eLLM: Achieving Lossless Million-Token LLM Inference on CPUs Faster Than GPUs

Yaguang Huangfu*
lucienhuangfu@outlook.com

Abstract

LLMs are foundational to AI, yet cloud vendors offer inference APIs that depend on expensive GPUs. There is a growing need for an affordable and efficient solution for long-sequence inference. LLMs require accumulating context (KV cache) and computing attention over this context, where attention computation is memory-bound and relies on contiguous memory access to achieve high memory bandwidth. Existing frameworks repeatedly construct temporary tensors to handle inputs with dynamic shapes. Furthermore, due to limited GPU memory and unpredictable sequence lengths, these frameworks are forced to split KV caches into chunks forming logically unbounded tensors. In contrast, CPU-only computers with large memory capacities and substantial memory bandwidth present a viable alternative. We introduce **eLLM**, a framework preconstructs a elastic computation graph using permanent tensors on CPU-only machines for maximum shapes. Our experimental results demonstrate that **eLLM** outperforms the baseline system in attention computation when the batch size is normalized so that each user receives a comparable share of memory bandwidth and cost. This approach enables contiguous memory access for KV caches and leverages hardware prefetching to fully utilize peak memory bandwidth. It efficiently adapts computation to support ragged input and output shapes without additional overhead, and scales seamlessly to handle millions of tokens across multiple rounds of dialogue without compromising accuracy. The code is available at: https://github.com/lucienhuangfu/eLLM

Keywords: Machine Learning System, Inference Framework, LLM, Computational Graph

1 Introduction

Large Language Models (LLMs) [21] serve as the foundation of modern AI applications. However, cloud providers offer inference services powered by expensive GPUs, driving the demand for affordable and efficient solutions, particularly for long-sequence [8]. LLM inference accumulates contextual information in a key-value (KV) cache and performs attention computation [30] over this context. Attention computation is memory-bound [34], depending on contiguous memory access to achieve high bandwidth. As sequence length increases, attention computation becomes the dominant factor in inference time. Existing frameworks repeatedly construct temporary tensors to handle inputs with dynamic shapes. Furthermore, due to limited GPU memory and unpredictable sequence lengths, these frameworks are forced to split KV caches into chunks [14] forming logically unbounded tensors. In contrast, CPU-only computers [33] with large memory capacities [12] and substantial memory bandwidth offer a promising alternative. We introduce **eLLM**, a novel inference framework that adopts a space-for-time strategy. **eLLM** preconstructs a elastic computation graph using permanent tensors on CPU-only machines for maximum shapes. To accommodate dynamic-shaped inputs, the framework adopts a dimension-first data layout. To save memory, intermediate tensors are reused across position and block dimensions. Our experimental results demonstrate that **eLLM** outperforms the baseline system in attention computation when the batch size is normalized so that each user receives a comparable share of memory bandwidth and cost. As sequence length increases, eLLM outperforms GPUs in positional time and, ultimately, in total time. This approach enables contiguous memory access for KV caches and leverages hardware prefetching to fully utilize peak memory bandwidth. It efficiently adapts computation to support ragged input and output shapes without additional overhead, and scales seamlessly to handle millions of tokens across multiple rounds of dialogue without compromising accuracy.

2 Problem

LLMs serve as foundational components for a wide range of AI applications, including deep research and fiction writing, both of which require processing sequences that range from thousands to millions of tokens. For example, deep research often involves reading at least 10 papers, with each paper easily exceeding 10,000 words—resulting in a total of over 100,000 tokens. [7].

Long Sequence Definition: As the sequence length increases, the total memory bandwidth consumed by attention computation eventually surpasses that of embedding computation. When the total number of tokens exceeds this threshold, we define the scenario as *long-sequence inference*. For example, with LLaMA3-70B (FP16) [10] , a total of 400,000 tokens results in the KV cache consuming 140 GB of memory bandwidth—matching the bandwidth required by the model parameters. This token count can arise from various configurations, such as: ❶ Batch size = 1, sequence length = 400,000; ❷ Batch size = 100, sequence length = 4,000.

*Part of the work was done at The Hong Kong Polytechnic University.

2.1 Issues

High-performance LLM inference is primarily conducted on high-end multi-GPU machines, but GPUs face three major issues when infering long sequences:

(1) Deficient memory: LLMs require memory both for storing model parameters and accumulating KV cache [8]. For LLaMA3-70B (FP16), model parameters alone need 140 GB. The KV cache grows with total token count. For instance, 1 million tokens require 328 GB of memory, such as: ❶ Batch size = 1, Sequence length = 1,000,000; ❷ Batch size = 100, Sequence length = 10,000. However, GPUs are short of memory (A100 80GB). Many GPUs are required to meet memory demands, significantly increasing costs. CPUs (CPU-only computers) have two orders of magnitude more memory capacity than GPUs, Such as Xeon 4TB. CPUs and their memory are both cheap.

(2) Enlarged Latency: Attention time scales linearly with both batch size and sequence length. As the sequence length increases, attention computation gradually dominates the overall inference time. To meet real-time requirements in online services, the batch size must be reduced as sequence length grows. If the batch size ratio between GPU and CPU matches the ratio of their memory bandwidths, then each data sample receives the same memory bandwidth, and their latency becomes comparable. If the GPU batch size is too large, each sample gets less memory bandwidth, causing GPU latency to exceed that of the CPU. Conversely, if the GPU batch size is too small, each sample gets more memory bandwidth, and GPU will be faster than CPU.

(3) Underutilized Cores: Inference consists of two fundamentally different components: embedding computation is compute-bound, while attention computation is memory-bound [34]. On GPUs with many-core architectures, core utilization is often suboptimal during inference. For embedding computation, GPUs require large batch sizes to fully utilize their compute capacity. However, long-sequence inferences restrict the batch size, leading to underutilization. For attention computation, each core receives insufficient memory bandwidth. For example, an A100 GPU with 2 TB/s bandwidth must serve 108×64 cores, severely limiting per-core bandwidth. In contrast, CPUs with multi-core architectures can maintain better utilization during inference. For embedding computation, CPUs can achieve adequate task parallelism with moderate batch sizes. For attention computation, CPUs provide sufficient memory bandwidth per core. For instance, a Xeon CPU with 0.5 TB/s memory bandwidth share across 48 cores.

2.2 Formulation

We focus on **token-wise inference** for LLMs, where the model takes an input prompt and incrementally generates one token at a time until an end-of-sequence (EOS) token is produced. This scenario encompasses both prompt and generation workloads. The inference framework must handle dynamic inputs with the following characteristics:

- **Sequence Length:** Variable, ranging from 1 to 1 million tokens
- **Batch Size:** Variable, ranging from 1 to the maximum supported size

To enable a fair comparison between GPU and CPU performance, we assume:

> *The ratio of GPU batch size to CPU batch size is equal to the ratio of their memory bandwidths.*

This assumption ensures that each sample is allocated the same amount of memory bandwidth, allowing for per-user cost equivalence across hardware platforms.

Our goal is to design a novel LLM inference framework on a **CPU-only computer**, targeting the following:

- Scaling to million-token sequences
- Minimizing latency
- Reducing cost

Given the substantial latency of CPU embedding computation, the following conditions must hold for CPU inference to surpass GPU inference in attention computations:

- The CPU must fully utilize its peak memory bandwidth
- The GPU must fail to reach its peak memory bandwidth

3 Background

This section introduces the LLM stack [38], which consists of three hierarchical layers: model, framework, and hardware. The framework bridges the model and hardware layers, providing programming APIs for algorithm engineers and determining how to execute models on various hardware platforms [16]. Beyond model complexity, the framework plays a critical role in system performance [24], which is the focus of this paper. Its performance is limited by model complexity and hardware architecture.

3.1 Model

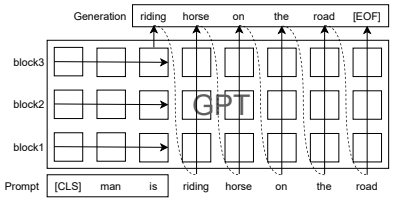

Figure 1. GPT Inference

The GPT inference [21] includes two stages: prefill and decode. The prefill stage processes the initial prompt sequence layer by layer (layer-wise traversal), while the decode stage

Layer	Decoder
Prediction	Softmax
Post-mapping	Linear and Nonlinear Mapping
Recombination	Column Matrix Multiplication
Weighting	Softmax
Score	Vector Matrix Multiplication
Pre-mapping	Linear Mapping
Lookup	Hash

Table 1. Transformer Block

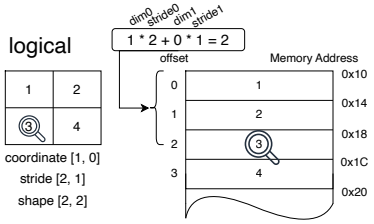

Figure 2. Tensor Offset

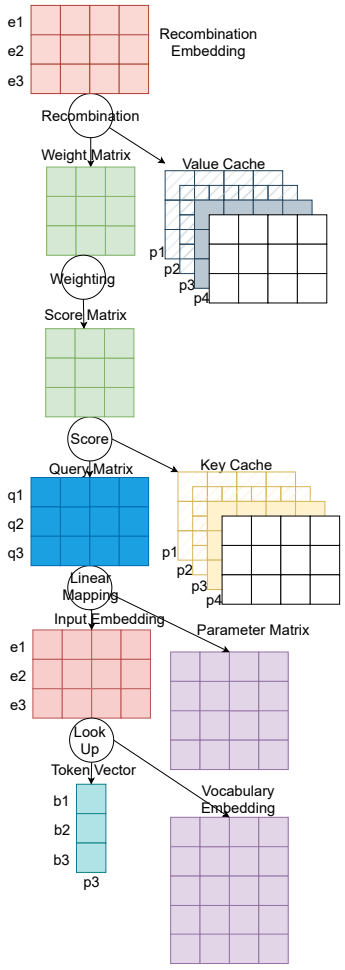

Figure 3. Computational Graph

generates subsequent tokens position by position (position-wise traversal) until an end-of-sequence (EOS) token is encountered. Both the length of prompt and generation are unpredictable. Interact rounds are unpredictable.

In this paper, we focus on the decode stage. Figure 1 illustrates the GPT inference with three Transformer blocks. The GPT model is based on decoder-only Transformer blocks [27]. Transformers utilize attention mechanisms [30] to compute embeddings and align outputs with inputs. Table 1 outlines the components of a Transformer block, consisting of seven parts:

a) **Lookup:** Maps input tokens with vocabulary embeddings to generate token embeddings.
b) **Pre-mapping:** Applies a linear transformation to the token embeddings, producing input embeddings.
c) **Score:** Multiplies the query with keys from the input to generate a scoring matrix.
d) **Weighting:** Applies the Softmax function to obtain the weight matrix.
e) **Recombination:** Computes a weighted average of the context using the weight matrix and value embeddings.
f) **Post-mapping:** Applies linear and non-linear transformations to the aligned embeddings, generating output embeddings.
g) **Prediction:** Performs Softmax function on the output embeddings to predict token IDs.

Transformer computations are divided into two parts: embedding computations (Pre-mapping and Post-mapping) and attention computations (Score, Weighting, and Recombination). Embedding computations primarily involve matrix multiplications with fixed shapes, while attention computations involve vector multiplications that scale with sequence length. Embedding computations are compute-bounded, while attention computations are memory-bounded [18]. At the meanwhile, KV cache increases with length. As sequence length increases, attention computations dominate both memory bandwidth and memory capacity requirements.

3.2 Framework

The computational graph [11] encapsulates the computations and data flow within neural networks. Nodes represent operators, while edges denote tensors. Operators are broadly categorized into two types: *shape operators* and *data operators*. Shape operators, such as `VIEW` and `TRANSPOSE`, modify the shape attributes of tensors without altering their data arrays. In contrast, data operators, such as `ADD` and `MULTIPLY`, modify both the shape attributes and the data arrays. The graph adheres to functional programming principles, ensuring immutability of input tensors. Each operator produces a new output tensor instead of modifying the input tensors.

Shape operations only adjust tensor metadata and share the data arrays with input tensors, as the underlying data remains unchanged. In contrast, data operations create new output tensors with newly allocated memory, as they involve modifications to the data.

Figure 3 depicts a computational graph for a Transformer block. Permanent tensors include parameter matrices, while all other tensors are temporary. In this example, the batch size is 3, and the position index is also 3. The first two KV embeddings have been computed and stored. During the linear mapping operators, the input embedding matrix and Query matrix are constructed. Parameter matrices are constant and preallocated. Once the linear mapping computation is complete, the input embedding matrix is deallocated, and the Query matrix is passed to the subsequent attention operator as the input tensor.

Tensors, which store input data, model parameters, or intermediate results, are organized as high-dimensional arrays stored contiguously in memory. Element access is determined through stride-based indexing, as illustrated in Figure 2.

During training, a programmer defines the computational graph, which is converted into an intermediate representation (IR) [15] and stored as a model file. Computational graphs are classified into two main types: **Static Graphs:** These graphs [1] are compiled once before execution, resulting in a fixed structure with pre-defined tensor shapes. **Dynamic Graphs:** These graphs [22] are constructed on-the-fly during execution, adapting to variable-length sequences.

The inference is divided into two primary stages: **initialization** and **prediction**. **Initialization Stage:** The framework loads the graph definition from the model file, constructs parameter tensors, and preallocates memory. **Prediction Stage:** During inference, a computation graph must be constructed sequentially for each position. While the graph structure remains identical across positions, the shapes of tensors may vary depending on the batch size. The framework traverses the graph, construct node and execute node. In graph node construction, it constructs intermediate tensors by computing their shapes and strides. Memory is allocated for these tensors. In graph node execution, tensors are divided into chunks, and package chunks and arithmetic functions into tasks. Each task is assigned arithmetic functions and dispatched to the target hardware as kernel functions for execution.

3.3 Hardware

CPUs are designed for low-latency, small-batch tasks, whereas GPUs are optimized for high-throughput, large-batch workloads. Table 2 shows a CPU workstation and a GPU work workstation. We compare CPUs and GPUs across three key aspects:

CPU Server	Metrics	GPU Server	
		CPU	GPU
13.3	Arithmetic Power (TFLOPS)	7.526	624
0.5	Memory Bandwidths (TB/S)	0.2	2
4	Memory Capacity (TB)	4	0.078
0.755	Configured Memory Capacity (TB)	0.866	0.078
Xeon 8488C	Type	AMD 7V13	A100
1	Number	4	4

Table 2. Comparison of CPU and GPU Server Specifications

Arithmetic Computation: While GPUs have significantly higher overall computational throughput, CPUs possess stronger single-core performance. CPUs typically feature multiple cores (generally fewer than 100), whereas GPUs follow a many-core architecture with thousands of cores.

Memory Bandwidth: GPUs offer significantly higher overall memory bandwidth compared to CPUs. However, achieving this bandwidth requires a large number of threads accessing different memory addresses concurrently. Additionally, GPU memory and cache latencies are an order of magnitude higher than those of CPUs. To offset these latencies, GPUs depend on large memory transactions per access. GPUs also provide kilobytes of programmable cache (shared memory) for temporarily storing intermediate results, a feature not directly available on CPUs. Furthermore, CPU hardware prefetchers are more advanced and efficient than those in GPUs

Memory Capacity: CPUs have a substantially larger memory capacity than GPUs. A single CPU [12] can accommodate terabytes of memory, making it suitable for storing large model parameters and long contexts without NUMA constraints. In contrast, a single GPU typically has limited memory (e.g., 80 GB), necessitating a NUMA architecture to interconnect multiple GPUs into a unified memory pool.

4 Motivation

Current CPU-only inference frameworks often inherit design principles from GPU-based systems, resulting in suboptimal performance. While these frameworks aim to minimize porting efforts, it fails to account for the fundamental architectural differences between CPUs and GPUs. Unlike CPUs, GPUs have limited memory capacity, which necessitates to build dynamic computation graphs . However, dynamic graph construction introduces significant inefficiencies. Due

to unpredictable sequence lengths, these frameworks are forced to split KV caches into chunks.

Our objective is to develop an efficient LLM inference framework tailored for CPU-only machines. The implementation of the computation graph has a profound impact on framework performance. To achieve this, we must rethink the computation graph design—eliminating graph management overhead and fully leveraging peak memory bandwidth.

4.1 Computation Graph

Due to limited memory and unpredictable sequence lengths, current frameworks must repeatedly construct dynamic computation graphs—particularly for managing KV caches. Constructing a single graph node involves computing tensor shapes, chunk offsets, and managing memory allocations. Several factors contribute to the high overhead:

- **Deep Graph Traversal:** Frameworks must traverse deep networks position by position, sequentially visiting all layers. For example, LLaMA3 70B [10] has 80 decoder blocks, each with 12 layers. For a sequence length of 1024, this results in the traversal of 983,040 nodes.
- **Sequential Task Deduction:** Each operator's task must be derived through a sequence of steps. Asynchronous pipelines store task data structures to decouple dependencies, enabling parallel execution of current tasks while precomputing future tasks. However, moving these task data structures between pipelines increases memory bandwidth consumption.

Moreover, GPU-based frameworks delegate graph construction to the host CPU, allowing the GPU to focus exclusively on task execution. This construction process can dominate the host CPU's utilization [29]. In contrast, CPU-only systems must handle both graph management and task execution on the same processor, placing additional strain on CPU resources.

This raises a key question: Can we generate executable tasks directly without repeatedly constructing computation graphs?

4.2 KV Cache

KV caches occupy the majority of memory during long-sequence inference. Their shape is determined by two dimensions: *batchsizesequencelength*. GPUs cannot construct fixed-shape tensors for the KV cache due to limited memory and unpredictable sequence lengths during inference. This limitation results in two scenarios: When the batch size is small and sequence length is large, the actual sequence length may be shorter than the preset length, leading to reduced throughput [13]. When the batch size is large and sequence length is small, the actual sequence length may exceed the preset length, restricting the maximum sequence length that can be processed.

Since the sequence length is unpredictable [13] , KV caches cannot be stored in statically bounded tensors, and reallocating them dynamically is costly. Therefore, KV caches must reside in logically unbounded tensors. To simulate such unbounded tensors, frameworks typically split the KV cache into chunks and manage them via a chunk table. However, this chunked representation cannot achieve peak memory bandwidth due to several overheads [14]: ❶ maintain a hash table for memory address mapping, ❷ perform address lookups prior to every KV chunk access, ❸ execute a large number of small tasks in attention computation

In contrast, CPUs can construct fixed tensor for KV Cache thanks to their large memory and narrower batch size range. Even under large batch size configurations, CPUs are capable of sustaining near-unbounded sequence lengths.

CPUs can keep KV Cache in Contiguous Layout: Storing the KV cache in contiguous memory enables efficient address computation using coordinate-based indexing. Can we preconstruct maximum KV cache which can hold KV values during inferences?

5 Approach

Since CPUs offer much larger memory capacities, so we propose a novel inference framework that leverages a space-for-time strategy. Our proposed solution, eLLM, employs static memory allocation to build elastic computational graphs on CPU-only machines.

5.1 Elastic Graph

A static computation graph needs to be compiled only once, but it supports only fixed input shapes. If the input shape does not match the compiled shape, all inputs must be aligned to the maximum shape, resulting in redundant computation. We improve upon the static graph by enabling the framework to build elastic computation graphs that support dynamic input shapes. To handle variable-shaped inputs efficiently, the framework adopts a dimension-first data layout, which allows tensors of varying shapes to be managed flexibly. Additionally, to reduce memory usage, intermediate tensors are reused across both the position and block dimensions.

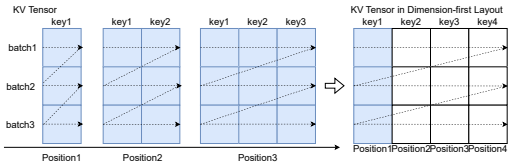

Figure 4. KV Tensor along Position

5.1.1 Dimension-first Layout.

Since the input varies across both the batch dimension and the sequence dimension, the

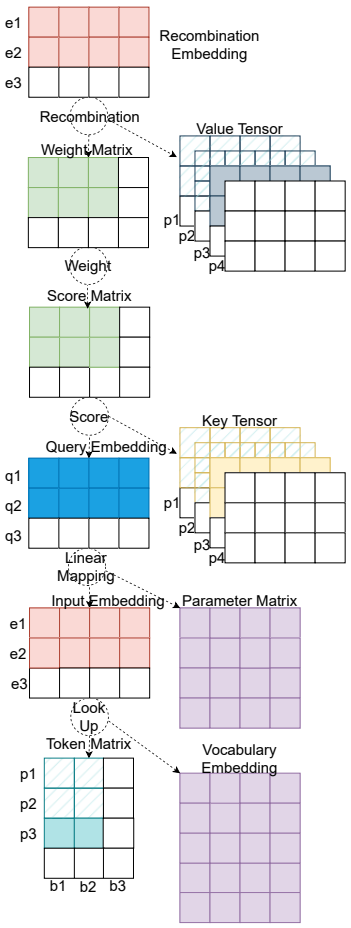

Figure 5. Permanent Graph

corresponding tensors in the computation graph must also change accordingly. As a result, a single pre-constructed computation graph cannot be universally reused, necessitating dynamic graph construction. In Transformers, aside from the fixed parameter matrices, the primary sources of shape variation come from two types of matrices: **embedding matrices** and **attention matrices**.

- **Embedding Matrices:**
  - Row size corresponds to the sequence length, while the column size remains fixed.
  - They have consistent strides, meaning their element offsets scale linearly with token positions.
- **Attention Matrices:**
  - Both row and column sizes depend on the sequence length.
  - Unlike embedding matrices, their strides vary, causing the same coordinates in different matrices to have different offsets.

Figure 4 illustrates attention matrices, where the position indices range from 1 to 3. The KV tensor is projected along the head size dimension; the head size dimension is not visible. The column sizes of these matrices correspondingly range from 1 to 3. Despite these variations, all matrices can fit within the largest (last) matrix. However, due to differing strides, mapping coordinates across these matrices becomes non-trivial.

To handle variable-shaped inputs, the framework employs a dimension-first data layout to accommodate tensors with varied shapes. The tensor shapes are designed to accommodate the maximum batch size and sequence length, ensuring that all data can fit within the allocated maximum tensors during inference. We propose **dimension-first layouts**, a novel layout that ensures element offsets scale consistently with token positions for varied-length sequences. Despite changes in tensor shapes, the memory layout (stride) remains the same, offering a scalable approach for handling sequences of different lengths. Traditional tensors represent a special case of dimension-first tensors. Each matrix begins at the start of the maximum-sized matrix. Each vector begins at the start of its respective storage space within the maximum-sized vector. For vectors smaller than the maximum size, gaps may exist between consecutive vectors, leaving unoccupied memory spaces. This trade-off ensures stride uniformity and scalability for varied-shape tensors.

Embedding tensors retain their traditional layout, while KV and score tensors leverage the dimension-first design to store data across their dimensions. Figure 5 provides a snapshot of tensor usage within a Transformer block during inference:

- **Setup:** Maximum sequence length = 4, maximum batch size = 3, and hidden size = 4. Input sequence length = 3, and batch size = 2.
- **Tensor Dimensions:**
  - **Input Token Matrix:** Dimensions: [sequence, batch], maximum shape = 5×3. Predicted tokens for positions 1 and 2 are cached, and position 3 is under prediction.
  - **Embedding Matrix:** Dimensions: [batch, hidden], maximum shape = 3×4, current shape = 2×4.
  - **KV Cache:** Dimensions: [sequence, batch, hidden], maximum shape = $4 \times 2 \times 4$. Cached embeddings for positions 1 and 2, with position 3 currently being cached.
  - **Score Matrix and Weight Matrix:** Dimensions: [batch, sequence], maximum shape = 3×5, current shape = 2×5. The first two rows are filled, each with three columns corresponding to completed attention computations.

5.1.2 Graph Compilation.

The framework constructs a elastic computational graph for the longest sequence length. Figure 6 illustrates this process, where the blue squares represent allocated tensors, indicating their permanent status. The compiler prepares computation by traversing the forward computational graph in topological order, allocating large contiguous memory blocks for all tensors, particularly

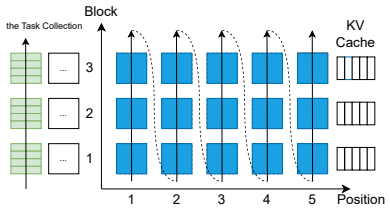

Figure 6. Elastic Graph Construction

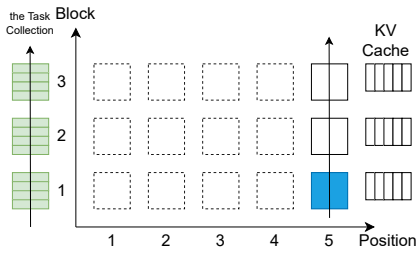

Figure 7. Reusing Graph Construction

the KV cache. It assumes that a single batch of data is processed at any given time. Input embeddings are allocated with dimensions [batch, hidden] for position-wise decoding, while other tensor dimensions are adjusted accordingly. The compilation process consists of five key steps:

a) Define the maximum batch size and sequence length based on available memory capacity;
b) Perform a forward pass to visit each node;
c) Allocate static memory for tensors using their maximum shapes;
d) Split tensors into chunks;
e) Place chunk offsets (memory address) into address frames;
f) Package frames with arithmetic functions as chunked operator;
g) Collect all chunked operators into a topologically sorted task collection.

Shape operators execute directly since they do not modify tensor data. Conversely, data operators alter both tensor shapes and values, requiring tensors to be divided into smaller chunks. With fixed memory addresses, the framework precomputes tensor chunk offsets and partitions tensors into chunks. Tensors are initially split into matrices, which are further divided into rows, storing coordinates into address frames . For matrix multiplication, matrices are partitioned into smaller blocks, with block coordinates (memory addresses) stored for execution.

Each chunk is paired with its corresponding arithmetic function to form tasks, encapsulated within a **chunked operator**. Different operators generate chunked operators with distinct shapes and data access patterns. The compiler collects all chunked operators into a **task collection**, representing a topologically ordered computational graph. This task collection serves as the foundation for efficient runtime execution.

5.1.3 Tensor Reuse. Permanent allocation of all tensors and tasks can quickly deplete terabytes of memory , posing a significant scalability challenge for long sequences. To overcome these limitations, we introduce tensor reuse along two key dimensions: **position** and **block**. Instead of expanding all nodes in the graph, the framework focuses on expanding only the tensors required for the *last positional graph* and reuses the first Transformer block across all others. Figure 7 illustrates this process, highlighting the memory-efficient traversal of the last positional graph.

a) **Reusing Position-based Tensors:** Since the shapes of embedding tensors at different positions remain consistent (given preset maximum batch size and sequence length), the framework reuses tensors across positions. Although KV and attention tensors have varying shapes at each position, their sequence dimension sizes increase progressively with position. The largest tensors at the last position can therefore accommodate all prior positions, obviating the need to allocate separate memory for intermediate positions.
b) **Reusing Block-based Tensors:** Excluding parameter tensors and the KV cache, all Transformer blocks share the same tensor shapes. This uniformity allows the framework to construct only the first Transformer block and reuse it for all subsequent blocks. The compute cores index tasks within the collection based on the position, ensuring efficient execution. *Embedding computation* is independent of position, making it straightforward to reuse tensors. *Attention computation*, which spans from the beginning of the sequence to the current position

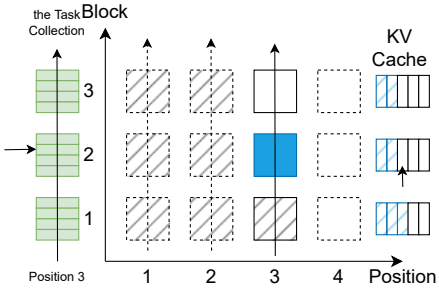

Figure 8. Prediction Process

5.1.4 Prediction. The runtime processes incoming batch data, structured as [sequence, batch], using a FIFO scheduling strategy. Batches are processed sequentially, ensuring that each batch starts only after the previous one completes. During inference, the framework predicts tokens *position-wise*, handling both prompt tokens and unknown tokens.

Prompt tokens do not require prediction, whereas unknown tokens must be predicted.

Each thread iterates through the precompiled **global task collection**. Figure 8 illustrates the prediction process. Once all tasks for a chunked operator are completed, the thread proceeds to the next operator, executing tasks in five steps:

a) **Task retrieval:** Fetch a chunked operator from the task collection.
b) **Task assignment:** Assign tasks from the chunked operator based on batch size and position.
c) **Core dispatch:** Distribute tasks evenly among compute cores according to thread ID.
d) **Task execution:** Independently execute assigned tasks on compute cores.
e) **Thread synchronization:** Synchronize with other threads via a barrier to maintain consistency.

5.2 Attention Per Core

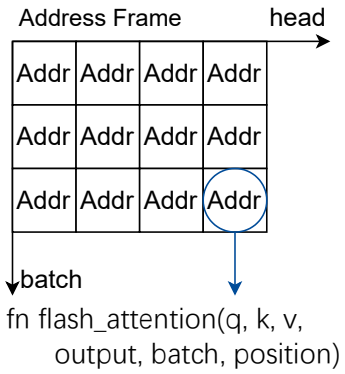

Figure 9. Attention Process

The LLaMA 3 70B model employs Grouped Query Attention (GQA) [14] with 64 query heads and 8 shared KV heads. Figure 9 illustrates the attention process. eLLM utilizes the FlashAttention algorithm for attention computation, which can be broken down into three key aspects: For **data storage**, the system maintains only 8 shared KV heads instead of expanding them into 64 separate KV heads, significantly reducing memory consumption. For **data movement**, the system loads the 8 shared KV heads into the L3 cache only once. Subsequent accesses are efficiently served from the L3 cache, minimizing redundant memory I/O. For **data computation**, each attention head's computation is executed independently on separate CPU cores, with all 64 query heads processed in parallel.

The computation follows a position-wise approach, leveraging SIMD (Single Instruction, Multiple Data) operations to accelerate vector multiplications. All intermediate results, particularly the accumulated output vectors, are stored within CPU registers, avoiding unnecessary memory accesses. Only the input query (q), key (k), value (v), and the final output vectors involve memory I/O. Since the KV cache is structured as a multi-dimensional tensor, the memory address of the next row can be calculated using its coordinates and strides. By utilizing cache prefetch instructions, the CPU asynchronously loads the next row of key and value data, significantly reducing cache misses and establishing a contiguous data pipeline.

6 Experiment

We are currently seeking high-performance machines to conduct our experiments.

7 Analysis

Contrary to mainstream belief, this study argues that CPUs are better suited for long-sequence inference than GPUs. We analyze how the CPU-based eLLM framework achieves superior performance and manage memory efficiently.

7.1 Latency

$$t = e + \frac{(p+1)}{2} \cdot a$$

The average positional latency t consists of:

- e: embedding time (a constant)
- a: attention time per position (a constant)
- p: position (a variable)

The relationship between average positional latency and position (i.e., sequence length) is theoretically linear, as described by the formula above. Our experimental results show that CPU curve follow this linear trend and align well with theoretical expectations.

In contrast, when memory is sufficient, the GPU-based baseline exhibits a roughly linear latency trend. However, its attention computation is slower than when using fully contiguous GPU memory—and even slower compared to the CPU. When batch sizes are adjusted so that the cost per user is equal between GPU and CPU systems, the GPU initially outperforms the CPU. Yet as the sequence length increases, the CPU eventually surpasses the GPU.

When GPU memory becomes insufficient, latency rises sharply. This is due to the need to either recompute KV values or offload KV chunks to CPU memory, then reload them into GPU memory over PCIe during inference. Since PCIe bandwidth is an order of magnitude lower than CPU memory bandwidth, this severely impacts performance. Even under equal batch sizes, the CPU ultimately outperforms the GPU as the sequence length grows. Since we do not have full control over the deployment environment, we are unable to perform detailed profiling to fully explain the long-sequence latency curve.

Nonetheless, we identify five key factors that likely contribute to the high latency observed in existing frameworks: ❶ Costly Graph Execution, ❷ Chunked KV Cache, ❸ Costly

Memory Management, ❹ Limited Parallelism, ❺ Redundant Memory I/O.

7.1.1 Costly Graph Execution. The impact of CPU on LLM performance is significant. GPT is often considered a CPU-bound model [31] for GPU inference, where graph compilation and execution workloads primarily occupy CPU cores. This bottleneck degrades overall system performance.

Graph compilation involves a sequence of tightly coupled steps, including computing tensor shapes and strides, allocating memory, chunking tensors, and mapping these chunks to arithmetic operations. To minimize idle compute resources, task generation and execution must be organized as an asynchronous pipeline. These steps impose significant overhead on the CPU.

Graph execution further exacerbates the GPU performance, as the CPU is responsible for launching numerous GPU kernels—particularly for attention and MLP layers. CPU-side operators trigger GPU kernel execution. The CPU becomes a bottleneck due to the overhead of issuing and queuing kernel launches. At small batch sizes, GPUs often remain idle while waiting for CPU instructions.

7.1.2 Chunked KV Cache. vLLM adopts a chunked key-value (KV) cache design. Chunk management is handled by the CPU, which maintains a global chunk table and pre-allocates sufficient chunk memory. As new key and value vectors are generated during decoding, the CPU assigns them to available chunks and updates the chunk-to-address mapping table. This table must be synchronized to the GPU before each attention computation.

Each KV chunk typically stores 128 tokens (i.e., 128 rows of key/value vectors), amounting to tens of kilobytes per block. The GPU loads these blocks into shared memory in preparation for attention. However, shared memory is highly constrained (e.g., 100–160 kB per SM), so KV data cannot all reside in shared memory at once. Instead, chunks must be loaded in rotating windows, demanding precise coordination between memory access and compute to maintain pipeline efficiency.

Before data loading, the GPU must first resolve logical page indices to physical addresses via the mapping table, introducing additional indirection latency. Then, threads explicitly fetch and stage the chunked KV data into shared memory. Compute can only begin once the full required chunk is resident, which adds at least one global memory latency cycle. Since the final chunk is often only partially filled (e.g., <128 tokens), compute units still wait for the entire block to be loaded, resulting in underutilization.

The latency of vLLM's paged decode kernel is highly sensitive to chunk size. [23] This is because the chunk size directly impacts:

- the number of page lookups per token (i.e., jump frequency),
- the sparsity and locality of physical memory addresses,
- shared-memory utilization efficiency,
- warp-level load balancing, and
- the effective L2 cache hit rate.

Smaller chunks lead to more fragmented KV access, increasing synchronization and cache pressure; larger chunks reduce fragmentation but increase wasted memory and underutilization in boundary cases.

Attention computation for a single head is decomposed into many small operations across disjoint KV pages. This fragmentation introduces overhead from frequent thread scheduling, shared-memory contention, and synchronization barriers between warps and stages.

In contrast, CPUs benefit from tightly integrated caches and wide register files that allow both intermediate attention state and KV vectors to be retained in fast-access memory. Moreover, CPUs leverage hardware-managed, row-wise cacheline prefetching, which reduces the need for explicit data movement and improves asynchronous dataflow. These architectural traits give CPUs an advantage in terms of fine-grained scheduling, pipelining, and latency hiding for long sequence inference.

7.1.3 Costly Memory Management. Managing massive KV chunks and tensors incurs substantial overhead, which escalates sharply as memory resources become depleted. GPU memory management is inherently dependent on the host CPU and, to some extent, constrained by CPU memory bandwidth. The host CPU is responsible for issuing commands related to memory allocation, deallocation, and merging, while the GPU merely executes these instructions.

Memory management consists of two components: algorithms and execution. Complex memory allocation algorithms operate entirely on the host CPU. Since dynamic memory allocation often relies on greedy heuristics [35], it fails to achieve optimal memory utilization. As memory capacity dwindles, the overhead associated with memory reclamation and consolidation increases dramatically, further throttling performance. If the memory is not well prepared, the GPU computation may stall.

7.1.4 Limited Parallelism. FlashAttention must process each block sequentially along the sequence dimension because the output of the online softmax depends on the attention scores of the previous token. This sequential dependency results in a low degree of parallelism, which limits the efficiency of the GPU's many-core architecture. In contrast, CPUs can fully utilize all cores and achieve near-peak memory bandwidth.

For each attention head, the corresponding KV vectors must be loaded into shared memory, and the entire computation must be executed within a single Streaming Multiprocessor (SM), where CUDA cores share a common shared memory. On GPUs, parallelism is primarily achieved along

the batch and head dimensions. However, multiple heads are often grouped into a single chunk—e.g., in the default configuration of vLLM, eight heads are packed into one chunk—which must be loaded in its entirety. To fully utilize all SMs (e.g., 108 SMs on A100), the batch size must be increased significantly. Yet, long-sequence inputs inherently limit the batch size due to memory and latency constraints. This trade-off prevents full utilization of available SMs and, consequently, the full GPU memory bandwidth.

7.1.5 Redundant Memory I/O. The NUMA architecture of GPUs introduces redundant memory I/O, which must be transferred via inter-GPU links or PCIe. Both inter-GPU bandwidth and PCIe bandwidth are an order of magnitude lower than CPU memory bandwidth, leading to increased latency.

Tensor parallelism partitions large matrix multiplications across multiple GPUs. During each prediction step, the matrix A must be copied to four GPUs, and the resulting submatrices C must be transferred into their respective KV cache memory chunks. These submatrices may be stored in local GPU memory or even offloaded to CPU memory [2]. The necessity of frequent data transfers across different memory regions significantly impacts performance, limiting the efficiency of computations.

7.2 Memory Usage

CPUs facilitate long-sequence inference due to their large memory capacity and the high memory efficiency of eLLM. The available memory determines the maximum sequence length that can be supported for KV cache allocation. While dynamic memory allocation inherently results in some memory waste, static memory allocation largely avoids this issue. Although static allocation may slightly increase memory consumption for intermediate tensors and task-related data structures, the only inefficiency occurs when processing short sequences, in which case part of the KV cache remains unused. However, for long-sequence inference, KV cache utilization is comparable between dynamic and static allocation strategies.

Static memory allocation eliminates memory fragmentation by preallocating space for all necessary data structures, thereby minimizing redundancy. Apart from the memory used for model parameters, intermediate tensors and task-related data structures consume minimal space, enabling the vast majority of available memory to be allocated to the KV cache. We analyze memory usage under three main components:

- **Intermediate Tensors:** Require a fixed memory allocation corresponding to the tensors used in each transformer block and the final prediction layer.
- **Task Data Structures:** Only need to represent the computational graph for a single token position.
- **KV Cache:** Must be allocated in advance based on the maximum supported sequence length and batch size.

By contrast, dynamic memory allocation inevitably leads to fragmentation and memory waste. We analyze dynamic allocation in the same three categories:

- **Intermediate Tensors and Task Data Structures:** Require significantly more memory, as each operator needs its own allocation. Additionally, multiple operators may execute in parallel or shortly thereafter, requiring further memory reservations.
- **KV Cache:** Must be dynamically allocated based on the sequence length and batch size at runtime.

Since intermediate tensors and task-related structures are allocated based on the maximum batch size and do not scale with sequence length, static allocation provides predictable and efficient memory usage. In contrast, dynamic allocation causes the memory footprint of intermediate tensors and task structures to grow linearly with batch size, though still independent of sequence length.

Based on our analysis of eLLM's memory design, it is clear that only the KV cache grows with sequence length, while the memory footprint of other tensors and task structures remains constant. This design enables eLLM to efficiently support million-token-scale sequences. For example, with a batch size of 1 and a sequence length of 1 million tokens, the KV cache alone consumes 327.7 GB of memory. Notably, the KV cache required by a single user in this case exceeds the total memory capacity of four A100 GPUs combined.

8 Benefits

eLLM enables affordable long-sequence inference—scaling to millions of tokens—on CPU-only servers, with latency that eventually surpasses GPU-based systems. This approach democratizes access to LLM inference, as CPU-only servers are widely available, and DDR memory upgrades are straightforward and cost-effective.

eLLM is particularly well-suited for non-production and lower-priority tasks such as model evaluation, data enrichment, and asynchronous workloads, much like OpenAI's Flex processing. Moreover, eLLM presents a compelling solution for online inference. Future work will extend eLLM to multi-CPU systems (e.g., 4×Xeon CPUs).

Finally, eLLM is also ideal for on-device inference with small models. It delivers efficient inference for both short and long texts without requiring GPU support. In AI PCs and AI Phones—where typically only a single user is present and the batch size is fixed at one—inference workloads transition from compute-bound to memory-bound, aligning well with CPU's design strengths.

9 Related Work

While general machine learning frameworks [32] provide programming patterns for machine learning algorithms, existing solutions [1] [22] [20] [6] are inefficient for LLM inference. As a result, researchers have abandoned large machine learning frameworks and developed dedicated LLM inference frameworks from scratch, such as LLama.cpp [9], vLLM [14], and TensorRT-LLM [19], which focus on accelerating and optimizing inference performance for the latest LLMs. However, existing frameworks [15] [4] [26] [3] rely on temporary computation graphs, which introduce inefficiencies. They try to improve efficiencies by the following two key optimization techniques.

9.1 Computation Graph Optimization

LLMs have complex computational graphs that contain numerous operations and parameters. Compilers [17] [25] [5] optimize these graphs to improve efficiency, reduce redundant computations, and eliminate unnecessary operations.

- **Operator Fusion:** Combines consecutive operations into a single fused operation, reducing memory access and improving efficiency. *Example:* A Silu followed by elementwise multiplciation can be fused to reduce memory reads and writes, speeding up computation.
- **Constant Folding:** Computes constant expressions at compile time to avoid redundant calculations during inference. *Example:* Fixed weights or biases can be computed during compilation rather than at runtime.
- **Operator Reordering:** Changes the order of operations to enhance computational efficiency, often to better leverage parallelism. *Example:* Reordering matrix multiplications in a transformer model to optimize parallel hardware utilization.
- **Subgraph Extraction and Reuse:** Reuses subgraphs (such as layers or blocks) that appear multiple times, eliminating redundant computations. In particular, CUDA Graphs leverage **static memory allocation** to construct computational graphs. Once launched, the entire graph executes without requiring further CPU interaction. However, CUDA Graphs require fixed input shapes, making them unsuitable for varied input shapes. *Example:* vLLM reuses precompiled CUDA Graphs during the generation phase by matching them to the corresponding batch size. If a different batch size is encountered, a new computational graph must be created.

9.2 Dynamic Shape Optimization

Dynamic shapes, which result from variable-length input tokens, cause changes in tensor shapes during computation. Compilers [36] [39] [28] [37] optimize tensors with dynamic shapes through several strategies:

a) **Shape Inference and Propagation** Compilers use shape inference to deduce tensor sizes throughout the computation graph based on operations applied to them.
   - **Shape Propagation:** Known input shapes are propagated through the graph to infer the shapes of intermediate tensors, aiding memory allocation and optimization decisions. *Example:* In matrix multiplication, the compiler infers the resulting tensor shape and adjusts memory allocation accordingly.
   - **Partial Shape Information:** For partially dynamic shapes (e.g., dynamic batch size), the compiler uses placeholders for unknown dimensions, which are refined at runtime. *Example:* Variable-length sequences may have a dynamic sequence length while keeping the batch size fixed, with partial shapes inferred at compile time.

b) **Tensor Reshaping and Layout Optimization** Dynamic shapes may require reshaping and layout adjustments to optimize performance.
   - **Shape Transformation (Reshaping):** The compiler inserts reshaping operations to ensure tensors are in the correct shape for computations. *Example:* If an operation expects a 2D tensor but receives a 3D tensor, reshaping ensures the proper shape is used.
   - **Padding and Stride Optimization:** Padding ensures tensor dimensions align with hardware requirements, preventing memory inefficiencies. *Example:* In NLP models with varying sequence lengths, padding is used to align sequence lengths, optimizing matrix multiplication.

c) **Lazy Evaluation and Just-in-Time (JIT) Compilation** Lazy evaluation and JIT compilation allow optimizations to be made based on actual tensor shapes at runtime.
   - **Deferred Computation:** The compiler delays optimizations until the actual tensor shapes are known, ensuring shape-specific optimizations are applied only when needed.
   - **JIT Compilation for Dynamic Shapes:** The compiler generates optimized code at runtime, adapting to different input sizes while maintaining performance.

10 Conclusion

CPU-only and GPU computers differ significantly in both performance and cost. However, by normalizing batch size such that each user receives a similar share of memory bandwidth and cost, a fair comparison becomes feasible. Under these conditions, CPU-only systems not only support longer sequence lengths but also deliver faster inference. CPUs can fully utilize their peak memory bandwidth through the elastic computation graph, enabling contiguous memory access. In contrast, GPUs suffer from memory bandwidth degradation caused by dynamic computation graphs and chunked

KV cache. Despite their high hardware costs, GPU cores remain underutilized during inference workloads. Furthermore, GPUs lack sufficient memory capacity to store large KV caches without sacrificing memory bandwidth. Their reliance on host CPUs for graph compilation and task execution imposes additional bottlenecks. By removing all GPU cards, a GPU-free computer powered by the eLLM framework becomes the optimal solution for long-sequence LLM inference.

Acknowledgments

We gratefully acknowledge the part-time student helpers Peiduo Liu, Zhanzhi Lin, and Yuhang Dai from The Hong Kong Polytechnic University for their invaluable contributions to the project's software implementation.

References

[1] Martin Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). 2016, pp. 265–283. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi (visited on 12/20/2024).

[2] Reza Yazdani Aminabadi et al. *DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale*. June 30, 2022. DOI: 10.48550/arXiv.2207.00032. arXiv: 2207.00032[cs]. URL: http://arxiv.org/abs/2207.00032 (visited on 02/28/2025).

[3] Tianqi Chen et al. *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*. tex.pubstate: preprint. Oct. 5, 2018. arXiv: 1802.04799[cs]. URL: http://arxiv.org/abs/1802.04799 (visited on 12/04/2023).

[4] Scott Cyphers et al. *Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning*. Jan. 30, 2018. DOI: 10.48550/arXiv.1801.08058. arXiv: 1801.08058[cs]. URL: http://arxiv.org/abs/1801.08058 (visited on 12/16/2024).

[5] Tri Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. Number: arXiv:2205.14135. June 2022. arXiv: 2205.14135[cs]. (Visited on 04/18/2024).

[6] Alexander Demidovskij et al. "OpenVINO Deep Learning Workbench: Comprehensive Analysis and Tuning of Neural Networks Inference". In: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. Place: Seoul, Korea (South) tex.eventtitle: 2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW). IEEE, Oct. 2019, pp. 783–787. ISBN: 978-1-7281-5023-9. DOI: 10.1109/ICCVW.2019.00104. URL: https://ieeexplore.ieee.org/document/9022274/ (visited on 11/20/2023).

[7] Jiayu Ding et al. *LongNet: Scaling Transformers to 1,000,000,000 Tokens*. July 19, 2023. DOI: 10.48550/arXiv.2307.02486. arXiv: 2307.02486[cs]. URL: http://arxiv.org/abs/2307.02486 (visited on 06/17/2025).

[8] Yao Fu. *Challenges in Deploying Long-Context Transformers: A Theoretical Peak Performance Analysis*. May 14, 2024. DOI: 10.48550/arXiv.2405.08944. arXiv: 2405.08944[cs]. URL: http://arxiv.org/abs/2405.08944 (visited on 06/25/2024).

[9] *Ggerganov/llama.cpp: LLM inference in C/C++*. URL: https://github.com/ggerganov/llama.cpp (visited on 04/19/2024).

[10] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. Nov. 23, 2024. DOI: 10.48550/arXiv.2407.21783. arXiv: 2407.21783[cs]. URL: http://arxiv.org/abs/2407.21783 (visited on 05/06/2025).

[11] Yizeng Han et al. *Dynamic Neural Networks: A Survey*. Dec. 2, 2021. DOI: 10.48550/arXiv.2102.04906. arXiv: 2102.04906[cs]. URL: http://arxiv.org/abs/2102.04906 (visited on 12/10/2024).

[12] Joel Hestness, Stephen W. Keckler, and David A. Wood. "A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior". In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 2014 IEEE International Symposium on Workload Characterization (IISWC). Raleigh, NC, USA: IEEE, Oct. 2014, pp. 150–160. ISBN: 978-1-4799-6454-3 978-1-4799-6452-9. DOI: 10.1109/IISWC.2014.6983054. URL: http://ieeexplore.ieee.org/document/6983054/ (visited on 12/14/2024).

[13] Yunho Jin et al. "S3: increasing GPU utilization during generative inference for higher throughput". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., Dec. 10, 2023, pp. 18015–18027. (Visited on 06/17/2025).

[14] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. Issue: arXiv:2309.06180 tex.pubstate: preprint. Sept. 12, 2023. arXiv: 2309.06180[cs]. URL: http://arxiv.org/abs/2309.06180 (visited on 11/20/2023).

[15] Chris Lattner et al. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. tex.pubstate: preprint. Feb. 29, 2020. arXiv: 2002.11054[cs]. URL: http://arxiv.org/abs/2002.11054 (visited on 12/04/2023).

[16] Jinhao Li et al. *Large Language Model Inference Acceleration: A Comprehensive Hardware Perspective*. Oct. 14, 2024. DOI: 10.48550/arXiv.2410.04466. arXiv: 2410.04466[cs]. URL: http://arxiv.org/abs/2410.04466 (visited on 12/09/2024).

[17] Mingzhen Li et al. "The Deep Learning Compiler: A Comprehensive Survey". In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (Mar. 1, 2021), pp. 708–727. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2020.3030548. arXiv: 2002.03794[cs]. URL: http://arxiv.org/abs/2002.03794 (visited on 11/20/2023).

[18] Sambit Mishra et al. "Impact of Memory Bandwidth on the Performance of Accelerators". In: *Practice and Experience in Advanced Research Computing 2024: Human Powered Computing*. PEARC '24. New York, NY, USA: Association for Computing Machinery, July 17, 2024, pp. 1–9. ISBN: 979-8-4007-0419-2. DOI: 10.1145/3626203.3670540. URL: https://dl.acm.org/doi/10.1145/3626203.3670540 (visited on 12/09/2024).

[19] Nvidia. *Nvidia/TENSORRT: NVIDIA® TENSORRTTM, an SDK for high-performance deep learning inference, includes a Deep Learning Inference Optimizer and runtime that delivers low latency and high throughput for inference applications*. URL: https://github.com/NVIDIA/TensorRT.

[20] Onnx. *Onnx/onnx: Open standard for machine learning interoperability*. URL: https://github.com/onnx/onnx.

[21] OpenAI et al. *GPT-4 Technical Report*. Mar. 4, 2024. DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774[cs]. URL: http://arxiv.org/abs/2303.08774 (visited on 12/14/2024).

[22] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. tex.pubstate: preprint. Dec. 3, 2019. arXiv: 1912.01703[cs, stat]. URL: http://arxiv.org/abs/1912.01703 (visited on 12/04/2023).

[23] Ramya Prabhu et al. *vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention*. Jan. 29, 2025. DOI: 10.48550/arXiv.2405.04437. arXiv: 2405.04437[cs]. URL: http://arxiv.org/abs/2405.04437 (visited on 06/25/2025).

[24] Alexander Ratner et al. *MLSys: The New Frontier of Machine Learning Systems*. Dec. 1, 2019. DOI: 10.48550/arXiv.1904.03257. arXiv: 1904.03257[cs]. URL: http://arxiv.org/abs/1904.03257 (visited on 12/20/2024).

[25] Jared Roesch et al. *Relay: A High-Level Compiler for Deep Learning*. Aug. 24, 2019. DOI: 10.48550/arXiv.1904.08368. arXiv: 1904.08368[cs]. URL: http://arxiv.org/abs/1904.08368 (visited on 12/18/2024).

[26] Amit Sabne. *XLA : Compiling machine learning for peak performance*. 2020.

[27] Noam Shazeer. *Fast Transformer Decoding: One Write-Head is All You Need*. Number: arXiv:1911.02150. Nov. 2019. arXiv: 1911.02150[cs]. (Visited on 04/18/2024).

[28] Haichen Shen et al. "Nimble: Efficiently Compiling Dynamic Neural Networks for Model Inference". In: *Proceedings of Machine Learning and Systems* 3 (Mar. 15, 2021), pp. 208–222. (Visited on 06/18/2025).

[29] vLLM Team. *vLLM v0.6.0: 2.7x Throughput Improvement and 5x Latency Reduction*. vLLM Blog. Sept. 5, 2024. URL: https://blog.vllm.ai/2024/09/05/perf-update.html (visited on 12/18/2024).

[30] Ashish Vaswani et al. *Attention Is All You Need*. Aug. 2, 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762[cs]. URL: http://arxiv.org/abs/1706.03762 (visited on 12/09/2024).

[31] Prabhu Vellaisamy et al. *Characterizing and Optimizing LLM Inference Workloads on CPU-GPU Coupled Architectures*. arXiv.org. Apr. 16, 2025. URL: https://arxiv.org/abs/2504.11750v1 (visited on 06/12/2025).

[32] Yaguang Huangfu et al. "MatrixMap: Programming Abstraction and Implementation of Matrix Computation for Big Data Applications". In: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. Place: Melbourne, VIC tex.eventtitle: 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS). IEEE, Dec. 2015, pp. 19–28. ISBN: 978-0-7695-5785-4. DOI: 10.1109/ICPADS.2015.11. URL: http://ieeexplore.ieee.org/document/7384274/ (visited on 11/20/2023).

[33] Renchao Yu. "Application of CPU in AI and Machine Learning". In: *Proceedings of the International Conference on Decision Science & Management*. ICDSM '24. New York, NY, USA: Association for Computing Machinery, Nov. 18, 2024, pp. 221–224. ISBN: 979-8-4007-1815-1. DOI: 10.1145/3686081.3686118. URL: https://doi.org/10.1145/3686081.3686118 (visited on 12/09/2024).

[34] Zhihang Yuan et al. *LLM Inference Unveiled: Survey and Roofline Model Insights*. May 1, 2024. DOI: 10.48550/arXiv.2402.16363. arXiv: 2402.16363[cs]. URL: http://arxiv.org/abs/2402.16363 (visited on 12/13/2024).

[35] Junzhe Zhang et al. *Efficient Memory Management for GPU-based Deep Learning Systems*. Feb. 19, 2019. DOI: 10.48550/arXiv.1903.06631. arXiv: 1903.06631[cs]. URL: http://arxiv.org/abs/1903.06631 (visited on 02/28/2025).

[36] Bojian Zheng et al. "DietCode: Automatic Optimization for Dynamic Tensor Programs". In: ().

[37] Zhen Zheng et al. "BladeDISC: Optimizing Dynamic Shape Machine Learning Workloads via Compiler Approach". In: *Proceedings of the ACM on Management of Data* 1.3 (Nov. 13, 2023), pp. 1–29. ISSN: 2836-6573. DOI: 10.1145/3617327. URL: https://dl.acm.org/doi/10.1145/3617327 (visited on 12/18/2024).

[38] Zixuan Zhou et al. *A Survey on Efficient Inference for Large Language Models*. July 19, 2024. DOI: 10.48550/arXiv.2404.14294. arXiv: 2404.14294[cs]. URL: http://arxiv.org/abs/2404.14294 (visited on 12/10/2024).

[39] Kai Zhu et al. *DISC: A Dynamic Shape Compiler for Machine Learning Workloads*. tex.pubstate: preprint. Nov. 23, 2021. arXiv: 2103.05288[cs]. URL: http://arxiv.org/abs/2103.05288 (visited on 12/04/2023).