GLINTHAWK: A Two-Tiered Architecture for High-Throughput LLM Inference

Pouya Hamadanian
MIT

Sadjad Fouladi
Microsoft Research

Abstract

Large Language Models (LLM) have revolutionized natural language processing, but their inference demands substantial resources, while under-utilizing high-end accelerators like GPUs. A major bottleneck arises from the attention mechanism, which requires storing large key-value caches, limiting the maximum achievable throughput way below the available computing resources. Current approaches attempt to mitigate this issue through memory-efficient attention and paging mechanisms, but remained constrained by the assumption that all operations must be performed on high-end accelerators.

In this work, we propose Glinthawk, a two-tiered architecture that decouples the attention mechanism from the rest of the Transformer model. This approach allows the memory requirements for attention to scale independently, enabling larger batch sizes and more efficient use of the high-end accelerators. We prototype Glinthawk with NVIDIA T4 GPUs as one tier and standard CPU VMs as the other. Compared to a traditional single-tier setup, it improves throughput by 5.9× and reduces cost of generation by 2.8×. For longer sequence lengths, it achieves 16.3× throughput improvement at 2.4× less cost. Our evaluation shows that this architecture can tolerate moderate network latency with minimal performance degradation, making it highly effective for latency-tolerant, throughput-oriented applications such as batch processing. We shared our prototype publicly at https://github.com/microsoft/glinthawk.

1 Introduction

Large Language Models (LLMs), a class of deep neural networks characterized by their underlying *Transformer* architecture [49], have found remarkable success in natural language processing tasks [39]. LLMs predict probability distributions over sequences of sub-words called “tokens” and operate autoregressively. At each step, the model looks at all previous tokens to predict the next token, generating text one token at a time.

Serving LLMs requires top-of-the-line accelerators (e.g., GPUs) and interconnects (e.g., InfiniBand). Due to the high costs of such infrastructure, there is a strong interest in maximizing the number of *tokens* processed per second, i.e., the inference throughput, to spread the cost across many tokens [25, 26, 38, 43, 52]. This is particularly useful for throughput-oriented application [43] such as mass document processing [22], model benchmarking [54], and data cleaning [33]. Providers also offer services tailored to these use cases [36] at a discount compared to interactive sessions like chatbots. This work focuses on improving throughput for such throughput-oriented, latency-tolerant applications.

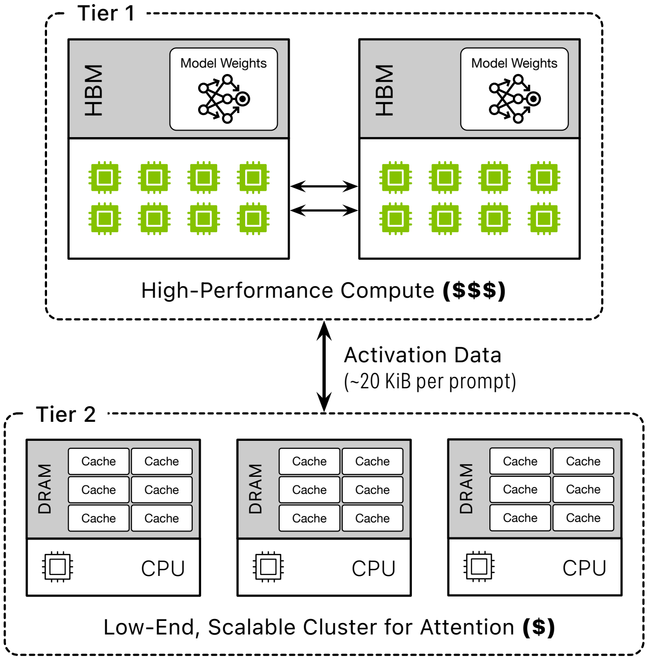


Figure 1: Glinthawk dissects attention memory and compute to a second cluster of low-end nodes, and improves the utilization of costly GPUs working on compute-heavy operations, improving the total system throughput and reducing inference costs.

The key challenge with LLM inference is low overall utilization of compute resources, sometimes as low as 1% [38]. For instance, in LLaMA2-70B—an open-weight LLM from Meta [48]—in one stage, an 896 MiB weight matrix must be multiplied by a 16 KiB input to compute a 112 KiB output for a *single* token. This operation is bottlenecked by the rate at which the model weight can be loaded into the compute cores, leaving compute resources heavily underutilized, as each loaded weight participates in only one multiply-accumulate operation.

A common technique to boost utilization is *batching*, where multiple tokens from multiple prompts are processed together to amortize the cost of loading model weights [26, 38, 43]. For instance, processing 256 prompts of the aforementioned LLaMA2-70B model leads to 64× higher throughput compared to processing only one at a time.

However, the degree of batching is highly constrained by GPU memory capacity. This is due to the *attention* mechanism—a key step in the computation graph of the Transformer architecture—which requires storing information about previously processed tokens in the form of a cache. For LLaMA2-70B, a single prompt with 32K tokens needs a 10 GiB cache, rivaling the model weights in size. An accelerator with 80 GiB of memory would be limited to a batch size of 8, while other non-attention operations can benefit from scaling to batch sizes in the *hundreds*.

Several efforts have been made to reduce attention memory requirements, but none eliminate this constraint. One line of work focuses on reducing attention's memory footprint per prompt [13, 26, 42, 51]. Others suggest paging the attention cache in and out to secondary storage, such as host's DRAM [14, 26] or SSDs [43], but are severely limited by slow GPU-CPU interconnects, such as PCIe. These state-of-the-art inference engines are built with an unquestioned assumption: all computation, including attention, must be done on the high-end accelerator.

In this paper, we argue that attention is fundamentally different from other operations in the Transformer architecture: it is relatively compute-light, embarrassingly parallel, and the only stateful operation in the computation graph. Leveraging these insights, we propose a novel inference architecture that fully dissects attention from the rest of the model; storing the cache *and* performing attention computation at a second set of low-end compute machinery. This technique allows attention subsystem to trivially scale, increasing possible batch sizes by multiple orders of magnitude. In our design, Glinthawk[1], high-performance accelerators ("Tier 1") handle the core model computations involving the model weights, while a set of independent, lower-end compute nodes ("Tier 2") manage the attention mechanism.

To demonstrate the feasibility of this approach, we implement and evaluate one possible realization of Glinthawk on commodity cloud infrastructure, with NVIDIA T4 GPUs as Tier-1 accelerators, and CPU-based virtual machines with inexpensive DRAM as the Tier-2 nodes (Figure 1). These two subsystems communicate with each other over Ethernet links. This prototype achieves a 5.9× increase in throughput and 2.8× reduction in cost compared to a traditional architecture with only the Tier-1 accelerators (details provided in §6).

This prototype has modest networking requirements and needs less than 50 Gbps of inter-tier bandwidth (§6.3.3) and can tolerate tens of milliseconds of inter-tier latency with minimal degradation in token throughput (§6.5). It is particularly useful for processing long context prompts, outpacing a 'single-tier' baseline by 16× in throughput (§6.6). Last, we show that the aforementioned configuration is just one of many feasible realizations and highlight how Glinthawk can be used to improve the throughput of other high-end GPUs such as NVIDIA H100s (§6.3.2).

The remainder of this paper is structured as follows; in §2, we provide a brief background on large language models from an operational point of view. In §3 we discuss attention and non-attention operations with regards to memory bandwidth and compute load. We analyze the inter-play of network characteristics and the cache requirements in §4. We discuss how we design, configure, and implement Glinthawk in §5, and evaluate our prototype with end-to-end benchmarks in §6. Finally, we discuss Glinthawk's limitations and future directions in §7.

We release Glinthawk as open-source software, along with the artifacts necessary to reproduce the experimental results.

2 Background

In this section, we provide a brief background on the architecture of LLMs and Transformers, with a focus on its high-level computational characteristics. We specifically discuss decoder-only transformers, as they are the most common architecture for LLMs, including GPT-3 [16], Meta's LLaMA [21, 47, 48], and Google's Gemma [46].

2.1 Large Language Models

Operationally, a large language model can be abstracted as a function $\mathcal{L}$ that, given a sequence of *tokens* (words or subwords), outputs a probability distribution over the next possible token in the sequence. This distribution is then sampled to produce the next token in the sequence. This process is repeated autoregressively with the new sequence until a special token marking the end of the sequence is sampled, or a certain maximum length is reached. The input sequence is called the "prompt," while the autoregressively generated tokens are called the "completion."

To avoid redundant work, the intermediate computations for past tokens can be cached and reused when generating new tokens, commonly referred to as the *key-value (KV) cache* or the context. We use these terms interchangeably in the paper.

The core of this function is a chain of functionally identical *Transformer blocks* or "layers." Each layer has its own weights and KV cache, receives a vector of size D, and produces a vector of the same size. Stacking multiple of these units, along an embedding block at the beginning and a classifier at the end[2], forms the Transformer architecture.

While a layer's output is dependent on both the weights and the cache, the computations can be decoupled. More specifically, we break down a Transformer block into three stages

$$\mathcal{F}_1(W_k;x) \to qkv$$
$$\mathcal{F}_2(C_k,qkv) \to C_k,x'$$
$$\mathcal{F}_3(W_k;x,x') \to y,$$

[1] A Glinthawk is "an aerial scavenger designed to keep the land clean of broken machines."

[2] The embedding block converts the tokens into vectors of size D, while the classifier maps the hidden state to a probability mass function.

Table 1: Naming conventions for Transformer parameters.

Symbol	Description	Symbol	Description
B	Batch size	S	Sequence length
D	Activation dimension	H	Number of attention heads
D_{kv}	Key/Value dimension	H_{kv}	Number of key/value heads
D_h	Hidden dimension		

where x marks the activation data. In this notation, $\mathcal{F}_2$ is the attention mechanism that operates on the context C_k and the output from $\mathcal{F}_1$, whereas the non-attention operations, $\mathcal{F}_{1,3}$, deal with the immutable model weights. This framework divides Transformer operations into two groups, which we refer to as attention and non-attention operations for the rest of this paper.

Non-attention operations. The bulk of computation in $\mathcal{F}_{1,3}$ consists of multiple General Matrix Multiplys (GEMMs) between model weights and transformed inputs. Since the model weights are the same for all inputs passing through a layer, their memory access cost—i.e., loading model parameters into registers for multiply-accumulate operations—can be amortized across multiple inputs, a technique known as *batching*. The memory requirements of $\mathcal{F}_{1,3}$ remains relatively constant, with negligible increase in the working set, as batch size grows.

Attention operation. While in $\mathcal{F}_{1,3}$ the same set of model weights are applied to different inputs, in $\mathcal{F}_2$ each input (that belongs to a different sequence) requires its own dedicated cache. The core of the attention mechanism involves General Matrix-Vector Multiply (GEMV) operations with this cache, and these must be repeated for each prompt independently, offering no opportunity for batching.

While processing multiple prompts through batching improves inference throughput, its effectiveness is constrained by the memory required to store the KV cache, which often becomes a limiting factor well before other system resource limits are reached [26].

3 Characterizing Transformer Computations

To motivate our two-tiered design, we argue attention and non-attention operations have fundamentally different resource requirements. In this section, we will analyze these requirements of the individual operations and, using first-order analytical models, we will demonstrate that in the traditional (single-tier) approach, accelerators are consistently bottlenecked by attention, while much better suited to perform non-attention operations at high throughput.

We focus on two key metrics: (1) number of memory load/store operations which reflects the demand on memory bandwidth, and (2) the minimum number of floating-point operations (FLOPs) required for computation. We use the LLaMA2 model family [48] as a running example throughout the following sections, but these arguments hold for other model families as well. Table 1 describes the symbols we will use in these discussions.

3.1 Non-attention Operations

Excluding attention, and some smaller operators such as `RMSNorm` and residual connections, a Transformer layer consists of several key tensor computations, including the generation of *query*, *key*, and *value* vectors, as well as four projections in the Feed-Forward Network (FFN) stage, all of which are GEMM operations. The size of these tensors, number of memory load/store operations, and computational load of these operators are detailed in Table 2.

Cost factor. For small batch sizes, much of the time the kernel spends is on memory loads, and these operators are far more efficient at larger batches where the load cost is amortized over the batch. While memory pressure is relatively constant with batch size, compute requirements scale linearly with it. As batch size increases, these operations eventually cross into compute-bound territory, with no discernible benefits in increasing it beyond that point. These batch sizes are the optimal operating point for non-attention operations.

Scaling. Compute power can be linearly scaled with parallelism, e.g., pipeline or tensor parallelism [31, 44]. GEMM operations are moderately parallel, i.e., they can be parallelized but require synchronization steps that involve communication across nodes. The communication latency for synchronization has to be comparable to the computation time, and with commodity networks, the gap is significant. Some parallelism approaches are more resilient than others, as we will cover in §4.2.

3.2 Attention Operations

Attention consists of a set of small, independent operations. For each prompt in a batch, H pairs of GEMV computations must be performed. The size, and memory and computational load of these operators are shown in Table 2. Notably, both compute and memory transfers scale linearly with batch size, as there are no common weights in attention—each prompt operates on its own dedicated set of tensors.

Cost factor. The memory and computational load of attention scales with both batch size B and sequence length S. Note that compared to non-attention operations, attention is far less compute-heavy; the compute to memory load ratio for non-attention is approximately B, but is a constant D/D_{kv} for attention. This has important implications.

First, attention can be carried out on machines with weaker compute capabilities. For example, in §6, we show we can carry out attention fully in CPU nodes, and keep up with GPUs

Table 2: Memory load/store and compute characteristics of non-attention and attention operations in a single Transformer layer. Smaller operations such as `RMSNorm` are omitted. For compute, we count each multiply-accumulate operation as one.

	Operation	Sizes	Memory Load/Store	Compute
Non-Attention	$x^T.w_q$	$w_q \in \mathbb{R}^{D\times D}, x \in \mathbb{R}^{B\times D}$	$2BD+D^2$	BD^2
	$x^T.w_{kv}$	$w_{kv} \in \mathbb{R}^{D\times 2D_{kv}}, x \in \mathbb{R}^{B\times D}$	$BD+2DD_{kv}+2BD_{kv}$	$2BDD_{kv}$
	$x^T.w_o$	$w_o \in \mathbb{R}^{D\times D}, x \in \mathbb{R}^{B\times D}$	$2BD+D^2$	BD^2
	$x^T.w_{1,2}$	$w_{1,2} \in \mathbb{R}^{D\times 2D_h}, x \in \mathbb{R}^{B\times D}$	$BD+2DD_h+2BD_h$	$2BDD_h$
	$x^T.w_3$	$w_3 \in \mathbb{R}^{D_h\times D}, x \in \mathbb{R}^{B\times D_h}$	BD_h+D_hD+BD	BDD_h
	Total	—	$D(2D+3D_h+2D_{kv})+B(8D+3D_h+2D_{kv})$	$BD(2D+3D_h+2D_{kv})$
Attention	$Q_i^T.K_i\ (i \in \{1..H\})$	$Q_i \in \mathbb{R}^{B\times D/H}, K_i \in \mathbb{R}^{D/H\times S}$	$B(D+SD_{kv}+SH)$	SBD
	$A_i^T.V_i\ (i \in \{1..H\})$	$A_i \in \mathbb{R}^{B\times S}, V_i \in \mathbb{R}^{D/H\times S}$	$B(SH+SD_{kv}+D)$	SBD
	Total	—	$B(2D+2SH+2SD_{kv})$	$2SBD$

computing non-attention operations. Second, the cost of running attention is not dominated by compute resources, but on a balance of compute and memory. The most cost-effective approach to running attention is to find compute nodes where the cost perFloating-point Operations Per Second (FLOPS) and memory operations is low.

Scaling. Attention computation is *embarrassingly parallel*: batches can be parallelized across both the batch dimension and the head dimension (e.g., Llama models have 32–128 heads). This means, in theory, attention can be distributed across BH compute node without significant overhead.[3] In contrast, such parallelization is impractical for GEMM operations, as the large layer weights cannot be efficiently partitioned across nodes without inflating memory requirements. For example, if each node computes a single element of the resulting matrix, an entire row and column must be loaded, followed by a dot-product operation. If a single node computes the full GEMM, the loaded rows and columns are reused across many elements, reducing memory overhead and improving efficiency.

3.3 Verdict

The difficulty in synchronizing GEMM operations and their sizable computational load means they are, unsurprisingly, a perfect fit for high-end GPUs. But the same cannot be said for attention; attention is relatively low in compute, scales similarly in compute and memory requirements, and is highly parallelizable. We could, in theory, run attention on a swarm of tiny nodes, as long as they can process one sequence length; this parallelization will not increase computation, memory or network operations.

4 Key-Value Cache Storage

In §3, we discussed how attention, from the perspective of compute intensity and memory bandwidth requirements, scales differently from non-attention operations. Here, we explore another aspect of Transformer scalability: memory size of the key-value (KV) cache. In order to avoid recomputations when generating tokens autoregressively, Transformers require holding onto large context vectors until their corresponding sequence is completed. Context vectors are large enough that they rival the the size of LLM weights [26], and limit the max number of *in-flight* prompts during inference.

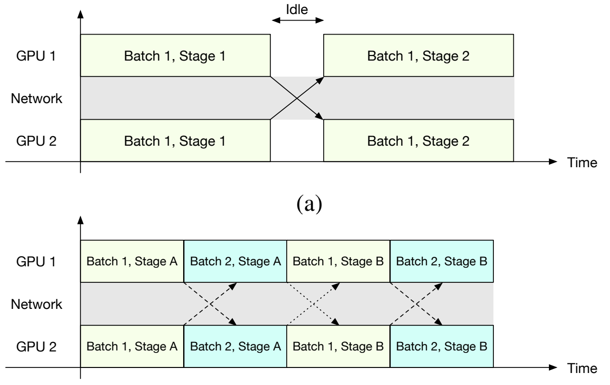
We examine how different parallelization strategies impact the number of in-flight batches, and consequently, the maximum batch size that can be used during inference. Using a first-order model of these dynamics, we show that in traditional parallelization techniques, the maximum batch size *does not* increase linearly with the number of GPUs, and network latency imposes a significant overhead on it. In contrast, Glinthawk can scale the maximum batch size much more effectively, and can mitigate inter-tier latency by increasing the number of in-flight batches.

4.1 Single-GPU Setup

As discussed in §2.1, Transformers benefit from storing certain intermediate data in the form of a cache. This data—the key and value vectors—is stored per each layer and token. For any layer, all cache entries produced in prior tokens will be required for processing the next token. Therefore, the cache remains 'alive' until the prompt is completed. We will call a prompt 'in-flight,' if it is has not completed and has an allocated cache, and similarly we call a batch of prompts 'in-flight' if all the prompts in the batch are incomplete.

We'll compute the memory requirements for the cache. Assuming a data width of 2 bytes (i.e., half-precision floating-point [10]), a singular token and layer will require $4D_{kv}$ bytes of memory (Table 1 describes the symbols we use

[3] Attention can also be loosely parallelized along sequence length using a distributed implementation of FlashAttention [20, 28]



(a)

(b)

Figure 2: **(a)** Synchronous vs. **(b)** asynchronous parallelism. Asynchronous parallelism has the potential to hide communication, but requires more than 1 in-flight batch.

in this discussion). For a Transformer with N layers processing a prompt with a length of S tokens before termination, the total memory required is $M := 2NSD_{kv}$. If we intend to run computation in batches of tokens from B different prompts at a time, we need to store one full entry per each, i.e., BM. For reference, for the LLaMA2-70B, we need to store $M = 640$MiB of data per prompt at 2048 tokens; processing a batch of size 128 requires $BM = 80$GiB of memory just to store the context.

Consider a simple case where the Transformer fully resides on a single accelerator. It computes layers one at a time until the last layer, generates the next set of tokens, and moves back to computing the first layer again. The GPU is never idle—always actively working on some kernel—and anytime a prompt finishes, a new one takes its place and reuses its context memory space [52]. Only one in-flight batch is needed, and the maximum batch size will be the same as the maximum number of in-flight prompts. Supposing $C_{\max}$ is the free memory space after allocating LLM parameters, the maximum number of in-flight prompts is $\lfloor \frac{C_{\max}}{M} \rfloor$, same as the maximum batch size.

4.2 Single-Tier Parallelism

If the model is too large to fit in one GPU, we have to employ a parallelization strategy to split model weights among GPUs. Below, we discuss context memory dynamics for two popular parallelization techniques: tensor and pipeline parallelism[4].

Tensor parallelism. In this approach, each GPU hosts a fraction of each layer, and after each computation the GPUs share their fractional results via an AllGather operator. If the GPUs run *synchronously*, computation cannot continue until the AllGather operator is finished, as shown in Figure 2a. Individual computation steps can be sub-millisecond, and this data transfer can quickly become the major bottleneck with distributed GPUs. A simple workaround is to run inference *asynchronously* depicted in Figure 2b, e.g., GPUs work on stage A for batch 1, and stage A for batch 2 while batch 1 is going through AllGather.

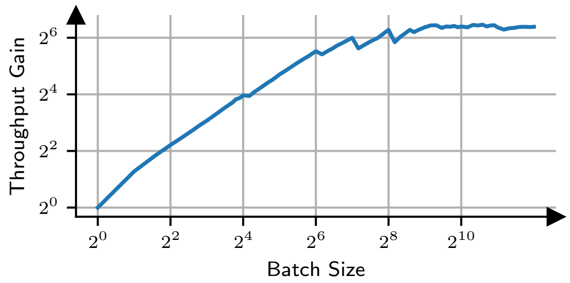


Figure 3: Throughput gain vs. batch size, for the Llama2-70B transformer running on a NVIDIA T4 GPU.

The key question is how many unique 'in-flight' batches are needed to hide the transmission latency. This is important, since the maximum batch size we can run is inversely proportional to this value, i.e., if we need $IF_{tp}(B)$ in-flight batches of size B for full utilization, B should be such that $B \leq \lfloor \frac{C_{\max}}{M \times IF_{tp}(B)} \rfloor$. The reduction in batch size, $IF_{tp}(B)$, is the toll that transmission latency takes on throughput.

Suppose the computation latency for the smallest stage between synchronization barriers is $t^B_{c,min}$ for batch size B on one GPU. As tensor parallelism splits matrices to K shards between K GPUs, the computation would take at least $t^B_{c,min}/K$ per GPU node. If the communication latency of a batch t^B_n is negligible compared to compute $t^B_n/t^B_{c,min} \approx 0$, i.e., the GPUs are co-located, we only need 1 unique batch in flight at a time to keep all GPUs busy. If not, we need more than 1. With $0 \ll t^B_n \leq t^B_{c,min}/K$, we need 2 unique batches; One batch is in transit while the other undergoes computation, and the roles are reversed when the transit and computation is over, similar to the example in Figure 2. In general, we need

$$IF_{tp}(B) := \left\lceil 1 + \frac{t^B_n}{t^B_{c,min}/K} \right\rceil \qquad (1)$$

in-flight batches to keep all GPUs busy.

For instance, with the Llama2-70B transformer the computation before the smallest synchronization barrier is $t^1_{c,min} \approx 0.48$ms on an NVIDIA T4 GPU at a batch size of $B = 1$. The communication involves 16KiB of data transfer, and assuming an Round-trip Time (RTT) of 2ms and 8Gbps Ethernet bandwidth, takes $t^B_n = 1 + (16\text{KiB}/8\text{Gbps}) \approx 1ms$[5]. We need $IF_{tp}(1) \approx \lceil 2.08K + 1 \rceil$ in-flight batches for full utilization. At 10-way tensor parallelism ($K = 10$, hypothetical minimum T4

[4] We do not discuss parallelism strategies based on transferring weights, e.g., FSDP, Zero-3 [40, 41, 53], whom have substantial network requirements.

[5] The latency will be higher in practice as AllGather latency is max transit time along all nodes, and sensitive to stragglers.

GPUs needed to evenly host the model), we need 22 in-flight batches of size $B = 1$. As we have context space for $\frac{C_{\max}}{M} = 32$ prompts, $B = 1$ is the maximum batch size we can use. For reference, under no network latency $IF_{tp}(\cdot) = 1$ and $B = 32$, under the specified link. The non-attention operations of this Transformer do not saturate compute until $B = 256$, as observed in Figure 3. The communication overhead would have been negligible under NVLink and tolerable with InfiniBand, hence the popularity of tensor parallelism under strong interconnects.

Pipeline parallelism. Alternatively, we can split the model layer by layer across multiple nodes; with K GPUs, each node hosts $\frac{N}{K}$ layers. Each GPU takes in a batch, passes it through the layers it hosts, and sends it to the next GPU.[6] Pipeline parallelism is by nature asynchronous, and to keep all GPUs busy, we need at least K in-flight batches. However, the computation steps involved in pipeline parallelism are far larger than tensor parallelism, and the network overhead on number of in-flight batches will be smaller.

Similar to tensor parallelism, suppose the computation latency is t_c^B per GPU, and the communication latency of one batch from one GPU to the next is t_n^B, for a batch of size B. With a similar argument to tensor parallelism, we need

$$IF_{pp}(B) = \left\lceil 1 + \frac{t_n^B}{t_c^B} \right\rceil \times K' \tag{2}$$

batches to keep all GPUs busy. Under the previous example and at 10-way pipeline parallelism, we have $t_c^1 \approx 56$ms on an NVIDIA T4 GPU, $t_n^1 = 1 + (16\text{KiB}/8\text{Gbps}) \approx 1ms$ at a batch size of $B = 1$. We would need $IF_{pp}(1) \approx = 2K = 20$ in-flight batches of size 1 for full utilization, and cannot go higher.

Fundamental limitations. Table 3 summarize the context dynamics discussed before. In both parallelism approaches the available context memory $C_{\max}$ will scale linearly with the number of GPUs K. Unfortunately, so will the number of in-flight batches required to maintain full utilization ($IF_{tp}(B)$,$IF_{pp}(B) \propto K$). As such, the batch size will not improve with parallelism, and token throughput will at best scale linearly with K. Note that a linear increase in throughput with K is not useful; We can achieve the same throughput gains by hosting M parallelism sessions, with each session using K/M-way parallelism.

4.3 Glinthawk: Two-Tier Parallelism

Our two-tiered scheme can scale available context memory $C_{\max}$ linearly with the number of Tier-2 nodes, with minimal impact on the number of in-flight batches. In this scheme, the context memory and attention computation is moved to a dedicated second tier of compute nodes; the first tier only works on non-attention operators, which mainly involve GEMM operations. Besides the points mentioned in §3 (attention is embarassingly parallel and computationally cheap), this architecture has another interesting benefits; *attention is the only stateful computation*. Non-attention operators are stateless, i.e., the expensive GPUs computing them can be swapped, downscaled or upscaled without disrupting the entire inference. The nodes hosting attention are stateful, and difficult to alter.

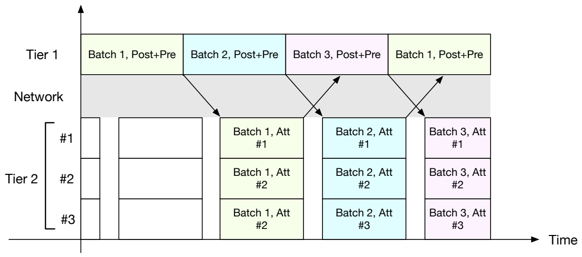


Figure 4: Glinthawk's batching schedule. Glinthawk hides the inter-tier transit time by utilizing multiple inflight batches.

Context dynamics. The Glinthawk parallelization schedule is visualized in Figure 4. When a Tier-1 node has finished non-attention operations on a batch of size $B \times K'$, the batch is split to K' pieces of size B and sent to K' Tier-2 nodes. Attention operations are performed on these batches in $t_{c2,att}^B$, and sent back to the Tier-1 node. Suppose the overall transit time takes $t_n^{BK'}$. To keep Tier-1 busy in the meantime, we need other in-flight batches that undergo non-attention operations. If the Tier-1 computation latency is $t_{c1,no-att}^{BK'}$ for the full batch, we need

$$IF_{gh}(B,K') = \left\lceil 1 + \frac{t_{c2,att}^B + t_n^{BK'}}{t_{c1,no-att}^{BK'}} \right\rceil \tag{3}$$

in-flight batches to hide the Tier-2 total latency from Tier-1.

This overhead is practically invariant to K'. For small K', we have $t_n^{BK'} \ll t_{c2,att}^B$ and therefore $IF_{gh}(B,\,K') \approx \lceil 1 + t_{c2,att}^B / t_{c1,no-att}^{BK'} \rceil$, which does not increase with K'. With larger values of K', we have $t_n^{BK'} \gg t_{c2,att}^B$ and $IF_{gh}(B,K') \approx \lceil 1 + t_n^{BK'} / t_{c1,no-att}^{BK'} \rceil$; Since the Tier-1 batch size BK' grows large, the non-attention latency is compute-bound and grows linearly with K', cancelling the linear growth in transit time. Therefore, the number of in-flight batches needed, $IF_{gh}(B,K')$, does not scale with K'. The Tier-1 batch size BK' is depicted against the number of attention nodes per Tier-1 node K' in Table 4, for the running example we have used thus far. The Tier-1 batch size grows super-linearly with K', highlighting the scalability of this approach.

This in-flight batch overhead is additional to whatever parallelization scheme we use to split non-attention operators in Tier 1. In this work, Glinthawk parallelizes Tier-1 nodes with pipeline parallelism, mainly for latency resilience, but it is trivial to extend Glinthawk to tensor parallelism.

[6] We focus on LLM inference, and do not consider training. Pipeline parallelism has challenges in training, due the existence of a backward pass [31].

Table 3: Communication latency, minimum in-flight batches and max batch size for different parallelism schemes.

Parallelism	Context Memory	Communication latency	Minimum Inflight-Batches	Max Batch Size
Single-GPU	–	–	1	$\lfloor \frac{C_{max}}{M} \rfloor$
Tensor	$C_{max} \propto K$	$t_n^B \approx \frac{RTT}{2} + \frac{2BD}{BW}$	$IF_{tp}(B) = \lceil 1 + K \frac{t_n^B}{t_{c,min}^B} \rceil$	$\max_B$ if $B \leq \lfloor \frac{C_{max}}{M \times IF_{tp}(B)} \rfloor$
Pipeline	$C_{max} \propto K$	$t_n^B \approx \frac{RTT}{2} + \frac{2BD}{BW}$	$IF_{pp}(B) = \lceil 1 + \frac{t_n^B}{t_c^B} \rceil \times K$	$\max_B$ if $B \leq \lfloor \frac{C_{max}}{M \times IF_{pp}(B)} \rfloor$
Glinthawk	$C_{max} \propto K' \times K$	$t_n^{BK'} \approx RTT + \frac{2BK'(4D+2D_{kv})}{BW}$	$IF_{gh}(B,K') = \lceil 1 + \frac{t_{c2,att}^B + t_n^{BK'}}{t_{c1,no-att}^{BK'}} \rceil$	$\max_{BK'}$ if $BK' \leq \lfloor \frac{C_{max}}{M \times IF_{gh}(B) \times IF_{pp}(BK')} \rfloor$

Table 4: Unique batches needed to achieve full utilization, using NVIDIA T4 GPUs as Tier-1 and AMD EPYC 7V12 (16 cores, 110 GiB RAM) as Tier 2.

K’	Tier-1 batch size BK'	Tier-2 batch size B	Overhead $IF_{gh}(B,K')$
1	72	72	3
2	124	62	3
4	252	63	3
8	632	79	3
16	1280	80	2

5 Design

So far, we have discussed how a two-tier inference architecture allows us to better utilize Tier-1 accelerators by scaling out the attention mechanism to Tier-2 nodes. In this section, we go over the practical design of this architecture (§5.1), explain how we determine configuration parameters such as batch sizes and the number of Tier 1 and 2 nodes (§5.2), and discuss implementation details (§5.3).

5.1 Glinthawk: Architecture

Glinthawk comprises of three main components: (1) a Dispatcher, which creates and populates in-flight batches, collects completions tokens, and assigns context storage, (2) a set of Tier-1 worker nodes that compute non-attention operations, and (3) a set of Tier-2 worker nodes that compute attention operations.

Dispatcher. This component is a low-overhead controller, logically located within the worker node responsible for processing the initial operations of the Transformer. It serves three primary functions in managing inference pipeline: (1) initial handshake with worker nodes, and assigning roles in the computation graph, based on its global view of the worker pool, (2) maintaining in-flight batches by organizing and tracking prompts as they move through the system (i.e., continuous batching [52]), (3) controlling context assignment of prompts to Tier-2 nodes.

When Glinthawk is starting, the Dispatcher creates a set of in-flight *batch state objects*. These objects store essential information related to prompts being processed, such as a global prompt identifier, a global context location identifier, the position of the latest token, the generation temperature, and inter-stage communication data (e.g., activation vectors, queries, etc.). The Dispatcher creates a finite number of batch state objects, which is a configurable system parameter, further discussed in §5.2). Once initialized, these batches are updated and modified as prompts progress through the system.

The Dispatcher then allocates context space for each unallocated prompt in the batch state objects. It assigns a unique identifier that corresponds to a memory region within a Tier-2 node for storing key-value cache vectors. This assignment ensures two key guarantees: first, no attention worker will run out of context memory while another has space available, and second, the computation load of attention is proportionally distributed across workers. The latter is guaranteed since each Tier-2 worker is responsible for applying attention to the slots assigned to it, and the number of slots per worker is predetermined during configuration.

The Dispatcher maintains a list of queued unprocessed prompts. After creating all batch state objects, the Dispatcher fills them with prompt information from this queue, and sends them to worker nodes for processing. When a prompt finishes (by reaching end-of-sequence tokens, or getting to the max sequence length), the Dispatcher replaces the prompt in that slot with a new one from the queue. There is no need to announce the finish of the previous prompt nor to release context assignments. These assignments will be reused for new prompts automatically.

The Dispatcher does not discriminate between input and output tokens. If there are enough unprocessed prompts available, and the batch size is sufficiently high, the number of tokens processed per second will be no different in ‘prefill’ or ‘generation’ phases. Of course, if these conditions are not met, we can create batch state objects that are filled with tokens from only one prompt. Also, if the latency of generation is important, we can speed up the prefill phase with a dedicated tier, as proposed in recent work [12, 37].

Tier-1. Tier-1 nodes are in charge of processing non-attention operations. They are also responsible for the distribution/aggregation of work to/from Tier-2 nodes. These nodes consist of three separate abstractions.

- **Network Controller**: All manners of communication are abstracted in network controller that resides on each node. The brunt of this communication is with other worker nodes and involves serializing/deserializing state objects. Since nodes run at high throughput (e.g., an NVIDIA T4 GPUs as a Tier-1 nodes typically processes 10K prompts per second, or 4 Gbps of activation data), they have been optimized to carry out these operations by avoiding memory copies and reusing previously allocated memory.
- **Tier Router**: Each Tier-1 node is in charge of a slice—one or more layers—of the Transformer (since Glinthawk is currently based on pipeline parallelism), and has a dedicated set of Tier-2 nodes for its attention workload. The Tier Router splits a ready-for-attention batch based on their context locations to 'shards' and routes the shards to their respective Tier-2 nodes. It later receives shards back from Tier-2 nodes, merges them back and computes the next non-attention operators.

  The 'Tier Router' does not necessitate that each Tier-2 node participates one shard per each merge; rather, one node can participate more than its equal share or none at all. This makes the merge operation resilient to stragglers, but does not put unequal computational load across nodes. As covered before, this is because computational load is determined by the number of context assignments.
- **Compute Kernel**: The computational power of the node is abstracted to a compute kernel, that takes in a batch/shard and applies the operators it hosts to them. The compute kernel abstracts hardware choice, as it could use a GPU, TPU, CPU, FPGA, etc. with no relevance for other components. The compute kernel always gives priority to batches in later layers/stages of the Transformer to make sure enough work for subsequent workers is always in the pipeline.

Tier-2. Tier-1 and Tier-2 nodes share much of their structure, with two key differences. First, Tier-2 only focuses on layer-agnostic attention operations. Second, Tier-2 nodes do not have a 'Tier Router' component; they receive batches, compute attention, and send them back.

Note that the attention operation is independent of the layer. In theory, we could have all Tier-2 nodes serving all Tier-1 nodes at the same time, instead of each Tier-2 node being tied to a specific Tier-1 node. This may be particularly beneficial in cases where Tier-1 nodes have heterogeneous compute resources. The downside is vulnerability to stragglers while collecting shards. We leave the exploration of such assignment strategies to future work.

5.2 Glinthawk: Configuration

Given a set of resources in the form of eligible Tier-1 and Tier-2 nodes, we aim to maximize the number of tokens per second we can achieve, or alternatively minimize the cost of token generation. These metrics can be bottlenecked by a multitude of factors such as the maximum compute kernel processing speed, network bandwidth, maximum in-flight batches, etc. Maximizing gains is about balancing all these components in tandem.

Concretely, given a specific LLM, a Glinthawk configuration is uniquely determined by the tuple (K, K', B), where K is the number of Tier-1 nodes, $K' \times K$ is the number of Tier-2 nodes, and $K' \times B$ is the running batch size in Tier 1. If we define $f(K, K', B)$ as the intended metric (e.g., throughput, cost per throughput unit, etc.) for this configuration, we aim to solve the following optimization problem:

$$K^*, K'^*, B^* \leftarrow \underset{(K,K',B)}{\operatorname{argmax}} f(K, K', B).$$

Since the optimization space it not too large, we can solve this problem with an exhaustive search over the parameter space. To compute $f(K, K', B)$, we simulate the full pipeline. Specifically, we model compute resources and link bandwidth as "pipeline elements" that, given a batch with a known size, are occupied for a known amount of time and send the result to the next element in the pipeline. We also model the link latency, i.e., the raw RTT, as a "delay" element that is not occupy-able and introduces a fixed delay for each batch.

With this model, we can compute the available context memory C_{max}, create the maximum number of in-flight batches possible $\frac{C_{max}}{B \times K' \times M}$, and send them through the pipeline. Transient pipeline effects stabilize after 1 generated token, at which point we can measure the number of tokens generated per second.

We implement the simulation as a Discrete Event Simulation stack (300 Python LoC + 250 C++ LoC). We approximate link occupation times by a simple α-β model [23]. To calculate the compute resource occupation time, we run a one-time profiling of the Tier-1 and Tier-2 compute kernels at various batch sizes. The profile takes about 20 min per accelerator type. To simulate a cluster with 80 NVIDIA T4 GPUs, 80 CPU nodes, and a maximum batch size of 4096, the full optimization takes less than 2 minutes to run.

5.3 Implementation

We implement Glinthawk in C++ and CUDA [35] (16K LoC), and Python (2.4K LoC). For Transformer operations, we use a mix of cuBLAS [1] routines and custom CUDA kernels for CUDA-powered nodes. For CPU-powered nodes we implement all operations with OpenMP [8]. Kernel computations run at FP32, while kernel results are stored in the model's native data type. We use CUDA Virtual Memory Management [5] to implement PagedAttention [26].

Our implementation abstracts away communication between nodes, which is handled at the host level with all data transmission carried out over userspace POSIX sockets. However, alternative approaches, such as NCCL [7] or custom transport layers, can also be implemented if needed.

We used a standard implementation of Multi-Head Attention (MHA) with support for Grouped-Query Attention (GQA) [13]. The implementation can be enhanced with FlashAttention [19, 20] and FlashInfer [2] for higher throughput. Alternative kernel implementations can be used for all stages as long as Glinthawk has visibility into context slot assignments, and can break layers to attention and non-attention stages.

6 Evaluation

The primary goal of our proposed architecture is to improve the utilization of our most expensive and valuable resources—the Tier-1 accelerators—by satisfying attention's memory requirements by scaling up less expensive second-tier nodes. By evaluating different configurations of Glinthawk, and comparing them against baselines described in §6.1, we aim to investigate two primary questions: (1) how much does Glinthawk's two-tiered architecture improve throughput compared to standard pipeline parallelism? (§6.3) (2) How much network bandwidth do we need to sustain this throughput? (§6.3.3) (3) What are the costs associated with the proposed architecture? (§6.4) Moreover, using a series of microbenchmarks, we study how throughput is affected as we change inter-tier latency (§6.5) and sequence length (§6.6) on throughput.

6.1 Setup

Testbed. We use a testbed comprising of NVIDIA Tesla T4 GPUs (16 GiB of memory per GPU), and AMD EPYC 7V12 CPUs (16 cores with 110 GiB of RAM). All nodes are equipped with 8 Gbps networking. For various microbenchmarks, we also use other compute nodes such as an NVIDIA Quadro RTX 6000 (24 GiB of memory), NVIDIA A100 (40 GiB of memory), NVIDIA H100 (80 GiB of memory) and an NVIDIA A6000 (48 GiB of memory).

Metrics. Our main focus is on two primary metrics: (1) throughput, measured as tokens per second (§6.3), (2) cost per throughput unit, measured as U.S. Dollars per token per second (§6.4). We also report time-per-token, although token latency is a non-goal for Glinthawk.

Dataset. Since token processing time does scale with sequence length, the length of prompts does affect inference throughput. We use ShareGPT [9] dataset, specifically the first human message and AI response, as the prompt and completion length distribution of our experiments.

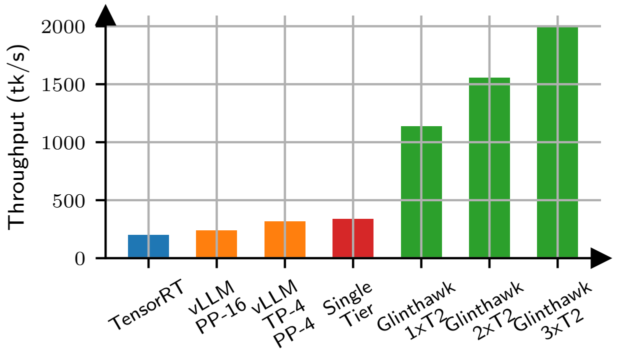


Figure 5: Inference throughput for various schemes using 16 NVIDIA T4 GPUs. Glinthawk extracts more throughput from high-end Tier 1 machines compared to baselines.

Models. For these evaluations, we primarily focus on the Llama 2 [21, 48] model class families, due to their popularity and the fact many well-known open-source models such as Alpaca [45], Vicuna [18] are based on this model class. We use the model's native FP16 weights without quantization.

6.2 Baselines

Single-Tier. This baseline utilizes Glinthawk without a second tier, and allows for a direct analysis on how much the second tier helps. This is equivalent to standard pipeline parallelism.

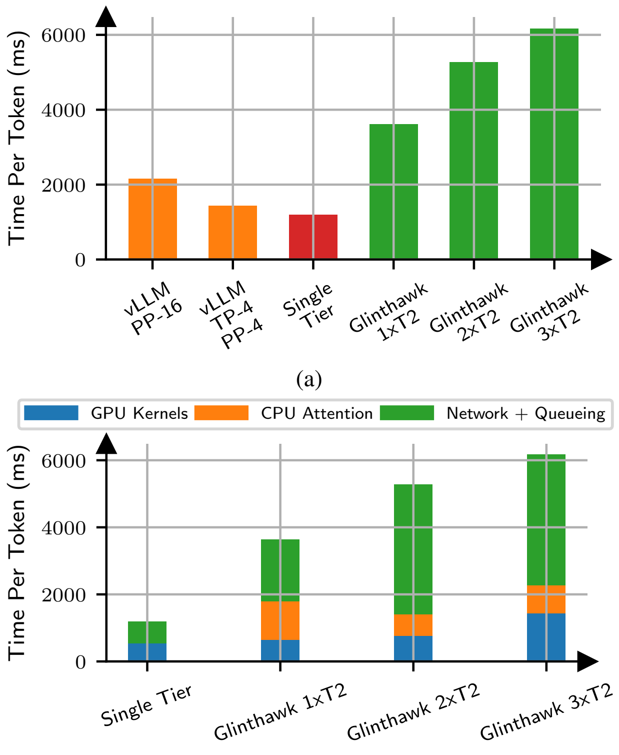
TensorRT-LLM. TensorRT-LLM [3] is one of the leading inference engines on NVIDIA hardware, and supports pipeline and token-level parallelism for various models including the Llama model family.

vLLM. vLLM [4] is a widely known inference framework that utilizes PagedAttention [26] for efficient management of KV-cache slots. We compare different configurations of Glinthawk against vLLM running with pipeline parallelism and/or tensor parallelism.

6.3 End-to-end Performance

For our end-to-end evaluation, we compare the performance of the baseline systems, as defined in §6.2, running on 16× NVIDIA T4 GPUs to Glinthawk running with same Tier-1 nodes, supplemented with one to three additional Tier-2 nodes per each Tier-1 node. While this comparison isn't entirely fair, as Glinthawk uses more resources, we demonstrate in §6.4 that Glinthawk also provides lower cost per throughput unit across the evaluated configurations, simultaneously achieving better throughput and cost efficiency.

Figure 5 summarizes these results. Notably, our single-tier setup achieves performance on-par or better than both vLLM (16-way pipeline parallel and 4-way pipeline parallel + 4-way tensor parallel) and TensorRT (16-way pipeline



(a)

(b)

Figure 6: **(a)** Time per token across different schemes. Glinthawk gains throughput at the cost of higher time per token, due to large batch sizes and using a second tier. **(b)** Breakdown of time per token. With more Tier 2 nodes, Glinthawk runs at higher batch sizes, increasing compute time in GPUs.

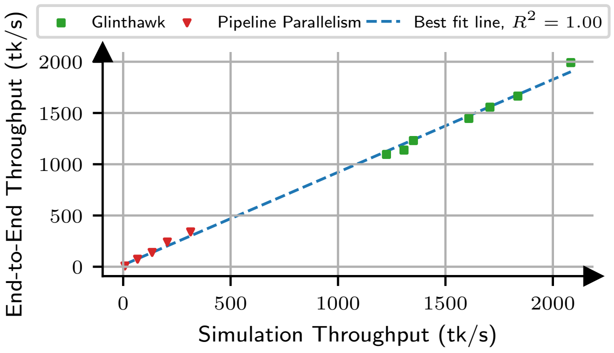


Figure 7: Predicted throughput vs. end-to-end measurements for various Glinthawk configurations with NVIDIA T4 GPUs as Tier 1 and AMD EPYC 7V12 16 Core CPUs as Tier 2.

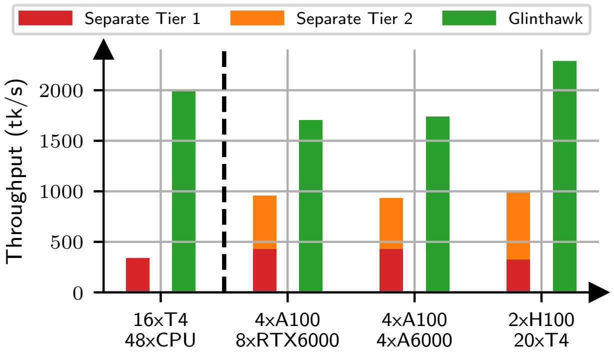


Figure 8: Glinthawk's throughput compared to running each Tier as a separate pipeline. The T4+CPU config is from experiment data, and other configs are simulated.

parallel). Adding one CPU node per GPU boosts Glinthawk's throughput by 3.4× over single-tier baseline. The optimal configuration for Glinthawk, based on simulation results, involves 3 CPU nodes per GPU, resulting in 5.9× overall improvement.

Figure 6a shows time per token (either processed or generated) for the same schemes (with the exception of TensorRT-LLM). Glinthawk has the highest time per token (up to 5.2× higher compared to the traditional pipeline parallel setup), due to several reasons, including using a higher batch size, slower attention computation on Tier 2 devices, and the inter-tier communication latency.

Furthermore, Figure 6b shows the breakdown of time per token for Glinthawk. With more Tier-2 nodes, Glinthawk runs at higher batch sizes which increases the latency of GPU kernels; the single-tier runs at a batch size of 14, while Glinthawk runs at batch sizes of 112, 116 and 246 with 1, 2 or 3 CPU nodes per GPU. With 1× CPU, inference is bottlenecked by the rate that Tier-2 can process attention. We can see this since when we add a second Tier-2 node, the optimizer maintains a similar batch size (112 → 116) and decreases work done per CPU, but the overall throughput increases. With the addition of the third Tier-2 node, the optimizer can increase the Tier 1's batch size enough that it allows Tier 2 batch sizes to also increase, from 58 prompts (×2 nodes) to 82 prompts (×3 nodes) per batch.

In all these cases, more than 50% of the time per token is spent in transit—either being transferred over the network or waiting in a queue. The amount of DRAM these nodes have is nearly twice as much as needed for the minimum number of in-flight batches, which causes queueing in Tier-2. We can reduce the time per token by lowering the number of in-flight batches; however, keeping a small queue ensures that transient slowdowns in straggling machines would not cause pipeline bubbles, as there is always some non-empty queue of batches.

6.3.1 Simulation Fidelity

As discussed in §5.2, we use simulations to decide the optimal configurations to run Glinthawk at. In Figure 7, we compare the empirical throughput in the end-to-end configurations tested in §6.3 against the predicted throughput in simulations. As observed, the simulation matches empirical measurements with high accuracy.

Table 5: Inter-tier network traffic rates (in Gbps) for several configurations and the corresponding token throughput. Since Glinthawk only needs to send small activation data, the bandwidth requirements remain modest.

				Tier-1 Egress		Tier-2 Egress	
	Tier 1	Tier 2	Token/sec	Per	Total	Per	Total
Empirical	9×T4	27×CPU	1096	2.91	26.2	0.86	23.3
	16×T4	16×CPU	1138	1.68	26.9	1.49	23.9
	16×T4	32×CPU	1557	2.30	36.7	1.02	32.7
	16×T4	48×CPU	1992	2.94	47.0	0.87	41.8
Simulated	4×A100	8×RTX6000	1705	8.66	34.6	3.85	30.8
	4×A100	4×A6000	1738	8.87	35.5	7.88	31.5
	2×H100	20×T4	2286	24.4	48.8	2.17	43.4

6.3.2 Hypothetical Configurations

While these experiments highlight the improvements, this is just one realization of Glinthawk's two-tier proposal, using NVIDIA T4 GPUs and CPU nodes. Glinthawk should be beneficial with a wide array of compute cluster pairings, e.g., using NVIDIA H100 GPUs as Tier-1 nodes and NVIDIA T4 GPUs as Tier 2. Unfortunately, we did not have access to sufficient equipment for additional hardware pairings and therefore relied on simulations. In accordance to the procedure described in §5.2, we profile the performance of several high-end to medium-end GPUs, and use those real-world profiles to simulate their end-to-end performance in a first order model of the computation and networking pipeline. While these results are not end-to-end experiments, our simulations have previously matched real experiments with high fidelity, as shown in §5.2.

Figure 8 shows the predicted throughput of Glinthawk using these clusters, compared to what these clusters could achieve as separate single-tier schemes. In all cases, the sum of these two tiers could achieve greater throughput with Glinthawk than traditional pipeline parallelism. As covered in §3 and §4, Glinthawk's main strategy is to assign operations with significant differences in memory-to-compute ratio to hardware that optimally matches their resource demands. As long as this contrast exists, Glinthawk is effective. This disparity can be observed at many scales, whether comparing NVIDIA T4 GPUs with CPUs, or NVIDIA H100 GPUs with NVIDIA T4s.

6.3.3 Networking Requirements

Table 5 shows the required network bandwidth between Tier-1 and Tier-2 nodes for the end-to-end experiments and simulated configurations. These requirements are modest by modern cloud networking standards, despite Glinthawk's state-of-the-art token generation rate.

The total inter-tier traffic rate grows linearly with token throughput. Network transfers are dominated by activation data, and per each processed token, a Tier-1 node has to send activations data to Tier-2 nodes for as many layers as it hosts, i.e., $\lceil \frac{N}{K} \rceil$. Therefore, total Tier-1 egress can be approximated with

$$K \times \left\lceil \frac{N}{K} \right\rceil \times 2(2D+2D_{kv}) \times T \approx c_0 \times T$$

where T is the token throughput, and c_0 is a constant that depends on the LLM. Total Tier-2 egress is similar, but with smaller activation data ($2D$) leading to a smaller constant. The activation data may be amenable to compression [32], but that is out of the scope of this work.

Table 6: Retail prices of the equipment used in our evaluation at the time of writing, September 2024 [34].

Device	Retail Price (USD)
DDR5 Memory (128 GiB)	$211
AMD EPYC 7H12 (64 cores)	$2,078
NVIDIA T4 (16 GiB)	$1,780
NVIDIA RTX 6000 (24 GiB)	$2,280
NVIDIA A6000 (48 GiB)	$4,820
NVIDIA A100 (40 GiB)	$8,798
NVIDIA H100 (80 GiB)	$30,979

6.4 Cost Analysis

We evaluate Glinthawk's throughput improvements in terms of associated costs. The fundamental aspect of our approach is the ability to meet part of the memory requirements with less expensive computing nodes, which can directly contribute to lower costs. We use the cost of the computing equipment used in out setup for our primary configuration, and extend the cost comparison to alternative configurations tested through simulation.[7] While our prototype demonstrates the potential of Glinthawk, we expect that reproducing this architecture in more efficient form factors could yield greater cost efficiencies over time. Custom hardware setups or optimized infrastructure could further reduce operational costs and improve performance scalability.

Table 6 shows the retail unit prices of the compute and memory devices used in our setup, and Figure 9 depicts what the cost per throughput unit would be across empirical and simulated schemes. Glinthawk manages to lower the cost of inference across all schemes, particularly compared to schemes that only use high-end GPUs. Note that the cost reduction will ultimately depend on how well the tier pairs 'fit', i.e., the contrast in their memory to compute.

6.5 Effects of Inter-Tier Latency

To evaluate how network latency between Tier-1 and Tier-2 nodes affects Glinthawk's performance, we induce inter-tier

[7] It is noteworthy that these costs are subject to market conditions, and this analysis may become less relevant over time.

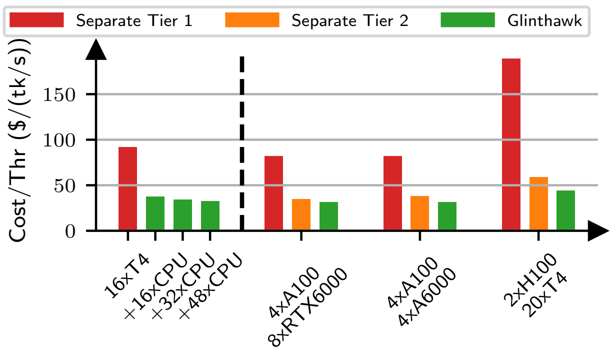


Figure 9: Glinthawk's cost per throughput unit compared to running each Tier as a separate pipeline. The T4+CPU configuration is from experiment data, and others are simulated.

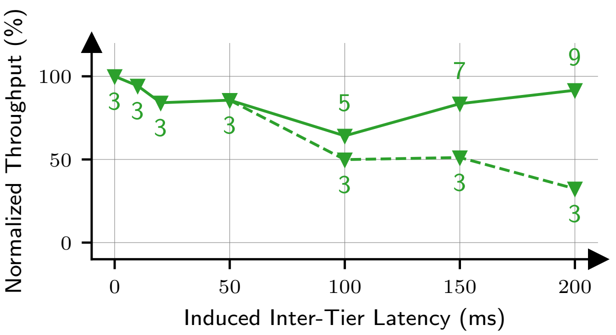
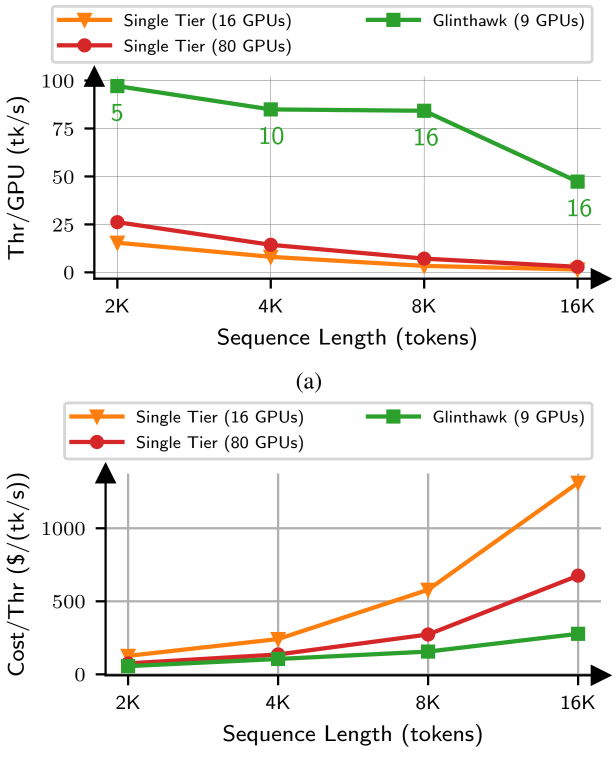


Figure 10: Glinthawk's inference throughput with various inter-tier RTT values, normalized to throughput without induced latency. The number below data points denotes the number of Tier-2 nodes per GPU. Glinthawk mitigates inter-tier latency with more Tier-2 nodes.



(a)

(b)

Figure 11: **(a)** Throughput per GPU and **(b)** Cost per Inference Throughput (CIT) compared to a single tier setup for various sequence lengths. The number below Glinthawk's data points denotes the number of Tier 2 nodes per GPU. As sequence length grows, KV-cache slots become more scarce, limiting batch size. This makes Glinthawk's two-tier separation more valuable.

latency from 2ms to 200ms. Figure 10 demonstrates the inference throughput variation at different latency values, and for different number of Tier 2 nodes per GPU. Higher latency increases the end-to-end prompt latency, which in turn increases the number of in-flight prompts, and reduce batch size. If we do not scale Tier-2 machines, throughput will drop with higher latencies. However, if we scale the Tier-2 size, we can increase in-flight batches and hide the latency from Tier-1 machines. Note that as covered in §6.3.3, the network bandwidth needed only scales with token throughput, and large inter-tier latencies do not increase required bandwidth.

6.6 Scaling to Longer Sequence Lengths

Higher sequence lengths will require larger KV-cache slots and proportionally reduce the batch size the scheme can run at. However, as covered in §3 the computational load of attention scales with the context memory; regardless of sequence lengths, since the total context memory is bounded, the total computational load remains constant. In other words, Tier-2 can still keep up with Tier-1. Furthermore, Glinthawk can scale the number of Tier-2 nodes to scale the batch size back up again. The effect can be observed in Figures 11a and 11b. Glinthawk outperforms the single-tier baseline both in throughput gained per GPU, and the cost per throughput unit. Note that we normalize throughput here per GPU to be able to compare across different GPU cluster sizes.

7 Limitations

In this section, we discuss some limitations of Glinthawk, around both its approach and its evaluation.

7.1 Limitations of Glinthawk's approach

High token latency. Glinthawk is by design oriented towards maximizing throughput through extensive batching. Batching prompts improves throughput, but exacerbate latency. In general, inference throughput and latency are conflicting goals [11]. Additionally, separating attention and non-attention computations affects end-to-end token throughput for individual prompts, especially over high-latency network links.

No prefill. As mentioned in §5.1, Glinthawk currently does not treat input or output tokens any differently. Our problem-setting of batch inference assumes that Glinthawk has as many prompts needed to fill in-flight batches. With full and sufficiently large batch sizes, running inference on a batch of input tokens, i.e., 'prefill,' is not faster than running inference with a batch of distinct prompts. If these two assumptions—i.e., large batch sizes and a sufficient backlog of prompts—are not met, it is beneficial to implement 'prefill' batches in Glinthawk.

7.2 Limitations of our evaluation

Inter-Tier Latency. In §6.5, we measured the resilience of Glinthawk's two-tier architecture to latency increments between the two-tiers. For this experiment, we injected the stated latency in Glinthawk's code stack. We did not perform those experiments on actual high-latency links and did not study the effects of congestion control on performance. This may be particularly important in links with high bandwidth-delay product (BDP).

Glinthawk + Tensor Parallelism. A key design decision we made was to build Glinthawk's two-tier architecture atop pipeline parallelized Tier-1 Nodes. However, Glinthawk could also be applied to a tensor parallelized cluster of Tier-1 Nodes. This will be beneficial if the Tier-1 cluster has strong interconnects, e.g., NVLink or InfiniBand, and can improve token generation despite tensor parallelism synchronization steps.

8 Related Work

Efficient Kernels. The most direct way to improve inference is to improve the efficiency of the implemented kernels. Faster kernels leads to better utilization of GPU resources, producing a higher rate of tokens per second. On one end are techniques that offer better parallelization for models without altering the computation, such as better packing of Attention work [19, 20]. Orca demonstrated the effectiveness of token-level parallelism in contrast to request-level parallelism, i.e, batching tokens instead of requests [52]. Alternatively, a line of work focuses on alterations to computation accuracy, trading off speed for minimal degradations in quality [29, 50].

Weight Memory. Another way to improve inference is through better management of weight memory locality and bandwidth needs. For high-end LLMs, model weights are commonly larger than the memory available in one GPU and need to be distributed among multiple GPUs, *e.g.,* GPT-3 [16] requires 325GiB of memory to host model weights. A range of techniques exist for model distribution, *e.g.,*, model/tensor parallelism [27, 55], offloading [43], pipeline parallelism [15, 24].

Key-Value Memory. KV Cache memory grows linearly with batch size and sequence length, and as discussed in length in the paper, bottlenecks the maximum inference batch size. One class of techniques focus on frugal memory management. PagedAttention demonstrated the effectiveness of allocating KV Cache as needed, instead of allocating the entire sequence memory upfront, which allowed running at higher batch sizes [26]. Another line of work focuses on cases where the prompts in a batch share an initial number of tokens, and can effectively reuse the same KV Cache space [25, 56]. Some works suggest loading/unloading KV Cache to/from GPU, such as to CPU DRAM through PCIe [14] or SSDs [43]. Another approach is to quantize only the KV Cache to save space [43]. These works proved effective at reducing KV cache footprint, but they cannot ultimately scale attention memory to nominal batch sizes or growing sequence lengths [6]. Another line of work suggests aggregating GPU memories in a cluster and running distributed attention algorithms for long context requests [28, 30]. Concurrent to our work, Lamina suggests similar ideas of offloading attention to a second set of memory-optimized devices [17]. However, we propose Glinthawk as a blueprint for flexible and scalable inference clusters with commodity level networking hardware. We show that our prototype is resilient to hundreds of milliseconds of latency between tiers, while Lamina needs RDMA-level network performance to maintain improvements.

9 Conclusion

We presented Glinthawk, a two-tiered inference architecture for large language models, where the expensive high-performance accelerators handle non-attention operations, while a swarm of lower-end resources manages attention. Through extensive experiments, we demonstrated the end-to-end performance improvements of Glinthawk, its scaling characteristics, cost-effectiveness vs. well-known baselines, and resilience to network conditions including inter-tier latency.

While our prototype has proven successful, several new directions warrant exploration. First, alternative parallelization strategies for Tier-1 nodes, such as tensor parallelism, need to be examined to better understand their implications in Glinthawk's architecture. Second, our analysis suggests that the ideal devices for Tier-2 would be small compute nodes with enough memory and compute for one prompt. However, the hardware and network design of these nodes is as of yet unclear. We invite the community to further explore the implications of this architecture on designing the next generation of accelerators for LLM inference.

References

[1] cuBLAS — developer.nvidia.com. https://developer.nvidia.com/cublas. [Accessed 18-09-2024].

[2] GitHub - flashinfer-ai/flashinfer: FlashInfer: Kernel Library for LLM Serving — github.com. https://github.com/flashinfer-ai/flashinfer. [Accessed 18-09-2024].

[3] GitHub - NVIDIA/TensorRT-LLM at release/0.5.0 — github.com. https://github.com/NVIDIA/TensorRT-LLM/tree/release/0.5.0?tab=readme-ov-file. [Accessed 07-02-2024].

[4] GitHub - vllm-project/vllm: A high-throughput and memory-efficient inference and serving engine for LLMs — github.com. https://github.com/vllm-project/vllm. [Accessed 18-09-2024].

[5] Introducing Low-Level GPU Virtual Memory Management | NVIDIA Technical Blog — developer.nvidia.com. https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/. [Accessed 18-09-2024].

[6] Long context | Generative AI on Vertex AI | Google Cloud — cloud.google.com. https://cloud.google.com/vertex-ai/generative-ai/docs/long-context. [Accessed 18-09-2024].

[7] NVIDIA Collective Communications Library (NCCL) — developer.nvidia.com. https://developer.nvidia.com/nccl. [Accessed 18-09-2024].

[8] openmp.org. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. [Accessed 18-09-2024].

[9] ShareGPT: Share your wildest ChatGPT conversations with one click. — sharegpt.com. https://sharegpt.com/. [Accessed 18-09-2024].

[10] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[11] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association.

[12] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. SARATHI: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

[13] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[14] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. DeepSpeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022.

[15] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[17] Shaoyuan Chen, Yutong Lin, Mingxing Zhang, and Yongwei Wu. Efficient and economic large language model inference with attention offloading, 2024.

[18] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing GPT-4 with 90%* ChatGPT quality, March 2023.

[19] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning, 2023.

[20] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness, 2022.

[21] Abhimanyu Dubey et al. The llama 3 herd of models, 2024.

[22] Akshay Goel, Almog Gueta, Omry Gilon, Chang Liu, Sofia Erell, Lan Huong Nguyen, Xiaohong Hao, Bolous Jaber, Shashir Reddy, Rupesh Kartha, Jean Steiner, Itay Laish, and Amir Feder. Llms accelerate annotation for medical information extraction. In Stefan Hegselmann, Antonio Parziale, Divya Shanmugam, Shengpu Tang, Mercy Nyamewaa Asiedu, Serina

Chang, Tom Hartvigsen, and Harvineet Singh, editors, *Proceedings of the 3rd Machine Learning for Health Symposium*, volume 225 of *Proceedings of Machine Learning Research*, pages 82–100. PMLR, 10 Dec 2023.

[23] Roger W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, 1994.

[24] Ke Wen James Reed, Pavel Belevich. PiPPy: Pipeline parallelism for pytorch. https://github.com/pytorch/PiPPy, 2022.

[25] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput LLM inference with shared prefixes, 2024.

[26] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention, 2023.

[27] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving, 2023.

[28] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, Shen Li, Zhigang Ji, Tao Xie, Yong Li, and Wei Lin. Infinite-LLM: Efficient LLM service for long context with DistAttention and distributed kvcache, 2024.

[29] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration, 2024.

[30] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023.

[31] Xiaodong Liu, Kevin Duh, Liyuan Liu, and Jianfeng Gao. Very deep transformers for neural machine translation, 2020.

[32] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. CacheGen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 38–56, New York, NY, USA, 2024. Association for Computing Machinery.

[33] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. Can foundation models wrangle your data?, 2022.

[34] Newegg. Newegg Electronic Store. https://www.newegg.com. [Accessed 19-09-2024].

[35] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, mar 2008.

[36] OpenAI. OpenAI Batch API. https://platform.openai.com/docs/guides/batch. [Accessed 18-09-2024].

[37] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting, 2024.

[38] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 606–624. Curan, 2023.

[39] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[40] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models, 2020.

[41] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing billion-scale model training, 2021.

[42] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[43] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a Single GPU, 2023.

[44] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: training multi-billion parameter

language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[45] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[46] Gemma Team. Gemma: Open models based on Gemini research and technology, 2024.

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models, 2023.

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

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[50] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 23–29 Jul 2023.

[51] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024.

[52] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[53] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on scaling fully sharded data parallel, 2023.

[54] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[55] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning, 2022.

[56] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang, 2023.