compare the results of training our NeRF + SR pipeline with grid-based or random patch sampling. However there are many more configurations possible. In this section, we compare the results of

1) using bilinear interpolation as the SR method, 2) use out-of-box pretrained SR model without finetuning, 3) training the NeRF and SR model both from scratch, 4) finetuning the pipeline with pretrained SR on the NeRF dataset with grid-based patch sampling, 5) same as (4) but uses random patch sampling, 6) use the distillation method proposed by [40] and train on a larger training set augmented by a teacher NeRF at HR.

We train the pipeline on NeRF-synthetic dataset and show the comparison in Table 3.2. Using bilinear interpolation as SR has the shortest training time (only requires warming up the NeRF backbone) but has a significant PSNR decrease. Directly using a pretrained EDSR model can also cut down the training time and has a better PSNR than bilinear. However, training on NeRF dataset still help it to achieve a better accuracy. Among those training methods, finetuning SR using random patch sampling achieves the best results while paying a little more training time (exclude distillation) due to the random sampling overhead. For training the pipeline, although we see a promising PSNR increase with 1K extra training images generated by a teacher TensoRF, the training time becomes much longer as we need to train the NeRF model at HR first and also need to render many HR images. Note that the PSNR of distillation might increase if we generate more training images, but the training time will be even longer. We do not further optimize our distillation procedure as it's not the focus in this paper. To sum up, using random patch sampling and finetuning a pretrained SR model gives us the best trade-off between time and quality under our pipeline setup.

3.5 Summary

In this work, we study the limit of SR-based NeRF model. We propose FastSR-NeRF and find a cohesive and simple NeRF + SR pipeline can actually achieve impressive quality while also being compute and memory efficient. The key result of this approach is that, although it's not the fastest nor the smallest model, it remains efficient for all of training time, rendering latency and model size. We achieve this by leveraging the lightweight technique called random patch sampling and pretrained SR model – both of these interventions can boost our pipeline's accuracy without introducing prohibitive computational overhead. Our pure Python-based approach (without any customized GPU kernels) allows the whole training & inference pipeline to run on consumer-grade devices such as a laptop with a reasonable time. We believe this work and comprehensive analysis will help the development of an end-to-end NeRF application that can purely be deployed on personal devices for improved compute efficiency and user privacy.

Remarks on Author Contributions

My major contributions are as follow:

- Led the project development and paper writing.

Chapter 4

TeleRAG: Efficient Retrieval-Augmented Generation Inference with Lookahead Retrieval

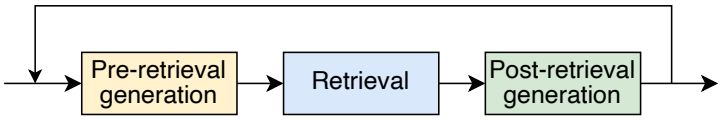
Remarks on Chapter Material

The content of this chapter is adapted from the following paper (* indicates equal contributions):

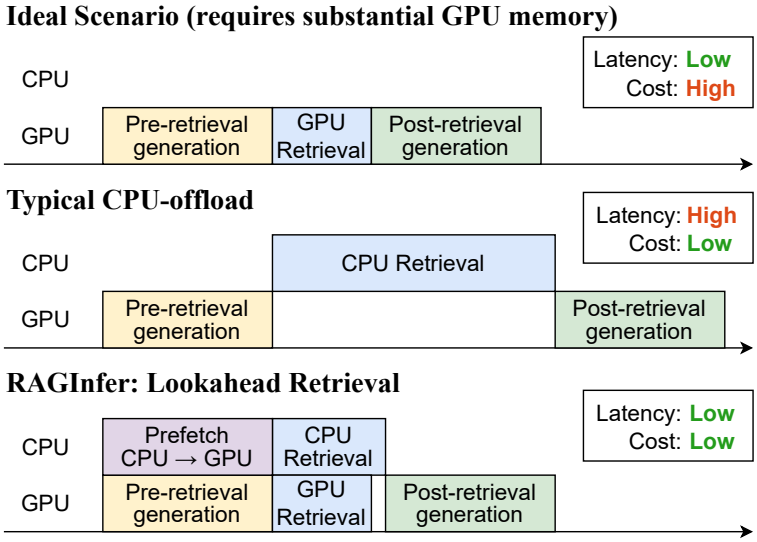
- Chien-Yu Lin*, Keisuke Kamahori*, Yiyu Liu*, Xiaoxiang Shi, Madhav Kashyap, Yile Gu, Rulin Shao, Zihao Ye, Kan Zhu, Stephanie Wang, Arvind Krishnamurthy, Rohan Kadekodi, Luis Ceze, Baris Kasikci, "TeleRAG: Efficient Retrieval-Augmented Generation Inference with Lookahead Retrieval", arXiv:2502.20969 2025.

4.1 Introduction

Retrieval-augmented generation (RAG) has emerged as a powerful technique to enhance large language models (LLMs) by integrating them with external databases [84; 26; 162]. During inference, RAG *retrieves* relevant content from external data sources, usually indexed as vector datastores, to mitigate issues such as hallucinations [185; 221; 142] and incorporate up-to-date or private information [121; 191].



(a) Typical pipeline stages of a RAG application.



(b) Different scenarios for RAG and the illustration of the proposed *lookahead retrieval* mechanism. It prefetches relevant data for retrieval from CPU to GPU, overlaps data transfer with the pre-retrieval stage, and accelerates retrieval with GPU-CPU cooperation.

Figure 4.1: (a) Illustrations of RAG pipeline stages. (b) Overview of TELERAG and comparison to the baseline.

Modern RAG applications share two key characteristics. (1) RAG applications are built as modular pipelines as shown in Figure 4.1a, where a single query undergoes *multiple rounds* of LLM calls and retrievals, each of which serves different functions to improve overall output quality. For example, query transformation rounds [81; 315; 212; 23] generally occur before retrieval to refine the user's query with LLMs (pre-retrieval generation). (2) RAG's *datastores are typically large*, supported by recent works demonstrating that larger datastore consistently improves the performance of RAG applications [191; 142; 35; 101; 231].

These characteristics create several significant challenges for efficient RAG inference, especially in latency-sensitive applications such as customer chatbots [43; 19; 257], financial analysis [183; 195], and emergency medical diagnosis [149; 89]. First, the latency of RAG systems is inherently increased due

to multiple rounds of LLM generation and retrieval. Second, although GPUs can speed up both LLM generation and retrieval, the combination of these processes poses significant memory requirements, often exceeding available GPU resources (see §4.3). Hence, it is expensive or even infeasible to fully host both operations on GPUs.

Consequently, in local or custom deployments, which are common for RAG applications with private or sensitive data, the retrieval datastore is often offloaded to CPU memory to alleviate GPU memory constraints. However, while CPU offloading resolves memory limitations, it significantly increases retrieval latency (see §4.3), diminishing overall system efficiency. This memory pressure is not limited to local deployments; even for serving scenarios in data centers, where many concurrent requests are processed together and GPUs are plentiful, managing large index sizes still presents a significant challenge for achieving high throughput with latency service level objectives (SLOs). Allocating additional GPU memory to the retrieval data store directly reduces the memory available for the KV cache of the LLM serving system, limiting the effective batch sizes [156]. In contrast, CPU retrieval would increase the per-request latency by a lot, highlighting the need for techniques that reduce GPU memory consumption while preserving low latency.

To address the latency challenge, several recent works have been proposed to accelerate RAG inference. These approaches include hardware accelerators for retrieval [131; 218], reuse of KV-cache [134; 180; 288], and speculative document retrieval [268; 310]. However, these works do not address the substantial memory demands associated with retrieval. Therefore, designing a memory-efficient system for low-latency RAG deployments is crucial.

In this paper, we argue that a promising direction to reduce GPU memory usage while maintaining low latency lies in improving the design of the retrieval index. Specifically, the inverted file index (IVF) [240] reduces retrieval latency by pre-clustering data and limiting runtime search to relevant clusters. It offers a way to leverage GPU-accelerated search while not increasing memory consumption by dynamically transferring only the necessary clusters from the CPU to the GPU. However, this data transfer overhead is a key bottleneck, and needs to be minimized [141] (see §4.3).

Our proposal. To tackle latency and memory bottlenecks in RAG inference, we introduce a novel mechanism called **lookahead retrieval**, which predicts the subsets of IVF clusters that will likely be accessed

during runtime and proactively transfers them from the CPU to the GPU before the retrieval stage.

The key observation behind lookahead retrieval is that there is substantial semantic overlap between queries *before* and *after* the pre-retrieval stage. Namely, IVF clusters used by the initial input query, which is available well before the retrieval stage, are also likely to be relevant to the actual retrieval query (see §4.3).

Using this insight, we propose TELERAG, an efficient inference system designed to optimize RAG latency while minimizing GPU memory consumption. TELERAG employs lookahead retrieval to preemptively load relevant IVF clusters onto the GPU, effectively hiding CPU-GPU data transfer overhead during concurrent LLM generation. As illustrated in Figure 4.1b, this approach significantly reduces retrieval latency without exceeding GPU memory constraints, enabling efficient execution of RAG applications. To maintain retrieval accuracy while adopting lookahead retrieval, TELERAG simultaneously searches for clusters that were not prefetched (*i.e.*, mispredicted) from the CPU and merges the results. TELERAG also supports batch and multi-GPU inference.

A key challenge in deploying lookahead retrieval is determining the optimal number of IVF clusters to prefetch: Prefetching too many clusters increases data-transfer overhead, whereas prefetching too few could result in higher retrieval latency due to increased CPU processing. To address this, we propose a profile-guided approach coupled with an analytical model that dynamically determines the ideal prefetch amount based on pipeline characteristics and hardware configurations.

Results summary. We evaluate TELERAG with a popular Wikipedia-based datastore [16] across six popular RAG pipelines built with the Llama model family [251] (3B, 8B, and 13B). Remarkably, TELERAG supports retrieval from a **61GB datastore** alongside a **Llama3-8B (16GB)** LLM on a single RTX 4090 GPU (**24GB memory**), significantly outperforming the CPU retrieval baseline. Experiments demonstrate that TELERAG achieves up to a 1.53× average reduction in end-to-end latency for single-query inference on an RTX 4090. In batching scenarios, TELERAG provides even greater throughput enhancements as the batch size increases, achieving 1.83× higher throughput on average at batch size 8 on an H100 GPU. These results underscore TELERAG's capability to efficiently manage large-scale RAG inference tasks under tight GPU memory constraints, validating its practical benefits for resource-constrained deployments.

In summary, we make the following key contributions:

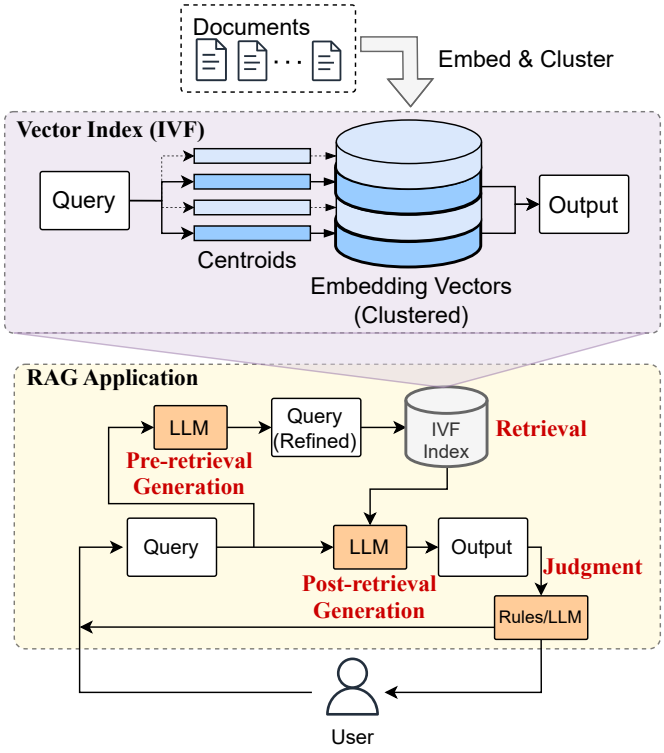


Figure 4.2: Overview of RAG.

- Analyzing the correlation of the queries between the pre-retrieval generation and retrieval stages, revealing significant overlap in their corresponding IVF clusters.

- Proposing *lookahead retrieval*, which prefetches likely IVF clusters to the GPU, and hides CPU-GPU data transfer time during pre-retrieval generation.

- Developing TELERAG, an efficient RAG inference system that integrates *lookahead retrieval*, resulting in significant acceleration of RAG with minimal GPU memory usage.

4.2 Background

RAG

RAG is a technique that enhances the capabilities of LLMs by integrating them with information retrieval to generate more accurate and relevant text [84; 26; 162]. The core idea behind RAG is to augment the LLM with relevant information retrieved from a large corpus of documents, improving the LLM's ability to answer questions without hallucinations [185; 221; 142] and generate contents based on up-to-date or private information [121; 191].

RAG workflow. A basic RAG workflow involves several phases, including data store building, retrieval, and interactions with LLMs [67; 118; 60; 271; 260; 210; 83]. In order to build a data store, raw data in various formats is cleaned, converted to plain text, and divided into *chunks*. These chunks are then encoded into vectors using an embedding model and stored in a *vector index*, enabling efficient searching based on similarity. When a user provides a *query*, it is converted into a vector using the same embedding model, and the most similar chunks from the indexed database are retrieved. For a large-scale index, approximate algorithms such as the inverted file index (IVF) [240] (see §4.2 for detail) are commonly used to accelerate the search process. The retrieved chunks, along with the original user query, are combined and given as a prompt to the LLM. The LLM then generates a response that relies on the information provided in the retrieved documents. This approach enables dynamic and informative responses in both single- and multi-turn dialogues [162].

Modularity in modern RAG applications. However, naively integrating document retrieval into LLM generation can cause several issues. For example, the retrieval often struggles to find relevant content and may select irrelevant or mismatched chunks [81; 315; 182], and a single retrieval may not provide sufficient context, necessitating multiple retrieval rounds [84].

To solve these issues, most state-of-the-art RAG models adopt a *modular* approach that employs multiple rounds of LLM calls and retrievals for a single query [84; 73; 118; 83; 60; 271]. Typically, they have the following types of steps (shown in Figure 4.2):

- *Pre-retrieval generation*, used to assess whether retrieval is needed or to generate queries for retrieval. An example of a *pre-retrieval* technique is query transformation [182; 123; 81; 319; 291; 217; 213;

315; 171], which reformulates the original query to make it clearer and more suitable for the retrieval task.

- *Retrieval*, used to identify relevant documents from the vector data store. This stage takes the output of pre-retrieval generation and generates data for the next stage.

- *Post-retrieval generation*, generates response based on user query and retrieved documents. It can also perform additional process such as summarization [128; 144] or reranking[321; 242] on the retrieved documents.

- *Judgment*, dynamically determines the execution flow. For example, it decides if more iteration is needed to enhance the response. Heuristics or LLMs can be used for judgement stage.

By proceeding through these functions, RAG applications can deliver more precise and contextually appropriate responses, significantly enhancing the capabilities of LLMs for various applications [135; 84].

Vector Index and Inverted File (IVF)

The *vector index* is a crucial component of RAG applications that retrieves similar items from large datasets of high-dimensional vectors. Given the query vector $x \in \mathrm{R}^D$ and vector database $Y = \{y_0, \ldots, y_{N-1}\} \subset \mathrm{R}^D$, which comprises N vectors, the vector index aims to find the k nearest neighbors of x from the database:

$$\text{k-argmin}_{i=0:N} d(x, y_i),$$

where D is the dimensionality of the vector determined by the embedding model, and d denotes the distance function, which is typically the L2-norm or inner product [136].

To improve search efficiency, the inverted file index (IVF) algorithm is widely used for large-scale vector databases due to its simplicity and effectiveness. IVF partitions the data store into *clusters* and restricts searches to the most relevant clusters, reducing the computational cost. To obtain the cluster assignment of each stored vector, it performs clustering algorithm such as k-means [198] and partitions the database into

N_c clusters:

$$\{C_1, C_2, \ldots, C_{N_c}\} = k\text{-means}(Y, N_c),$$

where C_j is the set of vectors assigned to the j-th cluster, and the cluster centroids $\{c_1, c_2, \ldots, c_{N_c}\}$ are obtained by taking the mean of each vectors in $\{C_1, C_2, \ldots, C_{N_c}\}$. Then, each database vector $y_i \in Y$ is assigned to the nearest cluster center:

$$\text{Cluster}(y_i) = \text{argmin}_{j=1:N_c} d(y_i, c_j).$$

With the trained centroids and cluster assignment, we can perform a more efficient index search. There are two steps involved in the IVF index searching. First, it will identify the nearest L cluster centroids of a query vector x:

$$\{c_{j_1}, c_{j_2}, \ldots, c_{j_L}\} = L\text{-argmin}_{j=1:C}\ d(x, c_j).$$

This step is also referred to as the coarse-grained search in IVF, as it identifies candidates at the cluster level. Second, the fine-grained search is performed in the L nearest clusters and identify k closest vectors of x:

$$k\text{-argmin}_{y_i \in \cup_{l=1}^{L} C_{j_l}} d(x, y_i).$$

which involves sorting. In this way, IVF reduces search space and accelerates the retrieval process. Here, the hyperparameter L from the first step is also referred as `nprobe` [69]. When `nprobe` is larger, the IVF retrieval will be more accurate as it search more clusters. However, the retrieve latency is longer as more computation and data are needed.

Since the search process is highly parallelizable among each cluster and each vector, this search algorithm can be highly accelerated by GPUs. Open-source libraries offer efficient GPU implementations [136; 222]. However, IVF does not reduce the memory requirement for the index since the data for all clusters must be stored. As a result, IVF still requires a substantial GPU memory footprint, which becomes a bottleneck when GPU memory capacity is the constraint of RAG systems. Moreover, recent work reported that

increasing the size of the data store positively affects RAG application performance [191; 142; 35; 101; 231], exacerbating this issue.

Use Cases and Motivation

In this paper, we aim to achieve low-latency RAG inference without driving up GPU memory consumption, which is relevant for both local and data center settings. This goal is critical for efficiently serving a wide range of real-world applications where users expect near-instantaneous responses, thus imposing strict latency SLOs. Examples include customer chatbots [43; 19; 257; 28], financial analysis [183; 195], autonomous driving [38; 296], and emergency medical diagnosis [149; 89].

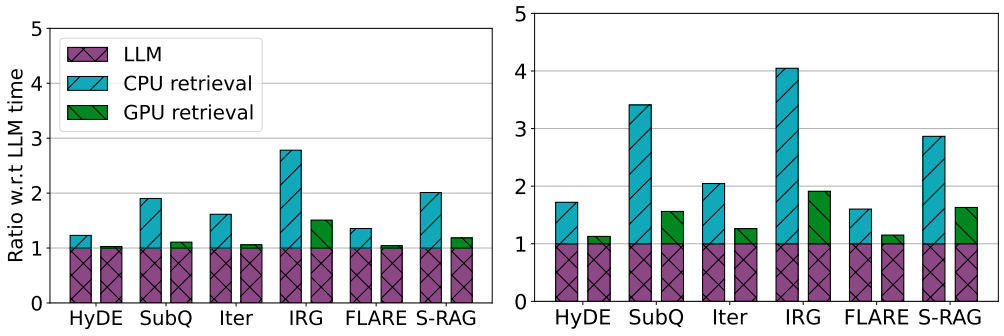
On the one hand, there is significant demand for RAG applications that involve user-specific private data in local settings [270; 158; 55; 279; 86]. In such scenarios, the available GPU resources are usually limited, and the system serves a single request at a time. Therefore, CPU offloading is necessary when using a large vector index, which introduces substantial latency as we will show in the next section.

On the other hand, for large-scale RAG services in data centers [9; 4; 23], the goal is to serve many concurrent user requests cost-effectively, necessitating high system throughput. However, maximizing LLM serving throughput is often bottlenecked by the GPU memory available for the KV cache, which dictates the maximum possible batch size [156]. If a large retrieval datastore is loaded onto GPUs, it consumes memory that could otherwise be used for a larger KV cache, thereby limiting throughput. Hence, falling back to slow CPU retrieval to save GPU memory is a common option [130], but this is not desirable for latency-sensitive applications due to latency overhead.

In both cases, a straightforward way to reduce inference latency is scaling up GPU resources to allow both the LLM and datastore to reside in GPU memory, which is usually cost-prohibitive. Therefore, in this paper, we aim to design an efficient RAG inference system that can satisfy latency SLOs without requiring a significant amount of GPU memory.

4.3 Analyzing Latency of RAG Pipelines

In this section, we analyze state-of-the-art RAG applications and identify their underlying systems challenges in achieving low latency. To conduct the analysis, we construct a 61GB vector index with the Faiss



(a) Batch = 1. **(b)** Batch = 4.

Figure 4.3: Latency breakdown of six RAG pipelines on NQ dataset [155]. `nprobe` is 256.

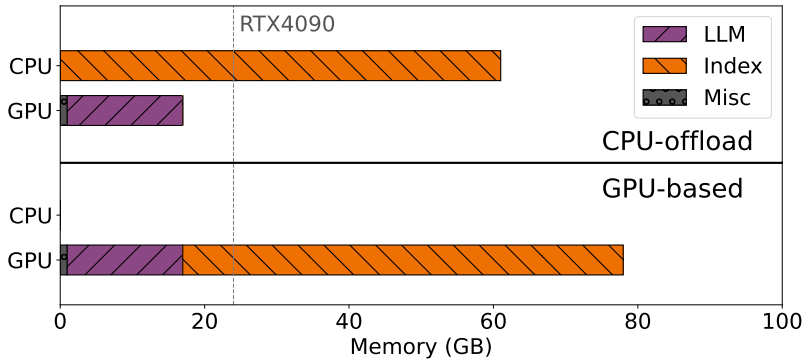


Figure 4.4: The breakdown of memory consumption at GPU and CPU for two different strategies: CPU offloading and GPU-based retrieval. The dotted line indicates the memory capacity of a RTX4090 GPU, which is a common used GPU for local deployment.

library [69], and set the IVF clusters to 4096 following common practice [24]. We use Llama-3-8B [17] as the LLM, and run our analysis on an RTX 4090 (24GB memory) and H100 (80GB memory). §4.5 provides more details about our experimental setups.

End-to-end Latency of RAG Pipelines

We first analyze the end-to-end latency of six representative RAG pipelines (see §4.5 for details) in two scenarios: (1) the LLM runs on the GPU, while retrieval is offloaded to the CPU, minimizing GPU memory usage, and (2) both the LLM and the vector index reside in GPU memory, enabling GPU-based retrieval for lower latency. If the GPU's memory is insufficient, CPU offloading is necessary.

We set the `nprobe` to 256, a commonly used setting under this index's configuration (see §4.5 for details), to measure retrieval latency. Figure 4.3 shows the breakdown of end-to-end latency into LLM infer-

ence part and retrieval part, based on 1024 randomly sampled queries from the NQ dataset [155]. We observe that, in the CPU-based retrieval baseline (first scenario), the retrieval phase dominates the latency, consuming 41.1% and 60.5% of total latency for batch sizes 1 and 4, respectively. In contrast, GPU-accelerated retrieval in the second scenario dramatically reduces this bottleneck, accounting for only 10.5% and 28.3% of the latency in comparable configurations. On average, GPU retrieval is $5.96\times$ (batch size 1) or $3.87\times$ (batch size 4) faster than CPU retrieval, reducing overall latency by approximately $1.5\times$ or $1.8\times$, respectively. Thus, accelerating retrieval on the GPU is crucial for improving end-to-end latency.

However, GPU acceleration comes with a significant memory cost. Figure 4.4 shows the memory requirements for both GPU and CPU. As Figure 4.4 shows, putting both LLM weights and the retrieval index on the GPU requires around 77 GB of memory, which exceeds the capacity of consumer GPUs like the RTX 4090 with 24 GB. Thus, GPU acceleration on retrieval is often not feasible when running on a lower-end GPU or indexing with a large datastore.

Even on GPUs capable of holding the entire index in memory (*e.g.*, H100), the index's memory footprint limits throughput in data center serving scenario. Batched inference maximizes throughput, but it is bottlenecked by the KV cache capacity for LLM serving [156]. For the case of the example above, storing the index on the GPU leaves minimal memory for the KV cache (*e.g.*, ~3 GB, or ~20k tokens worth), restricting batch size, while offloading the index to the CPU significantly increases the available memory (*e.g.*, to >500k tokens worth). This issue is exacerbated since RAG typically involves long contexts [134; 180; 288] and there is a trend of growing model and index sizes [191; 142; 35; 101; 231], increasing memory pressure and necessitating offloading. However, CPU offloading is suboptimal for latency-sensitive applications as discussed above.

Therefore, in the rest of this section, we try to answer the following question: *Is it possible to achieve the latency of GPU-based retrieval while using much less GPU memory?*

Opportunities of GPU-accelerated Retrieval with Limited Memory

A straightforward approach to enable GPU retrieval with limited GPU memory is to fetch the necessary data from CPU to GPU on-demand for each query, leveraging the IVF index structure that narrows the search space (§4.2). While this method enables faster searching on the GPU, data fetching becomes the bottleneck.

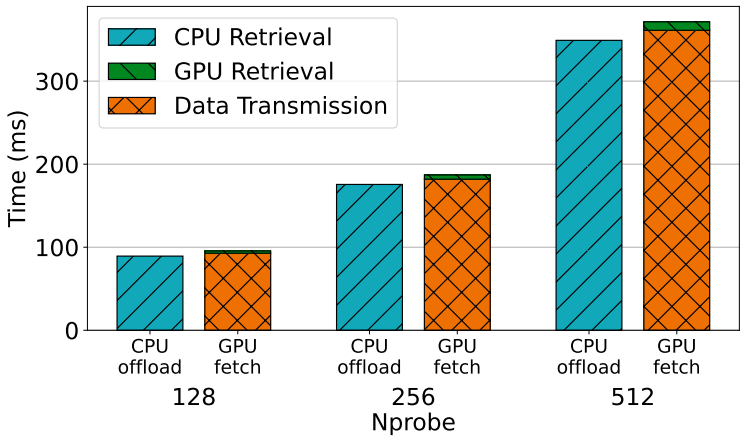


Figure 4.5: Latency breakdown of CPU-offload and runtime-fetch GPU retrieval, averaged over 512 random NQ queries.

Dataset	HyDE [81]	SubQ [171]	Iter [171]	IRG [233]	FLARE [133]	S-RAG [24]
NQ [155]	73.1%	63.2%	91.5%	83.8%	79.1%	100.0%
HotpotQA [287]	75.3%	62.5%	89.6%	89.4%	80.2%	100.0%
TriviaQA [138]	73.1%	61.6%	86.2%	86.1%	81.7%	100.0%

Table 4.1: IVF cluster overlapping rate between the input and output of the pre-retrieval generation. The `nprobe` is set to 512. Since Self-RAG does not incorporate query transform, its coverage is always 100%.

In Figure 4.5, we compare the latency of the on-demand fetching system against CPU retrieval on an RTX 4090 GPU. We show three different `nprobe` values that determine the amount of data fetched. Overall, fetch time dominates latency due to the limited PCIe bandwidth between the CPU and GPU (32 GB/s). Although the GPU search is substantially faster than the CPU counterpart, the fetch overhead results in a higher end-to-end latency, which is about 3% slower on average across `nprobe` values. Thus, to realize any meaningful speedup with this approach, the data fetching latency must be effectively hidden.

To hide data fetch costs, CPU-to-GPU transfer must precede retrieval, which requires predicting which data will be accessed. Fortunately, as we show next, the modern RAG pipelines offer a valuable hint as a query at the previous step.

Overlapping of IVF Clusters

While the exact data to be retrieved is only known after the pre-retrieval generation is done, we observe that there are high similarities of the IVF clusters assignments between the queries at different stages.

Similarity of queries at different stages. During the pre-retrieval process of RAG pipelines (*e.g.*, query transformation [182; 123; 81; 319; 291; 217; 213; 315; 171]), an LLM call is issued to refine an initial user query q_{in} into a transformed query q_{out}, which is then used for retrieval. This process often rewrites the query into a different format [81] or simplifies its complexities [171], which intuitively preserves the query's core semantic content. Hence, the embedding vectors of q_{in} and q_{out} are likely to be similar. This similarity, in turn, suggests that the IVF clusters to which these queries would be assigned will overlap. Therefore, q_{in} can serve as a valuable hint for predicting subsequent memory accesses.

Prediction coverage. To verify this hypothesis, we evaluate the average cluster coverage rate between prefetched clusters and clusters used for retrieval in three popular question-answer datasets (NQ [155], HotpotQA [287], and TriviaQA [138]) and six RAG pipelines. Table 4.1 shows the coverage when we prefetch 256 clusters. From the table, we observe that IVF cluster overlap rates are consistently high across a range of datasets and pipelines. For instance, even the lowest reported values remain above 61.6% (for SubQ).

Opportunity. This data shows an opportunity for predicting required clusters, which can make it possible to hide data transfer overhead during LLM generation. In this paper, we aim to leverage this observation to accelerate the inference latency for RAG.

4.4 Design of TELERAG

Based on the high IVF cluster overlapping observation presented in §4.3, we propose TELERAG, an efficient RAG inference system that utilizes lookahead retrieval to prefetch a minimal set of likely IVF clusters to GPU, and accelerate the retrieval process. In this section, we detail the design of lookahead retrieval and implementation of TELERAG.

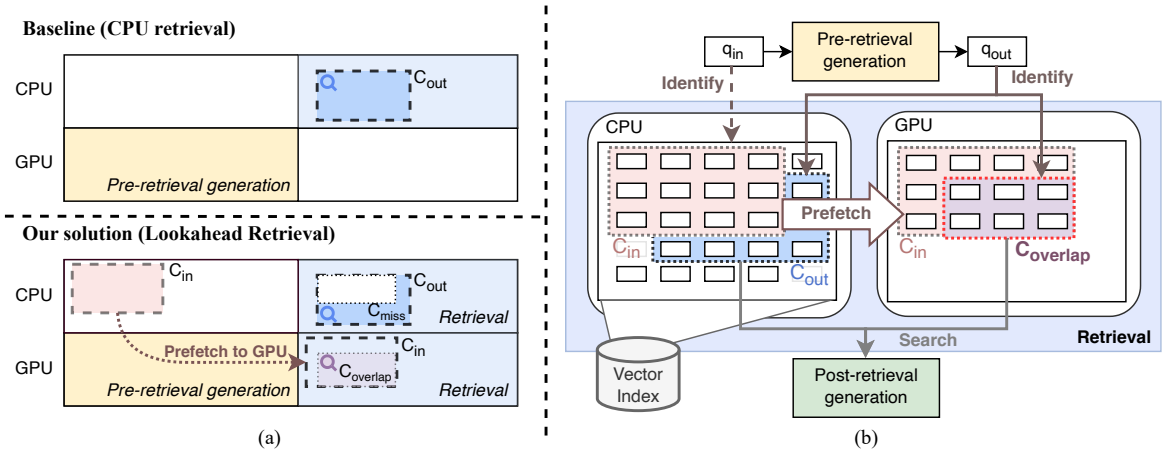


Figure 4.6: (a) The overview of **lookahead retrieval** compared against CPU-offloaded retrieval, and (b) the system design of TELERAG. After identifying clusters (C_{in}) for the initial query q_{in}, C_{in} is transferred to the GPU while concurrently generating q_{out}. At retrieval time, the GPU searches prefetched clusters while the CPU handles remaining ones, before results are merged.

Overview

Figure 4.6 presents the overview of TELERAG with lookahead retrieval. In Figure 4.6, q_{in} denotes the initial input to the pre-retrieval stage and q_{out} denotes its output, *i.e.*, the query given to the subsequent retrieval stage. We highlight the IVF clusters corresponding to q_{in} and q_{out} with a red background (C_{in}), and a blue background (C_{out}), respectively. Due to the semantic similarity of q_{in} and q_{out}, there is significant overlap between C_{in} and C_{out}, marked with a purple background (C_{overlap}).

Given this, the proposed lookahead retrieval technique in TELERAG operates in three key steps:

1. **Pre-retrieval & data transfer:** During LLM generation, identify C_{in} and transfer corresponding data to GPU memory, prioritizing clusters closest to q_{in}. This leverages GPU DMA for concurrent transfer of clusters data and compute of LLM generation.

2. **GPU similarity search:** After identifying C_{out}, the GPU efficiently performs a similarity search on the predicted clusters (C_{overlap}) already in its memory.

3. **CPU similarity search:** Concurrently with the GPU search, the CPU processes the similarity search for the missed clusters (C_{miss}) that were not prefetched to the GPU, reducing the CPU's overall workload.

In summary, our lookahead retrieval speeds up retrieval by overlapping data prefetching to the GPU with LLM generation and performing concurrent similarity searches on both the GPU (for predicted clusters) and CPU (for the remainder), significantly reducing the CPU's computational workload.

Finding the Optimal Prefetch Amount

A key challenge in TELERAG is to balance the benefit of reducing retrieval latency by prefetching data against the overhead of CPU-GPU transfers. Prefetching more clusters reduces the subsequent retrieval time but can also extend the transfer phase; if it extends beyond the LLM generation window, we lose the advantage of overlap and potentially introduce additional delay. Still, additional delays in data transfer may be worthwhile if they substantially reduce retrieval latency. To guide the choice of the optimal amount of data to prefetch, we develop a mathematical model based on profile information.

Mathematical model. Here, we denote b_p as the number of bytes to prefetch and B as the CPU-GPU bandwidth. The optimal amount of data to prefetch is denoted as b_p^*. To start, we let t_1 represent the combined time of prefetching and pre-retrieval LLM generation, and t_2 represents the retrieval time. We have: $t_1 = \max(t_{\text{LLM}}, t_p)$ and $t_2 = \max(t_c, t_g)$, where t_{LLM} is the time for LLM generation, t_p is prefetching time, t_c is the CPU retrieval time, and t_g is the GPU retrieval time. The objective is to minimize $t_1 + t_2$.

Since prefetching time t_p is proportional to b_p, we express t_1 as a piecewise function:

$$t_1 = \begin{cases} t_{\text{LLM}}, & \text{if } b_p \leq B \cdot t_{\text{LLM}}, \\ \frac{b_p}{B}, & \text{if } b_p > B \cdot t_{\text{LLM}}, \end{cases} \tag{4.1}$$

From Eq. 4.1, if we prefetch fewer bytes than can be transferred during LLM generation, t_1 is effectively just t_{LLM} because the transfer overlaps completely with LLM execution.

As shown in §4.3, GPU retrieval (t_g) is generally much faster than CPU retrieval (t_c). Thus, we assume $t_c \gg t_g$. As CPU has a limited parallelism, t_c usually grows proportionally with the number of clusters to

be processed [129], such that:

$$t_2 = t_c = r_{\text{miss}} \times n_{\text{probe}} \times t_{cc}, \tag{4.2}$$

where r_{miss} is the miss rate (percentage of IVF clusters are not caught on GPU), n_{probe} is the total number of clusters to search, and t_{cc} is the CPU time to search a single cluster. Increasing b_p can only decrease or maintain the current miss rate, *i.e.*, $\frac{\mathrm{d}r_{\text{miss}}}{\mathrm{d}b_p} \leq 0$. Moreover, because clusters are prefetched in order of descending likelihood, we assume r_{miss} is either linear or a concave up function[1] of b_p, *i.e.*, $\frac{\mathrm{d}^2 r_{\text{miss}}}{\mathrm{d}b_p^2} \geq 0$.

We now examine two cases:

- **Case 1:** $b_p^* \leq B \cdot t_{\text{LLM}}$. Here, $t_1 = t_{\text{LLM}}$ is constant because prefetching is fully overlapped with LLM generation. Since increasing b_p in this regime will not increase t_1 and cannot worsen the miss rate, pushing b_p to the boundary $B \cdot t_{\text{LLM}}$ minimizes $t_1 + t_2$. Hence,

$$b_p^* = B \cdot t_{\text{LLM}}. \tag{4.3}$$

- **Case 2:** $b_p^* > B \cdot t_{\text{LLM}}$. In this region, t_1 grows linearly with b_p, and we have: $\frac{\mathrm{d}^2}{\mathrm{d}b_p^2}(t_1 + t_2) = \frac{\mathrm{d}^2 r_{\text{miss}}}{\mathrm{d}b_p^2} \cdot n_{\text{probe}} \cdot t_{cc} \geq 0$. Therefore, $t_1 + t_2$ is concave up, allowing at most one minimum. At the minimum point, we have:

$$\frac{\mathrm{d}}{\mathrm{d}b_p}(t_1 + t_2) = 0 \implies \frac{1}{B} + \frac{\mathrm{d}r_{\text{miss}}}{\mathrm{d}b_p} \cdot n_{\text{probe}} \cdot t_{cc} = 0. \tag{4.4}$$

From this, we obtain:

$$b_p^* = B \times n_{\text{probe}} \times t_{cc} \times \Delta r_{\text{miss}}, \tag{4.5}$$

where Δr_{miss} is the decrement of the miss rate for this round. If b_p^* is indeed larger than $B \cdot t_{\text{LLM}}$, it becomes the global minimum; otherwise, the solution reverts to Case 1.

In summary, our analysis shows that the optimal prefetch amount b_p^* can only lie at one of two points: (1) Prefetch until LLM generation completes (i.e. $b_p = B \cdot t_{\text{LLM}}$). (2) A point determined by Eq. 4.5.

[1] An upward U-shaped function whose second derivative is positive.

However, under typical CPU-GPU bandwidth (*e.g.*, 55 GB/s on PCIe 5), the time spent loading additional clusters often outweighs any retrieval latency reduction from lowering r_{miss}. Consequently, the second scenario in Eq. 4.5 becomes nearly infeasible in practice. Therefore, on current hardware, *prefetching exactly until LLM execution ends* is generally the most effective choice.

Profiling-guided approach. Although b_p^* can be derived from the analysis above, it depends on knowing t_{LLM} for each query, which cannot be obtained ahead of time. To address this, we use a *profiling-guided approach*, leveraging the observation that, despite differences in query content, the output length (and thus generation time) for a RAG pipeline often remains similar across most queries. Accordingly, for each RAG pipeline, we can measure t_{LLM} on a calibration set containing n queries, and get estimated $\hat{b}_p^* = B \cdot \bar{t}_{\text{LLM}}$, where $\bar{t}_{\text{LLM}} = \text{mean}\{t_{\text{LLM},1}, t_{\text{LLM},2}, \ldots, t_{\text{LLM},n}\}$. This estimated $\hat{b}_p^*$ can then be used for incoming queries, ensuring a near-optimal prefetch amount.

Design Details

Sorting Optimization. TELERAG optimizes the final k-argmin sorting step of the IVF search by leveraging the GPU. Since each cluster can have thousands of data points, sorting benefits from the compute power of GPUs [22]. Hence, TELERAG transfers the distance values computed by the CPU for C_{miss} to the GPU. Since this is a scalar value, unlike the vector data of the cluster, it incurs minimal overhead. The GPU then performs a fast global sort on the combined distances from C_{overlap} and C_{miss}, achieving end-to-end acceleration often lacking in CPU-only retrieval implementations.

Prefetch Target. TELERAG's prefetching mechanism targets a specific number of bytes (b_p) rather than the number of clusters to ensure more predictable performance. Since IVF cluster sizes are often highly uneven, targeting a fixed cluster count can lead to unstable data loading times, whereas a byte limit provides a clearer upper bound on the transfer duration based on bandwidth.

When selecting clusters to meet the prefetch byte budget (b_p), the system adds whole clusters sequentially based on proximity to the query. If adding the next closest cluster would exceed the budget, it is omitted entirely rather than being partially loaded, ensuring a clean division where each cluster is processed exclusively by either GPU or CPU.

Multi-Round Support. For multi-round RAG involving the same input query, the system performs

a full prefetch (up to the budget) only in the first round, leveraging high cluster similarity across rounds. In subsequent rounds, it incrementally fetches only the additional required clusters that were not loaded previously, optimizing data transfer while respecting the memory budget.

Batch Support. For batch inference, a challenge for lookahead retrieval is that each query requires a different set of IVF clusters. In our design, we fix the prefetching budget to the amount we got from §4.4 irrespective of batch size, and equally distribute the budget to each query in a batch. As a result, although the average cluster hit rate per request will decrease for larger batch sizes, we can ensure each query's retrieval search time is equally accelerated, forming a balanced batched retrieval.

Multi-GPU support. For multi-GPU systems, we perform a greedy search based on embedding similarities of queries to group similar queries into mini-batches, and distribute the mini-batches among GPUs. The greedy search is efficient and incurs negligible overhead: In our profiling for up to the batch size of 256, the latency to perform greedy search is less than 0.1s, which is minimal compared to the end-to-end latency of each batch.

Implementation Details

TELERAG is implemented in Python and also leverages PyTorch's [21] operator ecosystem for efficient computation. The datastore index is initially constructed using FAISS [69], and its data structures, such as IVF centroids and cluster data, are converted to PyTorch tensors.

At runtime, cluster data is loaded into a contiguous pinned memory region on the CPU, enabling non-blocking memory copies to the GPU. A fixed-size contiguous buffer on the GPU is allocated based on the user's configuration or GPU memory capacity during runtime.

To enable concurrent CPU-GPU data transferring and LLM generation, we utilize PyTorch's `_copy(non_blocking=True)` API and use separate CUDA streams for prefetching and LLM. In this way, the data copy operations will not block the GPU to perform computation, and thus the TELERAG can do prefetching in parallel to the pre-retrieval LLM generation.

To implement the index search with GPU-CPU cooperation, for the GPU part, we use a single matrix-vector multiplication that computes distances for all prefetched vectors; for the CPU part, we utilize `multithreading` in Python to parallelize similarity searches across clusters. We then move the

Specification	Value
Dataset	wiki_dpr [140]
Dataset size	2.1 billion tokens
# of chunks	21 million
# of IVF cluster	4096
Embed model	Contriever [120]
Embed dimension	768
Index type	`FLAT`[2]
Distance metric	`Inner Product`
Index size	61GB

Table 4.2: Detailed configurations of our retrieval index.
distances computed from CPU to GPU, merge with distances on GPU and perform global sorting on GPU.

4.5 Evaluation

We conduct extensive experiments to evaluate the effectiveness of TELERAG. In this section, we describe the necessary details on how we set up the evaluations, present experimental results, and provide in-depth analysis and discussions.

Evaluation Datasets and RAG Models

Datastore. We build a datastore based on the wiki_dpr dataset [140], a popular dataset that contains 2.1 billion tokens from Wikipedia. Following previous works [24; 140; 191], we chunk the passages by every 100 tokens, and use Contriever [120] to generate an embedding for each chunk. The embeddings have a hidden dimension of 768.

Vector index. For the baseline retrieval, we build an IVF vector index using Faiss [69] on the datastore. As described in §4.4, we convert the Faiss index to a customized index in PyTorch for TELERAG. See Table 4.2 for the detailed configurations of our vector index and datastore.

LLMs. We evaluate TELERAG on the Llama model family [251] in three different sizes (Llama-3.2-3B, Llama-3-8B, Llama-2-13B) to represent different use cases.

RAG pipelines. We evaluate TELERAG with six popular RAG pipelines. Figure 4.7 shows the overview

[2]Original embedding without compression for the best retrieval precision.

Setup	Desktop	Server
CPU	Threadripper 5975	EPYC 9554
CPU memory size	512GB	1.5TB
GPU	RTX4090 [202]	H100 [205]
GPU memory size	24GB	80GB
CPU-GPU Bus	PCIe 4	PCIe 5
Bandwidth	32 (24) GB/s	64 (51) GB/s

Table 4.3: Hardware specifications for our setups. In bandwidth, the number in the parentheses is the actual bandwidth we measured from our system.

of each pipeline. Note that although some pipelines lack pre-retrieval generation, the post-retrieval generation serves similar functionality for the retrieval of the next iteration. Below are brief descriptions of the evaluated RAG pipelines.

1. **HyDE** [81] prompts LLM to generate a hypothetical paragraph and perform retrieval based on the embedding of the generated paragraph.

2. **SubQestion (SubQ)**[3] [171] prompts LLM to generate multiple sub-questions and performs retrievals for each generated sub-question.

3. **Iterative (Iter)**[4] [171] prompts LLM to generate narrower questions first and iteratively refine them based on previous answers. At the end of each iteration, it prompts LLM to judge if the answer is good enough.

4. **Iter-RetGen (IRG)** [233] iteratively do retrieval and LLM generation for 3 iterations.

5. **FLARE** [133] iteratively issues retrievals based on the confidence (probability score) of predicted tokens for the upcoming sentence.

6. **Self-RAG (S-RAG)**[5] [24] uses the LLM to judge for retrieval, generate responses, and self-critique on the responses. We use fine-tuned models based on Llama2-7B and Llama2-13B from their official repository [25].

[3]Implemented in LlamaIndex as `SubQuestionQueryEngine`.
[4]Implemented in LlamaIndex as `MultiStepQueryEngine`.
[5]We evaluate on the short-form version which only has one iteration.

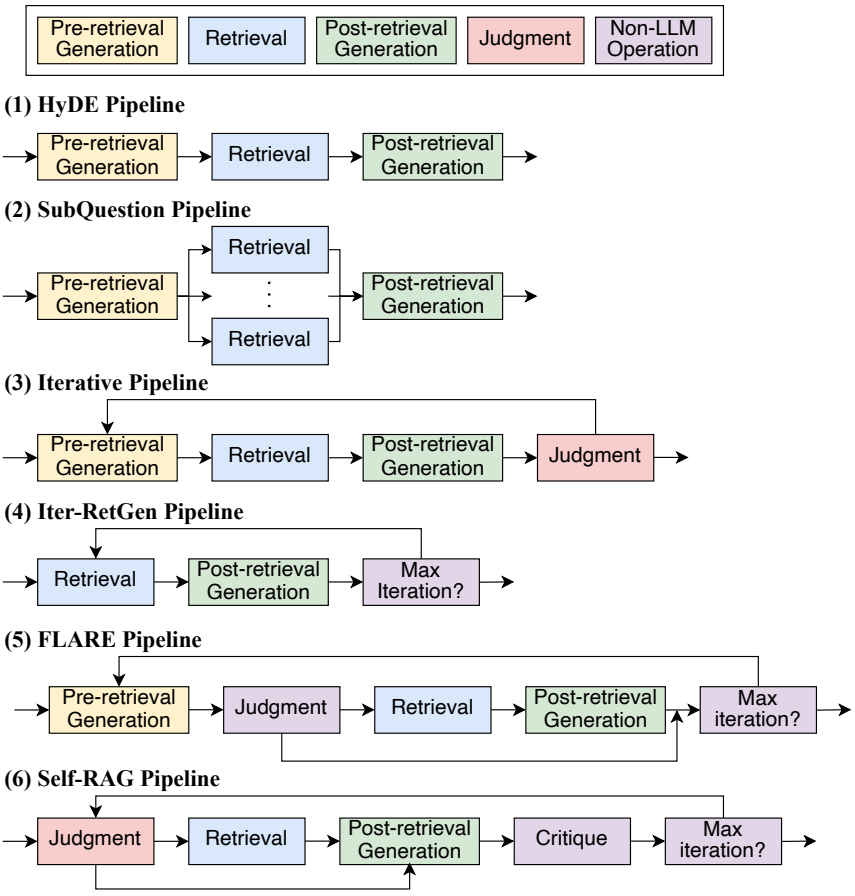
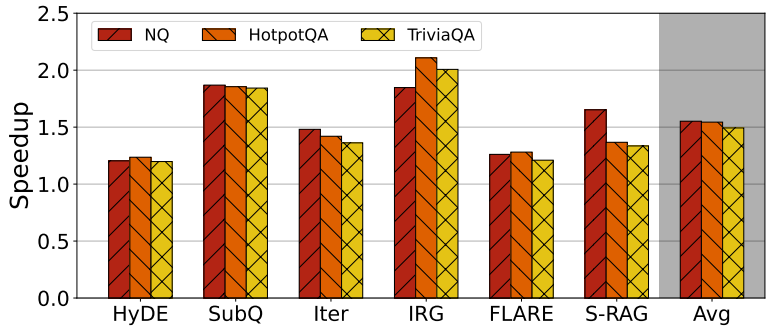


Figure 4.7: Overview of six RAG pipelines that we evaluate.

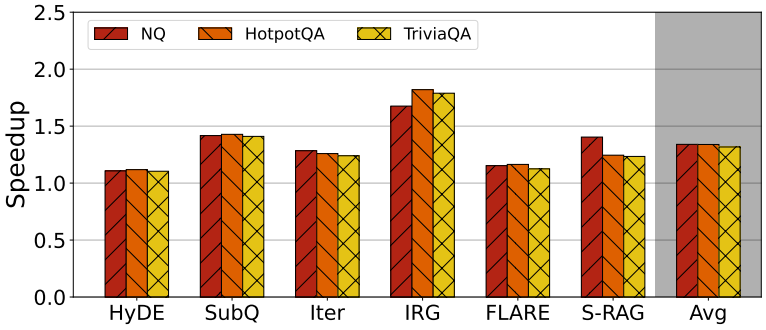
Evaluation Datasets. We use three commonly used question-answering datasets, NQ [155], HotpotQA [287], and TriviaQA [138]. For each dataset, we randomly sample 1024 queries and report the average unless otherwise specified.

Experiment Setups

Hardware setups. We evaluate TELERAG on two hardware environments, `Desktop` and `Server`, which are equipped to represent the settings for the desktop and data center use cases. The `Desktop` has the RTX4090 (24 GB memory), and we use 3B and 8B models. The `Server` has the H100 (80 GB memory), and we use 8B and 13B models. These sizes represent common model sizes for RAG applications [24; 23]. We do not evaluate the 13B model on `Desktop` as its model size (26 GB) exceeds RTX4090's memory capacity. Table 4.3 summarizes the hardware configurations.



(a) End-to-end latency speedup with Llama-3.2-3B.



(b) End-to-end latency speedup with Llama-3-8B.

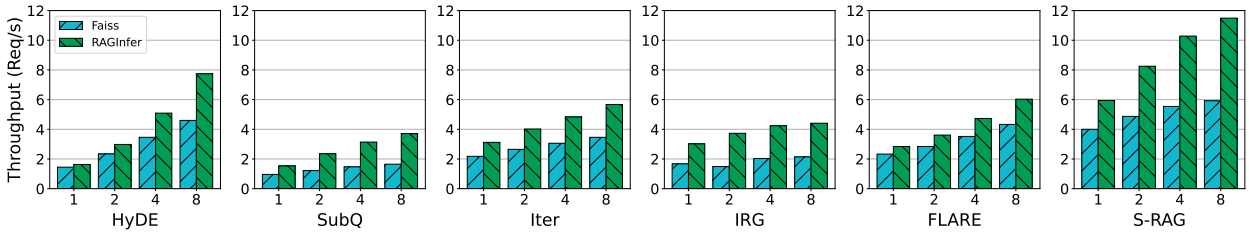
Figure 4.8: End-to-end latency speedup of TELERAG and baseline on six RAG pipelines and three datasets, with RTX4090 GPU. `nprobe` is 256.

Nprobe and top-k. For the IVF index, `nprobe` is a hyperparameter that controls the trade-off between search efficiency and retrieval quality. A common heuristic is to set `nprobe` to $4\sqrt{N_c}$ [322]. Given our index size of $N_c = 4096$, we use `nprobe` = 256 ($= 4\sqrt{4096}$) by default, unless otherwise specified. For retrieval, top-k denotes the number of most relevant documents to return. We use top-k = 3, which is a standard choice in RAG.

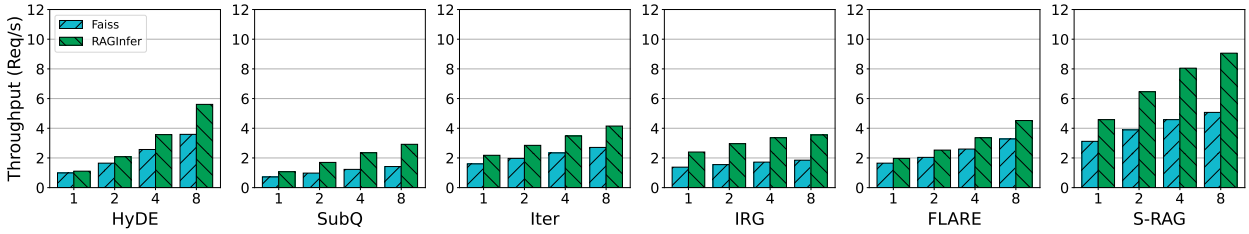
RAG pipeline implementation. We implement the RAG pipelines with the FlashRAG framework [135]. For IRG, FLARE, and S-RAG, we use the framework's default implementations. For the other pipelines, we reimplement them using FlashRAG's APIs.

Benchmark methodology. We follow a benchmarking methodology from [316] and use GPT-3.5-Turbo [206] to run through each pipeline and record the input and output text of each step. During latency evaluation, we set the LLM's output length based on the recorded text. This way, we ensure a fair latency comparison across different LLM models.

Baseline systems. To evaluate the latency of each pipeline, we construct a clean execution flow in



(a) End-to-end throughput with six pipelines and different batch sizes using Llama-3-8B.



(b) End-to-end throughput with six pipelines and different batch sizes using Llama-2-13B.

Figure 4.9: End-to-end throughput of TELERAG and the baseline across six RAG pipelines on the NQ dataset using Llama-3-8B and Llama-2-13B at different batch sizes on a H100 GPU. The `nprobe` is set to 256, and the x-axis represents the batch size.

Python that only contains LLM generation, datastore retrieval, and other necessary logical operations to fulfill each pipeline. For LLM generation, we use SGLang [316], which is a state-of-the-art LLM inference engine. For retrieval, we use a popular retrieval library, Faiss[69], as the CPU-offloaded baseline. [6]

Prefetching budget setups. Based on the methodology we described in §4.4, we profile each RAG pipeline with a small amount of 64 random samples from NQ [155] and derive the prefetching budget of each pipeline.

Max prefetching memory. We set a maximum GPU memory limit for prefetching in each configuration. For `Server`, we allocate up to 16 GB. For `Desktop`, we allocate up to 10 GB and 3.75 GB for the 3B and 8B models, respectively. These settings demonstrate that TELERAG can efficiently operate using only a small fraction (up to 40% for RTX4090 and 20% for H100) of total GPU memory.

Evaluation Results

Single-query latency on RTX4090. We evaluate the end-to-end RAG latency for single queries on `Desktop` equipped with RTX4090, representing a typical local scenario. Figure 4.8 shows the latency

[6] This baseline resembles frameworks like LlamaIndex and LangChain, which primarily function as wrappers around LLM serving and retrieval libraries. We build a system from the ground up to avoid the overhead introduced by their complex layers designed for advanced functionality.

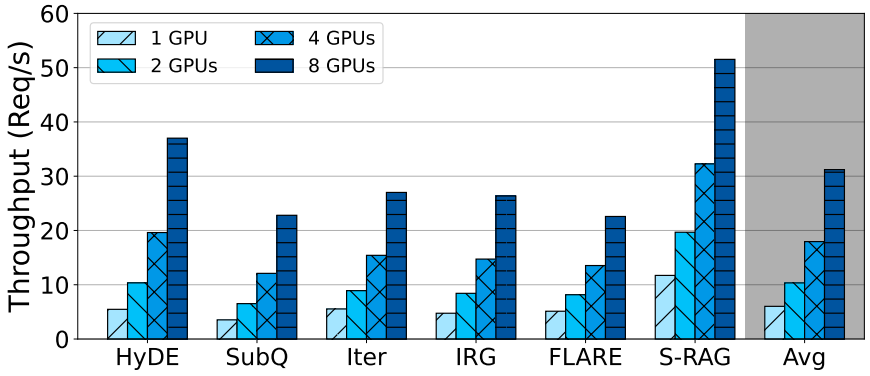


Figure 4.10: Simulated throughput[7] of TELERAG on NQ dataset with different number of H100 GPUs.

reduction of TELERAG across three datasets and two LLMs (Llama-3.2-3B and Llama-3-8B).

As shown in Figure 4.8, TELERAG consistently outperforms the baseline across all tested configurations. With Llama-3.2-3B, TELERAG achieves average speedups of $1.55\times$, $1.54\times$, and $1.49\times$ on NQ, HotpotQA, and TriviaQA, respectively. The best speedup of $2.11\times$ is achieved in the Iter-RetGen pipeline on HotpotQA, because this pipeline involves frequent retrieval operations and generally short LLM outputs, which enhances the relative impact of retrieval acceleration.

Another notable improvement is in the SubQuestion pipeline, where TELERAG achieves approximately $1.85\times$ speedup across all datasets. This pipeline uses LLM-generated sub-questions and performs batched retrievals of 3 to 4 queries simultaneously. CPU-based retrieval suffers from limited parallelism in such scenarios, but TELERAG efficiently utilizes GPU parallelism, significantly enhancing performance.

When deploying Llama-3-8B, speedups with TELERAG slightly decrease compared to Llama-3.2-3B, primarily due to increased LLM latency and reduced available memory for prefetching. Still, TELERAG achieves approximately 1.3x speedup across datasets, with a peak improvement of $1.82\times$ for Iter-RetGen pipeline on HotpotQA. Achieving these results with only 3.75 GB of remaining GPU memory (after accounting for Llama-3-8B (16 GB), the embedding model (1 GB), and other miscellaneous tensors) highlights TELERAG's robust capability in accelerating RAG inference even under tight GPU memory constraints.

Multi-query throughput on H100. To evaluate the TELERAG's performance in batched inference, we evaluate the end-to-end throughput on `Server` (H100) in batch sizes 1, 2, 4, and 8. We show the results of six evaluated RAG pipelines with Llama-3-8B and Llama-2-13B in Figure 4.9.

[7] Results for 1 GPU are real, not simulated.

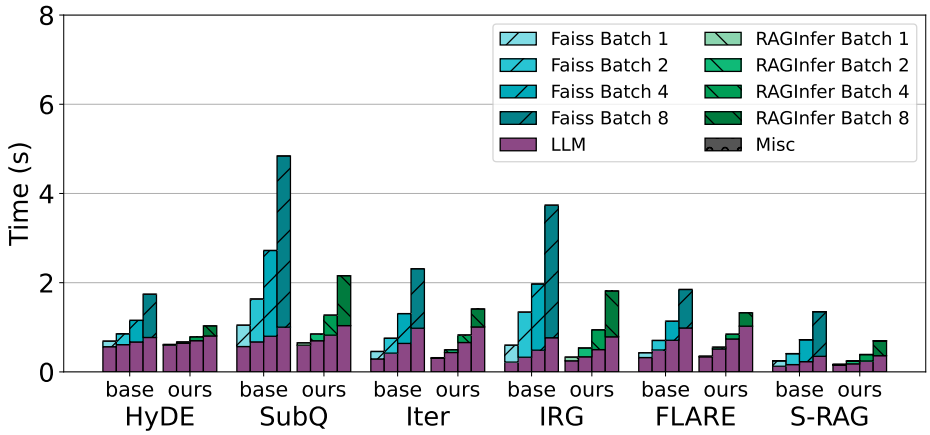


Figure 4.11: Latency breakdown for Llama-3-8B on NQ with a H100 GPU in different batch sizes. `nprobe` is 256.

As shown in Figure 4.9, TELERAG consistently outperforms the Faiss baseline across all pipelines and batch sizes in both LLM sizes. At batch size 1 and running with Llama-3-8B (equivalent to the single-query setting), TELERAG delivers a throughput increase of $1.1\times$ to $2.2\times$, with an average of $1.46\times$. As the batch size increases, the performance gains of TELERAG continually grow. The Faiss baseline demonstrates near-linear scalability up to a batch size of 4, but reaches a noticeable plateau at a batch size of 8, indicating that the CPU baseline's capacity to handle a high volume of simultaneous queries is limited. In contrast, TELERAG continues to scale nearly linearly through the largest batch evaluated, reaching 1.4–$2.2\times$ higher throughput and an average throughput increase of $1.83\times$ at batch 8. The biggest throughput gain is achieved for SubQuestion, up to $2.2\times$ at batch 8. This is again due to their higher demands on the retrieval, leaving higher optimization space for TELERAG.

Multi-GPU Throughput. We also evaluate TELERAG in a multi-GPU system to further show the scalability. For the evaluation, we construct a simulation framework around our single-GPU implementation. We first define a global batch size that the whole system will process at once, and then we construct mini-batches to distribute among each GPU. To form a mini-batch, we perform a greedy search based on the embedding similarity of queries, grouping similar queries into the same mini-batch to maximize the cluster hit rate. We present the simulated throughput of TELERAG with a global batch size of 128 and mini-batch size of 4 with 1 to 8 GPUs. As shown in Figure 4.10, TELERAG's throughput scales well with the number of GPUs. When compared to 1 GPU results, the averaged throughput enhancements of TELERAG are $1.7\times$,

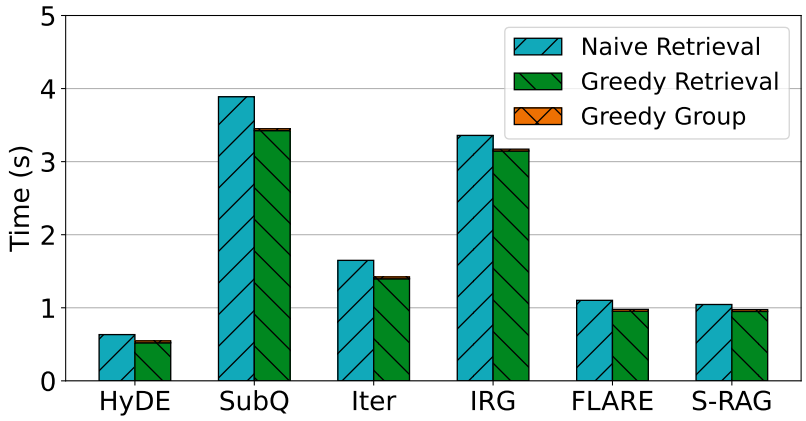


Figure 4.12: Comparison of end-to-end retrieval latency for two batching strategies: naive mini-batching and similarity-aware greedy grouping.

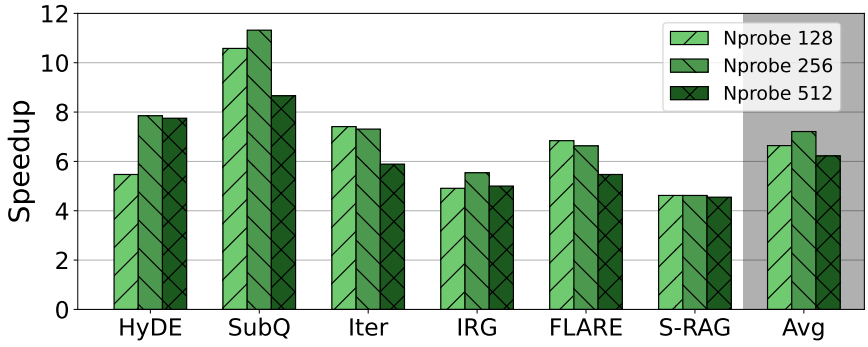
3.1×, and 5.4× with 2, 4, and 8 GPUs, respectively.

Analysis and Sensitivity Study

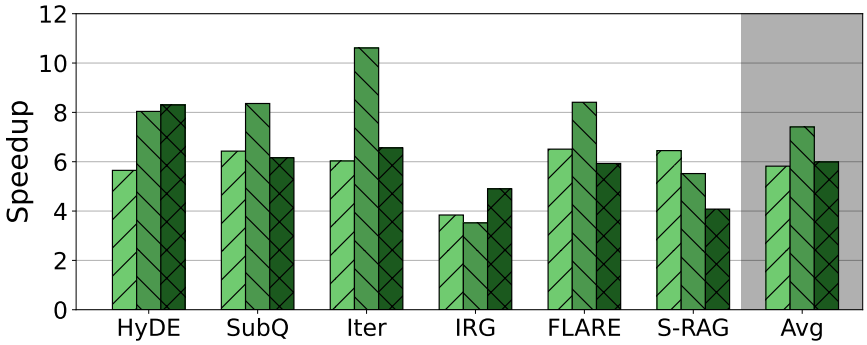
Latency breakdown. We further show the latency breakdown of running RAG pipelines with Llama-3-8B and Llama-2-13B on a single H100 GPU in different batch sizes in Figure 4.11. From Figure 4.11, we can observe that LLM latency grows sub-linearly with larger batch sizes. However, the latency for Faiss retrieval on CPU grows linearly with the batch size, dominating the overall latency when batch size is large. These results echo our findings in Figure 4.9, and show limited scalability of CPU retrieval in serving scenario. In contrast, TELERAG significantly accelerates across all batch sizes and achieves a higher speedup from 1.4× to 1.8× when the batch size increases from 1 to 8.

Ablation on mini-batch strategies. In our multi-GPU results, we implement a greedy grouping strategy for mini-batching based on queries' similarity in a global batch. Here, we examine the benefits and overhead of this strategy in Figure 4.12, where we show the end-to-end retrieval latency for mini-batch size 4 with four GPUs and a global batch of 128 queries. The grouping overhead is minimal and the overall retrieval latency of greedy grouping mini-batching consistently outperform naive mini-batching across all pipelines.

Retrieval speedups across `nprobe`. Figure 4.13 shows the retrieval latency reduction on NQ. We observe consistent speedups for all `nprobe` values. The greatest speedups are achieved at `nprobe` 256, with average speedups of 7.21× and 7.41× on RTX4090 and H100, respectively. As a larger `nprobe`



(a) Llama-3.2-3B with a RTX4090 GPU.



(b) Llama-3-8B with a H100 GPU.

Figure 4.13: Single-query retrieval speedup on NQ with different `nprobe` values.

value is used, the retrieval performance of TELERAG becomes constrained by missed clusters on the CPU, given our fixed prefetch budget across varying `nprobe` values. However, RAG inference with higher `nprobe` will result in longer latency spent on the retrieval and hence, TELERAG still has a significant latency improvement against the CPU retrieval baseline.

Prefetch budgets and cluster hit rates. Table 4.4 shows the prefetch budgets we set with the profile-guided approach on RTX4090 and H100 for NQ. It also presents the average cluster hit rate achieved with this prefetch budget. From the table, we can see that it generally achieves a high cluster hit rate (>50%) when TELERAG has a large prefetching budget. For cases where the budget is less than 2 GB, we observe a relatively lower hit rate (<50%), limiting the benefits of reducing the CPU's search workloads. However, as observed from Figure 4.8, achieves from 1.2× to 1.6× end-to-end speedups for these pipelines, thanks to the combined benefit of reducing CPU workload and utilizing the GPU to perform sorting on similarity distances.

Pipeline	H100 (Llm3-8B) Budget	H100 (Llm3-8B) Hit Rate	4090 (Llm3-3B) Budget	4090 (Llm3-3B) Hit Rate
HyDE	9 GB	91.95%	7 GB	87.42%
SubQ	8 GB	79.04%	7 GB	76.38%
Iter	5 GB	95.51%	3 GB	84.59%
IRG	4 GB	59.52%	2.5 GB	50.34%
FLARE	6 GB	79.35%	3 GB	56.67%
S-RAG	3 GB	71.29%	1.25 GB	29.96%

Table 4.4: The prefetch budget and corresponding averaged cluster hit rate for each pipeline and hardware setup on NQ dataset. The target retrieval `nprobe` is 256.

4.6 Related Work

Systems for RAG

RAGCache [134] proposes a caching system that stores KV caches from datastore chunks in an order-aware manner to improve the time to the first token. This approach only reduces prefill latency, leaving retrieval and decode times unaffected, despite the fact they typically dominate total latency [13]. Moreover, it assumes repeated use of the same document across multiple requests, limiting its scalability over large data stores. Similarly, TurboRAG [180] precomputes KV caches from the data store, but it also only optimizes prefill latency. CacheBlend introduces a selective KV-cache fusion technique for RAG to reuse pre-computed caches [288].

RaLMSpec [310] proposes speculative retrieval and batched verification, Chameleon [131] proposes a CPU-GPU-FPGA heterogeneous architecture for accelerating the retrieval process, and PipeRAG [132] is an algorithm-system co-design technique to overlap the retrieval and generation by modifying the algorithms. However, these works focus on the paradigm of RAG that retrieves documents once every few tokens and do not apply to modular RAG applications that are widely used now, as discussed in §4.2.

Speculative RAG [268] introduces drafting by smaller LLMs to reduce RAG latency, but it does not target the retrieval latency. APIServe (InferCept) [12] proposes a novel KV cache management strategy that can support the interception of LLM generation by other workloads, including retrieval. However,

this work again does not focus on optimizing retrieval latency. EdgeRAG [230] reduces the memory requirement of retrieval by re-generating and caching document embeddings at runtime, targeting extremely resource-constrained environments. RAGO [130] introduces an abstraction for RAG to automatically pick task placement, resource allocation, and batching policies.

Unlike all these prior works, we tackle system challenges of long retrieval latency and large memory requirement for modular RAG pipelines with large-scale datastores.

Systems for Compound LLM Applications

Apart from RAG, there is a growing interest in compound or agentic LLM applications, where multiple LLM calls and other applications are combined to serve complex functionalities [301; 33; 263]. LLM-Compiler [146] is a framework that optimizes the execution of multiple functions in large language models by enabling parallel function calling. AI Metropolis [283] accelerates LLM-based multi-agent simulations with out-of-order execution. RAG is a specific type of application in this broader direction, and we propose systems techniques to optimize its execution latency, focusing on the characteristics of retrieval workload.

Vector Index

Vector index is a key component of RAG [36; 84], and many works have been proposed to improve their efficiency. ScaNN proposes an anisotropic quantization method for better efficiency [95]. DiskANN and SPANN propose memory-disk hybrid indexing systems that work beyond the limitation of memory capacity [48; 125]. Other prior works propose hardware acceleration of vector index with GPUs [136], FPGAs [303; 129], TPU [51], or ray tracing hardware [320; 177]. BANG proposes a method to scale graph-based ANN beyond GPU memory [141], and Rummy allows the index to scale beyond GPU memory capacity with reordered pipelining [307]. These methods either require algorithm modifications or are bottlenecked by CPU-GPU bandwidth. Our proposal, focuses on the context of modular RAG applications, where queries for retrieval are usually generated by LLMs, and optimizes the latency and the GPU memory consumption without altering the algorithm of the IVF index.

4.7 Conclusion

In this paper, we introduced TELERAG, an inference system that tackles the system challenges of RAG pipelins under latency-sensitive scenarios. By using lookahead retrieval, which overlaps data transfer and GPU computation for faster retrieval, a profile-guided approach to determine optimal prefetching data amount, and GPU-CPU cooperation, TELERAG speeds up the end-to-end latency with minimal GPU memory requirement. Our evaluation shows that TELERAG significantly improves performance compared to existing state-of-the-art solutions.

Remarks on Author Contributions

My major contributions are as follow:

- Brainstormed the core ideas and designed system with Keisuke Kamahori.
- Implemented the RAG pipelines frontend the TeleRAG system backend.
- Led the evaluations on the pipeline accuracy and IVF overlapping rates.
- Led the performance evaluations.
- Led the design and implementation of batching and multi-GPU systems.
- Led and co-authored the paper writing.

Part II

Efficient Machine Learning Systems with Compression

Chapter 5

SPIN: An Empirical Evaluation on Sharing Parameters of Isotropic Networks

Remarks on Chapter Material

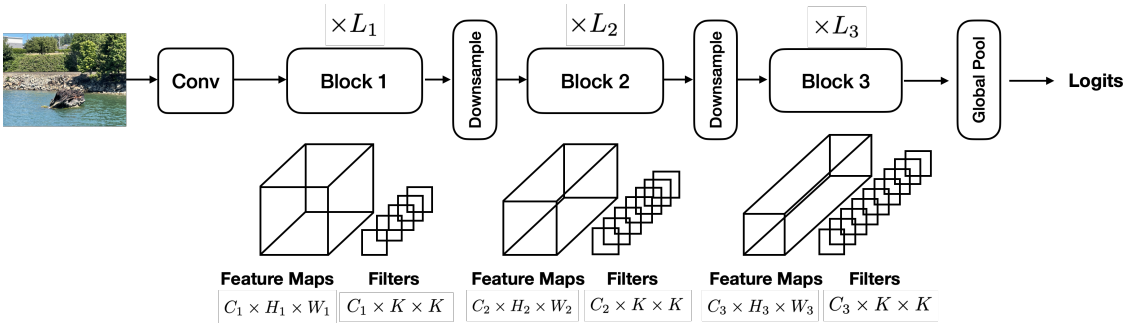
The content of this chapter is adapted from the following paper (* indicates eqaul contributions). The work is done during an internship with Apple.

- Chien-Yu Lin*, Anish Prabhu*, Thomas Merth, Sachin Mehta, Anurag Ranjan, Maxwell Horton, Mohammad Rastegari, "SPIN: An Empirical Evaluation on Sharing Parameters of Isotropic Networks", In Proceedings the 17th European Conference on Computer Vision (ECCV), 2022.

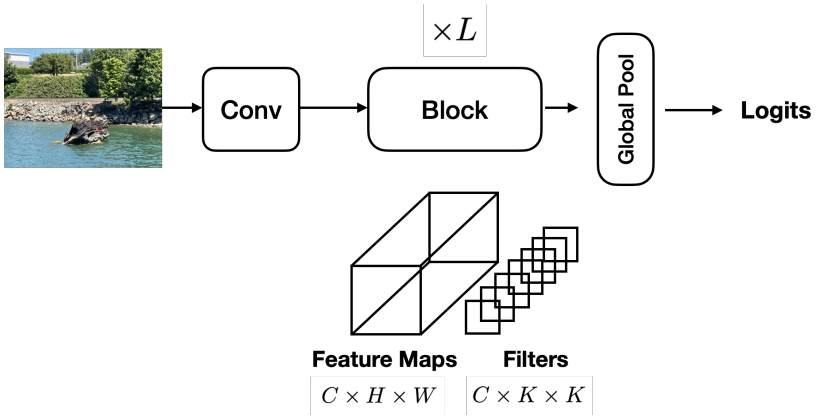
5.1 Introduction

Isotropic neural networks have the property that all of the weights and intermediate features have identical dimensionality, respectively (see Figure 5.1). Some notable convolutional neural networks (CNNs) with isotropic structure [255; 176] have been proposed recently in the computer vision domain, and have been applied to different visual recognition tasks, including image classification, object detection, and action recognition. These isotropic CNNs contrast with the typical "hierarchical" design paradigm, in which spatial resolution and channel depth are varied throughout the network (e.g., VGG [239] and ResNet [103]).

The Vision Transformer (ViT) [68] architecture also exhibits this isotropic property, although softmax



(a) Architecture of regular CNNs.



(b) Architecture of isotropic CNNs.

Figure 5.1: Basic architectures of regular and isotropic CNNs. (a) Regular CNNs vary the shape of intermediate features and weight tensors in the network while (b) isotropic CNNs fix the shape of all intermediate features and weight tensors in the network.

self-attention and linear projections are used for feature extraction instead of spatial convolutions. Follow-up works have experimented with various modifications to ViT models (e.g. replacing softmax self-attention with linear projections [250], factorized attention [305], and non-learned transformations [262]); however, the isotropic nature of the network is usually retained.

Recent isotropic models (e.g., ViT [68], ConvMixer [255], and ConvNext [176]) attain state-of-the-art performance for visual recognition tasks, but are computationally expensive to deploy in resource constrained inference scenarios. In some cases, the parameter footprint of these models can introduce memory transfer bottlenecks in hardware that is not well equipped to handle large amounts of data (e.g. microcontrollers, FPGAs, and mobile phones) [159]. Furthermore, "over-the-air" updates of these large models can become impractical for continuous deployment scenarios with limited internet bandwidth. Parameter (or

weight) sharing[1], is one approach which compresses neural networks, potentially enabling the deployment of large models in these constrained environments.

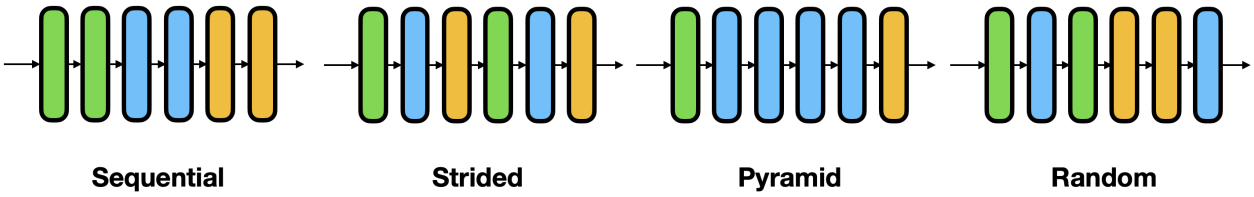
Isotropic DNNs, as shown Figure 5.1, are constructed such that a layer's weight tensor has identical dimensionality to that of other layers. Thus, cross-layer parameter sharing becomes a straightforward technique to apply, as shown in ALBERT [159]. On the other hand, weight tensors within non-isotropic networks cannot be shared in this straightforward fashion without intermediate weight transformations (to coerce the weights to the appropriate dimensionality). In Appendix A, we show that the search space of possible topologies for straightforward cross-layer parameter sharing is significantly larger for isotropic networks, compared to "multi-staged" networks (an abstraction of traditional, non-isotropic networks). This rich search space requires a comprehensive exploration. Therefore, in this paper, we focus on isotropic networks, with the goal of finding practical parameter sharing techniques that enable high-performing, low-parameter neural networks for visual understanding tasks.

To extensively explore the weight sharing design space for isotropic networks, we experiment with different orthogonal design choices (Section 5.3). Specifically, we explore (1) different sharing topologies, (2) dynamic transformations, and (3) weight fusion initialization strategies from pretrained non-sharing networks. Our results show that parameter sharing is a simple and effective method for compressing large neural networks versus standard architectural scaling approaches (e.g. reduction of input image size, channel size, and model depth). Using a weight sharing strategy discovered from our design space exploration, we achieve nearly identical accuracy (to non-parameter sharing, iso-FLOP baselines) with significantly reduced parameter counts. Beyond the empirical accuracy versus efficiency experiments, we also investigate network representation analysis (Section 5.5) and model generalization (Appendix F) for parameter sharing isotropic models.

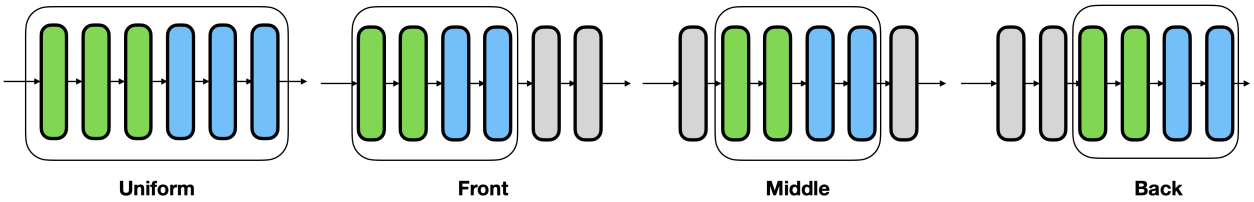
5.2 Related Works

Cross-layer Parameter Sharing. Cross-layer parameter sharing has been explored for both CNN- and Transformer-based models [153; 154; 241; 143; 62; 159; 245]. For instance, Kim et al. [143] applies cross-layer parameter sharing across an entire heterogeneous CNN. However, they share weights at the granularity

[1]We interchangeably use the terms parameter and weight sharing throughout this paper.



(a) Sharing mapping.



(b) Sharing distribution.

Figure 5.2: Sharing topologies. In (a), sharing mapping determines which layers share the same weights while in (b), sharing distribution determines how the weight sharing layers are distributed in the network. Layers with the same color share weights. Layers outside of the sharing section do not share weights. Best viewed in color.

of filters, whereas we share weights at the granularity of layers. In terms of our framework, Kubilius et al. [153] experiments with Uniform-Strided, proposing a heterogeneous network based off of the human visual cortex. With isotropic networks, we can decouple parameter sharing methods from the constraints imposed by heterogeneous networks. Thus, we expand the scope of weight sharing structures from their work to isotropic networks.

Cross-layer parameter sharing is explored for isotropic Transformer models for the task of neural language modeling [62; 159] and vision [245]. Lan et al. [159] experiments with Uniform-Sequential, and Dehghain et al. [62] experiments with universal sharing (i.e. all layers are shared). Takase et al. [245] experiments with 3 strategies, namely Uniform-Sequential, Uniform-Strided, and Cycle. In this paper, we extend these works by decomposing the sharing topology into combinations of different sharing mappings (Figure 5.2a) and sharing distributions (Figure 5.2b).

Dynamic Recurrence for Sharing Parameters. Several works [32; 161; 94; 236] explore parameter sharing through the lens of dynamically repeating layers. However, each technique is applied to a different model architecture, and evaluated in different ways. Thus, without a common framework, it's difficult to get a comprehensive understanding of how these techniques compare. While this work focuses only on static weight

sharing, we outline a framework that may encompass even these dynamic sharing schemes. In general, we view this work as complementary to explorations on dynamic parameter sharing, since our analysis and results could be used to help design new dynamic sharing schemes.

5.3 Sharing Parameters in Isotropic Networks

In this section, we first motivate why we focus on isotropic networks for weight sharing (Section 5.3), followed by a comprehensive design space exploration of methods for weight sharing, including empirical results (Section 5.3).

Why Isotropic Networks?

Isotropic networks, shown in Figure 5.1b, are simple by design, easy to analyze, and enable flexible weight sharing, as compared to heterogeneous networks.

Simplicity of Design. Standard CNN architectural design, whether manual [103; 228]) or automated through methods like neural architecture search [246; 108]), require searching a complex search space, including what blocks to use, where and when to downsample the input, and how the number of channels should vary throughout the architecture. On the other hand, isotropic architectures form a much simpler design space, where just a single block (e.g., attention block in Vision Transformers or convolutional block in ConvMixer) along with network's depth and width must be chosen. The simplicity of implementation for these architectures enables us to more easily design generic weight sharing methods across various isotropic architectures. The architecture search space of these networks is also relatively smaller than non-isotropic networks, which makes them a convenient choice for large scale empirical studies.

Increased Weight Sharing Flexibility. Isotropic architectures provide significantly more flexibility for designing a weight sharing strategy than traditional networks.

We define the *sharing topology* to be the underlying structure of how weight tensors are shared throughout the network. Suppose we have an isotropic network with $L \geq 1$ layers and a weight tensor "budget" of $1 \leq P \leq L$. The problem of determining the optimal sharing topology can be seen as a variant of the set

cover problem; we seek a set cover with no more than B disjoint subsets, which maximizes the accuracy of the resulting network. More formally, a possible sharing topology is an ordered collection of disjoint subsets $\mathcal{T} = (\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_P)$, where $\cup_{i=1}^{B} \mathcal{S}_i = \{1, 2, ..., L\}$ for some $1 \leq P \leq L$. We define $\frac{L}{P}$ to be the *share rate*.

We characterize the search space in Appendix A, showing that isotropic networks support significantly more weight sharing topologies than heterogeneous networks (when sharing at the granularity of weight tensors). This substantially increased search space may yield more effective weight sharing strategies in isotropic networks than non-isotropic DNNs, a reason why we are particularly interested in isotropic networks

Cross-layer Representation Analysis. To better understand if the weights of isotropic architectures are amenable to compression through weight sharing, we study the representation of these networks across layers. We hypothesize that layers with similar output representations will be more compressible via weight sharing. To build intuition, we use Centered Kernel Alignment (CKA) [151], a method that allows us to effectively measure similarity across layers.

Figure 5.3 shows the pairwise analysis of CKA across layers within the ConvMixer network. We find significant representational similarity for nearby layers. This is not unexpected, given the analysis of prior works on iterative refinement in residual networks [124]. Interestingly, we find that CKA generally peaks in the middle of the network for different configurations of ConvMixer. Overall, these findings suggest that isotropic architectures may be amenable to weight sharing, and we use this analysis to guide our experiments exploring various sharing topologies in Section 5.3.

Weight Sharing Design Space Exploration

When considering approaches to sharing weights within a neural network, there is an expansive design space to consider. This section provides insights as well as empirical evaluation to help navigate this design space. We first consider the weight sharing topology. Then, we introduce lightweight dynamic transformations on the weights to increase the representational power of the weight-shared networks. Finally, we explore how to use the trained weights of an uncompressed network to further improve accuracy in weight-sharing isotropic networks. All experiments done in this section are based on a ConvMixer model with 768 channels, depth of 32, patch extraction kernel size of 14, and convolutional kernel size of 3.

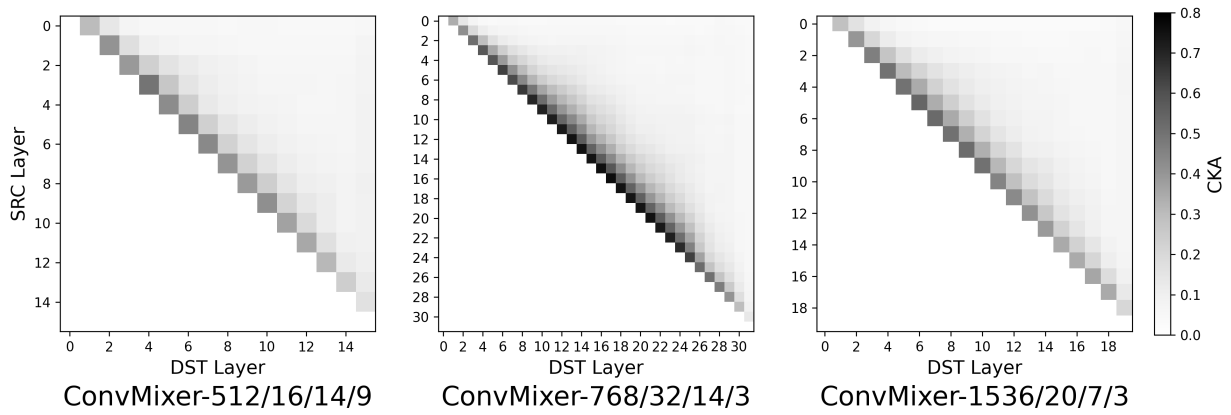


Figure 5.3: CKA similarity analysis on ConvMixer's intermediate feature maps shows that the output feature maps of neighboring layers and especially the middle layers have the highest similarity. Here, we compute the CKA similarity of each layer's output feature maps. The diagonal line and the lower triangle part are masked out for clarity. The CKA for the diagonal line is 1 since they are identical. The CKA for the lower triangle is the mirror of the upper triangle. Best viewed on screen.

Weight Sharing Topologies. Isotropic networks provide a vast design space for sharing topologies. We perform an empirical study of various sharing topologies for the ConvMixer architecture, evaluated on the ImageNet dataset. We characterize these topologies by the (1) *sharing mapping* (shown in Figure 5.2a), which describes the structure of shared layers, and (2) the *sharing distribution* (shown in Figure 5.2b), which describes which subset of layers sharing is applied to. We study the following sharing mappings:

1. **Sequential**: Neighboring layers are shared in this topology. There is motivated by our cross layer similarity analysis in Section 5.3 and Figure 5.3, which suggest that local structures of recurrence may be promising.

2. **Strided**: This topology defines the recurrence on the network level rather than locally. If we consider having P blocks with unique weights, we first run all of the layers sequentially, then we repeat this whole structure L/P times.

3. **Pyramid**: This topology is an extension of Sequential, which has increasingly more shared sequential layers as you approach the center of the network. This is inspired by (1) empirical results in Figure 5.3 that show a similar structure in the layer-wise similarity and (2) neural network compression methods (e.g. quantization and sparsity methods), which leave the beginning and end of the network uncompressed [98; 223].

Table 5.1: Effect of different sharing distributions and mappings on the performance of weight-shared (WS) ConvMixer with a share rate of 2. In order to maintain the fixed share rate 2 for non-uniform sharing distributions (i.e., Middle, Front and Back), we apply sharing to 8 layers with share rate $3\times$ and have 16 independent layers. For Middle-Pyramid, the network is defined as $[4\times1, 1\times2, 2\times3, 2\times4, 2\times3, 1\times2, 4\times1]$, where for each element $N \times S$, N stands for the number of sharing layers and S the share rate for the layer. All experiments were done with a ConvMixer with 768 channels, depth of 32, patch extraction kernel size of 14, and convolutional kernel size of 3.

Network	Sharing Distribution	Sharing Mapping	Params (M)	FLOPs (G)	Top-1 Acc (%)
ConvMixer	-	-	20.46	5.03	75.71
WS-ConvMixer	Uniform	Sequential Strided Random	11.02	5.03	**73.29** 72.80 Diverged
WS-ConvMixer	Middle	Sequential Pyramid	11.02	5.03	73.14 **73.22**
WS-ConvMixer	Front Back	Sequential	11.02	5.03	**73.31** 72.35

4. **Random**: We randomly select which layers are shared within the network, allowing us to understand how much the choice of topology actually matters.

For the sharing distribution, we consider applying (1) **Uniform**, where sharing mapping is applied to all layers, (2) **Front**, where sharing mapping is applied to the front of the network, (3) **Middle**, where sharing mapping is applied to the middle of the network, and (4) **Back**, where sharing mapping is applied to the back of the network. Note that front, middle and back sharing distributions results in a non-uniform distribution of share rates across layers.

Figure 5.2 visualizes different sharing topologies while Table 5.1 shows the results of these sharing methods on the ImageNet dataset. When share rate is 2, ConvMixer with uniform-sequential, middle-pyramid, and front-sequential sharing topology result in similar accuracy (2.5% less than the non-shared model) while other combinations result in lower accuracy. These results are consistent with the layer-wise similarity study in Section 5.3, and suggests that layer-wise similarity may be a reasonable metric for determining which layers to share. Because of the simplicity and flexibility of *uniform-sequential* sharing topology, we use it in the following experiments unless otherwise stated explicitly.

Lightweight Dynamic Transformations on Shared Weights. To improve the performance of a weight shared network, we introduce lightweight dynamic transformations on top of the shared weights for each individual layer. With this, we potentially improve the representational power of the weight sharing network without increasing the parameter count significantly.

To introduce the lightweight dynamic transformation used in this study, we consider a set of N layers to be shared, with a shared weight tensor W_s. In the absence of dynamic transforms, the weight tensor W_s would simply be shared among all N layers. We consider $W_i \in \mathbb{R}^{C\times C\times K\times K}$ to be the weights of the i-th layer, where C is the channel size and K is the kernel size. With a dynamic weight transformation function f_i, the weights W_i at the i-th layer becomes

$$W_i = f_i(W_s) \tag{5.1}$$

The choose f_i to be a learnable lightweight affine transformation that allows us to transform the weights without introducing heavy computation and parameter overhead. Specifically, $f_i(W) = \mathbf{a} * W + \mathbf{b}$ applies a grouped point-wise convolution with weights $\mathbf{a} \in \mathbb{R}^{C\times G}$ and bias $\mathbf{b} \in \mathbb{R}^{C}$ to W, where G is the number of groups. The number of groups, $G \in [1, C]$, can be varied to modulate the amount of inter-channel mixing.

Table 5.2 shows the effect of different number of groups in the dynamic weight transformation on the performance and efficiency (in terms of parameters and FLOPs) of ConvMixer on the ImageNet dataset. As Table 5.2 shows, using $G = 64$, the dynamic weight transformation slightly improves accuracy by 0.07% (from 73.29% to 73.36%) with 7% more parameters (from 11.02M to 11.8M) and 11.9% more FLOPs (from 5.03G to 5.63G). Despite having stronger expressive power, dynamic weight transformation does not provide significant accuracy improvement with under 10% of overhead on number of weights and FLOPs and sometimes even degrading accuracy.

Initializing Weights from Pretrained Non-sharing Networks. Here we consider how we can use the weights of a pretrained, uncompressed network to improve the parameter shared version of an isotropic network. To this end, we introduce transformations on the original weights to generate the weights of the shared network for a given sharing topology. We define $V_j \in \mathbb{R}^{C\times C\times K\times K}$ to be the j-th pretrained weight in the original network, and $u_j \in \mathbb{R}^{C}$ to be the corresponding pretrained bias. The chosen sharing topology

Table 5.2: Effect of affine transformations on the performance of Weight Shared ConvMixer model with a sharing rate of 2. All experiments were done with a ConvMixer with 768 channels, depth of 32, patch extraction kernel size of 14, and convolutional kernel size of 3.

Network	Weight Transformation?	Group Rate	Params (M)	FLOPs (G)	Top-1 Acc (%)
ConvMixer	-	-	20.46	5.03	75.71
WS-ConvMixer		-	11.02	5.03	73.29
	✓	1	11.05	5.04	72.87
	✓	16	11.20	5.17	73.20
	✓	32	11.40	5.31	73.14
	✓	64	11.80	5.63	**73.36**

Table 5.3: Effect of different fusion strategies (Section 5.3) on the performance of ConvMixer. All experiments were done with a ConvMixer with 768 channels, depth of 32, patch extraction kernel size of 14, and convolutional kernel size of 3. All weight sharing ConvMixer models share groups of 2 sequential layers.

Network	Fusion Strategy	Params (M)	FLOPs (G)	Top-1 Acc (%)
ConvMixer	-	20.5	5.03	75.71
WS-ConvMixer	-	10.84	5.03	73.23
	Choose First			74.81
	Mean			74.91
	Scalar Weighted Mean			**75.15**
	Channel Weighted Mean			**75.15**
	Pointwise Convoulution			Diverged

defines a disjoint set cover of the original network's layers, where each disjoint subset maps a group of layers from the original network to a single shared weight layer. Concretely, if the weight W_i is shared among S_i layers $\{i_1, i_2, ..., i_{S_i}\}$ in the compressed network, then we define $W_i = F_i(V_{i_1}, V_{i_2}, ..., V_{i_{S_i}})$, where we can design each F_i. We refer to F as the *fusion strategy*. In all experiments we propagate the gradient back to the original, underlying V_j weights. Importantly, F does not incur a cost at inference-time, since we can constant-fold this function once we finish training.

One simple fusion strategy would be to randomly initialize a single weight tensor for this layer. Note that this is the approach we have used in all previous experiments. We empirically explore the following fusion strategies:

- **Choose First**: In this setup we take the first of the set of weights within the set: $W_i = F_i(V_{i_1}, V_{i_2}, ..., V_{i_S}) = V_{i_1}$. The choice of the first weight (V_{i_1}), rather than any other weight, is arbitrary. Training this method from scratch is equivalent to our vanilla weight sharing strategy.

- **Mean**: We take the average of all the weight tensors within the set, $W_i = \frac{1}{S_i} \sum_{k=1}^{S_i} V_{i_k}$ and $b_i = \frac{1}{S_i} \sum_{k=1}^{S_i} u_{i_k}$.

- **Scalar Weighted Mean**: Same as the average, except each weight tensor gets a learned scalar weighting, $W_i = \frac{1}{S_i} \sum_{k=1}^{S_i} \alpha_{i_k} V_{i_k}$, $\alpha_i \in \mathbb{R}$. We take a simple mean of the bias, just as in the Mean strategy. The idea here is to provide the ability to learn more complex fusions, of which Choose First strategy, and Mean are special cases.

- **Channel Weighted Mean**: Rather than a scalar per layer, each weight tensor has a learned scalar for every filter, $W_i = \frac{1}{S_i} \sum_{k=1}^{S_i} \vec{\alpha_i} V_{i_k}$, $\vec{\alpha_i} \in \mathbb{R}^C$. Again, we take a simple mean of the bias, just as the Mean strategy. This strategy should allow the model to choose filters from specific weight tensors, or learn linear combinations.

- **Pointwise Convolution**: In this transformation, a pointwise convolution is applied to each layers weights, that maps to the same size filter, $W_i = \frac{1}{S_i} \sum_{k=1}^{S_i} A_i * V_{i_k}$, $A_i \in \mathbb{R}^{C \times C}$ .This should allow arbitrary mixing and permutations of the kernels of each layer.

Table 5.3 shows that the Channel Weighted Mean fusion strategy allows us to compress the model by $2\times$ while maintaining the performance of original network. Furthermore, in Section 5.5, we show that weight fusion strategies allow us to learn representations similar to the original network.

5.4 Effect of Parameter Sharing on Different Isotropic Networks on the ImageNet dataset

We evaluate the performance of the parameter sharing methods introduced in Section 5.3 on a variety of isotropic architectures. For more information on the training set-up and details, see Appendix C.

Table 5.4: Weight sharing vs. model scaling for the ConvMixer model on ImageNet. For a fair comparison, we generate models with similar FLOPs and network parameters to our family of weight sharing models using traditional model scaling methods. Weight sharing methods achieve significantly better performance than traditional model scaling. See Table 5.5 for more details on the weight sharing model.

Network (C/D/P/K)	Resolution	Weight Sharing?	Share Rate	Params (M)	FLOPs (G)	Top-1 Acc(%)
768/32/14/3	224		-	20.5	5.03	75.71
576/32/14/3	322		-	11.8	5.92	70.326
768/16/14/3	322		-	10.84	5.32	74.20
768/32/14/3	224	✓	2	11.02	5.03	**75.14**
384/32/14/3	448		-	5.5	5.23	58.83
768/8/14/3	448		-	6.04	5.38	68.31
768/32/14/3	224	✓	4	6.3	5.03	**71.91**
288/32/14/3	644		-	3.25	6.23	40.46
768/4/14/3	644		-	3.63	6.04	57.75
768/32/14/3	224	✓	8	3.95	5.03	**67.19**

Parameter Sharing for ConvMixer

Typically, when considering model scaling, practitioners often vary parameters including the network depth, width, and image resolution, which scale the performance characteristics of the model [247]. In Table 5.4, we show that weight sharing models can significantly outperform baselines with the same FLOPs and parameters generated through traditional scaling alone, for example improving accuracy by roughly 10% Top-1 in some cases. We also show a full family of weight sharing ConvMixer models across multiple architectures in Table 5.5, and find that weight sharing can reduce parameters by over $2\times$ in many architectures while maintaining similar accuracy. These results show that weight sharing, in addition to typical scaling methods, is an effective axis for model scaling.

Parameter Sharing for Other Isotropic Networks

Although our evaluations have focused on ConvMixer, the methods discussed in Section 5.3 are generic and can be applied to any isotropic model. Here, we show results of applying parameter sharing to ConvNeXt [176] and the Vision Transformer (ViT) architecture.

Table 5.5: Weight sharing family of ConvMixer model on ImageNet. Significant compression rates can be achieved without loss in accuracy across multiple isotropic ConvMixer models. We also generate a full family of weight sharing models by varying the *share rate*, which is the reduction factor in number of unique layers for the weight shared model compared to the original. C/D/P/K represents the dimension of channel, depth, patch and kernel of the model. If *weight fusion* is specified, the channel weighted mean strategy described in Section 5.3 is used.

Network (C/D/P/K)	Weight Sharing?	Share Rate	Weight Fusion?	Params (M)	FLOPs (G)	Top-1 Acc(%)
1536/20/7/3		-	-	49.4	48.96	78.03
	✓	2	✓	25.8		**78.47**
	✓	4	✓	14		75.76
	✓	10		6.9		72.27
768/32/14/3		-	-	20.5	5.03	75.71
	✓	2	✓	11.02		**75.14**
	✓	4	✓	6.3		71.91
	✓	8	✓	3.95		67.19
512/16/14/9		-	-	5.7	1.33	67.48
	✓	2	✓	3.63		**65.04**
	✓	4	✓	2.58		59.34
	✓	8		2.05		54.25

Table 5.6: Effect of weight sharing on the ConvNeXt model on ImageNet. WS-ConvNeXxt has 2x less number of parameters but still achieves similar accuracy to the original ConvNeXt model.

Network	Depth	Share Rate	Params (M)	FLOPs (G)	Top-1 Acc(%)
ConvNeXt	18	-	22.3	4.3	78.7
	9	-	11.5	2.2	75.3
WS-ConvNeXt	18	2	11.5	4.3	**78.07**
		4	6.7		76.11
		6	4.3		72.07
		9	3.1		68.75

ConvNeXt. Table 5.6 shows the results of parameter sharing on the ConvNeXt isotropic architecture. With parameter sharing, we are able to compress the model by $2\times$ while maintaining similar accuracy on the ImageNet dataset.

Table 5.7: Share rate and ImageNet accuracy comparison with existing weight sharing methods.

Network	Share Rate	Params (M)	FLOPs (G)	Top-1 Acc(%)
ConvMixer-768/32 [255] (baseline)	-	20.5	5.03	75.71
WS-ConvMixer-768/32-S2 (ours)	**1.86**	**11.02**	5.03	**75.14**
ConvMixer-1536/20 [255] (baseline)	-	49.4	48.96	78.03
WS-ConvMixer-1536/20-S2 (ours)	**1.91**	**25.8**	48.96	**78.47**
ConvNeXt-18 [176] (baseline)	-	22.3	4.3	78.7
WS-ConvNeXt-18-S2 (ours)	**1.92**	**11.5**	4.3	**78.07**
WS-ConvNeXt-18-S4 (ours)	3.33	6.7	4.3	76.11
ResNet-152 [103] (baseline)	-	60	11.5	78.3
IamNN [161]	12	5	2.5-9	69.5
ResNet-101 [103] (baseline)	-	44.54	7.6	77.95
DR-ResNet-65 [94]	1.58	28.12	5.49	78.12
DR-ResNet-44 [94]	2.2	20.21	4.25	77.27
ResNet-50 [103] (baseline)	-	25.56	3.8	76.45
DR-ResNet-35 [94]	1.45	17.61	3.12	76.48
ResNet50-OrthoReg [143]	1.25	20.51	4.11	76.36
ResNet50-OrthoReg-SharedAll [143]	1.6	16.02	4.11	75.65

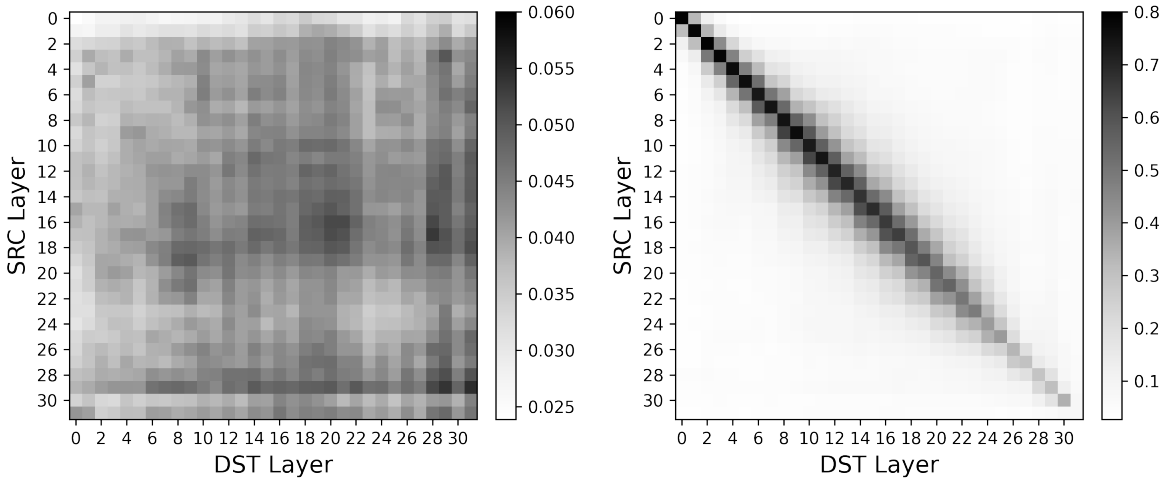
Vision Transformer (ViT). We also apply our weight sharing method to a Vision Transformer, a self-attention based isotropic network. Due to space limit, we report accuracy numbers in Appendix B. Furthermore, we discuss the differences between applying weight sharing methods to CNNs versus transformers.

Comparison with State-of-the-art Weight Sharing Methods.

Table 5.7 compares the performance of weight sharing methods discussed in Section 5.3 with existing methods [143; 94; 161] on ImageNet. Compared to existing methods, our weight sharing schemes are effective; achieving higher compression rate while maintaining accuracy. For example, ConvMixer-768/32, ConvMixer-156/20, and ConvNeXt-18 with weight sharing and weight fusion achieve 1.86x, 1.91x and 1.92 share rate while having a similar accuracy. Existing weight sharing techniques [143; 94] can only achieve at most 1.58x and 1.45x share rate at while maintaining accuracy. Although [161] can achieve 12x share rate, it results in a 8.8% accuracy drop.

These results show that isotropic networks can achieve a high share rate while maintaining accuracy with

simple weight sharing methods. The traditional pyramid style networks, while using complicated sharing schemes [143; 94; 161], the share rate is usually limited. Note that although our sharing schemes can achieve higher share rates, existing methods like [161; 143] are able to directly reduce FLOPs, which our method does not address.



(a) Vanilla WS-ConvMixer.

(b) WS-ConvMixer with Fusion.

Figure 5.4: (a) The CKA similarity analysis of a standard ConvMixer's intermediate feature maps compared to a vanilla weight shared ConvMixer, with share rate of 2. (b) The same analysis but compare to a weight shared ConvMixer initialized with weight fusion. The channel weighted mean fusion strategy is used (see Section 5.3).

5.5 Representation Analysis

In this sections, we perform qualitative analysis of our weight sharing models to better understand why they lead to improved performance and how they change model behavior. To do this, we first analyze the representations learned by the original network, compared to one trained with weight sharing. We follow a similar set-up to Section 5.3. We use CKA as a metric for representational similarity and compute pairwise similarity across all layers in both the networks we aim to compare. In Figure 5.4(a) we first compare the representations learned by a vanilla weight sharing method to the representations of the original network. We find that there is no clear relationship between the representations learned. Once we introduce the weight fusion initialization strategy (Section 5.3), we find significant similarity in representations learned, as shown

in Figure 5.4(b). This suggests that our weight fusion initialization can guide the weight shared models to learn similar features to the original network. In Appendix F, we further analyze the weight shared models and characterize their robustness compared to standard networks.

5.6 Summary

Isotropic networks have the unique property in which all layers in the model have the same structure, which naturally enables parameter sharing. In this paper, we perform a comprehensive design space exploration of shared parameters in isotropic networks (SPIN), including the weight sharing topology, dynamic transformations and weight fusion strategies. Our experiments show that, when applying these techniques, we can compress state-of-the-art isotropic networks by up to 2 times without losing any accuracy across many isotropic architectures. Finally, we analyze the representations learned by weight shared networks and qualitatively show that the techniques we introduced, specifically fusion strategies, guide the weight shared model to learn similar representations to the original network. These results suggest that parameters sharing is an effective axis to consider when designing efficient isotropic neural networks.

Remarks on Author Contributions

My major contributions are as follow:

- Brainstormed and discussed the core designs of the empirical framework with Anish.
- Led the model training and experiments of the paper.
- Co-led the structuring and writing of the paper

Chapter 6

Atom: Low-Bit Quantization for Efficient and Accurate Large Language Model Serving

Remarks on Chapter Material

The content of this chapter is adapted from the following paper:

- Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, Baris Kasikci, "Atom: Low-bit Quantization for Efficient and Accurate LLM Serving", In Proceedings of The Seventh Annual Conference on Machine Learning and Systems (MLSys), 2024.

6.1 Introduction

Large Language Models (LLMs) are increasingly being integrated into our work routines and daily lives, where we use them for summarization, code completion, and decision-making. Studies report that ChatGPT has over 100 million users, with more than 1 billion website accesses per month [70]. Furthermore, the size and capabilities of LLMs continue to grow to accommodate a broader range of tasks. The high inference demand and model complexity have significantly increased the operational costs, i.e., compute/memory and

energy, for LLM service providers to near $1 million daily [72].

Unsurprisingly, optimizing LLM serving is becoming a pressing concern. Most efforts have focused on improving LLM serving throughput, which is typically achieved by batching requests from various users [295; 46; 157]. Batching multiple requests increases compute intensity and amortizes the cost of loading weight matrices, thereby improving throughput. Prior work has explored LLM quantization techniques to further improve batching efficiency. These techniques employ smaller data types to replace 16-bit floating point (FP16) values, thereby reducing memory consumption and accelerating computation [169; 281].

However, current quantization schemes do not leverage the full extent of capabilities provided by emerging efficient low-bit hardware support (e.g., Nvidia Ampere [11] and Qualcomm Hexagon [274]). For instance, several prior approaches have explored weight-only quantization [169; 76]. In these quantization schemes, weights are quantized to a low-bit representation (e.g., INT3), whereas activations remain in a floating point representation (e.g., FP16). Consequently, weights must be dequantized to the appropriate floating point representation (e.g., FP16) before being multiplied with activations using floating point representation. Therefore, even though weight-only quantization reduces memory consumption, it still requires costly floating-point arithmetic, which is inefficient, especially for large batch sizes.

Another prominent quantization scheme is weight-activation quantization, where both weights and activations are quantized to low-bit representations. In this scheme, weights and activations can be directly multiplied using low-precision arithmetic units. This quantization approach has greater potential to achieve higher inference throughput than weight-only quantization due to the efficient low-bit hardware support. For example, A100 GPUs can reach 1248 TOPS of INT4 and 624 TOPS of INT8 as opposed to only 312 TFLOPS for FP16 with Tensor Cores [199]. Prior works such as LLM.INT8() [65] and SmoothQuant [281] explored INT8 weight-activation quantization and achieved near no accuracy loss. However, INT8 quantization still cannot utilize lower bit arithmetic such as INT4 Tensor Cores [200]. In addition, INT8 quantization remains sub-optimal for reducing the large memory consumption in LLM serving, where both model parameters and batched KV-cache consume large memory [237; 309]. For lower-bit weight-activation quantization, recent works such as OmniQuant [232] and QLLM [172] have proposed to quantize LLMs down to 4-bit. However, their techniques still show a significant perplexity increase compared to the FP16 baseline as shown in Figure 6.2. Therefore, determining how to accurately quantize LLMs into low-bit representations

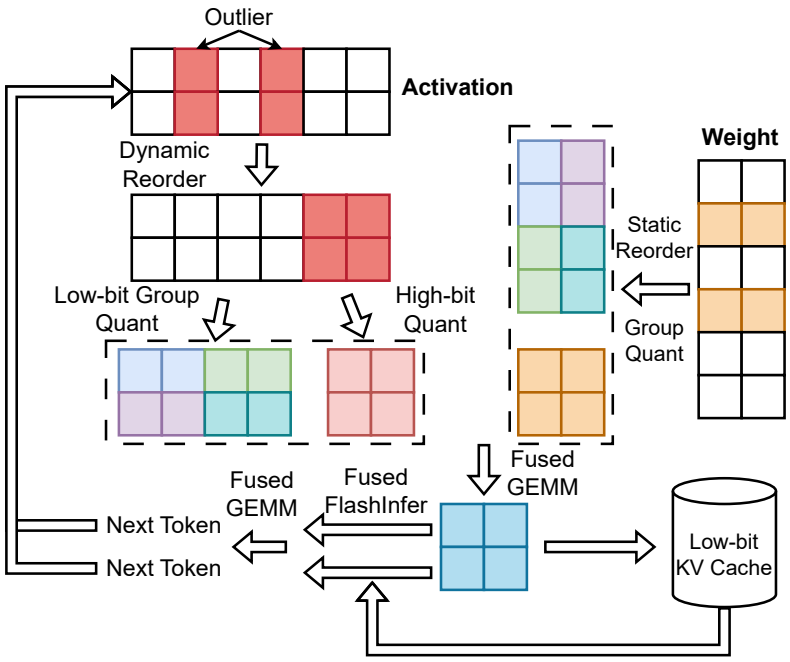


Figure 6.1: Overview of Atom's design.

while maintaining hardware efficiency remains an open area of research.

In this work, we introduce Atom, an accurate low-bit weight-activation quantization for LLMs that efficiently use modern hardware. To maintain accuracy, Atom incorporates three key quantization designs: (1) It adopts mixed-precision quantization, which retains a small but salient number of activations and weights in high precision to preserve accuracy. (2) It employs fine-grained group quantization on both weights and activations, which naturally reduces quantization errors. (3) Instead of pre-calculating quantization parameters for activations, Atom dynamically quantizes activations to best capture the distribution of each input.

Although these quantization optimizations can improve quantization accuracy, they may not utilize the underlying hardware efficiently without a bespoke design. For example, the mixed-precision technique could lead to irregular memory accesses and performance slowdown [93]; matrix multiplications with group quantization are not well-supported in kernel libraries; and dynamic quantization of activations incurs extra computation [281]. To ensure high hardware efficiency and minimize quantization overheads, Atom: (1) reorders activations and weights to maintain regular memory accesses for mixed-precision operations, (2) fuses quantization and reordering operations into existing operators to mitigate the overheads, (3) further quantizes outliers into 8-bit to keep a balance between accuracy and efficiency and (4) quantizes the KV-cache into low-bit representations to reduce memory movement. We illustrate Atom's quantization workflow

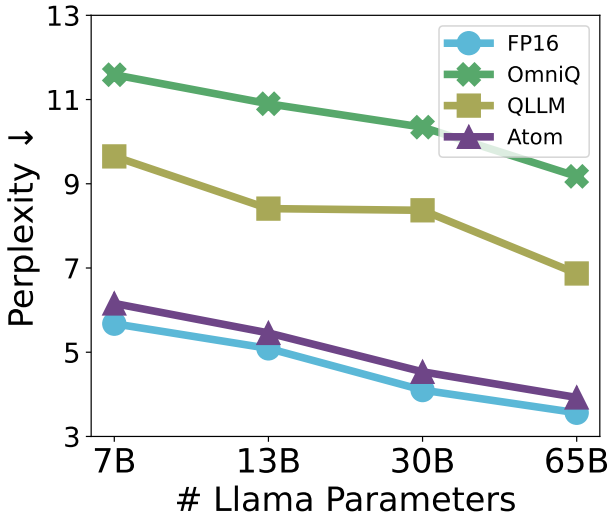


Figure 6.2: WikiText2 perplexity on Llama models with different 4-bit weight-activation quantization mechanisms.

in Figure 6.1.

To validate Atom's feasibility, we integrate it into an end-to-end serving framework [47]. For our special matrix multiplications with mixed-precision and group quantization, we implement customized CUDA kernels that utilize low-bit tensor cores. Experiments on popular datasets show that Atom has negligible accuracy loss (1.4% average zero-shot accuracy drop, 0.3 WikiText2 perplexity increase for Llama-65B) when quantizing models to 4-bit (for both weights and activations), while prior works suffer larger accuracy loss under the same precision (see Table 6.1).

When comparing end-to-end serving throughput to different precisions and quantization schemes, Atom improves throughput by up to 7.7×, 5.5×, and 2.5× relative to FP16, W4A16, and W8A8, respectively, while achieving similar latency (see Figure 6.11). These results show that Atom can accurately quantize LLMs into low-bit precision while achieving high serving throughput.

In summary, we contribute the following:

- A comprehensive performance analysis of LLM serving workloads that pinpoints the efficiency benefit of low-bit weight-activation quantization.
- Atom, an accurate low-bit weight-activation quantization algorithm that combines (1) mixed-precision with channel reordering, (2) fine-grained group quantization, (3) dynamic activation quantization to minimize quantization errors, and (4) KV-cache quantization.

- An integrated LLM serving framework for which we codesign an efficient inference workflow, implement low-bit GPU kernels and demonstrate practical end-to-end throughput and latency of Atom.
- A comprehensive evaluation of Atom, which shows that it improves LLM serving throughput by up to 7.7× with only a slight accuracy loss.

6.2 Quantization Basics

Quantization techniques use discrete low-bit values to approximate high-precision floating points. Since integers represent a uniform range, quantizing floating point values into integers is widespread due to simplicity and hardware efficiency [122; 97]. Typical quantization involves two steps: determining the quantization parameters (which consist of scale and zero point) and calculating the quantized tensor. For uniform asymmetric quantization, the scale s and zero point z are determined by [196]:

$$s = \frac{\max(X) - \min(X)}{2^n - 1} \cdot c, z = \lfloor \frac{-\min(X)}{s} \rceil, \tag{6.1}$$

where X is the input tensor, n is the quantization bit-width, and c is the clipping factor used to reduce the dynamic range of quantization to mitigate the effect of outlier values. The elements in quantized tensor can be calculated by:

$$\bar{X} = \text{clamp}(\lfloor \frac{X}{s} \rceil + z, 0, 2^n - 1).$$

We can further simplify this equation for symmetric quantization:

$$s = \frac{2 \cdot \max(|X|)}{2^n - 1} \cdot c$$

$$\bar{X} = \text{clamp}(\lfloor \frac{X}{s} \rceil, -2^{n-1}, 2^{n-1} - 1).$$

Quantization parameters s and z can be calculated either statically using calibration data or dynamically during inference time with runtime statistics. Thus, quantization approaches can be classified as *static* or *dynamic*.

For LLMs, we can apply quantization on both activation and weight matrices (weight-activation quan-

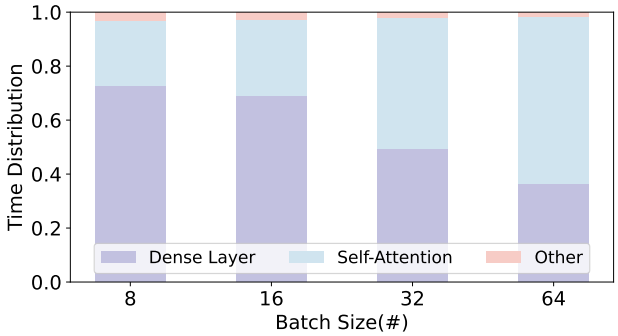


Figure 6.3: Runtime breakdown of Llama-7b inference with different batch sizes.

tization) or just the latter (weight-only quantization). However, asymmetric weight-activation quantization can lead to additional calculations during matrix multiplication since:

$$W \cdot X = s_W(\bar{W} - z_W) \cdot s_x(\bar{X} - z_x),$$

where three additional cross-terms need to be calculated for using low-bit arithmetic units. Therefore, we apply symmetric quantization in this work for efficiency.

Different trade-offs between accuracy and efficiency can be achieved by quantization with different granularity: For **per-tensor** quantization, all the values in the tensor share one set of scale and zero-point [196]. For **per-channel (token)** quantization, we calculate scale and zero-point for a row or a column of the tensor [281]. We denote the channel as the last dimension of the input matrix. Each channel can be further divided into several sub-groups, and quantization is individually performed on each group, which is called **per-group** quantization [169]. The finer the granularity, the more precise the quantization, but the higher the overhead. In this work, we adopt group quantization for higher accuracy with dedicated kernels to manage the overhead, as shown in § 6.4.

6.3 Performance analysis of low-bit LLM serving

In this section, we first analyze the performance bottleneck of LLM inference in serving scenarios and then establish the importance of low-bit weight-activation quantization.

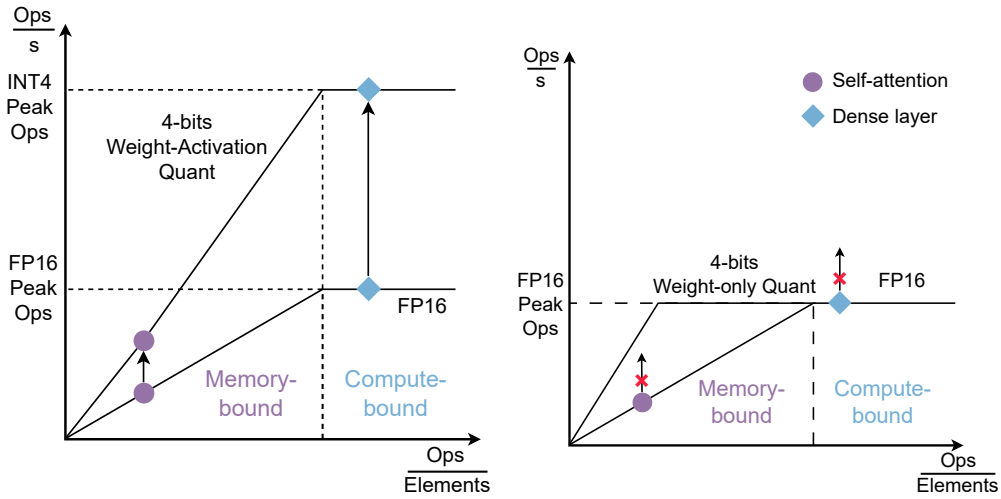
Due to high demand, LLM serving is throughput-oriented. However, the auto-regressive decode stage of LLM inference only takes one token as input and generates the next token, thus relying on matrix-vector

multiplication (GEMV) [14]. Since GEMV needs to load a large weight matrix while only performing a few multiplications, it is heavily memory-bound. It thus causes GPU under-utilization, which results in low compute intensity (computation-to-IO ratio) and, thereby, low throughput [275]. To mitigate this problem, batching is widely used by combining the input from multiple requests to perform dense layer (K,Q,V generation, O projection, and MLP) matrix multiplications and increase compute intensity, therefore GPU utilization [215; 295; 47; 317].

To further exploit the batching effect and boost throughput, the input matrices of the dense layer of the decode and prefill stages are batched together to form larger matrices [209]. Given large batch sizes, the dense layer ends up having compute-bound matrix-matrix multiplications (GEMM). However, though self-attention layers in the decode stage are also GEMV operations, they cannot benefit from batching. Since different inference requests do not share the KV-cache with different context histories, cross-request data cannot be batched for reuse, resulting in no efficiency benefit. Even with several optimizations such as FlashAttention [59] or Group Query Attention [18], the self-attention layers are still bounded by the large memory movement of KV-cache.

After applying the batching technique, we measure the time breakdown of different operators under different batch sizes. As Figure 6.3 shows, both the dense and self-attention layers act as bottlenecks to throughput, consuming over 90% of the processing time. Consequently, we employ quantization mechanisms to expedite both dense and self-attention layers. Note in Figure 6.3, the dense layer represents the batched K, Q, V generation, O projection, and MLP. The self-attention layer is implemented by FlashInfer [292] integrated with PageAttention [157]. Results indicate that the dense and self-attention layers together account for over 90% of the execution time, thereby constraining the throughput.

We use the Roofline model [275] to evaluate the effect of different quantization approaches in serving scenarios. As Figure 6.4 shows, weight-activation quantization has higher dense layer compute throughput due to the efficient low-bit hardware arithmetic. It also increases the throughput of the self-attention layer by reducing the size of the KV-cache, thus decreasing memory movement. However, as Figure 6.4 shows, weight-only quantization fails to improve dense layer throughput since dequantization must be performed before matrix multiplications, yielding calculations still in the floating point format. On the other hand, weight-only quantization fails to quantize the KV-cache, yielding no benefit for self-attention layers. We



(a) Weight-activation quantization **(b)** Weight-only quantization

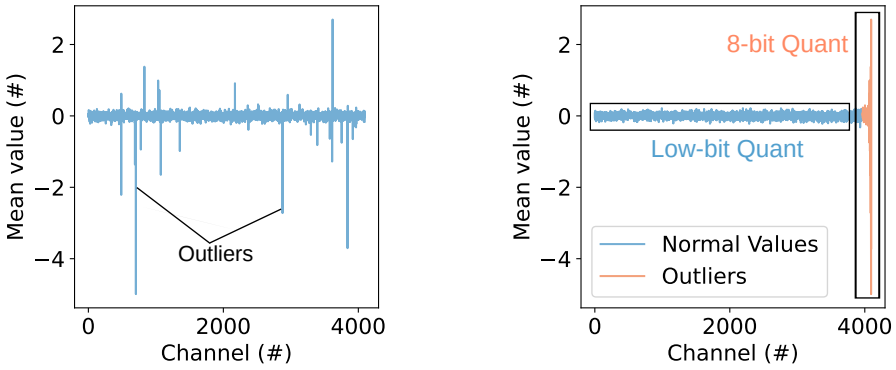
Figure 6.4: A roofline model of different quantization approaches that characterizes operators by their arithmetic intensity.

further quantify the effect of different quantization techniques in Figures 6.10 and 6.10 in §6.5 with kernel profiling. Note in Figure 6.4, arithmetic intensity is defined as Ops/Elements. At large batch sizes, the dense layer is compute-bound, which has a large arithmetic intensity, whereas self-attention consistently exhibits a lower arithmetic intensity.

In summary, the low-bit weight-activation quantization is superior to weight-only quantization in terms of enhancing the throughput in the serving scenario because it accelerates both the dense and self-attention layers. In the following sections, we demonstrate how Atom delivers high throughput while still maintaining high accuracy with the low-bit weight-activation quantization.

6.4 Design of Atom

Low-bit precision enables efficient utilization of the underlying hardware, leading to increased throughput. However, it is challenging to maintain high accuracy with a low-bit representation. To quantize LLMs to extremely low-bit precision while keeping accuracy, we incorporate a suite of quantization mechanisms tailored to LLM characteristics. These mechanisms include mixed-precision quantization with channel reordering, fine-grained group quantization, and dynamic quantization. We demonstrate the accuracy gain thanks to these techniques with ablation study in Table 6.3. Atom also applies low-bit quantization on



(a) Activation mean values per channel. **(b)** Mean values after reordering.

Figure 6.5: Sampled value of an activation matrix from Llama-7b.

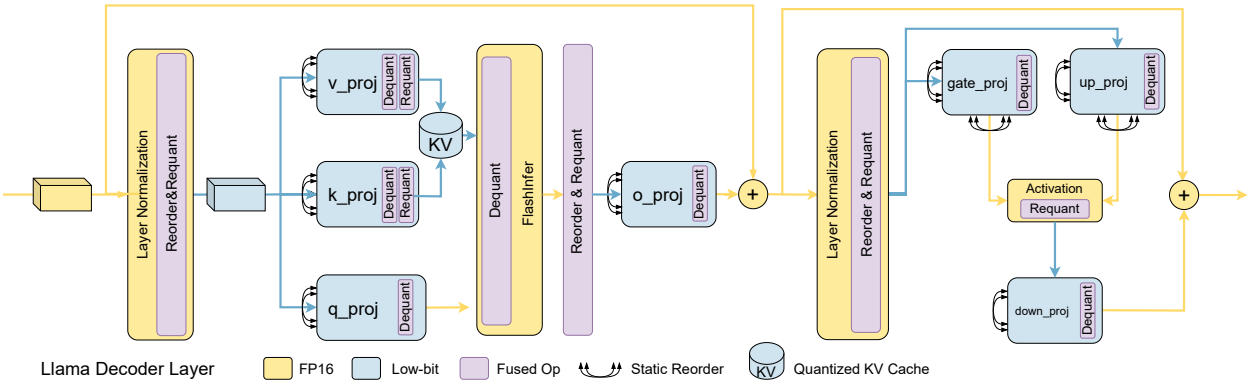


Figure 6.6: Overview of Atom workflow on Llama model family.

KV-cache, which further boosts the efficiency. The subsequent subsections delve into the specifics of each mechanism and its advantages, followed by a detailed description of the end-to-end workflow.

Mixed-precision quantization

Prior works observed that a key challenge of LLM quantization is the outlier phenomena in activations [65; 169]. As Figure 6.5 shows, a few channels exhibit large magnitudes that are several orders greater than those of other channels, which are called outliers. The large dynamic range of these outliers can substantially increase the quantization error. Therefore, efficiently handling the outliers is crucial in low-bit quantization. Note for Figure 6.5, in (a) The activation matrix contains outlier channels, which result in large quantization errors; in (b) Atom reorders these outlier channels to the end of the matrix and uses higher precision to quantize them while keeping regular memory access.

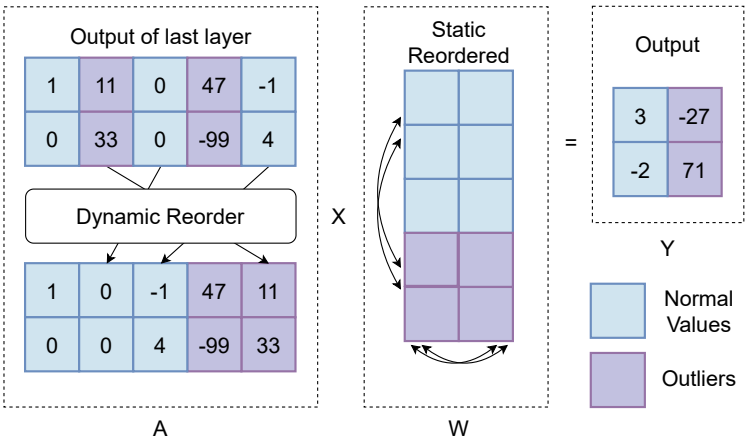


Figure 6.7: The overview of Atom's reordering mechanism.

One intuitive way to effectively mitigate this challenge is to quantize outliers and normal values separately, into low and high bits, which is referred to as a *mixed-precision* method. As Figure 6.5 shows, after we remove the outliers, the remaining channels are much more uniform, which can be effectively expressed by low-bit values. Our results indicate that 8-bit representations, such as FP8 [188] and INT8, are sufficient to express outliers (See Table 6.3). Since INT8 is widely supported by hardware implementations (e.g., NVIDIA Tensor Core [11]), Atom applies INT8 quantization for outliers.

The primary concern with mixed-precision quantization is its irregular memory accesses [65; 93], which leads to poor hardware efficiency. To apply mixed-precision quantization while maintaining regular memory access, Atom re-purposes the reordering technique introduced in RPTQ [297], where the objective is to improve quantization accuracy. As Figure 6.7 shows, Atom reorders the scattered outlier channels of activations to the end of the matrix, which enables the efficient implementation of mixed-precision. To guarantee the equivalence of the computation result, the weight matrices need to be reordered with the corresponding reorder indices of activations. Since the outlier channels can be identified offline using calibration data [65], the reordering of weight matrices incurs a one-time cost. However, the reordering of activation matrices still needs to be performed online, which can be expensive. To mitigate this, Atom fuses the activation matrix reordering operators into prior operators, which significantly reduces the reordering overhead to less than 0.5% of runtime.

Fine-grained group quantization

Even if Atom quantizes outliers and normal values separately, the latter is still challenging to perform accurately due to the limited representation capability of 4-bit precision (Section 6.5). To further enhance accuracy, *group quantization* is widely adopted [169; 196], which divides the matrix into subgroups and performs quantization within each subgroup. For example, a group size of 128 implies that every contiguous sequence of 128 elements is treated as a single group, which is quantized independently.

Group quantization offers a trade-off between accuracy improvements and dequantization overheads, especially in weight-activation quantization. Prior works have not investigated how to efficiently incorporate group dequantization into the delicate GEMM pipeline, i.e., MMA pipeline [248]. Atom proposes a fusion technique as shown in Figure 6.8, which contributes to an efficient GEMM kernel with practical speedup (See §6.5). Atom first calculates the matrix multiplication of the activation groups with the corresponding weight groups and obtains temporary results using efficient low-bit hardware, i.e. Tensor Cores (Step ①). Atom then adds multiple temporary results together to get the GEMM result. However, since Atom performs fine-grained quantization for each activation and weight group, each temporary result has different quantization parameters. Therefore, Atom first dequantizes all temporary results to the FP16 representation with CUDA Cores (Step ②) and then performs addition (Step ③). To manage the overhead, we fuse dequantization and summation into the GEMM kernel, to be specific, into the MMA pipeline. Therefore, the additional operations can be executed in place without extra memory movement and overlapped with the original MMA instructions. We demonstrate the efficiency of the fused GEMM operator in §6.5.

With a group size of 128 and a high precision channel size of 128, Atom has an effective bit of 4.25[1] on Llama-7b. The *effective bit* is defined as the average bits used for each element, including the quantization parameters. This metric is widely used in previous works on weight-only quantization [76; 169], mainly because it represents the actual compression ratio and, therefore, the speedup in the memory-bound setting. However, the main benefit of weight-activation quantization in serving scenarios is the computation efficiency of leveraging low-bit arithmetic units instead of the memory reduction. Therefore, we will not use this metric in the following discussions.

[1]With 4-bit for normal values, 8-bit for outliers, and 16-bit scale per group, the effective bit is calculated as $((4096 - 128) * 4 + 128 * 8)/4096 + 16/128 = 4.25$.

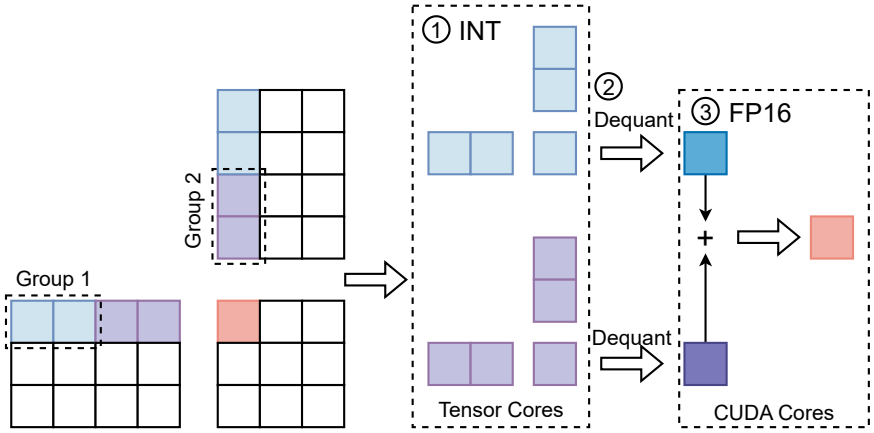


Figure 6.8: Overview of the fused Atom's GEMM operator.

Dynamic quantization process

Although fine-grained quantization can better preserve the local variations inside each channel of activations, this advantage would diminish if we statically calculated the quantization parameters based on calibration data, as the actual input might have a different local distribution.

Therefore, Atom adopts *dynamic quantization*, tailoring quantization parameters for each activation matrix during inference. To tame the overhead of dynamic quantization, we fuse quantization operations into the prior operator, akin to the implementation of ZeroQuant [289]. Since the additional operator is element-wise (with a reduction and an element-wise division), the run time of the fused operator is still negligible compared to the time-consuming dense and self-attention layers, as Figure 6.3 shows.

However, asymmetric quantization can lead to significant run-time overhead due to considerable additional computation (as discussed in §4.2). To strike a balance between throughput and accuracy, we choose symmetric quantization with a carefully chosen clip threshold. We also incorporate GPTQ [76] when quantizing the weight matrix since this is purely an offline process and offers an accuracy boost without sacrificing runtime efficiency.

KV-cache quantization

As described in §6.3, the self-attention layer in the decode stage is highly memory-bound. To mitigate this issue, Atom also applies low-bit quantization to the KV-cache. Atom loads the KV-cache in low-bit precision and directly dequantizes it before performing the FP16 calculation, which significantly boosts the

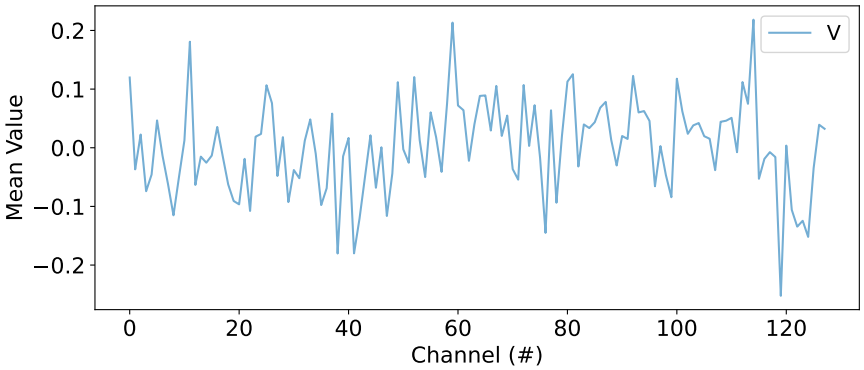


Figure 6.9: Sampled value of the V cache within a single attention head from Llama-7b.

throughput by large memory reduction. On the other hand, since the memory movement of asymmetric and symmetric quantized KV-cache are similar, they perform similarly on memory-bound self-attention layers. Therefore, Atom uses asymmetric quantization on KV-cache as it can provide accuracy benefits.

Compared with activation matrices, we argue that the KV-cache is more amenable to quantization. To perform self-attention, the Query vector of the incoming token is multiplied by the K cache. The result is normalized using Softmax and further multiplied with the V cache to obtain the output [259]. Due to the normalization of Softmax, the quantization error of the K cache has less influence on the output. Furthermore, our profiling in Figure 6.9 indicates that the V cache exhibits the outlier phenomenon less frequently, rendering it more suitable for quantization. Therefore, Atom directly applies asymmetric low-bit quantization with the granularity of attention head and preserves high accuracy as shown in §**??**.

Implementation of quantization workflow

To demonstrate the feasibility of our design choices, we implement Atom on Llama models [251], as shown in Figure 6.6. To leverage the benefit of quantization, Atom manages the overhead of the additional operators by kernel fusion: Atom fuses quantization operators, including reordering, quantization, and dequantization, into existing operators. For the compute-bound dense layer, Atom utilizes the low-bit units to boost throughput. For the memory-bound self-attention layer, Atom fuses dequantization with a kernel library for LLM serving, FlashInfer [292], so that only low-bit values from KV-cache are loaded. Atom also incorporates PageAttention [157] for efficient memory usage to enable large batch sizes.

6.5 Evaluation of Atom

We conduct a comprehensive evaluation of Atom's accuracy and efficiency. For accuracy, we evaluate Atom on widely used metrics, generation perplexity and zero-shot accuracy. For efficiency, we evaluate Atom from the bottom up, starting with per-kernel performance, followed by end-to-end throughput and latency. We also perform ablation studies to understand how different techniques affect Atom, which pinpoints the trade-off between the efficiency and accuracy of each design choice.

Quantization setup

Atom uses symmetric quantization on weights and activations while using asymmetric quantization on the KV-cache. We evaluate Atom using a group size of 128. To identify outlier channels, we use 128 randomly sampled sentences from WikiText2 [187] as calibration data, following prior works [160; 232; 172]. We select 128 channels with the highest square sum values as outlier channels and keep them in INT8. We then reorder activation and weight matrices according to the indices of outlier channels. After reordering, Atom adopts GPTQ [76] for the quantization on weight matrices. For clipping, we use a grid search to find optimal clipping factors 0.9 and 0.85 for activation and weight quantization, respectively.

For the preprocessing of weight quantization and outlier identification, we run Atom on a single RTX Ada 6000 and quantize the model layer-by-layer. For large Llama-65B, Atom takes roughly 4 hours to complete the process.

Accuracy evaluation

Benchmarks. We evaluate Atom on popular open-sourced Llama [251] models. We focus on low-bit settings, INT4 and INT3 weight-activation quantization. We adopt commonly used metrics of model accuracy, perplexity, and zero-shot accuracy. For perplexity, we evaluate on WikiText2 [187], PTB [186], and C4 [220] datasets. For zero-shot tasks, we use lm-eval [82], based on which we evaluate Atom on PIQA [34], ARC [57], BoolQ [56], HellaSwag [302], and WinoGrande [227] tasks.

Baselines. We compare Atom to recently released post-training quantization techniques: SmoothQuant [281], OmniQuant [232], and QLLM [172]. For SmoothQuant, we implement our own version as the official

Table 6.1: Zero-shot accuracy of quantized Llama models on six common sense tasks.

Llama	#Bits	Method	Zero-shot Accuracy ↑						
			PIQA	ARC-e	ARC-c	BoolQ	HellaSwag	Winogrande	Avg.
7B	FP16	-	77.42	52.61	41.47	73.12	72.98	67.01	64.10
	W4A4	SmoothQuant	63.11	40.03	31.57	58.47	43.38	52.80	48.23
		OmniQuant	66.15	45.20	31.14	63.51	56.44	53.43	52.65
		QLLM	68.77	45.20	31.14	-	57.43	56.67	51.84
		Atom	**75.57**	**51.30**	**37.88**	**72.05**	**69.84**	**62.35**	**61.50**
	W3A3	SmoothQuant	48.69	25.97	28.16	45.26	26.02	49.57	37.28
		OmniQuant	49.78	27.19	27.22	37.86	25.64	49.96	36.28
		Atom	66.10	42.21	31.40	62.48	52.90	52.25	51.22
13B	FP16	-	79.11	59.85	44.54	68.47	76.22	70.09	66.38
	W4A4	SmoothQuant	64.47	41.75	30.89	62.29	46.68	51.70	49.63
		OmniQuant	69.69	47.39	33.10	62.84	58.96	55.80	54.63
		QLLM	71.38	47.60	34.30	-	63.70	59.43	55.28
		Atom	**77.37**	**57.66**	**43.26**	**66.82**	**73.71**	**68.59**	**64.57**
	W3A3	SmoothQuant	47.99	26.30	27.65	46.91	25.65	49.64	37.36
		OmniQuant	50.22	26.77	27.82	37.83	25.77	51.07	36.58
		Atom	70.73	48.06	33.96	63.70	62.64	57.85	56.16
30B	FP16	-	80.20	58.92	45.31	68.38	79.23	72.69	67.46
	W4A4	SmoothQuant	59.30	36.74	28.58	59.97	34.84	49.96	44.90
		OmniQuant	71.21	49.45	34.47	65.33	64.65	59.19	57.38
		QLLM	73.83	50.67	38.40	-	67.91	58.56	57.87
		Atom	**78.73**	**58.92**	**45.82**	**68.47**	**77.40**	**73.09**	**67.07**
	W3A3	SmoothQuant	49.46	27.53	28.16	39.42	26.05	51.38	37.00
		Atom	72.47	49.54	37.80	65.75	66.99	60.14	58.78
65B	FP16	-	80.79	58.71	46.24	82.29	80.72	77.50	71.04
	W4A4	SmoothQuant	60.72	38.80	30.29	57.61	36.81	53.43	46.28
		OmniQuant	71.81	48.02	35.92	73.27	66.81	59.51	59.22
		QLLM	73.56	52.06	39.68	-	70.94	62.90	59.83
		Atom	**80.41**	**58.12**	**45.22**	**82.02**	**79.10**	**72.53**	**69.57**
	W3A3	SmoothQuant	49.56	26.64	29.10	42.97	26.05	51.14	37.58
		Atom	75.84	51.43	41.30	74.07	72.22	64.33	63.20

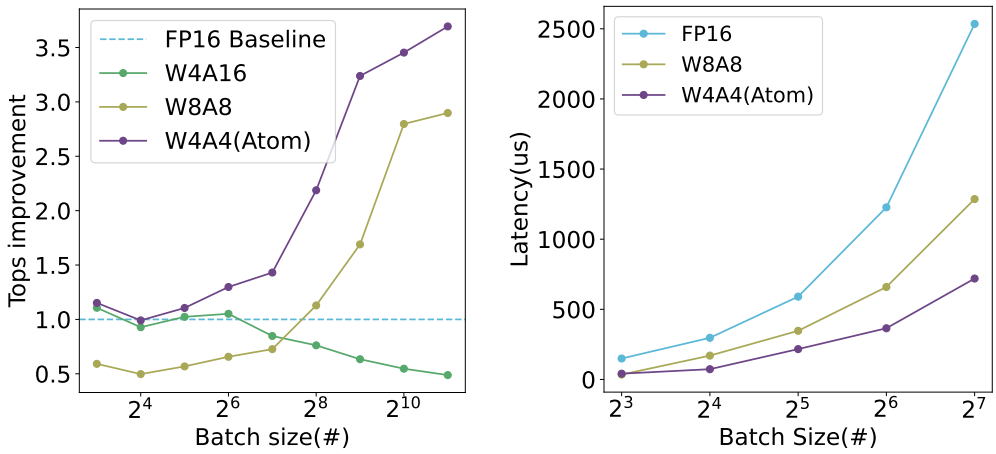
Table 6.2: Perplexity of quantized Llama models on WikiText2, PTB and C4 dataset.

Size	Bits	Method	Perplexity ↓ WikiText2	PTB	C4	Size	Bits	Method	Perplexity ↓ WikiText2	PTB	C4
7B	FP16	-	5.68	8.80	7.08	13B	FP16	-	5.09	8.07	6.61
	W4A4	SmoothQuant	22.62	40.69	31.21		W4A4	SmoothQuant	33.98	73.83	41.53
		OmniQuant	11.59	20.65	14.96			OmniQuant	10.90	18.03	13.78
		QLLM	9.65	-	12.29			QLLM	8.41	-	10.58
		Atom	**6.16**	**9.62**	**7.69**			Atom	**5.46**	**8.60**	**7.03**
	W3A3	SmoothQuant	2.7e4	3.5e4	2.6e4		W3A3	SmoothQuant	1.3e4	1.6e4	1.5e4
		OmniQuant	3.4e3	7.5e3	6.3e3			OmniQuant	7.2e3	1.6e4	1.3e4
		Atom	**10.67**	**18.10**	**13.65**			Atom	**8.40**	**14.31**	**10.76**
30B	FP16	-	4.10	7.30	5.98	65B	FP16	-	3.56	6.91	5.62
	W4A4	SmoothQuant	109.85	142.34	87.06		W4A4	SmoothQuant	88.89	278.76	283.80
		OmniQuant	10.34	14.91	12.49			OmniQuant	9.18	16.18	11.31
		QLLM	8.37	-	11.51			QLLM	6.87	-	8.98
		Atom	**4.54**	**7.67**	**6.35**			Atom	**3.93**	**7.54**	**5.92**
	W3A3	SmoothQuant	1.5e4	1.6e4	1.5e4		W3A3	SmoothQuant	6.6e8	3.7e8	4.4e8
		Atom	**6.99**	**11.07**	**9.14**			Atom	**5.79**	**9.30**	**7.84**

code does not support Llama models and only has W8A8 quantization. We conducted a grid search on the $alpha$ value defined in SmoothQuant and reported the best numbers for each benchmark. For OmniQuant, we use their pre-quantized weights for W4A4 evaluations and evaluate W3A3 by running their official code. To obtain the best W3A3 results for OmniQuant, we conduct a hyperparameter search and identify $lr = 1e^{-4}$ and $alpha = 0.75$ for their quantization process. We skip W3A3 OmniQuant on Llama-30B and Llama-65B due to the large resource requirement of its quantization process. For QLLM, we report the W4A4 numbers in their paper but do not evaluate W3A3 as their code was unavailable when we conducted experiments.

Zero-shot accuracy. Table 6.1 compares the zero-shot accuracy of six tasks between Atom and baselines on Llama models. Atom significantly outperforms the other weight-activation quantization methods. For W4A4, Atom shows only a 2.3%, 1.7%, 0.4% and 1.4% average accuracy loss for Llama at 7B, 13B, 30B and 65B sizes when compared to FP16. At the same time, previous works showed a 9.6% to 23.8% accuracy loss under the same settings.

Perplexity. Table 6.2 reports perplexity results of Atom and baselines on Llama models. As the table shows, though recent methods such as OmniQuant and QLLM successfully reduce the perplexity of W4A4



(a) Dense layer **(b)** Self-attention

Figure 6.10: Performance evaluation of different quantization approaches on Atom and baseline kernels.

to around 10, the accuracy loss is still significant. Atom further reduces the perplexity and achieves less than 0.4 perplexity increase on all three datasets with Llama-65b. For W3A3, Atom still largely maintains the perplexity, with an average 2.3 perplexity increase for Llama-65B. At the same time, existing works do not achieve acceptable perplexity. Note that Atom has less accuracy loss when quantizing larger models.

Efficiency evaluation

To demonstrate the efficiency of Atom, we conduct experiments profiling both per-kernel and end-to-end performance. Since the highly efficient INT4 arithmetic is supported by NVIDIA GPUs, we evaluate Atom with W4A4 quantization on a 24GB RTX 4090 with CUDA 11.3.

Kernel evaluation:

Matrix multiplication. We evaluate the fused GEMM operator implemented by Atom, as shown in Figure 6.10. We also implemented fused GEMM for 8-bit weight-activation quantization (W8A8) and 4-bit weight-only quantization (W4A16) following the existing work [281; 169] as baselines. For smaller batch sizes, GEMM is memory-bound; thus, weight-only quantization's memory reduction is effective. However, as the batch size increases, the efficiency of weight-only quantization diminishes in the compute-bound setting due to the expensive FP16 calculations. At the same time, 4-bit Atom outperforms all other approaches

due to its hardware efficiency. At batch size 512, Atom's matrix-multiplication achieves 3.4× and 1.9× speedup over FP16 and INT8 kernels.

Self-attention. For the self-attention layer, we fuse different quantization methods into FlashInfer [292], which is a performant kernel library for LLMs serving. We also integrate PageAttention [157] for efficient memory usage. We evaluate our implementation and show the results in Figure 6.10. The decrease in bits linearly reduces the memory usage of the KV-cache, therefore proportionally boosting the throughput in the memory-bound setting. At batch size 128, Atom achieves a 1.8× speedup over INT8 quantization and 3.5× over the FP16 baseline. Note that in Figure 6.10, we set up the evaluation configuration aligned with the Llama-7b config and 1024 sequence length. Kernels are evaluated by NVBench [203].

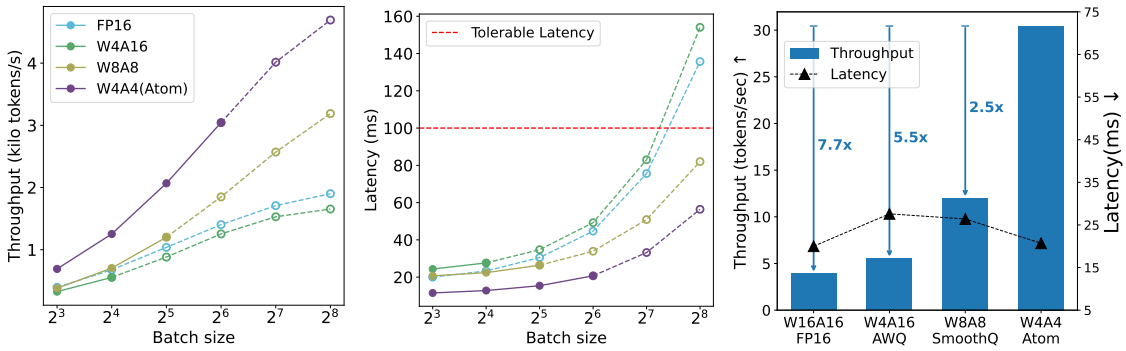
End-to-end evaluation:

Serving setup.

We integrate Atom into Punica, an LLM serving framework [47], to evaluate the performance in the end-to-end scenario. We also integrate W8A8 and W4A16 quantizations following previous works [281; 169] as baselines. To generate a representative workload, we use ShareGPT [117] to collect the distribution of prefill and decode request length. We treat multi-round conversations as requests from multiple users. Specifically, we concatenate all previous prompts and responses and use them as the prompt for the new user request. We vary the batch size from 8 to 256, which represents the practical range in LLM serving[2]. All requests are served in a First-Come-First-Served manner. When a request is finished, we re-fill the on-the-fly batch with a new request following *continous batching* as introduced in Orca [295]. Due to GPU memory limits, we only show the exact results on small batch sizes. When the memory requirement cannot be satisfied, we simulate the performance by reusing the KV-caches from a smaller batch size while preserving the data access pattern and amount of computation.

End-to-end throughput. We show the end-to-end throughput, i.e., generated tokens per second, in Figure 6.11. Solid lines represent exact evaluation results, while dashed lines represent our simulated results for the cases that exceed our GPU's memory capacity. As Figure 6.11 shows, Atom outperforms other quantization methods on all batch sizes. If we fix the available memory as shown in Figure 6.11, Atom can

[2]With quantization, pipelining, and tensor parallelism to amortize weights, it is practical to deploy a 180B model with a 256 batch size in the serving scenario [209].



(a) End-to-end throughput **(b)** Decode latency per token **(c)** Results with Fixed GPU memory

Figure 6.11: End-to-end evaluation of Atom.

achieve larger batch sizes so that its throughput further surpasses all baselines while still meeting the latency target. Atom achieves 7.73× throughput compared to the FP16 baseline and 2.53× throughput compared to INT8 quantization using the same amount of memory. In contrast, weight-only quantization is bounded by FP16 computation capacity in dense layers and large memory movement of the KV-cache in the self-attention layer. Note in Figure 6.11, solid lines are exact measurements, while dashed lines are estimations due to the limited memory capacity. (a) The number of generated tokens per second. (b) Average decode latency per token. Atom surpasses all other quantization methods for both throughput and latency. (c) Performance evaluated under a fixed amount of GPU memory. Note that Atom boosts the throughput by 2.5× more than W8A8 since it enables a larger batch size, which utilizes the batching effect.

End-to-end latency. We measure the latency as the average decoding time of each token, without considering the queuing time. Atom significantly outperforms other quantization methods on every batch size. When we achieve the highest practical performance at batch size 64, our latency is lower than INT8 or FP16 implementations, even under batch size 8. Notably, even at batch size 256, our latency is still lower than 100 ms, which has been shown to be the effective reading speed of human eyes by a prior study [254].

Ablation study of quantization techniques

In this subsection, we comprehensively evaluate the effectiveness of quantization techniques used in Atom, in terms of both accuracy and efficiency, to better illustrate our design choices and the trade-off between accuracy and efficiency.

Table 6.3: Ablation study on different quantization techniques used in Atom. The model used in this table is Llama-7B.

Quantization method	WikiText2 PPL↓
FP16 baseline	5.68
W4A4 RTN	2315.52
+ Keeping 128 outliers in FP16	11.34 (2304.2↓)
+ Quantizing outliers to INT8	11.39 (0.05↑)
+ Group size 128	6.22 (5.17↓)
+ Clipping	6.13 (0.09↓)
+ GPTQ	6.04 (0.09↓)
+ Quantizing KV-cache to INT4	6.16 (0.12↑)

Ablation study to evaluate accuracy

We examine the accuracy gain or loss of different quantization techniques used in Atom. We first use RTN and adopt per-channel quantization for weights and per-token quantization for activations, which is the standard quantization recipe [281], to quantize the model to W4A4. We then apply other quantization techniques used in Atom, i.e., mixed-precision, quantizing outliers, group quantization, clipping, GPTQ, and KV-cache quantization, and examine the perplexity case by case. As shown in Table 6.3, keeping outlier channels in FP16 significantly reduces the perplexity. Further quantizing outliers into INT8 only results in a very minor 0.05 perplexity increase, which indicates mixed precision effectively addresses the outlier issue. Besides, fine-grained group quantization brings another major perplexity reduction. Furthermore, using clipping and GPTQ lowers perplexity by 0.09 each. After all, quantizing KV-cache results in a slight 0.12 perplexity increase, which echoes our finding in Section 6.4.

Ablation study to evaluate efficiency

We then showcase the GEMM kernel throughput with different fused quantization techniques[3]. A pure INT4 GEMM implementation without any quantization operation achieves nearly 980 TOPS. Fusion of mixed precision, which keeps 128 channel calculations in INT8 Tensor Cores, leads to 8% overhead, with 900 TOPS throughput. Fine-grained group quantization contributes to the major overhead since it deeply affects the compute pipeline. The fusion of group dequantization decreases the performance to 770 TOPS.

[3]Kernel performance is profiled by NVBench [203] with the Llama-7b config and a batch size of 4096 on RTX 4090.

Table 6.4: WikiText2 perplexity for Llama-2 and Mixtral.

# Bits	Method	Llama2 7B	Llama2 13B	Llama2 70B	Mixtral 8x7B
FP16	-	5.47	4.88	3.32	3.84
W4A4	SmoothQuant	83.12	35.88	-	-
	OmniQuant	14.61	12.3	-	-
	Atom (INT)	**6.03**	**5.27**	**3.68**	**4.41**
	Atom (FP)	**6.14**	**5.35**	**3.78**	**4.50**

However, the fused GEMM kernel still outperforms the theoretical limit of INT8 throughput by nearly 18%.

Besides, to demonstrate the efficiency of channel reordering, we also conduct an ablation study on Atom and baseline. The baseline is implemented following the previous work [65], with matrix decomposition for mixed precision quantization. At the same time, Atom fuses quantization operators, including reordering and quantization, into existing operators. We evaluate batch sizes from 16 to 256 and measure the inference latency of a layer norm and a GEMM operation. Results show that Atom consistently outperforms the baseline from 25% to 35%.

6.6 Discussion

With innovations of model architectures like Mixture of Experts (MoE) [127; 58], State Space Models (SSMs) [92; 91], and evolvement of hardware accelerators (e.g., NVIDIA Blackwell GPU [204]), it's important that Atom can be used for new models and hardware. In this section, we provide evaluations on more LLMs and data formats.

Generality on models. Atom's main techniques to achieve high accuracy are mixed precision for outliers and fine-grained quantization for normal values. We empirically find these are generalizable to newer transformer-based LLMs. In Table 6.4, we show the perplexity results of two relatively new LLMs, Llama-2 [252] and Mixtral [127]. To generalize on MoE models, Atom only needs to adapt to using different reorder indices for different experts' FFN[4]. As Table 6.4 shows, Atom still outperforms baselines and maintains high accuracy.

[4]In practice, we find that accuracy is similar when Atom share reorder indices across all experts in an MoE layer. Therefore, we use shared indices for efficiency consideration.

Generality on data formats. With the support for emerging data formats such as FP4 and MX [175; 226] on new hardware, we also evaluate the effectiveness of Atom in FP4. As shown in Table 6.4, Atom maintains a similar accuracy to INT4 when quantizing both weights and activations into FP4. We conclude that the representation capability between INT4 and FP4 is similar. Additionally, group quantization with the MX format is supported by NVIDIA Blackwell GPUs. We expect this hardware feature can mitigate the group quantization overhead of Atom as described in § 6.5.

6.7 Related Work

LLM serving. Various works have been explored to improve LLM serving throughput. [215] investigated the batching effect when scaling up LLMs. Orca [295] proposed *continuous batching* to improve GPU utilization by refilling the on-the-fly batch. vLLM [157] utilized page tables to manage KV-cache, which significantly increases GPU memory utilization. FlexGen [237] proposed an offload mechanism to support larger batches for high serving throughput. However, unlike prior works, in this paper, we delve deep into the intersection between quantization and LLM serving.

Weight-only quantization. For LLMs, weight matrices lead to large memory movement, limiting decode efficiency. Weight-only quantization uses low-bit precision to approximate weight matrices. For instance, GPTQ [76] used 4-bit to quantize the weight based on the approximate second-order information. AWQ [169] further advanced accuracy by preserving salient weights. SqueezeLLM [145] handled outliers through non-uniform quantization and used a sparse format to keep outliers and sensitive weights at high precision. QuiP [42] successfully represented weights using 2-bit by an adaptive rounding method. Nonetheless, in the LLM serving scenario, the overhead of loading the weight matrix is amortized due to batching. Thus, the dense layer becomes compute-bound, while weight-only quantization fails to use efficient low-bit hardware to deliver ideal throughput.

Weight-activation quantization. Weight-activation quantization quantizes both the weight and activation matrices, which is considered more challenging due to the outlier phenomenon of the activation. LLM.INT8 [65] proposed mixed precision to preserve outlier values in activation matrices. [281; 232; 289; 272] used mathematical equivalent transformations to manage activation outliers. RPTQ [297] rearranges the channels to reduce the variance within one quantization group, further enhancing the accuracy.

Some works [172; 277] used low-rank matrices to compensate for quantization error. Others [93; 318] used algorithm and architecture co-design to accommodate outliers. However, these approaches either suffer significant accuracy loss at extremely low-bit precision or lack practical hardware support. In this work, our method achieves notable accuracy with low-bit representation and ensures practical speedup.

6.8 Summary

We presented Atom, a low-bit quantization method that leverages the underlying hardware efficiently to achieve both high accuracy and high throughput for LLM serving. We use mixed-precision quantization with reordering, fine-grained group quantization, dynamic quantization, and KV-cache quantization to preserve accuracy while fully exploiting emerging low-bit hardware support. We integrate Atom into an end-to-end serving framework, achieving up to $7.73\times$ throughput enhancement compared to the FP16 baseline as well as maintaining less than 1.4% zero-shot accuracy loss.

Remarks on Author Contributions

My major contributions are as follow:

- Brainstormed and discussed the quantization algorithm design with Yilong.
- Implemented advanced quantization techniques including clipping, FP quantization, and GPTQ into Atom.
- Implemented Atom quantization for newer LLMs and MoEs such as Llama2 and Mixtral.
- Led the accuracy evaluations.
- Co-led the structuring and writing of the paper

Chapter 7

Palu: Compressing KV-Cache with Low-Rank Projection

Remarks on Chapter Material

The content of this chapter is adapted from the following paper (* indicates equal contributions):

- Chi-Chih Chang*, Wei-Cheng Lin*, Chien-Yu Lin*, and Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, Mohamed S. Abdelfattah, Kai-Chiang Wu, "Palu: Compressing KV-Cache with Low-Rank Projection", In Proceedings of International Conference on Learning Representations (ICLR), 2025.

7.1 Introduction

Large language models (LLMs) have propelled AI into new applications and capabilities, providing a high-level intelligence that previous machine learning (ML) models could not achieve. To speed up inference, caching the Key-Value states (KV-cache) in memory is a simple yet effective technique. However, the size of the KV-cache can grow rapidly, straining memory capacity and bandwidth especially with long context lengths [79]; further, the memory-bounded nature of the decoding stage limits inference speed when loading KV-cache data [88]. Therefore, KV-cache compression has become a central research topic for running LLMs efficiently.

Although emerging attention mechanisms such as Multi-Query Attention (MQA) [235], Group-Query Attention (GQA) [18] and Multi-head Latent Attention (MLA) [61] can reduce KV-Cache size, it either requires model pre-training or has a significant impact on model's accuracy when converting from traditional Multi-Head Attention (MHA) [49]. In contrast, post-training KV-cache compression techniques offer an alternative approach to advance efficiency for existing models. Among various KV-cache compression methods, quantization [178; 107] and token eviction [308; 282] stand out as effective strategies to reduce the memory footprint of KV-cache.

Quantization methods aim to reduce the bit-width used to represent each piece of data, while token eviction techniques focus on retaining a partial set of KV-cache. However, both methods neglect the hidden dimensions of the KV-Cache, where substantial redundancy often resides. To capitalize on this untapped potential, we introduce *Palu*, a post-training KV-cache compression framework that leverages low-rank projection to reduce the hidden dimension of KV tensors, offering an additional and orthogonal compression dimension to existing quantization and token eviction methods.

A naive way to utilize low-rank projection for compressing the KV-cache is by directly mapping cached matrices into low-rank space [137; 311]. However, this approach imposes an unacceptably heavy overhead of computing the decomposition matrices during runtime. To avoid this, *Palu statically decomposes the Key and Value-projection weight matrices* and *caches the latent representations* of the low-rank decomposition (see Fig. 7.1). This innovative design enables *Palu* to reduce memory while mitigating the runtime overhead of KV-cache low-rank decomposition.

In designing an effective decomposition strategy for attention modules with multiple attention heads, we observed a clear trade-off between accuracy and reconstruction overhead. Decomposing the projection matrices across all attention heads together improves accuracy by preserving global information, but this approach significantly increases reconstruction costs. On the other hand, decomposing each head separately reduces reconstruction overhead but leads to a higher loss in accuracy. To address this, *Palu* introduces a medium-grained, group-head low-rank decomposition that strikes a balance between accuracy and reconstruction efficiency.

For LLMs, each linear projection module has a different sensitivity to compression [234; 298]. To exploit the sensitivity and improve accuracy, we design an efficient *rank search algorithm* based on Fisher

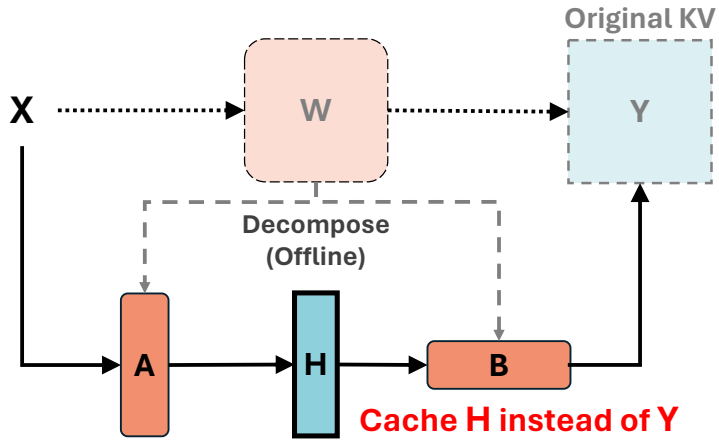


Figure 7.1: *Palu*'s *low-rank projection method* for KV-cache reduction. A weight matrix **W** of linear projection is decomposed into two low-rank matrices. Input **X** is down-projected to latent representation **H**, which is cached. **Y** can be reconstructed from **H** using the up-projection matrix **B**.

information [181; 174]. Our algorithm automatically assigns a higher rank for important matrices and lower ranks for less critical ones, boosting accuracy at the same overall KV-cache compression rate.

In addition to its low-rank decomposition, *Palu* is compatible with quantization techniques. We found that low-rank decomposition can introduce severe outliers in the latent representation, which significantly hinders accurate low-bit quantization. Although the Hadamard transformation has been shown to be effective for outlier elimination in recent studies [256; 27; 179; 53], its integration often introduces computational overhead during runtime. However, *Palu*'s inherent matrix pair structure makes it highly compatible with this technique, allowing the transformation matrices to be seamlessly fused into the forward and backward matrices, effectively mitigating outliers without impacting runtime efficiency.

We evaluate *Palu* on widely used LLMs and benchmarks. Our experiments demonstrate that *Palu* maintains strong zero-shot accuracy and perplexity with up to 50% low-rank compression. Moreover, when combining low-rank compression with quantization, *Palu* achieves an impressive **over 91.25% compression (11.4× reduction)** and yields a *significantly lower perplexity of 1.19* than KVQuant [107], a state-of-the-art KV-cache quantization method, which only achieves an 87.5% compression rate.

For latency evaluation, under a 50% KV-cache compression rate without quantization, *Palu* demonstrates up to *1.89× and 2.2× speedup* for RoPE-based and non-RoPE attention modules. When integrated with quantization, *Palu* achieves up to *2.91× and 6.17× acceleration* on RoPE-based and non-RoPE attention, respectively. These results underscore *Palu*'s ability to significantly reduce KV-cache memory footprint while boosting inference efficiency for LLMs.

Our key contributions include:

- *Palu*, a new post-training KV-cache compression framework that caches *low-rank latent representations* of Key and Value states.
- *Group-head low-rank decomposition (G-LRD)*, an optimization for balancing accuracy and reconstruction efficiency.
- An *automated rank search algorithm* for adaptively assigning ranks to each decomposed matrix, given a target compression rate.
- A co-designed *quantization compatibility optimization* that eliminates low-rank-induced outliers and imposes zero runtime overhead.

7.2 Background

Multi-Head Attention Mechanism

The multi-head attention (MHA) mechanism [258] is a core component of the transformer architecture. Given a new input token $\mathbf{x} \in \mathbb{R}^d$, an MHA with n heads projects the input into multiple queries, keys, and values using weight matrices $\mathbf{W}_i^q$, $\mathbf{W}_i^k$, and $\mathbf{W}_i^v$, respectively, for each head i, as shown by

$$\mathbf{q}_i = \mathbf{x}\mathbf{W}_i^q, \;\; \mathbf{k}_i = \mathbf{x}\mathbf{W}_i^k, \;\; \mathbf{v}_i = \mathbf{x}\mathbf{W}_i^v. \tag{7.1}$$

Here, $\mathbf{k}_i$ and $\mathbf{v}_i$ represent the key and value at time step t for head i. We can then compute the attention score for each head i and the corresponding attention output as

$$\mathbf{p}_{t,i} = \text{Softmax}\left(\frac{\mathbf{q}_i \mathbf{K}_i^T}{\sqrt{d_h}}\right), \;\; \mathbf{a}_i = \mathbf{p}_i \mathbf{V}_i, \tag{7.2}$$

where $\mathbf{K}_i$ and $\mathbf{V}_i$ denote the concatenation of current and all previous keys and values corresponding to the i-th head. The final MHA output is obtained by concatenating the outputs of all heads and then applying the out-projection layer $\mathbf{W}_o$, as shown by

$$\text{MHA}(\mathbf{x}) = \sum_{i=1}^{h} \mathbf{a}_i \mathbf{W}_i^o = \sum_{i=1}^{h} (\mathbf{p}_i \mathbf{V}_i)\mathbf{W}_i^o, \tag{7.3}$$

where $\mathbf{W}_i^o \in \mathbb{R}^{d_h \times d}$ represents the submatrices of the out-projection matrix for each head i.

Singular Value Decomposition (SVD)

SVD [137] is a commonly used technique for computing the low-rank approximation for a given matrix. SVD decomposes a given matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ into three matrices: $\mathbf{W} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. Here, $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices containing the left and right singular vectors, respectively. The matrix $\mathbf{\Sigma}$ is a diagonal matrix that consists of singular values. After decomposition, the low-rank approximation of $\mathbf{W}$ can be described as

$$\mathbf{W} \approx \mathbf{A}\mathbf{B}, \;\; \mathbf{A} = \mathbf{U}_r\sqrt{\mathbf{\Sigma}_r}, \;\; \mathbf{B} = \sqrt{\mathbf{\Sigma}_r}\mathbf{V}_r^T, \tag{7.4}$$

where $\mathbf{A} \in \mathbb{R}^{m \times r}$, $\mathbf{B} \in \mathbb{R}^{r \times n}$, $\mathbf{\Sigma}_r \in \mathbb{R}^{r \times r}$. $\mathbf{\Sigma}_r$ is a diagonal matrix containing the largest r singular values, and $\mathbf{U}_r$, $\mathbf{V}_r^T$ are corresponding singular vectors truncated from $\mathbf{U}$ and $\mathbf{V}^T$. This truncation and subsequent matrix formation let us approximate matrix $\mathbf{W}$ with two low-rank matrices $\mathbf{A}$ and $\mathbf{B}$, thereby reducing the storage by $\frac{mr+rn}{mn}$.

7.3 The Palu Framework

Compressing the KV-cache via Low-Rank Projection

A naïve approach to compress the KV-Cache with low-rank projection is to apply SVD directly on the KV-Cache and store the top-r singular vectors. However, this approach poses significant computational overhead during runtime that makes it impractical for deployments (see Appendix **??**).

To apply low-rank projection more efficiently than directly decomposing the KV-cache during runtime, *Palu* uses SVD to decompose the Key and Value projection matrices. This approach is based on the observation that low-rank decomposition rewrites the linear projection layer from $\mathbf{y} = \mathbf{x}\mathbf{W}$ into $\mathbf{y} = \mathbf{x}\mathbf{A}\mathbf{B}$.

Here, $\mathbf{A} \in \mathbb{R}^{d \times r}$ is the low-rank projection matrix, and $\mathbf{B} \in \mathbb{R}^{r \times d}$ is the reconstruction matrix derived by SVD. The forward process first *down-projects* the input token $\mathbf{x} \in \mathbb{R}^d$ into a low-dimensional latent

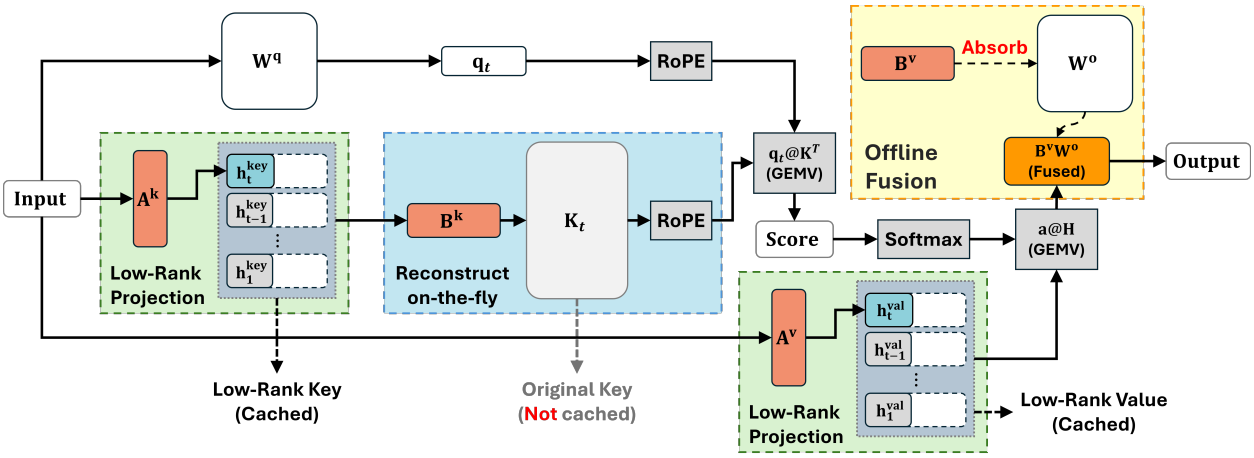


Figure 7.2: *Palu* uses low-rank decomposition ($\mathbf{W} \approx \mathbf{AB}$) to project the key (or value) to a lower-dimensional latent representation ($\mathbf{h}$), thereby reducing the size of the KV-cache. The original key ($\mathbf{K}_t$) is reconstructed on-the-fly with $\mathbf{B}^k$, and $\mathbf{B}^v$ is fused into $\mathbf{W}^o$ to avoid reconstruction overhead. The fusion also reduces the computational burden for output projection.

space $\mathbf{h} \in \mathbb{R}^r$ and then *up-projects* it back to the original space:

$$\mathbf{h} = \mathbf{A}\mathbf{x}, \quad \mathbf{y} = \mathbf{B}\mathbf{h} \tag{7.5}$$

This two-step process lets *Palu* (1) store the lower dimension latent representation instead of the origin key and value states, and (2) reconstruct them during decoding.

Integration with the Attention Mechanism and Offline Matrix Fusion

We now describe how *Palu* decomposes the key and value linear layers for the attention mechanism. For each attention head i, *Palu* applies SVD and maps the key-projection matrix $\mathbf{W}_i^k$ and value-projection matrix $\mathbf{W}_i^v$ into $\mathbf{A}_i^k\mathbf{B}_i^k$ and $\mathbf{A}_i^v\mathbf{B}_i^v$.

Based on the formula of attention output in Eq. 7.2, *Palu* absorbs the reconstruction matrix $\mathbf{B}_i^v$ into the output projection matrix $\mathbf{W}_i^o$ offline:

$$\mathbf{a}_i\mathbf{W}_i^o = (\mathbf{p}_i\mathbf{V}_i)\mathbf{W}_i^o = (\mathbf{p}_i\mathbf{H}_i^v\mathbf{B}_i^v)\mathbf{W}_i^o = \mathbf{p}_i\mathbf{H}_i^v(\mathbf{B}_i^v\mathbf{W}_i^o) \tag{7.6}$$

Such fusion lets *Palu* skip the explicit reconstruction of the full value vectors, reduce the number of matrix multiplications, and improve efficiency. A similar approach applies for calculating attention scores.

Matrix $\mathbf{B}_i^k$ can be fused into the query projection matrix $\mathbf{W}_i^q$ offline, as shown by

$$\mathbf{q}_i\mathbf{K}_i^T = \mathbf{q}_i(\mathbf{H}_i^k\mathbf{B}_i^k)^T = \mathbf{x}_t\mathbf{W}_i^q(\mathbf{B}_i^k)^T(\mathbf{H}_i^k)^T = \mathbf{x}_t\Big(\mathbf{W}_i^q(\mathbf{B}_i^k)^T\Big)(\mathbf{H}_i^k)^T. \tag{7.7}$$

Here, $\mathbf{B}_i^k \in \mathbb{R}^{r\times d_h}$ and $\mathbf{W}_i^q \in \mathbb{R}^{d\times d_h}$, so the fused matrix $(\mathbf{W}_i^q(\mathbf{B}_i^k)^T)$ has size $\mathbb{R}^{d\times r}$. This fusion boosts computational efficiency by reducing the matrix dimension during attention score calculation.

Compability with Positional Embedding

Recent LLMs, such as the Llama family, apply Rotary Positional Embedding (*i.e.,* RoPE [243]) onto the Query and Key states prior to their multiplication.

The non-linear nature of these positional embeddings prevents the matrix fusion of attention scores, as outlined in Eq. 7.7. To address this, *Palu* dynamically reconstructs the keys from latent representations on the fly. Specifically, *Palu* employs a custom GPU kernel that efficiently integrates key reconstruction, RoPE application, and subsequent Query-Key multiplication into a single fused operation. By transferring only the low-rank latent representations and performing reconstruction directly within GPU shared memory. By doing so, *Palu* substantially reduces the off-chip memory footprint, optimizing the memory-bound LLM decoding[299] process through a memory-computation trade-off. Detailed implementation specifics of this kernel are provided in Appendix 7.4.

Note that for some positional embedding methods, such as ALiBi [216], positional embedding is not directly applied to the Key states. Consequently, the fusion described in Eq. 7.7 remains valid. For these non-RoPE attention modules, *Palu* achieves greater speedup compared to RoPE-based attention, as their reconstruction can be avoided with matrix fusion.

Decomposition Granularity

Multi-Head Low-Rank Decomposition

We name the per-head decomposition scheme in Sec. 7.3 as *multi-head low-rank decomposition (M-LRD)*. We found M-LRD often causes a non-negligible accuracy degradation (discussed further in Sec. 7.4), possibly because SVD fails to capture the common information shared across heads. Therefore, alternative approaches are needed to preserve model accuracy.

Joint-Head Low-Rank Decomposition

An alternative approach is to jointly decompose weight matrices for all heads. By considering the combined weight matrix $\mathbf{W}_{\text{joint}} = [\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_n] \in \mathbb{R}^{d\times(d_h\cdot n_h)}$, we can perform a single low-rank decomposition $\mathbf{W}_{\text{joint}} \approx \mathbf{A}_{\text{joint}}\mathbf{B}_{\text{joint}}$, where $\mathbf{A}_{\text{joint}} \in \mathbb{R}^{d\times r_{\text{joint}}}$ and $\mathbf{B}_{\text{joint}} \in \mathbb{R}^{r_{\text{joint}}\times(d_h\cdot n_h)}$. We call this scheme *joint-head low-rank decomposition (J-LRD)*.

J-LRD has the advantage of preserving the common principal components shared among different heads. This occurs because SVD is particularly effective at capturing the dominant components when applied to a larger, combined matrix, resulting in a more accurate approximation.

For J-LRD, the joint latent representation shared among all heads can be computed with $\mathbf{h}_{\text{joint}} = \mathbf{x}\mathbf{A}_{\text{joint}}$. During decoding, the original states for each head can be reconstructed via

$$\big[\mathbf{y}_1, \ldots, \mathbf{y}_n\big] = \mathbf{h}_{\text{joint}}\mathbf{B}_{\text{joint}}.$$

Despite better-preserving model accuracy, J-LRD introduces *significant computational and memory overhead* during decoding. Specifically, the total number of floating point operations (FLOPs) to reconstruct the Key or Value state of all heads now becomes $L \cdot r_{\text{joint}} \cdot d_h \cdot n$. Assuming the same size as the total low-rank latent representations (*i.e.,* $r_{\text{joint}} = \sum_{i=1}^{n} r_i$), the total reconstruction cost is n times higher than M-LRD, whose total FLOPs is $L \cdot r_i \cdot d_h \cdot n$. When considering the matrix fusion in Sec. 7.3, the fused matrix of J-LRD has a size of $r_{\text{joint}} \cdot d \cdot n$, which is also n times larger than M-LRD, leading to substantial higher memory consumption.

Group-Head Low-Rank Decomposition

To balance the trade-off between accuracy and reconstruction cost, we propose *group-head low-rank decomposition* (G-LRD). G-LRD decomposes the matrices for a group of heads together. With combined weight matrices, it captures shared information within each group while limiting computational overhead and preserving accuracy.

To illustrate the G-LRD process, consider the weight matrices for a group of s heads, $\mathbf{W}_{g_j} = \big[\mathbf{W}_{j,1} \ldots \mathbf{W}_{j,s}\big]$, where $\mathbf{W}_{g_j} \in \mathbb{R}^{d\times(d_h\cdot s)}$. We low-rank decompose $\mathbf{W}_{g_j} \approx \mathbf{A}_{g_j}\mathbf{B}_{g_j}$, where $\mathbf{A}_{g_j} \in \mathbb{R}^{d\times r_g}$

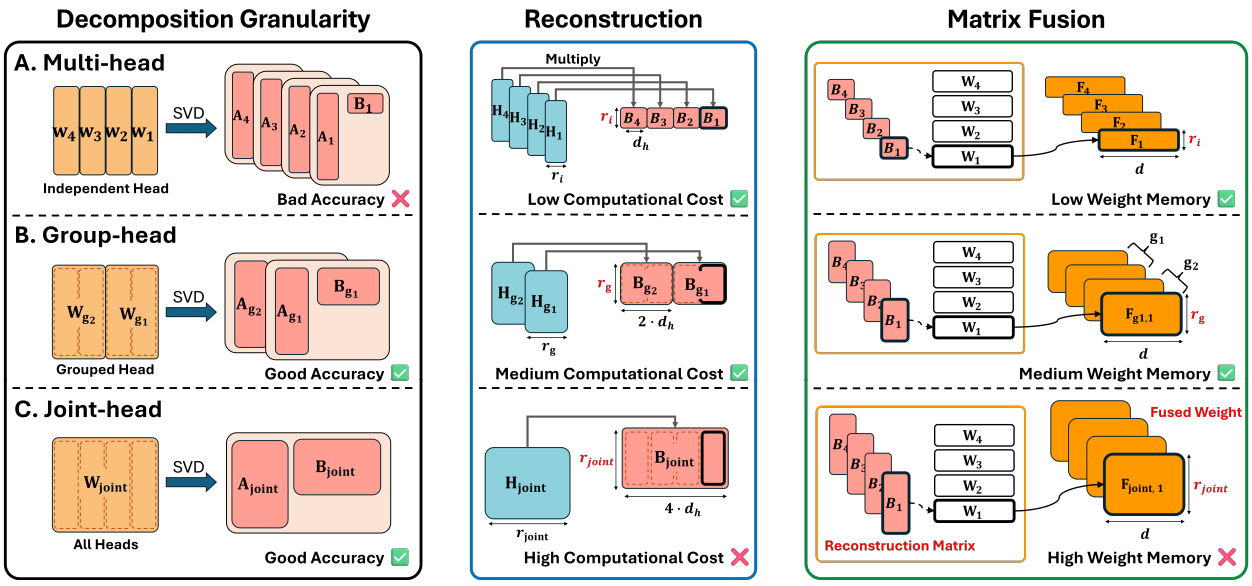


Figure 7.3: Performing decomposition at different granularities. Jointly decomposing multiple heads can achieve higher accuracy. Assuming the same total size of the latent representations (*i.e.*, $4 \cdot r_i = 2 \cdot r_g = r_{\text{joint}}$), the FLOPs for reconstruction overhead in joint-head decomposition schemes are 4 times larger than those in multi-head ones.

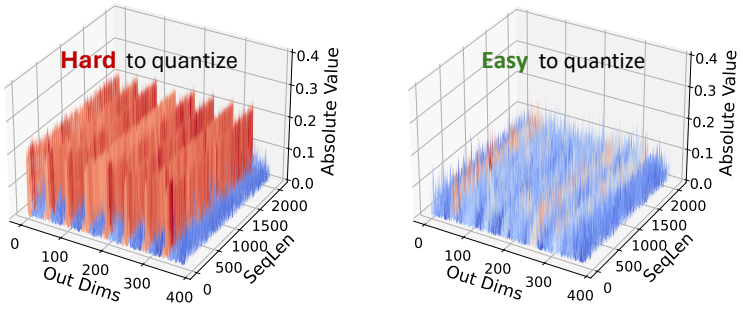
and $\mathbf{B}_{g_j} \in \mathbb{R}^{r_g \times (d_h \cdot s)}$. The latent representation shared among attention heads in the same group can be computed as $\mathbf{h}_{g_j} = \mathbf{x}\mathbf{U}_{g_j}$. During decoding, the original key or value for each head can be reconstructed via

$$[\mathbf{y}_{j,1} \dots \mathbf{y}_{j,s}] = \mathbf{h}_{g_j}\mathbf{B}_{g_j}.$$

The FLOPs for reconstructing the keys and values for all heads in G-LRD is $L \cdot r_g \cdot d_h \cdot n$. Comparing the cost to J-LRD and assuming the same total rank size ($r_g \cdot n_g = r_{\text{joint}}$), G-LRD reduces the reconstruction cost by n_g. Similarly, G-LRD also reduces the fused matrix size by n_g. To sum up, G-LRD offers a middle ground between computation overhead and approximation accuracy. We illustrate M-LRD, J-LRD and G-LRD in Fig. 7.3.

Automatic Rank Allocation

To allocate an ideal rank size to the decomposition target, it is crucial to accurately estimate the importance of the target matrix (*e.g.,* grouped weights). In *Palu*, we identify **Fisher information** [181; 174] as an accurate approximator since it can quantify the amount of information for each parameter. We then employ



(a) Low-Rank Key Cache (b) Low-Rank Key Cache (with Hadamard)

Figure 7.4: Activation distribution of the low-rank key caches at the 4^{th} Llama-2 attention layer.

the sum of Fisher information to estimate the importance of the weight matrix of each linear layer [10].

Assuming that the compression sensitivity is proportional to Fisher information, we determine the rank for each weight matrix by computing the ratio of its Fisher information to the total Fisher information across all decomposition targets. We use this ratio to allocate the compression rate (*i.e.,* rank level r), ensuring that more important layers retain higher rank levels.

Quantization Compatibility

We integrate quantization into *Palu* to compress the KV-cache further. We observe that low-rank compressed latent representations have severe outliers, which limit quantization applicability in *Palu*. Unlike natural outliers described in previous KV-cache quantization literature [178; 107], these outliers are induced by SVD-based low-rank factorization.

Fig. 7.4 (a) shows the distribution of low-rank compressed key states from a layer of Llama-2 with G-LRD. Repeating outlier patterns appear at the beginning of each decomposed group because SVD arranges larger eigenvalues in the initial rows or columns, resulting in rapidly descending values in the latent representation. This pattern stretches the data distribution and hurts quantization accuracy.

Inspired by recent LLM quantization literature [27; 256], we apply the Walsh-Hadamard transform (WHT) to eliminate outliers (Fig. 7.4 (b)), enabling a high quantization accuracy. However, this transformation introduces an extra matrix multiplication with associated runtime overhead. Unlike earlier methods [27] that must apply online WHT when quantizing KV-cache, we optimize this process by integrating the Hadamard matrix into low-rank decomposed weights with no additional compute overhead, as described by

$$\mathbf{W} \approx \mathbf{AB} = (\mathbf{AR})(\mathbf{R}^T\mathbf{B}) = \hat{\mathbf{A}}\hat{\mathbf{B}}, \tag{7.8}$$

where $\mathbf{R}$ is the Hadamard matrix. This optimization allows *Palu* to integrate the proposed low-rank compression technique with low-bit quantization. Our experiments show that, on top of the low-rank compression, our quantization method only *negligibly increases perplexity*, even at extreme levels such as *3-bit or 2-bit* with a simple per-token quantization scheme (see Sec. 7.4).

7.4 Experiments

Experiments Setup

Models and Tasks. We evaluate *Palu* on four LLM families, Llama-2 [253], Llama-3 [71], Mistral [126] and LongChat [163]. For accuracy evaluation, we measure perplexity on the WikiText-2 [187] and C4 [220] datasets and use LM-Evaluation-Harness [82] to measure zero-shot accuracy on six common sense tasks. We also evaluate long context accuracy on 16 tasks in LongBench [29]. Unless specification, we refer to baseline as a model with non-compressed KV-cache. See Appendix **??** for further details on the dataset and settings.

Compression Settings. We implemented *Palu* based on the Huggingface library [276]. Decomposition of the Key and Value projection layers was performed using the truncation-aware SVD method proposed by SVD-LLM [266]. Unless otherwise specified, *Palu*'s results are G-LRD with a group size of 4 (gs-4), with equal rank size for each group. To calculate Fisher information in rank searching, we used 2048 random samples from Wikitext-2, each with a sequence length of 1024. For quantization integration in *Palu*, we use a simple per-token, asymmetric integer quantization. For evaluation on quantization results, we compare *Palu* to advanced KV-cache quantization methods, including Atom [312], KVQaunt [107], and KIVI [178]. Refer to Sec. 7.5 for a brief summary of these methods.

GPU Kernels Implementation. We implemented a customized kernel for attention score with reconstruction in Triton [249] (See Appendix **??**). For quantization integration, we implemented kernels in CUDA for attention output and non-RoPE attention score, where matrix fusion can be applied (refer to Sec. 7.3 and Fig. 7.2). Our low-precision kernel fuses the dequantization process and the follow-up multiplication with low-rank compressed keys or values, enabling efficient processing on quantized latent KV-cache. When

evaluating speedup with quantization, we compare to the non-compressed baseline and KIVI [178], which we use their official code[1] in our experiments.

Results with Different Decomposition Granularity

We evaluate perplexity and zero-shot accuracy of *Palu* with a **50% low-rank compression rate** using M-LRD, G-LRD, and J-LRD on Llama2-7B and Llama3-8B-Instruct, and present the results in Table 7.1.

Table 7.1: Perplexity and zero-shot accuracy of *Palu* at 50% compression rate.

Model	Method	Perplexity ↓		Zero-Shot Accuracy (%) ↑						
		Wiki2	C4	OBQA	Hella	PIQA	ARC-e	ARC-c	Wino	Avg.
Llama2-7B	Baseline	5.47	7.26	44.20	76.00	78.07	76.30	46.42	69.30	65.05
	J-LRD	5.62	7.75	45.40	75.57	77.48	75.97	45.31	69.22	64.82
	G-LRD	6.01	9.82	43.60	73.39	76.33	73.02	42.57	66.77	62.61
	M-LRD	6.75	12.01	39.60	65.35	74.76	67.17	35.24	64.64	57.79
Llama3-8B-Inst	Baseline	8.28	13.01	43.20	75.80	78.62	81.61	56.83	71.90	67.99
	J-LRD	9.12	15.90	43.40	73.20	76.50	79.63	51.96	72.45	66.19
	G-LRD	10.11	17.87	42.60	70.36	76.06	76.30	48.99	72.38	64.45
	M-LRD	12.38	23.02	38.80	63.04	73.67	69.78	42.58	62.51	58.40

Perplexity Evaluation. As Table 7.1 shows, for the Llama2-7B model, *Palu*'s M-LRD method fails to maintain a low perplexity at a 50% compression rate. In contrast, despite having a high recomputation cost, J-LRD significantly outperforms M-LRD and achieves a 5.62 perplexity on WikiText-2.

For G-LRD, which still maintains a low computation cost, yields a 6.01 perplexity on Wikitext-2, showing a great balance between model accuracy and compression overheads. The same trend is observed in the Llama-3-8B model as well. More results Llama-2-13B can be found in Appendix **??**.

Zero-shot Evaluation Results. Similar to the perplexity evaluation, the J-LRD method demonstrates the best performance for the zero-shot accuracy on Llama-2-7B, with only a 0.23% average accuracy degradation. M-LRD method results in the lowest average performance, with a 7.26% drop in accuracy compared

[1] https://github.com/jy-yuan/KIVI

to the baseline. In comparison, G-LRD only has a 2.4% average accuracy decline, offering a sweet spot between model accuracy and compression overheads again.

Results of Quantization Integration

Table 7.2 showcases the impact of quantization on perplexity and KV-cache size when combined with *Palu*. With 3-bit quantization, *Palu* incurs only a **slight 0.08 and 0.23 perplexity increase** at 30% and 50% low-rank compression rate. These demonstrate a minimal accuracy trade-off for significant compression gains compared to the 16-bit baseline. Notably, at 2-bit quantization, *Palu* decisively outperforms the state-of-the-art KVQuant method, **reducing perplexity by 1.19 and 0.54**, while further slashing memory usage by 30% and 50%. These results establish *Palu* with quantization as a superior KV-cache compression method.

Table 7.2: Quantization perplexity and KV-cache size for Llama2-7B on WikiText-2. For perplexity, sequence length is 4096. KV-cache size is demonstrated for 128K sequence length.

Method	Bit	PPL	KV-cache Size (GB)	Comp. Rate
Baseline	16	5.12	64.0	-
Palu-30%	16	5.25	44.8	30%
Palu-50%	16	5.63	32.0	50%
Atom	3	6.15	12.6	80.32%
KVQuant	3	5.35	12.0	81.25%
Palu-30%	3	**5.33**	**8.4**	**86.87%**
Palu-50%	3	5.77	6.0	90.63%
Atom	2	117.88	8.6	86.56%
KVQuant	2	6.95	8.0	87.50%
Palu-30%	2	**5.76**	**5.6**	**91.25%**
Palu-50%	2	**6.41**	**4.0**	**93.75%**

Evaluation on Long Context Datasets

To access *Palu*'s ability for long-context scenarios, we evaluate baseline, KIVI and *Palu*'s accuracy on LongBench [29] Here, we evaluate the Mistral-7B and LongChat-7B models, which have up to 32K context length. We report the average score for each task type separately, as well as the overall average across all

Table 7.3: Experiment Results on LongBench: The average bit widths represent the total storage cost per element in the compressed KV-cache, including the overhead of quantization parameters. These values are calculated for each approach, assuming a context length of 10K.

Model	Method	Avg. Bits	Comp. Ratio	Multi-QA	Single-QA	Summarization	Few-Shot	Code	Synthetic	Avg.
Mistral-7B-v0.2	Baseline	16	1.00x	29.63	36.43	28.10	66.71	54.16	44.87	42.54
	Palu-30%	16	1.43x	29.83	36.52	27.48	65.70	55.16	37.92	41.55
	Palu-50%	16	2.00x	26.92	35.33	26.01	64.04	44.54	16.88	36.23
	KIVI-2	3.16	5.05x	28.81	35.07	27.60	66.45	54.47	40.28	41.45
	Palu-30% (3 bits)	3.13	7.59x	29.48	36.40	27.20	65.73	53.19	34.74	40.77
	Palu-50% (3 bits)	3.13	10.6x	26.73	32.72	25.73	63.25	44.43	18.57	35.71
LongChat-7B-v1.5	Baseline	16	1.00x	23.95	31.12	26.74	63.80	56.91	15.25	36.32
	Palu-30%	16	1.43x	22.42	29.43	25.52	62.87	58.99	14.25	35.45
	Palu-50%	16	2.00x	22.61	25.33	22.73	60.12	43.52	6.84	30.82
	KIVI-2	3.16	5.06x	23.24	30.19	26.47	63.54	53.51	16.13	35.60
	Palu-30% (3 bits)	3.13	7.59x	23.12	29.21	25.04	61.99	54.38	11.25	34.33
	Palu-50% (3 bits)	3.13	10.6x	18.56	24.14	22.35	58.76	40.50	6.02	29.03

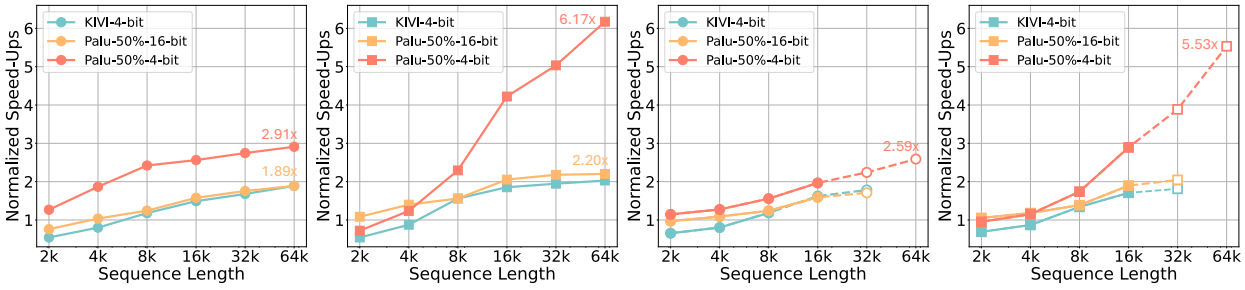
16 tasks. The results are shown in Table 7.3. We report the accuracy of KIVI using the configuration with a group size of 32 and 128-element fp16 residual [178].

As Table 7.3 indicates, we find that at a 50% low-rank compression level, *Palu* is relatively difficult to fully preserve accuracy. However, at a 30% compression level, *Palu* achieves only a minor average accuracy drop ($< 1\%$) compared to the baseline for both models. Furthermore, *Palu* can quantize the low-rank latent KV-cache down to 3 bits, with less than 1% further accuracy degradation. Overall, *Palu* maintains a strong 40.77% and 34.33% average accuracy for Mistral-7B and LongChat-7B, with an impressive 7.59x compression ratio. Compared to KIVI [178], *Palu* achieves a similar accuracy, while having an additional 30% compression rate from low-rank. Notably, *Palu* does not require the complex grouped quantization and mixed-precision techniques employed by KIVI, resulting in a high inference efficiency (see Sec. 7.4 for details).

Latency Evaluation

In this section, we provide latency and speedup evaluation, using Llama-2-7b as the base model. We measure decode latency on a single RTX 4090 GPU and compare *Palu* to the FP16 and KIVI-4-bit baselines. We evaluate *Palu*'s latency at a 50% compression rate, where we set compression rates for key and value to

75% and 25%, respectively. This allocation is based on our observations from the rank allocation results. For the FP16 baseline, we use the default implementation from HuggingFace. For KIVI, we use the CUDA kernels from its official repository. Due to the small memory capacity of RTX 4090 GPU, we adopt a 4-bit quantization [77] for the weights of all linear layers. Our results are the average of 100 runs.



(a) Attention Module (w/ RoPE) (b) Attention Module (w/o RoPE) (c) End-to-End Model (w/ RoPE) (d) End-to-End Model (w/o RoPE)

Figure 7.5: Normalized speedup for both the attention module and end-to-end model decoding. Solid lines represent exact measurements, while dashed lines indicate the FP16 baselines are out of memory, and the speedups are compared to the estimated baseline's latency.

Speedups of Attention Module and End-to-end Decoding.

Attention module speedup. We compare latency against standard attention without compression or quantization and show the speedups of *Palu* and KIVI-4-bit in Fig. 7.5 (a) and (b) for RoPE-based and non-RoPE attention. For RoPE-based attention, we applied our online reconstruction kernel for key and employed offline fusion for value as described in Sec 7.3.

As shown in Fig. 7.5 (a), *Palu* has minimal to no speedup when the sequence length is short, e.g. 4K. However, as sequence length increases, *Palu* delivers substantial performance gains. At 64K input length, *Palu* achieves a **1.89× speedup** over the FP16 baseline when using low-rank projection alone. By further applying 4-bit quantization to the Value states, the **speedup rises to 2.91**× for the same 64K context length, owing to our optimized low-precision kernel and reduced memory loading times. This performance notably surpasses KIVI-4-bit, which only achieves a 1.89× speedup at 64K, hindered by the overheads of its fined-grained group quantization. Notably, for RoPE-based attention, *Palu*-4-bit does not quantize key, as our online reconstruction kernel only supports FP16 precision for now.

For non-RoPE attention, we apply matrix fusion to both the Key and Value states (Eq. 7.7), effectively eliminating all reconstruction overhead. At a 64K sequence length with a 50% compression rate, *Palu*

achieves a **2.20× speedup over the FP16 baseline**. By further applying 4-bit quantization to both the Key and Value states, *Palu* boosts the speedup to **6.17×** for 64K input length. These results demonstrate that combining low-rank compression and quantization significantly enhances inference efficiency, particularly in long-context scenarios.

End-to-end speedup. We present the end-to-end speedups in Fig. 7.5 (c) and (d), measuring the decoding latency of generating the next token at various input lengths. Similar to the attention performance results, *Palu* shows minimal or no speedup for short sequences but delivers significant acceleration for longer sequences. Without quantization, *Palu* achieves up to **1.71× and 2.05× speedups** for RoPE-based and non-RoPE models, respectively. With a 50% compression rate, *Palu* runs up to 32K input length on an RTX 4090 GPU. By incorporating 4-bit quantization, *Palu* handles even longer sequences and delivers **2.59× and 5.53× end-to-end speedups** at a 64K sequence length. *Palu* integrated with quantization provides a substantial speed advantage over KIVI-4-bit, which only reaches 1.78× and 1.81× speedups at 32K sequence length for RoPE and non-RoPE scenarios, respectively, and is out-of-memory for longer sequences.

Kernel for RoPE-Based Attention Score

In this section, we evaluate the performance of our online reconstruction kernel for RoPE-based attention scores. We measure latency from the **pre-RoPE query vector** to **post-GEMV attention score**, and compare it with PyTorch's GEMV, which is used in the baseline attention (see Fig. 7.2).

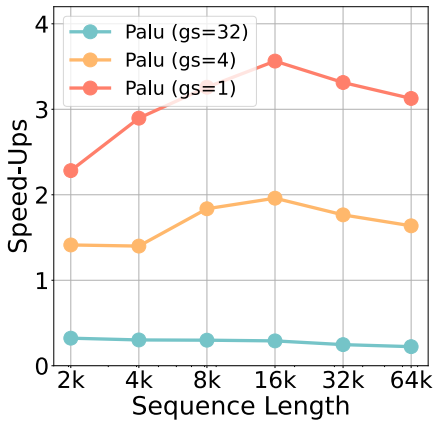


Figure 7.6: Speedup of *Palu*'s attention score kernel with online reconstruction.

We present speedups for group size 1, 4, and 32 at different sequence lengths in Fig. 7.6. For gs-32 (J-LRD), the highest accuracy decomposition, the high reconstruction cost causes a significant slowdown across all sequence lengths. For gs-1 (M-LRD), our kernel achieves up to a 3.56× speedup at sequence length 16K, showing strong performance when moderate accuracy loss is acceptable. For gs-4 (G-LRD), our kernel reaches up to 1.95× speedup. These results emphasize the need to explore various decomposition granularities for better accuracy and speed tradeoffs.

We also observe that speedup decreases for sequence lengths beyond 16K due to rising reconstruction costs, shifting the online reconstruction from memory- to compute-bound. A potential optimization is to quantize the decomposed weight matrices further and leverage high-throughput, low-precision hardware (*e.g.,* INT4 Tensor Cores) for online reconstruction, which we leave for future work. Despite the speedup drop at longer lengths, *Palu*'s overall attention speedup increases with longer input, thanks to matrix fusion on the Value state and the reduced memory footprint.

7.5 Related Work

SVD for LLM Compression. Several works have explored using SVD to compress LLMs. An early approach [197] applied standard SVD to weight matrices, resulting in significant compression errors. FWSVD [109] addressed this by using Fisher information to prioritize parameters, while ASVD [298] considered activation outliers. SVD-LLM [266] further minimized compression loss for each singular value. Unlike these methods, which compress model weights, *Palu* focuses on reducing KV-cache size.

KV-cache Quantization. Quantization is a widely used technique for compressing KV-cache. Atom [312] applies simple per-token quantization, while WKVQuant [300] introduces a two-level scheme to enhance accuracy. KIVI [178] uses per-channel and per-token quantization for Keys and Values, combined with ultra fine-grained group quantization. KVQuant [107] employs a similar setup but incorporates non-uniform quantization and sparse matrices to handle outliers. On top of these approaches, GEAR [139] adds a low-rank matrix to compensate for quantization errors. In *Palu*, we leverage low-rank techniques to exploit hidden dimension redundancy and achieve outstanding compression through simple per-token quantization.

MLA. The recently released DeepSeek-V2 model [61] introduces the MLA mechanism, which reduces KV-cache size by down-projecting Key and Value to a low-rank space and reconstructing them to full rank at runtime. Although MLA may seem similar to *Palu* at a high level, particularly with J-LRD, our design and derivation processes are fundamentally different. Unlike MLA, a new attention mechanism requiring pre-training, *Palu* is specifically designed for post-training integration. *Palu* focuses on converting existing models with MHA or GQA to support low-rank compressed KV-cache, preserving high accuracy while enhancing inference efficiency.

7.6 Summary

We introduce *Palu*, a novel KV-cache compression framework that decomposes linear projection weight matrices and caches the compressed latent representations. We propose various optimizations, including group-head low-rank decomposition, automatic rank allocation algorithm, quantization compatibility enhancement, and customized kernels with operator fusion. With these optimizations, *Palu* can maintain accuracy while achieving significant memory reduction and high inference speedup.

Remarks on Author Contributions

My major contributions are as follow:

- Brainstormed with Chi-Chih on the core ideas and co-designed the inference system and kernel with Chi-Chih and Wei-Cheng.
- Co-led the project direction, transitioning the focus from LLM weight compression to KV-Cache compression, and clearly differentiating our approach from Multi-Latent Attention (MLA) used in DeepSeek-V2 and DeepSeek-V3.
- Led and co-authored the structuring and writing of the paper.

Chapter 8

Conclusion

This thesis addresses the critical challenges posed by the growing computational and memory demands of modern machine learning workloads, including Convolutional Neural Networks (CNNs), Graph Neural Networks (GNNs), neural rendering (NeRFs), and Large Language Models (LLMs). By exploring two complementary strategies, sampling and compression, I have developed innovative approaches together with highly optimized system designs that significantly enhance efficiency. Sampling-based methods, such as CacheSample, FastSR-NeRF, and TeleRAG, selectively focus computation to substantially reduce memory and computational overhead. Concurrently, compression methods, including SPIN, Atom, and Palu, systematically minimize storage footprints and inference latency without sacrificing model performance. Together, these strategies form a synergistic approach, providing robust solutions that enhance scalability, lower resource usage, and maintain high accuracy. Ultimately, this integrated methodology paves the way toward broader accessibility and deployment of advanced AI systems.

Remark on Methodology. Although the works presented in this thesis employ a variety of techniques and target diverse workloads, they share a common methodological pattern for problem-solving.

Each project typically begins with a strong empirical observation. For example, *CacheSample* was motivated by the finding that GNNs can preserve accuracy even when a substantial number of edges are randomly dropped. Similarly, *TeleRAG* was initiated from the observation that inputs to different stages of RAG exhibit high similarity. This insight was further validated through analysis of IVF cluster hit rates using real RAG pipelines and datasets.

To uncover such observations, it is often necessary to re-implement or significantly modify open-source codebases and run them at scale with large datasets. These explorations yield deep insights that naturally point to promising solution directions.

However, another recurring theme across the projects in this thesis is that achieving real-world performance benefits typically requires nontrivial system-level optimization. For instance, in *CacheSample*, although sparsifying the input graph can theoretically reduce SpMM kernel time, naive edge sampling methods often incur overheads that outweigh the kernel speedup. In *Atom*, while group quantization enables accurate 4-bit inference for LLMs, practical inference would not be feasible without custom low-bit GEMM kernels tailored to group-quantized weights and activations.

Therefore, our approach consistently incorporates system considerations from the earliest stages of design. We prioritize solutions that are not only effective in theory but also feasible to build in practice. In many cases, implementing an algorithm efficiently presents challenges as intricate and innovative as the algorithm design itself.

The experience gained from this body of work strongly supports the view that algorithm–system co-design is essential to achieving machine learning workloads that are both accurate and accessible.

8.1 Future Work

This thesis presents effective sampling and compression techniques that enhance the efficiency of various deep learning models. However, the pace of advancement in artificial intelligence continues to accelerate rapidly. In this section, I outline my vision for future work aimed at further improving the efficiency of emerging machine learning workloads.

Exploring Redundancy in Multi-Modality

A prominent trend in modern machine learning is the development of unified models capable of handling multiple modalities. For instance, ChatGPT-4o demonstrates the ability to process and generate both text and images using a single model architecture. More broadly, we observe a convergence where Transformer-based models are increasingly applied beyond text, extending to images, videos, and audio. Similarly, diffusion-based models, originally developed for image generation, are being adapted to support text and

audio modalities.

These advancements, however, come at the cost of increased model complexity, larger parameter counts, and greater training data requirements—barriers that confine the development and deployment of such models to a few organizations with vast computational resources.

Despite the inherent complexity of multi-modal tasks—especially in domains like video and audio generation—there remains substantial potential to uncover and exploit redundancy, both within and across modalities. For example, video data naturally exhibits high temporal redundancy due to frame-to-frame similarity, while audio often contains significant sparsity, such as silent intervals between spoken words. Furthermore, cross-modal redundancy can also be leveraged; compact tensor representations may be shared across text and image modalities, or the most informative representation from one modality can guide the compression of others when multiple modalities are processed jointly.

As AI systems continue to evolve and tackle increasingly diverse and complex tasks, I believe that identifying and harnessing both intra- and inter-modal redundancy will be essential for sustaining scalability and democratizing access to advanced models. Efficient representation and compression strategies will play a critical role in enabling the broader adoption of multi-modal AI across diverse applications and resource settings.

Leveraging Hardware Advancements through Algorithm–System Co-Design

While much progress in AI has been driven by advances in model architectures, training methods, and data quality, there has also been rapid innovation in hardware from both major corporations and startups. Recent examples include group quantization support via the Microscaling format [226] on NVIDIA's Blackwell GPUs [204], high-speed chip-to-chip communication using silicon photonics in the upcoming Rubin architecture [273], wafer-scale chips from Cerebras [15], and fully on-chip inference with Groq's LPU [119].

To fully realize the potential of future large-scale models, it is essential to co-design algorithms and systems that exploit these hardware advancements. Efficient AI will require utilizing every part of the chip—not only as intended, but also in novel ways that repurpose existing components. For instance, rasterization units originally designed for 3D graphics have been used to accelerate neural rendering [50], while ray tracing cores have shown promise in supporting vector similarity search [177].

As models grow in scale and complexity, staying attuned to new hardware capabilities—and creatively mapping workloads to them—will be key to pushing the boundaries of performance and efficiency. Beyond merely adapting algorithms to hardware, a critical research direction is to re-architect models to better align with hardware strengths. I believe that deep, hardware-aware algorithm–system co-design will be central to enabling scalable, efficient, and accessible multi-modal AI in the future.

Enhancing Robustness for Efficient Methods

A commonly overlooked aspect of efficiency-focused methods is their robustness to failures and adversarial attacks. Since many efficient techniques work by eliminating redundancy, they may inadvertently reduce a model's ability to tolerate noise, faults, or unexpected inputs. Yet as we increasingly rely on these techniques to support large-scale models—especially in the context of limited global energy resources—robustness becomes even more critical.

In the future, large models may operate almost entirely on efficiency-driven designs. In such settings, a single point of failure could not only disrupt service but also lead to significant resource waste at the data center level. Ensuring robustness in these systems is therefore not just a matter of reliability, but also of sustainability.

Looking ahead, I plan to explore how robustness and efficiency can be jointly optimized. This includes identifying vulnerabilities introduced by compression or sampling techniques, and developing lightweight mechanisms that enhance fault tolerance without negating the efficiency gains.

Bibliography

[1] Apple metal performance shader. https://developer.apple.com/documentation/metalperformanceshaders/.

[2] Apple neural engine. https://machinelearning.apple.com/research/neural-engine-transformers.

[3] The blender software. https://www.blender.org/.

[4] Genspark. https://www.genspark.ai/.

[5] Huggingface edsr. https://huggingface.co/eugenesiow/edsr.

[6] Intel onednn. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.html.

[7] Mobilenerf official source code. https://github.com/google-research/jax3d/tree/main/jax3d/projects/mobilenerf.

[8] Nvidia tensor core. https://www.nvidia.com/en-us/data-center/tensor-cores/.

[9] Perplexity. https://www.perplexity.ai/.

[10] ABDELFATTAH, M. S., MEHROTRA, A., DUDZIAK, Ł., AND LANE, N. D. Zero-cost proxies for lightweight NAS.

[11] ABDELKHALIK, H., ARAFA, Y., SANTHI, N., AND BADAWY, A.-H. Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis, 2022.

[12] ABHYANKAR, R., HE, Z., SRIVATSA, V., ZHANG, H., AND ZHANG, Y. Apiserve: Efficient api support for large-language model inferencing. *arXiv preprint arXiv:2402.01869* (2024).

[13] AGRAWAL, A., KEDIA, N., PANWAR, A., MOHAN, J., KWATRA, N., GULAVANI, B. S., TUMANOV, A., AND RAMJEE, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310* (2024).

[14] AGRAWAL, A., KEDIA, N., PANWAR, A., MOHAN, J., KWATRA, N., GULAVANI, B. S., TUMANOV, A., AND RAMJEE, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve, 2024.

[15] AI, C. Cerebras wafer scale engine 3, 2025.

[16] AI@META. Dataset card for "wiki_dp". https://huggingface.co/datasets/facebook/wiki_dpr, 2020.

[17] AI@META. Llama 3 model card. https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md, 2024.

[18] AINSLIE, J., LEE-THORP, J., DE JONG, M., ZEMLYANSKIY, Y., LEBRÓN, F., AND SANGHAI, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[19] AKKIRAJU, R., XU, A., BORA, D., YU, T., AN, L., SETH, V., SHUKLA, A., GUNDECHA, P., MEHTA, H., JHA, A., ET AL. Facts about building retrieval augmented generation-based chatbots. *arXiv preprint arXiv:2407.07858* (2024).

[20] ALLAMANIS, M., BROCKSCHMIDT, M., AND KHADEMI, M. Learning to represent programs with graphs. In *International Conference on Learning Representations* (2018).

[21] ANSEL, J., YANG, E., HE, H., GIMELSHEIN, N., JAIN, A., VOZNESENSKY, M., BAO, B., BELL, P., BERARD, D., BUROVSKI, E., CHAUHAN, G., CHOURDIA, A., CONSTABLE, W., DESMAISON, A., DEVITO, Z., ELLISON, E., FENG, W., GONG, J., GSCHWIND, M., HIRSH, B., HUANG, S., KALAMBARKAR, K., KIRSCH, L., LAZOS, M., LEZCANO, M., LIANG, Y., LIANG, J., LU, Y., LUK, C. K., MAHER, B., PAN, Y., PUHRSCH, C., RESO, M., SAROUFIM, M., SIRAICHI,

M. Y., Suk, H., Zhang, S., Suo, M., Tillet, P., Zhao, X., Wang, E., Zhou, K., Zou, R., Wang, X., Mathews, A., Wen, W., Chanan, G., Wu, P., and Chintala, S. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2024), ASPLOS '24, p. 929–947.

[22] Arkhipov, D. I., Wu, D., Li, K., and Regan, A. C. Sorting with gpus: A survey. *arXiv preprint arXiv:1709.02520* (2017).

[23] Asai, A., He, J., Shao, R., Shi, W., Singh, A., Chang, J. C., Lo, K., Soldaini, L., Feldman, S., D'arcy, M., et al. Openscholar: Synthesizing scientific literature with retrieval-augmented lms. *arXiv preprint arXiv:2411.14199* (2024).

[24] Asai, A., Wu, Z., Wang, Y., Sil, A., and Hajishirzi, H. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations* (2023).

[25] Asai, A., Wu, Z., Wang, Y., Sil, A., and Hajishirzi, H. Original implementation of self-rag: Learning to retrieve, generate and critique through self-reflection. https://github.com/AkariAsai/self-rag, 2024.

[26] Asai, A., Zhong, Z., Chen, D., Koh, P. W., Zettlemoyer, L., Hajishirzi, H., and Yih, W.-t. Reliable, adaptable, and attributable language models with retrieval. *arXiv preprint arXiv:2403.03187* (2024).

[27] Ashkboos, S., Mohtashami, A., Croci, M. L., Li, B., Jaggi, M., Alistarh, D., Hoefler, T., and Hensman, J. Quarot: Outlier-free 4-bit inference in rotated llms. *arXiv preprint arXiv:2404.00456* (2024).

[28] AWS. Guidance for conversational chatbots using retrieval augmented generation on aws. https://aws.amazon.com/solutions/guidance/conversational-chatbots-using-retrieval-augmented-generation-on-aws/.

[29] BAI, Y., LV, X., ZHANG, J., LYU, H., TANG, J., HUANG, Z., DU, Z., LIU, X., ZENG, A., HOU, L., DONG, Y., TANG, J., AND LI, J. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508* (2023).

[30] BARRON, J. T., MILDENHALL, B., TANCIK, M., HEDMAN, P., MARTIN-BRUALLA, R., AND SRINIVASAN, P. P. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (October 2021), pp. 5855–5864.

[31] BATTAGLIA, P., HAMRICK, J. B. C., BAPST, V., SANCHEZ, A., ZAMBALDI, V., MALINOWSKI, M., TACCHETTI, A., RAPOSO, D., SANTORO, A., FAULKNER, R., GULCEHRE, C., SONG, F., BALLARD, A., GILMER, J., DAHL, G. E., VASWANI, A., ALLEN, K., NASH, C., LANGSTON, V. J., DYER, C., HEESS, N., WIERSTRA, D., KOHLI, P., BOTVINICK, M., VINYALS, O., LI, Y., AND PASCANU, R. Relational inductive biases, deep learning, and graph networks. *arXiv* (2018).

[32] BATTASH, B., AND WOLF, L. Adaptive and iteratively improving recurrent lateral connections. *CoRR abs/1910.11105* (2019).

[33] BERGER, E., AND ZORN, B. Ai software should be more like plain old software. https://www.sigarch.org/ai-software-should-be-more-like-plain-old-software/, 2024.

[34] BISK, Y., ZELLERS, R., BRAS, R. L., GAO, J., AND CHOI, Y. Piqa: Reasoning about physical commonsense in natural language, 2019.

[35] BORGEAUD, S., MENSCH, A., HOFFMANN, J., CAI, T., RUTHERFORD, E., MILLICAN, K., VAN DEN DRIESSCHE, G. B., LESPIAU, J.-B., DAMOC, B., CLARK, A., ET AL. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning* (2022), PMLR, pp. 2206–2240.

[36] BRIGGS, J., CULLEN, G., AND KOGAN, G. Vector search in the wild. https://www.pinecone.io/learn/series/wild/.

[37] BROWN, T. B., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., AGARWAL, S., HERBERT-VOSS, A., KRUEGER, G., HENIGHAN, T., CHILD, R., RAMESH, A., ZIEGLER, D. M., WU, J., WINTER, C., HESSE, C., CHEN, M., SIGLER, E., LITWIN, M., GRAY, S., CHESS, B., CLARK, J., BERNER, C., MCCANDLISH, S., RADFORD, A., SUTSKEVER, I., AND AMODEI, D. Language models are few-shot learners, 2020.

[38] CAI, T., LIU, Y., ZHOU, Z., MA, H., ZHAO, S. Z., WU, Z., AND MA, J. Driving with regulation: Interpretable decision-making for autonomous vehicles with retrieval-augmented reasoning via llm. *arXiv preprint arXiv:2410.04759* (2024).

[39] CAO, A., AND JOHNSON, J. Hexplane: A fast representation for dynamic scenes. *CVPR* (2023).

[40] CAO, J., WANG, H., CHEMERYS, P., SHAKHRAI, V., HU, J., FU, Y., MAKOVIICHUK, D., TULYAKOV, S., AND REN, J. Real-time neural light field on mobile devices. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2023), pp. 8328–8337.

[41] CHAN, E. R., LIN, C. Z., CHAN, M. A., NAGANO, K., PAN, B., MELLO, S. D., GALLO, O., GUIBAS, L., TREMBLAY, J., KHAMIS, S., KARRAS, T., AND WETZSTEIN, G. Efficient geometry-aware 3D generative adversarial networks. In *arXiv* (2021).

[42] CHEE, J., CAI, Y., KULESHOV, V., AND SA, C. D. Quip: 2-bit quantization of large language models with guarantees, 2023.

[43] CHEMMAGATE, B. Reducing rag pipeline latency for real-time voice conversations. https://developer.vonage.com/en/blog/reducing-rag-pipeline-latency-for-real-time-voice-conversations, 2024.

[44] CHEN, A., XU, Z., GEIGER, A., YU, J., AND SU, H. Tensorf: Tensorial radiance fields. In *European Conference on Computer Vision (ECCV)* (2022).

[45] CHEN, J., MA, T., AND XIAO, C. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations* (2018).

[46] CHEN, L. Dissecting batching effects in gpt inference, May 2023.

[47] CHEN, L., YE, Z., WU, Y., ZHUO, D., CEZE, L., AND KRISHNAMURTHY, A. Punica: Multi-tenant lora serving, 2023.

[48] CHEN, Q., ZHAO, B., WANG, H., LI, M., LIU, C., LI, Z., YANG, M., AND WANG, J. Spann: Highly-efficient billion-scale approximate nearest neighbor search. *arXiv preprint arXiv:2111.08566* (2021).

[49] CHEN, Y., ZHANG, C., GAO, X., MULLINS, R. D., CONSTANTINIDES, G. A., AND ZHAO, Y. Optimised grouped-query attention mechanism for transformers, 2024.

[50] CHEN, Z., FUNKHOUSER, T., HEDMAN, P., AND TAGLIASACCHI, A. Mobilenerf: Exploiting the polygon rasterization pipeline for efficient neural field rendering on mobile architectures. In *The Conference on Computer Vision and Pattern Recognition (CVPR)* (2023).

[51] CHERN, F., HECHTMAN, B., DAVIS, A., GUO, R., MAJNEMER, D., AND KUMAR, S. Tpu-knn: K nearest neighbor search at peak flop/s. In *Advances in Neural Information Processing Systems* (2022), vol. 35, pp. 15489–15501.

[52] CHETLUR, S., WOOLLEY, C., VANDERMERSCH, P., COHEN, J., TRAN, J., CATANZARO, B., AND SHELHAMER, E. cudnn: Efficient primitives for deep learning. *CoRR abs/1410.0759* (2014).

[53] CHIANG, H.-Y., CHANG, C.-C., FRUMKIN, N., WU, K.-C., AND MARCULESCU, D. Quamba: A post-training quantization recipe for selective state space models, 2024.

[54] CHIANG, W.-L., LIU, X., SI, S., LI, Y., BENGIO, S., AND HSIEH, C.-J. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining* (2019), KDD '19.

[55] CHUNG, N. C., DYER, G., AND BROCKI, L. Challenges of large language models for mental health counseling. *arXiv preprint arXiv:2311.13857* (2023).

[56] CLARK, C., LEE, K., CHANG, M.-W., KWIATKOWSKI, T., COLLINS, M., AND TOUTANOVA, K. Boolq: Exploring the surprising difficulty of natural yes/no questions, 2019.

[57] CLARK, P., COWHEY, I., ETZIONI, O., KHOT, T., SABHARWAL, A., SCHOENICK, C., AND TAFJORD, O. Think you have solved question answering? try arc, the ai2 reasoning challenge, 2018.

[58] DAI, D., DENG, C., ZHAO, C., XU, R. X., GAO, H., CHEN, D., LI, J., ZENG, W., YU, X., WU, Y., XIE, Z., LI, Y. K., HUANG, P., LUO, F., RUAN, C., SUI, Z., AND LIANG, W. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models, 2024.

[59] DAO, T., FU, D. Y., ERMON, S., RUDRA, A., AND RÉ, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[60] DATABRICKS. Rag (retrieval augmented generation) on databricks. https://docs.databricks.com/en/generative-ai/retrieval-augmented-generation.html, 2024.

[61] DEEPSEEK-AI, LIU, A., FENG, B., WANG, B., WANG, B., LIU, B., ZHAO, C., DENGR, C., RUAN, C., DAI, D., GUO, D., YANG, D., CHEN, D., JI, D., LI, E., LIN, F., LUO, F., HAO, G., CHEN, G., LI, G., ZHANG, H., XU, H., YANG, H., ZHANG, H., DING, H., XIN, H., GAO, H., LI, H., QU, H., CAI, J. L., LIANG, J., GUO, J., NI, J., LI, J., CHEN, J., YUAN, J., QIU, J., SONG, J., DONG, K., GAO, K., GUAN, K., WANG, L., ZHANG, L., XU, L., XIA, L., ZHAO, L., ZHANG, L., LI, M., WANG, M., ZHANG, M., ZHANG, M., TANG, M., LI, M., TIAN, N., HUANG, P., WANG, P., ZHANG, P., ZHU, Q., CHEN, Q., DU, Q., CHEN, R. J., JIN, R. L., GE, R., PAN, R., XU, R., CHEN, R., LI, S. S., LU, S., ZHOU, S., CHEN, S., WU, S., YE, S., MA, S., WANG, S., ZHOU, S., YU, S., ZHOU, S., ZHENG, S., WANG, T., PEI, T., YUAN, T., SUN, T., XIAO, W. L., ZENG, W., AN, W., LIU, W., LIANG, W., GAO, W., ZHANG, W., LI, X. Q., JIN, X., WANG, X., BI, X., LIU, X., WANG, X., SHEN, X., CHEN, X., CHEN, X., NIE, X., SUN, X., WANG, X., LIU, X., XIE, X., YU, X., SONG, X., ZHOU, X., YANG, X., LU, X., SU, X., WU, Y., LI, Y. K., WEI, Y. X., ZHU, Y. X., XU, Y., HUANG, Y., LI, Y., ZHAO, Y., SUN, Y., LI, Y., WANG, Y., ZHENG, Y., ZHANG, Y., XIONG, Y., ZHAO, Y., HE, Y., TANG, Y., PIAO, Y., DONG, Y., TAN, Y., LIU, Y., WANG, Y., GUO, Y., ZHU, Y., WANG, Y., ZOU, Y., ZHA, Y., MA, Y., YAN, Y., YOU, Y., LIU, Y., REN, Z. Z., REN, Z., SHA, Z., FU, Z., HUANG, Z., ZHANG, Z., XIE, Z., HAO, Z., SHAO, Z.,

WEN, Z., XU, Z., ZHANG, Z., LI, Z., WANG, Z., GU, Z., LI, Z., AND XIE, Z. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[62] DEHGHANI, M., GOUWS, S., VINYALS, O., USZKOREIT, J., AND KAISER, L. Universal transformers. *ArXiv abs/1807.03819* (2019).

[63] DENG, J., DONG, W., SOCHER, R., LI, L.-J., LI, K., AND FEI-FEI, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition* (2009), pp. 248–255.

[64] DENG, J., DONG, W., SOCHER, R., LI, L.-J., LI, K., AND FEI-FEI, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (2009), Ieee, pp. 248–255.

[65] DETTMERS, T., LEWIS, M., BELKADA, Y., AND ZETTLEMOYER, L. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022.

[66] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics* (2019).

[67] DIXIT, D. Advanced rag series: Generation and evaluation. https://div.beehiiv.com/p/advanced-rag-series-generation-evaluation, 2024.

[68] DOSOVITSKIY, A., BEYER, L., KOLESNIKOV, A., WEISSENBORN, D., ZHAI, X., UNTERTHINER, T., DEHGHANI, M., MINDERER, M., HEIGOLD, G., AND GELLY, S. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[69] DOUZE, M., GUZHVA, A., DENG, C., JOHNSON, J., SZILVASY, G., MAZARÉ, P.-E., LOMELI, M., HOSSEINI, L., AND JÉGOU, H. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[70] DUARTE, F. Number of chatgpt users, Jul 2023.

[71] DUBEY, A., JAUHRI, A., PANDEY, A., KADIAN, A., AL-DAHLE, A., LETMAN, A., MATHUR, A., SCHELTEN, A., YANG, A., FAN, A., GOYAL, A., HARTSHORN, A., YANG, A., MITRA,

A., SRAVANKUMAR, A., KORENEV, A., HINSVARK, A., RAO, A., ZHANG, A., RODRIGUEZ, A., GREGERSON, A., SPATARU, A., ROZIERE, B., BIRON, B., TANG, B., CHERN, B., CAUCHETEUX, C., NAYAK, C., BI, C., MARRA, C., MCCONNELL, C., KELLER, C., TOURET, C., WU, C., WONG, C., FERRER, C. C., NIKOLAIDIS, C., ALLONSIUS, D., SONG, D., PINTZ, D., LIVSHITS, D., ESIOBU, D., CHOUDHARY, D., MAHAJAN, D., GARCIA-OLANO, D., PERINO, D., HUPKES, D., LAKOMKIN, E., ALBADAWY, E., LOBANOVA, E., DINAN, E., SMITH, E. M., RADENOVIC, F., ZHANG, F., SYNNAEVE, G., LEE, G., ANDERSON, G. L., NAIL, G., MIALON, G., PANG, G., CUCURELL, G., NGUYEN, H., KOREVAAR, H., XU, H., TOUVRON, H., ZAROV, I., IBARRA, I. A., KLOUMANN, I., MISRA, I., EVTIMOV, I., COPET, J., LEE, J., GEFFERT, J., VRANES, J., PARK, J., MAHADEOKAR, J., SHAH, J., VAN DER LINDE, J., BILLOCK, J., HONG, J., LEE, J., FU, J., CHI, J., HUANG, J., LIU, J., WANG, J., YU, J., BITTON, J., SPISAK, J., PARK, J., ROCCA, J., JOHNSTUN, J., SAXE, J., JIA, J., ALWALA, K. V., UPASANI, K., PLAWIAK, K., LI, K., HEAFIELD, K., STONE, K., EL-ARINI, K., IYER, K., MALIK, K., CHIU, K., BHALLA, K., RANTALA-YEARY, L., VAN DER MAATEN, L., CHEN, L., TAN, L., JENKINS, L., MARTIN, L., MADAAN, L., MALO, L., BLECHER, L., LANDZAAT, L., DE OLIVEIRA, L., MUZZI, M., PASUPULETI, M., SINGH, M., PALURI, M., KARDAS, M., OLDHAM, M., RITA, M., PAVLOVA, M., KAMBADUR, M., LEWIS, M., SI, M., SINGH, M. K., HASSAN, M., GOYAL, N., TORABI, N., BASHLYKOV, N., BOGOYCHEV, N., CHATTERJI, N., DUCHENNE, O., ÇELEBI, O., ALRASSY, P., ZHANG, P., LI, P., VASIC, P., WENG, P., BHARGAVA, P., DUBAL, P., KRISHNAN, P., KOURA, P. S., XU, P., HE, Q., DONG, Q., SRINIVASAN, R., GANAPATHY, R., CALDERER, R., CABRAL, R. S., STOJNIC, R., RAILEANU, R., GIRDHAR, R., PATEL, R., SAUVESTRE, R., POLIDORO, R., SUMBALY, R., TAYLOR, R., SILVA, R., HOU, R., WANG, R., HOSSEINI, S., CHENNABASAPPA, S., SINGH, S., BELL, S., KIM, S. S., EDUNOV, S., NIE, S., NARANG, S., RAPARTHY, S., SHEN, S., WAN, S., BHOSALE, S., ZHANG, S., VANDENHENDE, S., BATRA, S., WHITMAN, S., SOOTLA, S., COLLOT, S., GURURANGAN, S., BORODINSKY, S., HERMAN, T., FOWLER, T., SHEASHA, T., GEORGIOU, T., SCIALOM, T., SPECKBACHER, T., MIHAYLOV, T., XIAO, T., KARN, U., GOSWAMI, V., GUPTA, V., RAMANATHAN, V., KERKEZ, V., GONGUET, V., DO, V., VOGETI, V., PETROVIC, V., CHU, W., XIONG, W., FU, W., MEERS, W., MARTINET, X., WANG,

X., TAN, X. E., XIE, X., JIA, X., WANG, X., GOLDSCHLAG, Y., GAUR, Y., BABAEI, Y., WEN, Y., SONG, Y., ZHANG, Y., LI, Y., MAO, Y., COUDERT, Z. D., YAN, Z., CHEN, Z., PAPAKIPOS, Z., SINGH, A., GRATTAFIORI, A., JAIN, A., KELSEY, A., SHAJNFELD, A., GANGIDI, A., VICTORIA, A., GOLDSTAND, A., MENON, A., SHARMA, A., BOESENBERG, A., VAUGHAN, A., BAEVSKI, A., FEINSTEIN, A., KALLET, A., SANGANI, A., YUNUS, A., LUPU, A., ALVARADO, A., CAPLES, A., GU, A., HO, A., POULTON, A., RYAN, A., RAMCHANDANI, A., FRANCO, A., SARAF, A., CHOWDHURY, A., GABRIEL, A., BHARAMBE, A., EISENMAN, A., YAZDAN, A., JAMES, B., MAURER, B., LEONHARDI, B., HUANG, B., LOYD, B., PAOLA, B. D., PARANJAPE, B., LIU, B., WU, B., NI, B., HANCOCK, B., WASTI, B., SPENCE, B., STOJKOVIC, B., GAMIDO, B., MONTALVO, B., PARKER, C., BURTON, C., MEJIA, C., WANG, C., KIM, C., ZHOU, C., HU, C., CHU, C.-H., CAI, C., TINDAL, C., FEICHTENHOFER, C., CIVIN, D., BEATY, D., KREYMER, D., LI, D., WYATT, D., ADKINS, D., XU, D., TESTUGGINE, D., DAVID, D., PARIKH, D., LISKOVICH, D., FOSS, D., WANG, D., LE, D., HOLLAND, D., DOWLING, E., JAMIL, E., MONTGOMERY, E., PRESANI, E., HAHN, E., WOOD, E., BRINKMAN, E., ARCAUTE, E., DUNBAR, E., SMOTHERS, E., SUN, F., KREUK, F., TIAN, F., OZGENEL, F., CAGGIONI, F., GUZMÁN, F., KANAYET, F., SEIDE, F., FLOREZ, G. M., SCHWARZ, G., BADEER, G., SWEE, G., HALPERN, G., THATTAI, G., HERMAN, G., SIZOV, G., GUANGYI, ZHANG, LAKSHMINARAYANAN, G., SHOJANAZERI, H., ZOU, H., WANG, H., ZHA, H., HABEEB, H., RUDOLPH, H., SUK, H., ASPEGREN, H., GOLDMAN, H., DAMLAJ, I., MOLYBOG, I., TUFANOV, I., VELICHE, I.-E., GAT, I., WEISSMAN, J., GEBOSKI, J., KOHLI, J., ASHER, J., GAYA, J.-B., MARCUS, J., TANG, J., CHAN, J., ZHEN, J., REIZENSTEIN, J., TEBOUL, J., ZHONG, J., JIN, J., YANG, J., CUMMINGS, J., CARVILL, J., SHEPARD, J., MCPHIE, J., TORRES, J., GINSBURG, J., WANG, J., WU, K., U, K. H., SAXENA, K., PRASAD, K., KHANDELWAL, K., ZAND, K., MATOSICH, K., VEERARAGHAVAN, K., MICHELENA, K., LI, K., HUANG, K., CHAWLA, K., LAKHOTIA, K., HUANG, K., CHEN, L., GARG, L., A, L., SILVA, L., BELL, L., ZHANG, L., GUO, L., YU, L., MOSHKOVICH, L., WEHRSTEDT, L., KHABSA, M., AVALANI, M., BHATT, M., TSIMPOUKELLI, M., MANKUS, M., HASSON, M., LENNIE, M., RESO, M., GROSHEV, M., NAUMOV, M., LATHI, M., KENEALLY, M., SELTZER, M. L., VALKO, M., RESTREPO, M., PATEL, M., VYATSKOV, M., SAMVELYAN, M., CLARK,

M., MACEY, M., WANG, M., HERMOSO, M. J., METANAT, M., RASTEGARI, M., BANSAL, M., SANTHANAM, N., PARKS, N., WHITE, N., BAWA, N., SINGHAL, N., EGEBO, N., USUNIER, N., LAPTEV, N. P., DONG, N., ZHANG, N., CHENG, N., CHERNOGUZ, O., HART, O., SALPEKAR, O., KALINLI, O., KENT, P., PAREKH, P., SAAB, P., BALAJI, P., RITTNER, P., BONTRAGER, P., ROUX, P., DOLLAR, P., ZVYAGINA, P., RATANCHANDANI, P., YUVRAJ, P., LIANG, Q., ALAO, R., RODRIGUEZ, R., AYUB, R., MURTHY, R., NAYANI, R., MITRA, R., LI, R., HOGAN, R., BATTEY, R., WANG, R., MAHESWARI, R., HOWES, R., RINOTT, R., BONDU, S. J., DATTA, S., CHUGH, S., HUNT, S., DHILLON, S., SIDOROV, S., PAN, S., VERMA, S., YAMAMOTO, S., RAMASWAMY, S., LINDSAY, S., LINDSAY, S., FENG, S., LIN, S., ZHA, S. C., SHANKAR, S., ZHANG, S., ZHANG, S., WANG, S., AGARWAL, S., SAJUYIGBE, S., CHINTALA, S., MAX, S., CHEN, S., KEHOE, S., SATTERFIELD, S., GOVINDAPRASAD, S., GUPTA, S., CHO, S., VIRK, S., SUBRAMANIAN, S., CHOUDHURY, S., GOLDMAN, S., REMEZ, T., GLASER, T., BEST, T., KOHLER, T., ROBINSON, T., LI, T., ZHANG, T., MATTHEWS, T., CHOU, T., SHAKED, T., VONTIMITTA, V., AJAYI, V., MONTANEZ, V., MOHAN, V., KUMAR, V. S., MANGLA, V., ALBIERO, V., IONESCU, V., POENARU, V., MIHAILESCU, V. T., IVANOV, V., LI, W., WANG, W., JIANG, W., BOUAZIZ, W., CONSTABLE, W., TANG, X., WANG, X., WU, X., WANG, X., XIA, X., WU, X., GAO, X., CHEN, Y., HU, Y., JIA, Y., QI, Y., LI, Y., ZHANG, Y., ZHANG, Y., ADI, Y., NAM, Y., YU, WANG, HAO, Y., QIAN, Y., HE, Y., RAIT, Z., DEVITO, Z., ROSNBRICK, Z., WEN, Z., YANG, Z., AND ZHAO, Z. The llama 3 herd of models, 2024.

[72] ELIMIAN, G. Chatgpt costs 700,000 to run daily, openai may go bankrupt in 2024, Aug 2023.

[73] FAN, W., DING, Y., BO NING, L., WANG, S., LI, H., YIN, D., CHUA, T.-S., AND LI, Q. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *Knowledge Discovery and Data Mining* (2024).

[74] FANG, J., XIE, L., WANG, X., ZHANG, X., LIU, W., AND TIAN, Q. Neusample: Neural sample field for efficient view synthesis. *arXiv:2111.15552* (2021).

[75] FEDUS, W., ZOPH, B., AND SHAZEER, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022.

[76] FRANTAR, E., ASHKBOOS, S., HOEFLER, T., AND ALISTARH, D. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[77] FRANTAR, E., CASTRO, R. L., CHEN, J., HOEFLER, T., AND ALISTARH, D. Marlin: Mixed-precision auto-regressive parallel inference on large language models, 2024.

[78] FRIDOVICH-KEIL, S., YU, A., TANCIK, M., CHEN, Q., RECHT, B., AND KANAZAWA, A. Plenoxels: Radiance fields without neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2022), pp. 5501–5510.

[79] FU, Y. Challenges in deploying long-context transformers: A theoretical peak performance analysis, 2024.

[80] GALE, T., ZAHARIA, M., YOUNG, C., AND ELSEN, E. Sparse gpu kernels for deep learning, 2020.

[81] GAO, L., MA, X., LIN, J., AND CALLAN, J. Precise zero-shot dense retrieval without relevance labels. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2023), pp. 1762–1777.

[82] GAO, L., TOW, J., BIDERMAN, S., BLACK, S., DIPOFI, A., FOSTER, C., GOLDING, L., HSU, J., MCDONELL, K., MUENNIGHOFF, N., PHANG, J., REYNOLDS, L., TANG, E., THITE, A., WANG, B., WANG, K., AND ZOU, A. A framework for few-shot language model evaluation, Sept. 2021.

[83] GAO, Y. Modular rag and rag flow: Part ii. https://medium.com/@yufan1602/modular-rag-and-rag-flow-part-ii-77b62bf8a5d3, 2024.

[84] GAO, Y., XIONG, Y., GAO, X., JIA, K., PAN, J., BI, Y., DAI, Y., SUN, J., AND WANG, H. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* (2023).

[85] GARBIN, S. J., KOWALSKI, M., JOHNSON, M., SHOTTON, J., AND VALENTIN, J. Fastnerf: High-fidelity neural rendering at 200fps. *ICCV* (2021).

[86] GHODRATNAMA, S., AND ZAKERSHAHRAK, M. Adapting llms for efficient, personalized information retrieval: Methods and implications. In *International Conference on Service-Oriented Computing* (2023), Springer, pp. 17–26.

[87] GHOLAMI, A., YAO, Z., KIM, S., HOOPER, C., MAHONEY, M. W., AND KEUTZER, K. Ai and memory wall. *IEEE Micro Journal* (2024).

[88] GHOLAMI, A., YAO, Z., KIM, S., HOOPER, C., MAHONEY, M. W., AND KEUTZER, K. Ai and memory wall, 2024.

[89] GM, A., AND DHAR, G. How apollo 24|7 leverages medlm with rag to revolutionize healthcare. https://cloud.google.com/blog/products/ai-machine-learning/how-apollo-247-leverages-medlm-with-rag-to-revolutionize-healthcare, 2024.

[90] GRAY, S., RADFORD, A., AND KINGMA, D. P. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224* (2017).

[91] GU, A., AND DAO, T. Mamba: Linear-time sequence modeling with selective state spaces, 2023.

[92] GU, A., GOEL, K., AND RÉ, C. Efficiently modeling long sequences with structured state spaces, 2022.

[93] GUO, C., TANG, J., HU, W., LENG, J., ZHANG, C., YANG, F., LIU, Y., GUO, M., AND ZHU, Y. OliVe: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (jun 2023), ACM.

[94] GUO, Q., YU, Z., WU, Y., LIANG, D., QIN, H., AND YAN, J. Dynamic recursive neural network. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), 5142–5151.

[95] GUO, R., SUN, P., LINDGREN, E., GENG, Q., SIMCHA, D., CHERN, F., AND KUMAR, S. Ac-

celerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning* (2020).

[96] HAMILTON, W. L., YING, R., AND LESKOVEC, J. Inductive representation learning on large graphs. In *NIPS* (2017).

[97] HAN, S., MAO, H., AND DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[98] HAN, S., POOL, J., TRAN, J., AND DALLY, W. J. Learning both weights and connections for efficient neural network. *ArXiv abs/1506.02626* (2015).

[99] HAN, S., POOL, J., TRAN, J., AND DALLY, W. J. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (2015), NIPS'15, p. 1135–1143.

[100] HAQUE, A., TANCIK, M., EFROS, A., HOLYNSKI, A., AND KANAZAWA, A. Instruct-nerf2nerf: Editing 3d scenes with instructions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (2023).

[101] HARDT, M., AND SUN, Y. Test-time training on nearest neighbors for large language models. *arXiv preprint arXiv:2305.18466* (2023).

[102] HASANZADEH, A., HAJIRAMEZANALI, E., BOLUKI, S., ZHOU, M., DUFFIELD, N., NARAYANAN, K., AND QIAN, X. Bayesian graph neural networks with adaptive connection sampling. In *Proceedings of the 37th International Conference on Machine Learning* (2020), vol. 119 of *Proceedings of Machine Learning Research*, PMLR, pp. 4094–4104.

[103] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), 770–778.

[104] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), 770–778.

[105] HEDMAN, P., SRINIVASAN, P. P., MILDENHALL, B., BARRON, J. T., AND DEBEVEC, P. Baking neural radiance fields for real-time view synthesis. *ICCV* (2021).

[106] HONG, C., SUKUMARAN-RAJAM, A., NISA, I., SINGH, K., AND SADAYAPPAN, P. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (2019), PPoPP '19, p. 300–314.

[107] HOOPER, C., KIM, S., MOHAMMADZADEH, H., MAHONEY, M. W., SHAO, Y. S., KEUTZER, K., AND GHOLAMI, A. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079* (2024).

[108] HOWARD, A. G., SANDLER, M., CHU, G., CHEN, L.-C., CHEN, B., TAN, M., WANG, W., ZHU, Y., PANG, R., VASUDEVAN, V., LE, Q. V., AND ADAM, H. Searching for mobilenetv3. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)* (2019), 1314–1324.

[109] HSU, Y.-C., HUA, T., CHANG, S., LOU, Q., SHEN, Y., AND JIN, H. Language model compression with weighted low-rank factorization. In *International Conference on Learning Representations* (2022).

[110] HU, T., LIU, S., CHEN, Y., SHEN, T., AND JIA, J. Efficientnerf efficient neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2022), pp. 12902–12911.

[111] HU, W., FEY, M., ZITNIK, M., DONG, Y., REN, H., LIU, B., CATASTA, M., AND LESKOVEC, J. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687* (2020).

[112] HU, Y., YE, Z., WANG, M., YU, J., ZHENG, D., LI, M., ZHANG, Z., ZHANG, Z., AND WANG, Y. Featgraph: A flexible and efficient backend for graph neural network systems. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)* (2020).

[113] HUANG, G. Ge-spmm open source code, 2020.

[114] HUANG, G., DAI, G., WANG, Y., AND YANG, H. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks, 2020.

[115] HUANG, W., ZHANG, T., RONG, Y., AND HUANG, J. Adaptive sampling towards fast graph representation learning. In *Advances in Neural Information Processing Systems (NIPS)* (2018).

[116] HUANG, X., LI, W., HU, J., CHEN, H., AND WANG, Y. Refsr-nerf: Towards high fidelity and super resolution view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2023), pp. 8244–8253.

[117] HUGGINGFACE. Sharegpt vicuna unfiltered, May 2023.

[118] ILIN, I. Advanced rag techniques: an illustrated overview. https://pub.towardsai.net/advanced-rag-techniques-an-illustrated-overview-04d193d8fec6, 2023.

[119] INC, G. What is a language processing unit?, 2025.

[120] IZACARD, G., CARON, M., HOSSEINI, L., RIEDEL, S., BOJANOWSKI, P., JOULIN, A., AND GRAVE, E. Unsupervised dense information retrieval with contrastive learning. *arXiv preprint arXiv:2112.09118* (2021).

[121] IZACARD, G., LEWIS, P., LOMELI, M., HOSSEINI, L., PETRONI, F., SCHICK, T., DWIVEDI-YU, J., JOULIN, A., RIEDEL, S., AND GRAVE, E. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning Research 24*, 251 (2023), 1–43.

[122] JACOB, B., KLIGYS, S., CHEN, B., ZHU, M., TANG, M., HOWARD, A., ADAM, H., AND KALENICHENKO, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2017.

[123] JAGERMAN, R., ZHUANG, H., QIN, Z., WANG, X., AND BENDERSKY, M. Query expansion by prompting large language models. *arXiv preprint arXiv:2305.03653* (2023).

[124] JASTRZEBSKI, S., ARPIT, D., BALLAS, N., VERMA, V., CHE, T., AND BENGIO, Y. Residual connections encourage iterative inference. *CoRR abs/1710.04773* (2017).

[125] JAYARAM SUBRAMANYA, S., DEVVRIT, F., SIMHADRI, H. V., KRISHNAWAMY, R., AND KADEKODI, R. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems 32* (2019).

[126] JIANG, A. Q., SABLAYROLLES, A., MENSCH, A., BAMFORD, C., CHAPLOT, D. S., DE LAS CASAS, D., BRESSAND, F., LENGYEL, G., LAMPLE, G., SAULNIER, L., LAVAUD, L. R., LACHAUX, M.-A., STOCK, P., SCAO, T. L., LAVRIL, T., WANG, T., LACROIX, T., AND SAYED, W. E. Mistral 7b, 2023.

[127] JIANG, A. Q., SABLAYROLLES, A., ROUX, A., MENSCH, A., SAVARY, B., BAMFORD, C., CHAPLOT, D. S., DE LAS CASAS, D., HANNA, E. B., BRESSAND, F., LENGYEL, G., BOUR, G., LAMPLE, G., LAVAUD, L. R., SAULNIER, L., LACHAUX, M.-A., STOCK, P., SUBRAMANIAN, S., YANG, S., ANTONIAK, S., SCAO, T. L., GERVET, T., LAVRIL, T., WANG, T., LACROIX, T., AND SAYED, W. E. Mixtral of experts, 2024.

[128] JIANG, H., WU, Q., LUO, X., LI, D., LIN, C.-Y., YANG, Y., AND QIU, L. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839* (2023).

[129] JIANG, W., LI, S., ZHU, Y., DE FINE LICHT, J., HE, Z., SHI, R., RENGGLI, C., ZHANG, S., REKATSINAS, T., HOEFLER, T., ET AL. Co-design hardware and algorithm for vector search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2023), pp. 1–15.

[130] JIANG, W., SUBRAMANIAN, S., GRAVES, C., ALONSO, G., YAZDANBAKHSH, A., AND DADU, V. Rago: Systematic performance optimization for retrieval-augmented generation serving. *arXiv preprint arXiv:2503.14649* (2025).

[131] JIANG, W., ZELLER, M., WALEFFE, R., HOEFLER, T., AND ALONSO, G. Chameleon: a heterogeneous and disaggregated accelerator system for retrieval-augmented language models. *arXiv preprint arXiv:2310.09949* (2023).

[132] JIANG, W., ZHANG, S., HAN, B., WANG, J., WANG, B., AND KRASKA, T. Piperag: Fast retrieval-augmented generation via algorithm-system co-design. *arXiv preprint arXiv:2403.05676* (2024).

[133] JIANG, Z., XU, F. F., GAO, L., SUN, Z., LIU, Q., DWIVEDI-YU, J., YANG, Y., CALLAN, J., AND NEUBIG, G. Active retrieval augmented generation.

[134] JIN, C., ZHANG, Z., JIANG, X., LIU, F., LIU, X., LIU, X., AND JIN, X. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *arXiv preprint arXiv:2404.12457* (2024).

[135] JIN, J., ZHU, Y., YANG, X., ZHANG, C., AND DOU, Z. Flashrag: A modular toolkit for efficient retrieval-augmented generation research. *arXiv preprint arXiv:2405.13576* (2024).

[136] JOHNSON, J., DOUZE, M., AND JÉGOU, H. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data 7*, 3 (2019), 535–547.

[137] JOLLIFFE, I. T., AND CADIMA, J. Principal component analysis: a review and recent developments. *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences 374*, 2065 (2016), 20150202.

[138] JOSHI, M., CHOI, E., WELD, D. S., AND ZETTLEMOYER, L. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2017), pp. 1601–1611.

[139] KANG, H., ZHANG, Q., KUNDU, S., JEONG, G., LIU, Z., KRISHNA, T., AND ZHAO, T. Gear: An efficient kv cache compression recipe for near-lossless generative inference of llm, 2024.

[140] KARPUKHIN, V., OGUZ, B., MIN, S., LEWIS, P., WU, L., EDUNOV, S., CHEN, D., AND YIH, W.-T. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2020).

[141] KHAN, S., SINGH, S., SIMHADRI, H. V., VEDURADA, J., ET AL. Bang: Billion-scale approximate nearest neighbor search using a single gpu. *arXiv preprint arXiv:2401.11324* (2024).

[142] KHANDELWAL, U., LEVY, O., JURAFSKY, D., ZETTLEMOYER, L., AND LEWIS, M. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations* (2019).

[143] KIM, D., AND KANG, W. Learning shared filter bases for efficient convnets. *CoRR abs/2006.05066* (2020).

[144] KIM, J., NAM, J., MO, S., PARK, J., LEE, S.-W., SEO, M., HA, J.-W., AND SHIN, J. Sure: Improving open-domain question answering of llms via summarized retrieval. In *The Twelfth International Conference on Learning Representations* (2023).

[145] KIM, S., HOOPER, C., GHOLAMI, A., DONG, Z., LI, X., SHEN, S., MAHONEY, M. W., AND KEUTZER, K. Squeezellm: Dense-and-sparse quantization, 2023.

[146] KIM, S., MOON, S., TABRIZI, R., LEE, N., MAHONEY, M. W., KEUTZER, K., AND GHOLAMI, A. An llm compiler for parallel function calling. *arXiv preprint arXiv:2312.04511* (2023).

[147] KINGMA, D., AND BA, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)* (San Diega, CA, USA, 2015).

[148] KIPF, T. N., AND WELLING, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations* (2017).

[149] KLANG, E., TESSLER, I., APAKAMA, D. U., ABBOTT, E., GLICKSBERG, B. S., ARNOLD, M., MOSES, A., SAKHUJA, A., SOROUSH, A., CHARNEY, A. W., ET AL. Assessing retrieval-augmented large language model performance in emergency department icd-10-cm coding compared to human coders. *medRxiv* (2024).

[150] KLICPERA, J., BOJCHEVSKI, A., AND GUNNEMAN, S. Predict then propagate: Graph neural networks meet personalized pagerank. In *Proceedings of the 7th International Conference on Learning Representations* (2019).

[151] KORNBLITH, S., NOROUZI, M., LEE, H., AND HINTON, G. E. Similarity of neural network representations revisited. *ArXiv abs/1905.00414* (2019).

[152] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1097–1105.

[153] KUBILIUS, J., SCHRIMPF, M., HONG, H., MAJAJ, N. J., RAJALINGHAM, R., ISSA, E. B., KAR, K., BASHIVAN, P., PRESCOTT-ROY, J., SCHMIDT, K., NAYEBI, A., BEAR, D., YAMINS, D. L. K., AND DICARLO, J. J. Aligning artificial neural networks to the brain yields shallow recurrent architectures.

[154] KUBILIUS, J., SCHRIMPF, M., HONG, H., MAJAJ, N. J., RAJALINGHAM, R., ISSA, E. B., KAR, K., BASHIVAN, P., PRESCOTT-ROY, J., SCHMIDT, K., NAYEBI, A., BEAR, D., YAMINS, D. L. K., AND DICARLO, J. J. Brain-like object recognition with high-performing shallow recurrent anns. *CoRR abs/1909.06161* (2019).

[155] KWIATKOWSKI, T., PALOMAKI, J., REDFIELD, O., COLLINS, M., PARIKH, A., ALBERTI, C., EPSTEIN, D., POLOSUKHIN, I., DEVLIN, J., LEE, K., ET AL. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics 7* (2019), 453–466.

[156] KWON, W., LI, Z., ZHUANG, S., SHENG, Y., ZHENG, L., YU, C. H., GONZALEZ, J. E., ZHANG, H., AND STOICA, I. Efficient memory management for large language model serving with page-dattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles* (2023).

[157] KWON, W., LI, Z., ZHUANG, S., SHENG, Y., ZHENG, L., YU, C. H., GONZALEZ, J. E., ZHANG, H., AND STOICA, I. Efficient memory management for large language model serving with pagedattention, 2023.

[158] LAM, M., JOHNSON, J., XIONG, W., MAENG, K., GUPTA, U., LI, Y., LAI, L., LEONTIADIS, I., RHU, M., LEE, H.-H. S., REDDI, V. J., WEI, G.-Y., BROOKS, D., AND SUH, G. E. Gpu-based private information retrieval for on-device machine learning inference, 2023.

[159] LAN, Z., CHEN, M., GOODMAN, S., GIMPEL, K., SHARMA, P., AND SORICUT, R. Albert: A lite bert for self-supervised learning of language representations. In *ICLR* (2020), OpenReview.net.

[160] LEE, C., JIN, J., KIM, T., KIM, H., AND PARK, E. Owq: Lessons learned from activation outliers for weight quantization in large language models. *ArXiv abs/2306.02272* (2023).

[161] LEROUX, S., MOLCHANOV, P., SIMOENS, P., DHOEDT, B., BREUEL, T. M., AND KAUTZ, J. Iamnn: Iterative and adaptive mobile neural network for efficient image classification. *CoRR abs/1804.10123* (2018).

[162] LEWIS, P., PEREZ, E., PIKTUS, A., PETRONI, F., KARPUKHIN, V., GOYAL, N., KÜTTLER, H., LEWIS, M., YIH, W.-T., ROCKTÄSCHEL, T., ET AL. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems 33* (2020), 9459–9474.

[163] LI, D., SHAO, R., XIE, A., SHENG, Y., ZHENG, L., GONZALEZ, J., STOICA, I., MA, X., AND ZHANG, H. How long can context length of open-source llms truly promise? In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following* (2023).

[164] LI, G., MÜLLER, M., THABET, A., AND GHANEM, B. Deepgcns: Can gcns go as deep as cnns? In *The IEEE International Conference on Computer Vision (ICCV)* (2019).

[165] LI, Q., HAN, Z., AND WU, X.-M. Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[166] LI, R., GAO, H., TANCIK, M., AND KANAZAWA, A. Nerfacc: Efficient sampling accelerates nerfs. *arXiv preprint arXiv:2305.04966* (2023).

[167] LIM, B., SON, S., KIM, H., NAH, S., AND LEE, K. M. Enhanced deep residual networks for single image super-resolution. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (July 2017).

[168] LIN, C.-H., GAO, J., TANG, L., TAKIKAWA, T., ZENG, X., HUANG, X., KREIS, K., FIDLER,

S., LIU, M.-Y., AND LIN, T.-Y. Magic3d: High-resolution text-to-3d content creation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2023).

[169] LIN, J., TANG, J., TANG, H., YANG, S., DANG, X., GAN, C., AND HAN, S. Awq: Activation-aware weight quantization for llm compression and acceleration, 2023.

[170] LIN, Z., LI, C., MIAO, Y., LIU, Y., AND XU, Y. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (New York, NY, USA, 2020), SoCC '20, Association for Computing Machinery.

[171] LIU, J. LlamaIndex. https://github.com/jerryjliu/llama_index, 11 2022.

[172] LIU, J., GONG, R., WEI, X., DONG, Z., CAI, J., AND ZHUANG, B. Qllm: Accurate and efficient low-bitwidth quantization for large language models, 2023.

[173] LIU, L., GU, J., LIN, K. Z., CHUA, T.-S., AND THEOBALT, C. Neural sparse voxel fields. *NeurIPS* (2020).

[174] LIU, L., ZHANG, S., KUANG, Z., ZHOU, A., XUE, J., WANG, X., CHEN, Y., YANG, W., LIAO, Q., AND ZHANG, W. Group fisher pruning for practical network compression. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event* (2021), M. Meila and T. Zhang, Eds., vol. 139 of *Proceedings of Machine Learning Research*, PMLR, pp. 7021–7032.

[175] LIU, S., LIU, Z., HUANG, X., DONG, P., AND CHENG, K. Llm-fp4: 4-bit floating-point quantized transformers, 2023.

[176] LIU, Z., MAO, H., WU, C.-Y., FEICHTENHOFER, C., DARRELL, T., AND XIE, S. A convnet for the 2020s. *arXiv preprint arXiv:2201.03545* (2022).

[177] LIU, Z., NI, W., LENG, J., FENG, Y., GUO, C., CHEN, Q., LI, C., GUO, M., AND ZHU, Y. Juno: Optimizing high-dimensional approximate nearest neighbour search with sparsity-aware algorithm and ray-tracing core mapping. *arXiv preprint arXiv:2312.01712* (2023).

[178] LIU, Z., YUAN, J., JIN, H., ZHONG, S., XU, Z., BRAVERMAN, V., CHEN, B., AND HU, X. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750* (2024).

[179] LIU, Z., ZHAO, C., FEDOROV, I., SORAN, B., CHOUDHARY, D., KRISHNAMOORTHI, R., CHANDRA, V., TIAN, Y., AND BLANKEVOORT, T. Spinquant–llm quantization with learned rotations. *arXiv preprint arXiv:2405.16406* (2024).

[180] LU, S., WANG, H., RONG, Y., CHEN, Z., AND TANG, Y. Turborag: Accelerating retrieval-augmented generation with precomputed kv caches for chunked text. *arXiv preprint arXiv:2410.07590* (2024).

[181] LY, A., MARSMAN, M., VERHAGEN, J., GRASMAN, R., AND WAGENMAKERS, E.-J. A tutorial on fisher information, 2017.

[182] MA, X., GONG, Y., HE, P., DUAN, N., ET AL. Query rewriting in retrieval-augmented large language models. In *The 2023 Conference on Empirical Methods in Natural Language Processing* (2023).

[183] MALEC, M. Rag in financial services: Use-cases, impact, & solutions. https://hatchworks.com/blog/gen-ai/rag-for-financial-services/, 2024.

[184] MALLEN, A., ASAI, A., ZHONG, V., DAS, R., KHASHABI, D., AND HAJISHIRZI, H. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Toronto, Canada, July 2023), Association for Computational Linguistics, pp. 9802–9822.

[185] MALLEN, A. T., ASAI, A., ZHONG, V., DAS, R., KHASHABI, D., AND HAJISHIRZI, H. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In *The 61st Annual Meeting Of The Association For Computational Linguistics* (2023).

[186] MARCUS, M., KIM, G., MARCINKIEWICZ, M. A., MACINTYRE, R., BIES, A., FERGUSON, M., KATZ, K., AND SCHASBERGER, B. The penn treebank: Annotating predicate argument structure. In

Proceedings of the Workshop on Human Language Technology (USA, 1994), HLT '94, Association for Computational Linguistics, p. 114–119.

[187] MERITY, S., XIONG, C., BRADBURY, J., AND SOCHER, R. Pointer sentinel mixture models, 2016.

[188] MICIKEVICIUS, P., STOSIC, D., BURGESS, N., CORNEA, M., DUBEY, P., GRISENTHWAITE, R., HA, S., HEINECKE, A., JUDD, P., KAMALU, J., MELLEMPUDI, N., OBERMAN, S., SHOEYBI, M., SIU, M., AND WU, H. Fp8 formats for deep learning, 2022.

[189] MILDENHALL, B., SRINIVASAN, P. P., ORTIZ-CAYON, R., KALANTARI, N. K., RAMAMOORTHI, R., NG, R., AND KAR, A. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics (TOG)* (2019).

[190] MILDENHALL, B., SRINIVASAN, P. P., TANCIK, M., BARRON, J. T., RAMAMOORTHI, R., AND NG, R. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV* (2020).

[191] MIN, S., GURURANGAN, S., WALLACE, E., SHI, W., HAJISHIRZI, H., SMITH, N. A., AND ZETTLEMOYER, L. Silo language models: Isolating legal risk in a nonparametric datastore. In *The Twelfth International Conference on Learning Representations* (2023).

[192] MIN, S. W., WU, K., HUANG, S., HIDAYETOĞLU, M., XIONG, J., EBRAHIMI, E., CHEN, D., AND MEI HWU, W. Large graph convolutional network training with gpu-oriented data communication architecture, 2021.

[193] MING CHEN, Z. W., ZENGFENG HUANG, B. D., AND LI, Y. Simple and deep graph convolutional networks.

[194] MÜLLER, T., EVANS, A., SCHIED, C., AND KELLER, A. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph. 41*, 4 (July 2022), 102:1–102:15.

[195] MYSCALE. 4 key benefits of rag algorithmic trading in financial markets. https://myscale.com/blog/benefits-rag-algorithmic-trading-financial-markets/, 2024.

[196] NAGEL, M., FOURNARAKIS, M., AMJAD, R. A., BONDARENKO, Y., VAN BAALEN, M., AND BLANKEVOORT, T. A white paper on neural network quantization, 2021.

[197] NOACH, M. B., AND GOLDBERG, Y. Compressing pre-trained language models by matrix decomposition. In *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing, AACL/IJCNLP 2020, Suzhou, China, December 4-7, 2020* (2020), K. Wong, K. Knight, and H. Wu, Eds., Association for Computational Linguistics, pp. 884–889.

[198] NOROUZI, M., AND FLEET, D. J. Cartesian k-means. In *Proceedings of the IEEE Conference on computer Vision and Pattern Recognition* (2013), pp. 3017–3024.

[199] NVIDIA. Nvidia a100 specifications.

[200] NVIDIA. Nvidia tensor core.

[201] NVIDIA. The api reference guide for cusparse, the cuda sparse matrix library, 2021.

[202] NVIDIA. Geforce rtx 4090. https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/, 2024.

[203] NVIDIA. Nvbench: Nvidia's benchmarking tool for gpus, 2024. Available online: https://github.com/NVIDIA/nvbench.

[204] NVIDIA. Nvidia blackwell platform arrives to power a new era of computing, March 2024.

[205] NVIDIA. Nvidia h100 tensor core gpu. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet, 2024.

[206] OPENAI. Gpt-3.5 turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo.

[207] ORTEGA, G., VÁZQUEZ, F., GARCÍA, I., AND GARZÓN, E. M. Fastspmm: An efficient library for sparse matrix matrix product on gpus. *The Computer Journal* (2014).

[208] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., DESMAISON, A., KOPF, A., YANG, E., DEVITO, Z., RAISON, M., TEJANI, A., CHILAMKURTHY, S., STEINER, B., FANG, L., BAI, J., AND CHINTALA,

S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*.

[209] PATEL, P., CHOUKSE, E., ZHANG, C., ÍÑIGO GOIRI, SHAH, A., MALEKI, S., AND BIANCHINI, R. Splitwise: Efficient generative llm inference using phase splitting, 2023.

[210] PATHWAY. Adaptive rag: How we cut llm costs without sacrificing accuracy. https://pathway.com/developers/showcases/adaptive-rag, 2024.

[211] PATIL, S. G., ZHANG, T., WANG, X., AND GONZALEZ, J. E. Gorilla: Large language model connected with massive apis, 2023.

[212] PENG, W., LI, G., JIANG, Y., WANG, Z., OU, D., ZENG, X., TONGXU, AND CHEN, E. Large language model based long-tail query rewriting in taobao search. *Companion Proceedings of the ACM on Web Conference 2024* (2023).

[213] PENG, W., LI, G., JIANG, Y., WANG, Z., OU, D., ZENG, X., XU, D., XU, T., AND CHEN, E. Large language model based long-tail query rewriting in taobao search. In *Companion Proceedings of the ACM on Web Conference 2024* (2024).

[214] POOLE, B., JAIN, A., BARRON, J. T., AND MILDENHALL, B. Dreamfusion: Text-to-3d using 2d diffusion. *arXiv* (2022).

[215] POPE, R., DOUGLAS, S., CHOWDHERY, A., DEVLIN, J., BRADBURY, J., LEVSKAYA, A., HEEK, J., XIAO, K., AGRAWAL, S., AND DEAN, J. Efficiently scaling transformer inference. *ArXiv abs/2211.05102* (2022).

[216] PRESS, O., SMITH, N., AND LEWIS, M. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations* (2022).

[217] PRESS, O., ZHANG, M., MIN, S., SCHMIDT, L., SMITH, N. A., AND LEWIS, M. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023* (2023), pp. 5687–5711.

[218] QUINN, D., NOURI, M., PATEL, N., SALIHU, J., SALEMI, A., LEE, S., ZAMANI, H., AND ALIAN, M. Accelerating retrieval-augmented generation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (New York, NY, USA, 2025), ASPLOS '25, Association for Computing Machinery, p. 15–32.

[219] RADFORD, A., KIM, J. W., HALLACY, C., RAMESH, A., GOH, G., AGARWAL, S., SASTRY, G., ASKELL, A., MISHKIN, P., CLARK, J., KRUEGER, G., AND SUTSKEVER, I. Learning transferable visual models from natural language supervision, 2021.

[220] RAFFEL, C., SHAZEER, N., ROBERTS, A., LEE, K., NARANG, S., MATENA, M., ZHOU, Y., LI, W., AND LIU, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res. 21*, 1 (jan 2020).

[221] RAM, O., LEVINE, Y., DALMEDIGOS, I., MUHLGAY, D., SHASHUA, A., LEYTON-BROWN, K., AND SHOHAM, Y. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics 11* (2023), 1316–1331.

[222] RAPIDSAI. Rapidsai/raft: Raft contains fundamental widely-used algorithms and primitives for data science, graph and machine learning. https://github.com/rapidsai/raft, 2022.

[223] RASTEGARI, M., ORDONEZ, V., REDMON, J., AND FARHADI, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV* (2016).

[224] REISER, C., PENG, S., LIAO, Y., AND GEIGER, A. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In *International Conference on Computer Vision (ICCV)* (2021).

[225] RONG, Y., HUANG, W., XU, T., AND HUANG, J. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations* (2020).

[226] ROUHANI, B. D., ZHAO, R., MORE, A., HALL, M., KHODAMORADI, A., DENG, S., CHOUDHARY, D., CORNEA, M., DELLINGER, E., DENOLF, K., DUSAN, S., ELANGO, V., GOLUB, M., HEINECKE, A., JAMES-ROXBY, P., JANI, D., KOLHE, G., LANGHAMMER, M., LI, A., MELNICK, L., MESMAKHOSROSHAHI, M., RODRIGUEZ, A., SCHULTE, M., SHAFIPOUR, R., SHAO, L., SIU,

M., DUBEY, P., MICIKEVICIUS, P., NAUMOV, M., VERRILLI, C., WITTIG, R., BURGER, D., AND CHUNG, E. Microscaling data formats for deep learning, 2023.

[227] SAKAGUCHI, K., BRAS, R. L., BHAGAVATULA, C., AND CHOI, Y. Winogrande: An adversarial winograd schema challenge at scale, 2019.

[228] SANDLER, M., HOWARD, A. G., ZHU, M., ZHMOGINOV, A., AND CHEN, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), 4510–4520.

[229] SARA FRIDOVICH-KEIL AND GIACOMO MEANTI, WARBURG, F. R., RECHT, B., AND KANAZAWA, A. K-planes: Explicit radiance fields in space, time, and appearance. In *CVPR* (2023).

[230] SEEMAKHUPT, K., LIU, S., AND KHAN, S. Edgerag: Online-indexed rag for edge devices, 2024.

[231] SHAO, R., HE, J., ASAI, A., SHI, W., DETTMERS, T., MIN, S., ZETTLEMOYER, L., AND KOH, P. W. Scaling retrieval-based language models with a trillion-token datastore. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

[232] SHAO, W., CHEN, M., ZHANG, Z., XU, P., ZHAO, L., LI, Z., ZHANG, K., GAO, P., QIAO, Y., AND LUO, P. Omniquant: Omnidirectionally calibrated quantization for large language models, 2023.

[233] SHAO, Z., GONG, Y., SHEN, Y., HUANG, M., DUAN, N., AND CHEN, W. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. In *Findings of the Association for Computational Linguistics: EMNLP 2023* (2023), pp. 9248–9274.

[234] SHARMA, P., ASH, J. T., AND MISRA, D. The truth is in there: Improving reasoning in language models with layer-selective rank reduction, 2023.

[235] SHAZEER, N. Fast transformer decoding: One write-head is all you need, 2019.

[236] SHEN, Z., LIU, Z., AND XING, E. P. Sliced recursive transformer. *CoRR abs/2111.05297* (2021).

[237] SHENG, Y., ZHENG, L., YUAN, B., LI, Z., RYABININ, M., FU, D. Y., XIE, Z., CHEN, B., BARRETT, C. W., GONZALEZ, J., LIANG, P., RÉ, C., STOICA, I., AND ZHANG, C. High-throughput

generative inference of large language models with a single gpu. In *International Conference on Machine Learning* (2023).

[238] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations* (2015).

[239] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition.

[240] SIVIC, J., AND ZISSERMAN, A. Video google: A text retrieval approach to object matching in videos. In *ICCV* (2003), IEEE Computer Society, pp. 1470–1477.

[241] SPOERER, C. J., KIETZMANN, T. C., MEHRER, J., CHAREST, I., AND KRIEGESKORTE, N. Recurrent networks can recycle neural resources to flexibly trade speed for accuracy in visual recognition. *bioRxiv* (2020).

[242] STEWART, D., AND LINSDELL, J. Say hello to precision: How rerankers and embeddings boost search. https://cohere.com/blog/say-hello-to-precision-how-rerankers-and-embeddings-boost-search, 2024.

[243] SU, J., LU, Y., PAN, S., WEN, B., AND LIU, Y. Roformer: Enhanced transformer with rotary position embedding. *CoRR abs/2104.09864* (2021).

[244] SUN, C., SUN, M., AND CHEN, H.-T. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. *CVPR* (2022).

[245] TAKASE, S., AND KIYONO, S. Lessons on parameter sharing across layers in transformers. *CoRR abs/2104.06022* (2021).

[246] TAN, M., CHEN, B., PANG, R., VASUDEVAN, V., AND LE, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), 2815–2823.

[247] TAN, M., AND LE, Q. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning* (09–15 Jun 2019), K. Chaud-

huri and R. Salakhutdinov, Eds., vol. 97 of *Proceedings of Machine Learning Research*, PMLR, pp. 6105–6114.

[248] THAKKAR, V., RAMANI, P., CECKA, C., SHIVAM, A., LU, H., YAN, E., KOSAIAN, J., HOEMMEN, M., WU, H., KERR, A., NICELY, M., MERRILL, D., BLASIG, D., QIAO, F., MAJCHER, P., SPRINGER, P., HOHNERBACH, M., WANG, J., AND GUPTA, M. CUTLASS, Jan. 2023.

[249] TILLET, P., KUNG, H.-T., AND COX, D. 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* (2019), pp. 10–19.

[250] TOLSTIKHIN, I. O., HOULSBY, N., KOLESNIKOV, A., BEYER, L., ZHAI, X., UNTERTHINER, T., YUNG, J., KEYSERS, D., USZKOREIT, J., LUCIC, M., AND DOSOVITSKIY, A. Mlp-mixer: An all-mlp architecture for vision. *CoRR abs/2105.01601* (2021).

[251] TOUVRON, H., LAVRIL, T., IZACARD, G., MARTINET, X., LACHAUX, M.-A., LACROIX, T., ROZIÈRE, B., GOYAL, N., HAMBRO, E., AZHAR, F., RODRIGUEZ, A., JOULIN, A., GRAVE, E., AND LAMPLE, G. Llama: Open and efficient foundation language models, 2023.

[252] TOUVRON, H., MARTIN, L., STONE, K., ALBERT, P., ALMAHAIRI, A., BABAEI, Y., BASHLYKOV, N., BATRA, S., BHARGAVA, P., BHOSALE, S., BIKEL, D., BLECHER, L., FERRER, C. C., CHEN, M., CUCURULL, G., ESIOBU, D., FERNANDES, J., FU, J., FU, W., FULLER, B., GAO, C., GOSWAMI, V., GOYAL, N., HARTSHORN, A., HOSSEINI, S., HOU, R., INAN, H., KARDAS, M., KERKEZ, V., KHABSA, M., KLOUMANN, I., KORENEV, A., KOURA, P. S., LACHAUX, M.-A., LAVRIL, T., LEE, J., LISKOVICH, D., LU, Y., MAO, Y., MARTINET, X., MIHAYLOV, T., MISHRA, P., MOLYBOG, I., NIE, Y., POULTON, A., REIZENSTEIN, J., RUNGTA, R., SALADI, K., SCHELTEN, A., SILVA, R., SMITH, E. M., SUBRAMANIAN, R., TAN, X. E., TANG, B., TAYLOR, R., WILLIAMS, A., KUAN, J. X., XU, P., YAN, Z., ZAROV, I., ZHANG, Y., FAN, A., KAMBADUR, M., NARANG, S., RODRIGUEZ, A., STOJNIC, R., EDUNOV, S., AND SCIALOM, T. Llama 2: Open foundation and fine-tuned chat models, 2023.

[253] TOUVRON, H., MARTIN, L., STONE, K., ALBERT, P., ALMAHAIRI, A., BABAEI, Y., BASH-

LYKOV, N., BATRA, S., BHARGAVA, P., BHOSALE, S., BIKEL, D., BLECHER, L., FERRER, C. C., CHEN, M., CUCURULL, G., ESIOBU, D., FERNANDES, J., FU, J., FU, W., FULLER, B., GAO, C., GOSWAMI, V., GOYAL, N., HARTSHORN, A., HOSSEINI, S., HOU, R., INAN, H., KARDAS, M., KERKEZ, V., KHABSA, M., KLOUMANN, I., KORENEV, A., KOURA, P. S., LACHAUX, M.-A., LAVRIL, T., LEE, J., LISKOVICH, D., LU, Y., MAO, Y., MARTINET, X., MIHAYLOV, T., MISHRA, P., MOLYBOG, I., NIE, Y., POULTON, A., REIZENSTEIN, J., RUNGTA, R., SALADI, K., SCHELTEN, A., SILVA, R., SMITH, E. M., SUBRAMANIAN, R., TAN, X. E., TANG, B., TAYLOR, R., WILLIAMS, A., KUAN, J. X., XU, P., YAN, Z., ZAROV, I., ZHANG, Y., FAN, A., KAMBADUR, M., NARANG, S., RODRIGUEZ, A., STOJNIC, R., EDUNOV, S., AND SCIALOM, T. Llama 2: Open foundation and fine-tuned chat models, 2023.

[254] TRAUZETTEL-KLOSINSKI, S., DIETZ, K., AND THE IRESTSTUDY GROUP. Standardized Assessment of Reading Performance: The New International Reading Speed Texts IReST. *Investigative Ophthalmology & Visual Science 53*, 9 (08 2012), 5452–5461.

[255] TROCKMAN, A., AND KOLTER, J. Z. Patches are all you need? *CoRR abs/2201.09792* (2022).

[256] TSENG, A., CHEE, J., SUN, Q., KULESHOV, V., AND DE SA, C. Quip#: Even better llm quantization with hadamard incoherence and lattice codebooks. *arXiv preprint arXiv:2402.04396* (2024).

[257] TUNG, K. Enhancing user experience by overcoming latency in the ion iq chatbot. https://www.ontinue.com/resource/enhancing-user-experience-by-overcoming-latency-in-the-ion-iq-chatbot/, 2024.

[258] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (2017), I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., pp. 5998–6008.

[259] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need, 2023.

[260] VIJEASWARAN, N., DHIMINE, A., DIAZ, A., BUSTILLO, S., SABIR, F., AND PUNIO, M. Advanced rag patterns on amazon sagemaker. https://aws.amazon.com/blogs/machine-learning/advanced-rag-patterns-on-amazon-sagemaker/, 2024.

[261] WANG, C., WU, X., GUO, Y.-C., ZHANG, S.-H., TAI, Y.-W., AND HU, S.-M. Nerf-sr: High quality neural radiance fields using supersampling. In *Proceedings of the 30th ACM International Conference on Multimedia* (New York, NY, USA, 2022), MM '22, Association for Computing Machinery, p. 6445–6454.

[262] WANG, G., ZHAO, Y., TANG, C., LUO, C., AND ZENG, W. When shift operation meets vision transformer: An extremely simple alternative to attention mechanism. *CoRR abs/2201.10801* (2022).

[263] WANG, L., MA, C., FENG, X., ZHANG, Z., YANG, H., ZHANG, J., CHEN, Z., TANG, J., CHEN, X., LIN, Y., ET AL. A survey on large language model based autonomous agents. *Frontiers of Computer Science 18*, 6 (2024), 186345.

[264] WANG, M., YU, L., ZHENG, D., GAN, Q., GAI, Y., YE, Z., LI, M., ZHOU, J., HUANG, Q., MA, C., HUANG, Z., GUO, Q., ZHANG, H., LIN, H., ZHAO, J., LI, J., SMOLA, A. J., AND ZHANG, Z. Deep graph library: Towards efficient and scalable deep learning on graphs.

[265] WANG, P., LIU, L., LIU, Y., THEOBALT, C., KOMURA, T., AND WANG, W. Neus: Learning neural implicit surfaces by volume rendering for multi-view reconstruction. *NeurIPS* (2021).

[266] WANG, X., ZHENG, Y., WAN, Z., AND ZHANG, M. Svd-llm: Truncation-aware singular value decomposition for large language model compression. *arXiv preprint arXiv:2403.07378* (2024).

[267] WANG, Z., LI, L., SHEN, Z., SHEN, L., AND BO, L. 4k-nerf: High fidelity neural radiance fields at ultra high resolutions. *arXiv preprint arXiv:2212.04701* (2022).

[268] WANG, Z., WANG, Z., LE, L., ZHENG, H. S., MISHRA, S., PEROT, V., ZHANG, Y., MATTAPALLI, A., TALY, A., SHANG, J., ET AL. Speculative rag: Enhancing retrieval augmented generation through drafting. *arXiv preprint arXiv:2407.08223* (2024).

[269] WANG, Z., WU, S., XIE, W., CHEN, M., AND PRISACARIU, V. A. NeRF–: Neural radiance fields without known camera parameters. *https://arxiv.org/abs/2102.07064* (2021).

[270] WANG, Z. J., AND CHAU, D. H. Mememo: On-device retrieval augmentation for private and personalized text generation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval* (New York, NY, USA, 2024), SIGIR '24, Association for Computing Machinery, p. 2765–2770.

[271] WEI, K. Advanced rag with azure ai search and llamaindex. https://techcommunity.microsoft.com/t5/ai-azure-ai-services-blog/advanced-rag-with-azure-ai-search-and-llamaindex/ba-p/4115007, 2024.

[272] WEI, X., ZHANG, Y., LI, Y., ZHANG, X., GONG, R., GUO, J., AND LIU, X. Outlier suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling, 2023.

[273] WIKIPEDIA. Nvidia rubin microarchitecture, 2025.

[274] WIKIPEDIA CONTRIBUTORS. List of qualcomm snapdragon systems on chips — Wikipedia, the free encyclopedia, 2023. [Online; accessed 26-October-2023].

[275] WILLIAMS, S., WATERMAN, A., AND PATTERSON, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM 52*, 4 (2009), 65–76.

[276] WOLF, T., DEBUT, L., SANH, V., CHAUMOND, J., DELANGUE, C., MOI, A., CISTAC, P., RAULT, T., LOUF, R., FUNTOWICZ, M., DAVISON, J., SHLEIFER, S., VON PLATEN, P., MA, C., JERNITE, Y., PLU, J., XU, C., LE SCAO, T., GUGGER, S., DRAME, M., LHOEST, Q., AND RUSH, A. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Online, Oct. 2020), Q. Liu and D. Schlangen, Eds., Association for Computational Linguistics, pp. 38–45.

[277] WU, X., YAO, Z., AND HE, Y. Zeroquant-fp: A leap forward in llms post-training w4a8 quantization using floating-point formats, 2023.

[278] WU, Z., JIN, Y., AND YI, K. M. Neural fourier filter bank. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2023), pp. 14153–14163.

[279] WUTSCHITZ, L., KÖPF, B., PAVERD, A., RAJMOHAN, S., SALEM, A., TOPLE, S., ZANELLA-BÉGUELIN, S., XIA, M., AND RÜHLE, V. Rethinking privacy in machine learning pipelines from an information flow control perspective, 2023.

[280] XIANGLI, Y., XU, L., PAN, X., ZHAO, N., RAO, A., THEOBALT, C., DAI, B., AND LIN, D. Bungeenerf: Progressive neural radiance field for extreme multi-scale scene rendering. In *The European Conference on Computer Vision (ECCV)* (2022).

[281] XIAO, G., LIN, J., SEZNEC, M., WU, H., DEMOUTH, J., AND HAN, S. Smoothquant: Accurate and efficient post-training quantization for large language models, 2023.

[282] XIAO, G., TIAN, Y., CHEN, B., HAN, S., AND LEWIS, M. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations* (2024).

[283] XIE, Z., KANG, H., SHENG, Y., KRISHNA, T., FATAHALIAN, K., AND KOZYRAKIS, C. Ai metropolis: Scaling large language model-based multi-agent simulation with out-of-order execution. *arXiv preprint arXiv:2411.03519* (2024).

[284] XU, K., LI, C., TIAN, Y., SONOBE, T., ICHI KAWARABAYASHI, K., AND JEGELKA, S. Representation learning on graphs with jumping knowledge networks, 2018.

[285] YANG, C., BULUC, A., AND OWENS, J. D. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing* (2018).

[286] YANG, C., AND MIHEER VAIDYA. Merge-spmm open source code, 2020.

[287] YANG, Z., QI, P., ZHANG, S., BENGIO, Y., COHEN, W. W., SALAKHUTDINOV, R., AND MANNING, C. D. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2018).

[288] YAO, J., LI, H., LIU, Y., RAY, S., CHENG, Y., ZHANG, Q., DU, K., LU, S., AND JIANG, J. Cacheblend: Fast large language model serving for rag with cached knowledge fusion, 2024.

[289] YAO, Z., AMINABADI, R. Y., ZHANG, M., WU, X., LI, C., AND HE, Y. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers, 2022.

[290] YARIV, L., GU, J., KASTEN, Y., AND LIPMAN, Y. Volume rendering of neural implicit surfaces. *NeurIPS* (2021).

[291] YE, F., FANG, M., LI, S., AND YILMAZ, E. Enhancing conversational search: Large language model-aided informative query rewriting. In *The 2023 Conference on Empirical Methods in Natural Language Processing* (2023).

[292] YE, Z., CHEN, L., LAI, R., ZHAO, Y., ZHENG, S., SHAO, J., HOU, B., JIN, H., ZUO, Y., YIN, L., CHEN, T., AND CEZE, L. Accelerating self-attentions for llm serving with flashinfer, February 2024.

[293] YEN-CHEN, L., FLORENCE, P., BARRON, J. T., RODRIGUEZ, A., ISOLA, P., AND LIN, T.-Y. iNeRF: Inverting neural radiance fields for pose estimation. *https://arxiv.org/abs/2012.05877* (2020).

[294] YU, A., LI, R., TANCIK, M., LI, H., NG, R., AND KANAZAWA, A. PlenOctrees for real-time rendering of neural radiance fields. In *ICCV* (2021).

[295] YU, G.-I., JEONG, J. S., KIM, G.-W., KIM, S., AND CHUN, B.-G. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 521–538.

[296] YUAN, J., SUN, S., OMEIZA, D., ZHAO, B., NEWMAN, P., KUNZE, L., AND GADD, M. Rag-driver: Generalisable driving explanations with retrieval-augmented in-context learning in multi-modal large language model. *arXiv preprint arXiv:2402.10828* (2024).

[297] YUAN, Z., NIU, L., LIU, J., LIU, W., WANG, X., SHANG, Y., SUN, G., WU, Q., WU, J., AND WU, B. Rptq: Reorder-based post-training quantization for large language models, 2023.

[298] YUAN, Z., SHANG, Y., SONG, Y., WU, Q., YAN, Y., AND SUN, G. Asvd: Activation-aware singular value decomposition for compressing large language models, 2023.

[299] YUAN, Z., SHANG, Y., ZHOU, Y., DONG, Z., ZHOU, Z., XUE, C., WU, B., LI, Z., GU, Q., LEE, Y. J., YAN, Y., CHEN, B., SUN, G., AND KEUTZER, K. LLM inference unveiled: Survey and roofline model insights. *CoRR abs/2402.16363* (2024).

[300] YUE, Y., YUAN, Z., DUANMU, H., ZHOU, S., WU, J., AND NIE, L. Wkvquant: Quantizing weight and key/value cache for large language models gains more. *arXiv preprint arXiv:2402.12065* (2024).

[301] ZAHARIA, M., KHATTAB, O., CHEN, L., DAVIS, J. Q., MILLER, H., POTTS, C., ZOU, J., CARBIN, M., FRANKLE, J., RAO, N., AND GHODSI, A. The shift from models to compound ai systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.

[302] ZELLERS, R., HOLTZMAN, A., BISK, Y., FARHADI, A., AND CHOI, Y. Hellaswag: Can a machine really finish your sentence?, 2019.

[303] ZENG, C., LUO, L., NING, Q., HAN, Y., JIANG, Y., TANG, D., WANG, Z., CHEN, K., AND GUO, C. FAERY: An FPGA-accelerated embedding-based retrieval system. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (2022), pp. 841–856.

[304] ZENG, H., ZHOU, H., SRIVASTAVA, A., KANNAN, R., AND PRASANNA, V. GraphSAINT: Graph sampling based inductive learning method. In *International Conference on Learning Representations* (2020).

[305] ZHAI, S., TALBOTT, W., SRIVASTAVA, N., HUANG, C., GOH, H., ZHANG, R., AND SUSSKIND, J. M. An attention free transformer. *CoRR abs/2105.14103* (2021).

[306] ZHANG, R., ISOLA, P., EFROS, A. A., SHECHTMAN, E., AND WANG, O. The unreasonable effectiveness of deep features as a perceptual metric. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), pp. 586–595.

[307] ZHANG, Z., LIU, F., HUANG, G., LIU, X., AND JIN, X. Fast vector query processing for large datasets beyond GPU memory with reordered pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (2024), pp. 23–40.

[308] ZHANG, Z., SHENG, Y., ZHOU, T., CHEN, T., ZHENG, L., CAI, R., SONG, Z., TIAN, Y., RÉ, C., BARRETT, C., ET AL. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems 36* (2024).

[309] ZHANG, Z., SHENG, Y., ZHOU, T., CHEN, T., ZHENG, L., CAI, R., SONG, Z., TIAN, Y., RÉ, C., BARRETT, C., WANG, Z., AND CHEN, B. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.

[310] ZHANG, Z., ZHU, A., YANG, L., XU, Y., LI, L., PHOTHILIMTHANA, P. M., AND JIA, Z. Accelerating retrieval-augmented language model serving with speculation. *arXiv preprint arXiv:2401.14021* (2024).

[311] ZHAO, J., ZHANG, Z., CHEN, B., WANG, Z., ANANDKUMAR, A., AND TIAN, Y. Galore: Memory-efficient LLM training by gradient low-rank projection. *CoRR abs/2403.03507* (2024).

[312] ZHAO, Y., LIN, C.-Y., ZHU, K., YE, Z., CHEN, L., ZHENGA, S., CEZE, L., KRISHNAMURTHY, A., CHEN, T., AND KASIKCI, B. Atom: Low-bit quantization for efficient and accurate llm serving. *arXiv preprint arXiv:2310.19102* (2023).

[313] ZHAO, Y., LIN, C.-Y. L., ZHU, K., YE, Z., CHEN, L., ZHENG, S., CEZE, L., KRISHNAMURTHY, A., CHEN, T., AND KASIKCI, B. Atom: Low-bit quantization for efficient and accurate llm serving. *MLSys* (2024).

[314] ZHENG, C., ZONG, B., CHENG, W., SONG, D., NI, J., YU, W., CHEN, H., AND WANG, W. Robust graph representation learning via neural sparsification. In *Proceedings of the 37th International Conference on Machine Learning* (2020), Proceedings of Machine Learning Research.

[315] ZHENG, H. S., MISHRA, S., CHEN, X., CHENG, H.-T., CHI, E. H., LE, Q. V., AND ZHOU, D. Take a step back: Evoking reasoning via abstraction in large language models. *arXiv preprint arXiv:2310.06117* (2023).

[316] ZHENG, L., YIN, L., XIE, Z., SUN, C., HUANG, J., YU, C. H., CAO, S., KOZYRAKIS, C., STOICA, I., GONZALEZ, J. E., BARRETT, C., AND SHENG, Y. Sglang: Efficient execution of structured language model programs, 2024.

[317] ZHONG, Y., LIU, S., CHEN, J., HU, J., ZHU, Y., LIU, X., JIN, X., AND ZHANG, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

[318] ZHOU, C., RICHARD, V., SAVARESE, P., HASSMAN, Z., MAIRE, M., DIBRINO, M., AND LI, Y. Sysmol: A hardware-software co-design framework for ultra-low and fine-grained mixed-precision neural networks, 2023.

[319] ZHOU, D., SCHÄRLI, N., HOU, L., WEI, J., SCALES, N., WANG, X., SCHUURMANS, D., CUI, C., BOUSQUET, O., LE, Q. V., ET AL. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations* (2022).

[320] ZHU, Y. RTNN: Accelerating neighbor search using hardware ray tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Apr. 2022), PPoPP '22, pp. 76–89.

[321] ZHUANG, S., LIU, B., KOOPMAN, B., AND ZUCCON, G. Open-source large language models are strong zero-shot query likelihood models for document ranking. In *Findings of the Association for Computational Linguistics: EMNLP 2023* (2023), pp. 8807–8817.

[322] ZILLIZ. How to select index parameters for ivf index. https://zilliz.com/blog/select-index-parameters-ivf-index, 2020.

[323] ZITNIK, M., AGRAWAL, M., AND LESKOVEC, J. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics 34*, 13 (2018), 457–466.