Aqua: Network-Accelerated Memory Offloading for LLMs in Scale-Up GPU Domains

Abhishek Vijaya Kumar
Cornell University
Ithaca, USA
abhishek@cs.cornell.edu

Gianni Antichi
Politecnico di Milano
Milan, Italy
gianni@polimi.it

Rachee Singh
Cornell University
Ithaca, USA
rachee@cs.cornell.edu

Abstract

Inference on large-language models (LLMs) is constrained by GPU memory capacity. A sudden increase in the number of inference requests to a cloud-hosted LLM can deplete GPU memory, leading to contention between multiple prompts for limited resources. Modern LLM serving engines deal with the challenge of limited GPU memory using admission control, which causes them to be unresponsive during request bursts. We propose that preemptive scheduling of prompts in time slices is essential for ensuring responsive LLM inference, especially under conditions of high load and limited GPU memory. However, preempting prompt inference incurs a high paging overhead, which reduces inference throughput. We present Aqua, a GPU memory management framework that significantly reduces the overhead of paging inference state; achieving both responsive and high throughput inference even under bursty request patterns. We evaluate Aqua by hosting several state-of-the-art large generative ML models of different modalities on servers with 8 Nvidia H100 80G GPUs. Aqua improves the responsiveness of LLM inference by 20× compared to the state-of-the-art. It improves LLM inference throughput over a single long prompt by 4×.[1]

CCS Concepts: **• Computing methodologies → Machine learning**; **• Networks → Data center networks**; **Data path algorithms**.

Keywords: Large language models; Generative AI; Paging; Virtual memory; GPU interconnects; Memory offloading;

ACM Reference Format:
Abhishek Vijaya Kumar, Gianni Antichi, and Rachee Singh. 2025. Aqua: Network-Accelerated Memory Offloading for LLMs in Scale-Up GPU Domains. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3676641.3715983

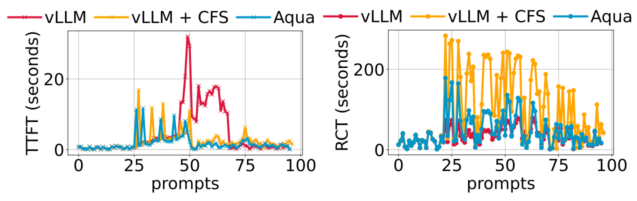
[1]Aqua's code is available at https://github.com/aquaml.

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1079-7/2025/03.
https://doi.org/10.1145/3676641.3715983

1 Introduction

Recent years have seen an explosive growth in popularity of generative machine learning (ML) models for a multitude of tasks ranging from text summarization to multimedia content creation [15, 38, 49, 53]. These models are typically hosted on virtualized multi-GPU cloud servers [9, 16, 47, 52], which interconnect a handful of ML accelerators on the board of the server with high-bandwidth links [42].

Inference is memory intensive. Generative ML models have a massive memory footprint [5–7, 40]. For example, hosting the Llama-405B model consumes 800 gigabytes of GPU high-bandwidth memory (HBM). Moreover, large-language models (LLMs) generate a key-value (KV) cache while processing input prompts using the attention mechanism [61]. The KV cache grows quadratically with the length of an input prompt and its generated sequence, taking several gigabytes of GPU memory per prompt [35]. Thus, a sudden increase in the number of inference requests to the LLM can deplete the GPU memory, leading to memory contention as multiple prompts compete for GPU memory.



(a) TTFTs **(b)** RCT

Figure 1. Responsiveness (measured using time-to-first-token or TTFT) and throughput (measured using request completion time or RCT) of inference queries on LLMs. Since vLLM batch processes queries, it has low RCT (high throughput) but high TTFT (low responsiveness). Fair-scheduling queries improves responsiveness but paging overheads dominate RCT. Aqua reduces paging overheads by offloading memory over high-speed multi-GPU interconnects (*e.g.,* Nvlinks), achieving responsive inference with low RCT.

Managing bursts in inference requests with queueing. Modern LLM serving engines deal with the challenge of limited GPU memory using admission control [35, 54]. They schedule a batch of prompts for inference and queue the remaining until free GPU memory is available. However, this paradigm of batch processing with admission control, is susceptible to head-of-line blocking [21] — a situation where processing a batch of prompts delays responses to other prompts that await their turn in the queue.

Queueing leads to unresponsiveness. We experimentally demonstrate that queuing prompts can severely degrade the responsiveness of an LLM. We serve an LLM on an Nvidia A100 GPU with 80GB HBM using the popular serving engine vLLM [35]. We initially issue 25 prompts to vLLM, then double the request rate for one minute. This surge of requests exhausts vLLM's KV cache capacity, resulting in subsequent prompts being queued. This causes *time-to-first-token* (TTFT), a measure of how quickly the model starts responding, to spike by 4X (red line in Fig. 1a). A different approach to handling bursts is to autoscale the model across additional GPUs [59]. However, scaling out can take several minutes due to virtual machine startup delays, during which the model will be unresponsive. Since even 3 seconds of unresponsiveness causes users to abandon services [27], delays of tens of seconds increase the risk of losing users.

Our proposal: preemptable inference. In this work, we posit that preemptive scheduling of prompts in time slices is essential for achieving responsive LLM inference, particularly under high load and limited GPU memory. Under this scheduling scheme, each prompt will receive time slices of GPU compute, allowing the model to stay responsive. A preemptive scheduler will stop execution of the current prompt at the time slice boundary and page out its relevant state from GPU memory to make space for executing the next prompt. Preemptive scheduling with time slices will also unlock the door to implementing fine-grained fair scheduling in existing inference serving systems, which only support coarse-grained fairness through admission control [54].

Preemption is slow due to paging overheads. However, preempting prompt inference requires paging the prompt's state out of GPU memory at the boundary of time slices. If the inference state of a prompt is paged to host DRAM [35, 55], transferring it back to GPU memory incurs a high overhead. The overhead of transferring state between DRAM and GPU memory is lower-bounded by the bandwidth of Peripheral Component Interconnect Express (PCIe) connections. In fact, we implement a preemptive fair scheduler in vLLM and find that while preemptive scheduling improves the time-to-first-token in vLLM (orange line Figure 1a), it causes request completion time or the time to finish inferring a prompt, to increase by more than $2X$ (orange line in Figure 1b).

Aqua: preemptable inference with low overhead. In this work, we build Aqua, a memory management framework for GPUs, that significantly reduces the overhead of paging inference state in and out of GPU memory to achieve efficient preemptive prompt scheduling. Reduced paging overheads enable Aqua to get the best of both worlds — responsive inference (Figure 1a) for individual prompts while still achieving high inference throughput (Figure 1b).

Aqua's key insight. Inference on ML models can be limited by memory capacity, memory bandwidth or compute, depending on the model type (*e.g.,* vision vs. text) and the inference query load [55, 59, 69]. As a result, at any time, some inference jobs in a cloud cluster underutilize GPU memory, while others face memory bottlenecks. Using this insight, Aqua decouples memory allocation from compute allocation by enabling GPUs to share spare memory with those that need more. Aqua limits these memory offloads to GPUs within a *scale-up domain* in cloud datacenters. GPUs in a scale-up domain are connected via high-bandwidth Nvlinks, allowing Aqua to achieve faster access to spare memory than traditional host DRAM over PCIe, significantly reducing preemption and paging overheads. In modern datacenters, multi-accelerator servers define a scale-up domain. Recently, the scale-up domain has expanded to encompass entire racks, like in Nvidia Blackwell's 72-GPU architecture, where high-bandwidth Nvlinks interconnect all 72 GPUs. These advancements extend the benefits of Aqua's decoupled memory allocation to entire racks [41].

Technical contributions. Aqua has three components: Aqua-profiler profiles the characteristics of incoming inference load and GPU memory utilization of hosted ML models in a deployment. Using this profile, Aqua-profiler labels models as producers or consumers and feeds this information to Aqua-placer. Producers do not fully utilize GPU memory and can offer the excess to other models. Consumers are bottlenecked on GPU memory and need more. Aqua-placer places models such that producers are paired with appropriate consumers within the same Nvlinks-domain. Finally, Aqua-lib is a library that dynamically manages tensors offloaded from consumers to producers' GPU memory.

Preemptive fair scheduling with low overheads. Using Aqua to manage memory between GPUs, we build a preemptive completely fair scheduler [60] for prompt inference. We design a new algorithm to partition the batch between prompts in different inference phases to achieve fairness while retaining throughput optimizations [4]. Due to Aqua-enabled fast access to spare memory, the overhead of paging is minimal. Fair scheduling (1) achieves fair allocation of compute resources to all prompts, and (2) allows the model to stay responsive even under high load and limited GPU memory. We note that fair scheduling can absorb temporary bursts of queries or manage sustained bursts until autoscaling can spawn new instances of the model.

Results. We evaluate Aqua by hosting several state-of-the-art large generative ML models of different modalities (*e.g.,* text, audio, vision) on a server with 8 cutting-edge Nvidia H100 80G GPUs. Using a representative distribution of these ML models and inference workloads, we show that Aqua improves the responsiveness of LLM inference, measured using time-to-first-token, by 20× compared to the state-of-the-art. Aqua improves the inference throughput over a single long prompt by 4× on memory constrained inference [55, 69].

2 Background and motivation

ML inference tasks are now critical for many organizations [49]. End users often host ML models on cloud servers, issuing queries without deploying their own hardware [16, 52]. Generative models can produce various media, including text [8], images [51], and audio [34]. LLMs, like ChatGPT, are particularly notable for their text generation capabilities.

2.1 Infrastructure for ML

At its core, ML inference entails performing several large matrix multiplications. Therefore, ML inference uses special-purpose accelerators like GPUs and TPUs, optimized for fast matrix multiplications [33]. In addition to the number of floating point operations (FLOPs) supported by an ML accelerator, the memory on the accelerator has become crucial in enabling fast ML training and inference due to the rapid growth in the size of ML models. Generative ML models have trillions of parameters, resulting in a large memory footprint. To meet the memory footprint of models, GPUs are equipped with tens of gigabytes of *high bandwidth memory* or HBM. Typically, GPUs are peripherals connected to a host server using PCIe. The host machine has a CPU and DRAM for general-purpose computation [43–45].

Server targets for ML. Multi-accelerator servers (*e.g.,* Nvidia DGX [46], Cerebras WSE [22], Intel Gaudi [32]) have become building blocks of larger cluster and datacenter deployments used for ML training and inference. A multi-accelerator server consists of a handful of ML accelerators, connected with high-speed on-board electrical interconnects (*e.g.,* Nvidia NVlink citenvlink, Google ICI [33]) that support up to hundreds of gigabytes per second of bandwidth between pairs of accelerators. Cloud operators connect racks of multi-accelerator servers into datacenter-scale deployments using a network fabric [33, 58].

Bandwidth asymmetry in ML clusters. The interconnect between GPUs on the same server is specialized for high bandwidth at several hundred gigabytes/second [42], while the connection between GPUs and the host (*e.g.,* CPU, DRAM) relies on general-purpose PCIe (50GB/s), which is significantly slower than Nvlinks in modern deployments.

2.2 Memory contention during LLM inference

LLMs process inference queries, or *prompts*, by converting them into tokens, which represent subword units. Each token is encoded as a multi-dimensional embedding vector that captures its meaning [24, 39]. During inference, these token embeddings are fed into the LLM. Given a user prompt, the LLM predicts the most probable next token in the vocabulary [20] via the attention mechanism [61], which computes *key* and *value* vectors for each token by multiplying its embedding with learned key and value matrices [24, 39]. Once the most likely token is generated, it is appended to the sequence, which includes both the prompt and previously generated tokens, and the process repeats.

The key and value vectors of all tokens in this sequence, known as the inference context, are cached for efficient future predictions. The key and value vectors of simultaneously processed sequences are stored in a key-value (KV) cache within the GPU memory [35]. The KV cache size is determined by the sequence length and number of sequences, growing with the number of generated tokens. The KV cache size is limited by the available GPU memory after loading the model weights on the GPU. Memory contention arises when the KV cache exceeds the available GPU memory.

In fact, even a single prompt can experience starvation during inference when the model is too large for the GPUs, or the combined memory of the prompt's inference context (KV cache) and model weights exceeds GPU memory. As model sizes and supported prompt lengths grow rapidly (*e.g.,* Llama-405B and Gemini's 1M prompt), the problem will worsen. This has prompted recent efforts to page context from the GPU to DRAM and load it only when necessary [55, 64, 69].

Memory contention hinders caching optimizations. Recently, LLM serving engines are caching the inference context of long common prefixes of prompts [13, 25]. As incoming prompts consume memory, the engine must discard the cached context from the GPU, rendering the optimization useless. Instead, the only choice the engine has is paging out the inference context to the DRAM or remote memory [37].

2.3 Existing ways of mitigating memory contention

One approach towards managing memory contention during LLM inference is to horizontally scale the number of GPUs hosting the model when a burst of inference requests arrives [59]. However, spawning model instances on more GPUs incurs startup delays of several minutes [11, 59]. These delays limit the utility of the scale out approach to requests that are already admitted and queued, as they will experience unresponsiveness until more resources are provisioned.

Paging suffers from bandwidth bottleneck. Paging resolves the issues arising from memory contention but suffers from poor performance because GPUs are idle while they wait for the paged data. The fundamental bottleneck in paging is the limited bandwidth between the GPU and the DRAM which is constrained by the PCIe bandwidth. While higher PCIe bandwidth may mitigate this problem, research has shown that the growth rate of PCIe bandwidth has been slow [26]. Thus, current inference serving engines that offload dynamic context to host DRAM must trade-off performance for responsiveness. In fact, to demonstrate this, we implement a completely fair scheduler in vLLM and find that while fair scheduling improves the time-to-first-token in vLLM (orange line Figure 1a), it causes request completion time or the time it takes to finish inferring a prompt, to increase by $\approx 50\%$ (orange line in Figure 1b).

3 Aqua Design

When inference load increases, modern serving systems take significantly longer to respond because of GPU memory capacity contention. Figure 1 shows that batch processing prompts with admission control [35, 54] queues prompts until the current batch of prompts finishes execution on the GPU, leading to head-of-line blocking delays. Serving can be unresponsive even when the inference load has a single prompt, when the GPU memory is insufficient for the model or the KV cache and model weights together exceed the available GPU memory [55]. Aqua ensures responsive inference under GPU memory capacity contention by reducing preemption overhead through fast paging and implementing a preemptive scheduler that uses fast paging.

Fast preemption with Aqua. Multi-accelerator servers in ML clusters are allocated to inference jobs executing a variety of ML models. The inference jobs have different GPU compute core, memory bandwidth and memory capacity utilization based on the model (*e.g.,* image vs text generation). GPUs with certain inference jobs and workloads are bottlenecked by compute with significant spare memory capacity, while other jobs are bottlenecked by memory capacity ([57],§4). We leverage this insight to decouple the memory and compute allocation on GPUs for inference jobs. This enables us to allocate memory on GPU with spare memory to another job bottlenecked by memory capacity on a different GPU. This allows the memory-constrained job to access additional memory via a high-speed GPU-GPU interconnect, such as Nvlinks, which offers an order of magnitude higher bandwidth compared to PCIe.

Aqua addresses bursts of inference requests in cloud-hosted LLMs by ensuring responsive inference (low time-to-first-token) and maintaining high throughput (low request completion time). Aqua achieves the goal of responsive and high throughput inference even under bursts by implementing preemptive scheduling with low paging overhead (§7). Aqua's fast preemption naturally benefits memory constrained inference systems reliant on paging [55, 64, 69] and also advanced caching optimizations [13, 25]. In §9, we show that Aqua also improves the performance of these systems in addition to achieving responsiveness. Aqua has three main components (Figure 2):

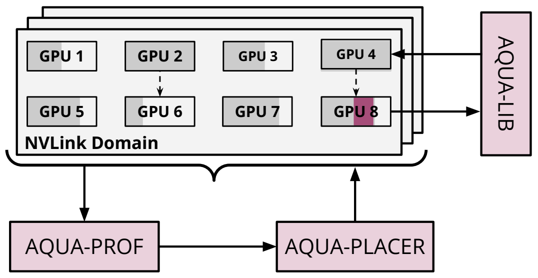


Figure 2. Design of Aqua.

1. Aqua-profiler (§4). Aqua-profiler profiles the characteristics of inference workloads towards any hosted model and the corresponding GPU memory utilization. Based on the profiled information, Aqua-profiler decides if a GPU has excess memory or needs more GPU memory to sustain its inference load. We term GPUs with excess memory as *producers* and those that need memory as *consumers.*

2. Aqua-placer (§5). Instead of relying on the chance existence of free GPU memory nearby, we develop an algorithm, Aqua-placer, to place models in clusters to maximize the opportunity to offload memory over the fast inter-GPU network. Aqua-placer places models on GPUs such that memory-bound ML models, *i.e.,* consumers, are hosted in proximity of memory-rich ML models, *i.e.,* producers.

3. Aqua-lib (§6). Inference workloads can change over time which means both the demand and supply of GPU memory during inference is elastic. This dynamic environment makes it hard to offload inference state to a different GPU that may require the memory back when it experiences high load of inference queries. To deal with this, we develop Aqua-lib, a memory management framework that exposes a new Aqua Tensors abstraction. Aqua Tensors are elastic — they allow producer GPUs to transparently re-claim memory they had exposed for offloads from memory consumer GPUs.

4 Aqua-profiler: profile memory consumption

Aqua-profiler profiles the current GPU utilization of inference jobs with the workload they serve in the steady state. Using this information, Aqua-profiler estimates if the GPU is a producer or a consumer. We call ML models on GPUs with spare memory capacity as producers and jobs that are bottlenecked by memory capacity as consumers.

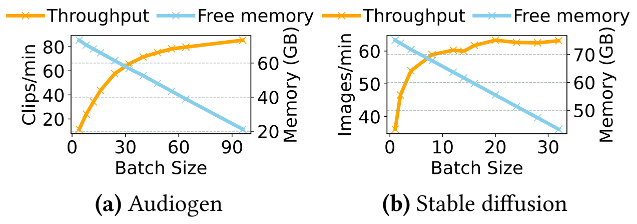
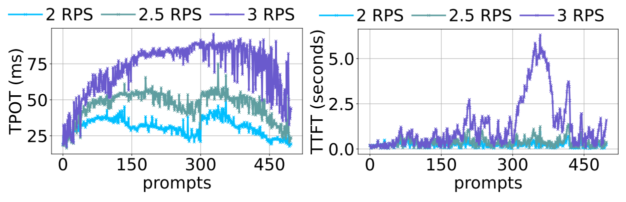


Figure 3. The plots show that when audio and image generation models reach the plateau of their throughput, there are 10s of GBs of free memory on an A100 80GB GPU. In fact, increasing the batch-size beyond a point results in diminishing increase in throughput.

Our experiments show that different generative ML models contend for different resources (*e.g.,* compute vs. memory). Specifically, vision and audio generative models have ample memory to spare even while operating at peak inference throughput (Figure 3). Since we need to estimate the size of free memory at any time, Aqua-profiler measures the free memory by increasing the batch size because the memory consumption of the model increases with batch size. We increase the batch size of inference until the model throughput saturates to measure the highest memory the model consumes. Since partial responses in images and audio offer limited benefits, increasing the batch size further is unnecessary. We consider a model a producer if free memory at peak throughput on the GPU is above a threshold (*e.g.,* 10 GB).

4.1 Can LLMs be producers of memory?

Our strategy of increasing inference batch size till throughput saturates is ineffective in determining the true memory requirement for LLMs. This is because LLM inference benefits from partial responses, especially in interactive use cases. Moreover, large batch sizes negatively impact the responsive time service-level agreements (SLA) of LLMs [4]. Therefore, we develop an alternate approach to classify LLMs.



(a) TPOT on ShareGPT **(b)** TTFT on ShareGPT

Figure 4. 4a shows the time per output token (TPOT) for the Llama 3.1 8B model on an 80 GB A100 GPU. TPOT increases with request rate. 4b illustrates the time to first token (TTFT), rising with the queue length from incoming requests.

Inference on an LLM has two main stages: the prefill stage and the decoding stage. In the prefill stage, key and value vectors are calculated for each token in the prompt, and self-attention scores are computed for every token with all previous tokens [61]. The output of the prefill stage is a vector per token, which is then multiplied by a matrix to generate the layer's output, preserving the input dimensions for the next layer. In the decoding stage, new tokens are generated with each forward pass. Unlike the prefill stage, where multiple vectors are processed, the decoding stage handles only one vector per prompt.

Recent work has proposed inference schedulers that batch the prefill stage of some input prompts with the decode stage of other prompts for overall efficiency of inference [4]. The optimal batch size, or chunk size, depends on the required SLA of the hosted LLM. Serving engines batch as many decode tokens as possible, supplementing the batch with tokens from new prompts when necessary.

We observe that GPUs hosting LLMs often have free memory due to strict responsiveness SLAs, as batching is limited by chunk size. We empirically demonstrate this using SLA requirements for interactive use cases from recent work [4], specifying a 100ms P99 Time Between Tokens (TBT) and under 1 second Time to First Token (TTFT). We also adopt their recommended chunk size of 512 for the ShareGPT [35] dataset to meet these SLAs.

Challenge. Given the SLA, chunk size, and dataset, Aqua-profiler searches for the highest request rate that meets the SLA since it leads to maximum memory consumption. Sweeping the entire request rate space is infeasible and challenging. To overcome this, we first perform a binary search within a range (*e.g.,* 1-10 RPS) followed by a linear search with 0.5 RPS increments and decrements to find local maxima. Using this, Aqua-profiler outputs the available memory at the highest traffic rate the LLM can sustain within the SLA.

In Figure 4, the Llama 3.1 8B model handles 2.5 RPS without violating the TTFT SLA. At 3 RPS, the queue size increases, resulting in a TTFT of 5-6 seconds (Figure 4b), which is unacceptable. Thus, Aqua-profiler reports the available memory (40 GB) at 2.5 RPS as free memory for this model and if this value exceeds a threshold (e.g., 10 GB), we classify the model as a producer.

Unpredictable loads. Unlike image generation models where the memory consumption of every data point in a batch is the same, the memory consumption of each prompt in a batch for LLM inference can vary significantly based on the length of the input and output tokens. Aqua-profiler samples requests for profiling using input and output token distributions from individual LLM deployments [17]. Its free memory estimation remains accurate as long as request patterns follow the historical distribution. Even with extensive profiling, a GPU's memory utilization fluctuates with unpredictable workload changes. To handle this, Aqua swiftly reclaims memory and returns it to the producer (§ 6).

Profling consumers. If the model is not a producer, Aqua-profiler assesses its need for paging and preemption. If the model has allocated swap space using Aqua-lib, the model is classified as a consumer, and Aqua-profiler determines the required swap space. For memory-constrained inference and prefill caching, this space is computed based on the number of prompts, the longest sequence required, and the memory needed for inference context. Aqua-profiler empirically determines the swap space required for CFS through artificial bursts of varying sizes (e.g., 2X, 3X the steady-state requests per second) (§ 9.4). If there is no need for swap space, we treat the model as a producer with 0 memory to offer.

Corner case performance. In corner cases, when all models are consumers of memory, Aqua defaults to using DRAM for paging, matching the performance of baselines like vLLM and FlexGen since they page to DRAM over PCIe.

4.2 Aqua in future datacenters

Recent trends in ML infrastructure suggest that Aqua's techniques are well-suited for modern datacenters. Aqua targets environments where multiple ML jobs co-exist within high-bandwidth interconnect domains like the Nvlinks/NVSwitch fabrics. These domains are growing in size, with examples like Nvidia's NVL72 racks (72 GPUs interconnected with NVSwitch) [41] and Google's TPU racks (64 TPUs per block) [33, 62]. The large size of these scale-up architectures increases the likelihood of diverse ML workloads co-existing within the same high-bandwidth domain. Diverse workloads enable Aqua to match a higher number of producers with consumers compared to smaller multi-GPU servers.

5 Aqua-placer: optimal model placement

Aqua-lib's ability to offload tensors on interconnected GPUs relies on the availability of free memory on these GPUs. Aqua-placer maps ML models for inference to GPUs in a cluster of servers, with the goal of maximizing opportunities to offload Aqua Tensors on interconnected GPUs. Mapping a single producer to multiple consumers is feasible but Aqua-placer does not allow that by design and maps one producer to one consumer. Sharing a producer with multiple consumers may oversubscribe NVlink bandwidth of the producer GPU, reducing the benefits of offloading memory over the inter-GPU network.

Differences from stable matching. Modeling the placement as a simple stable matching formulation will assume that any producer can be matched with any consumer. However, in the context of Aqua, we can only pair producers to consumers that are connected by the same fast server-scale inter-GPU network (*e.g.,* NVlinks or ICI). Aqua-placer solves the model assignment problem in two steps. First, it assigns models to multi-GPU servers and then within each server, it matches producers to consumers using simple stable matching. We encode the first step as an optimization to produce optimal model to server mappings.

Inputs and outputs. The optimization takes as input the number of servers in the cluster S, number of GPUs per server G, the number of models N and the total high-bandwidth memory capacity of each GPU G_{mem}. We make a simplifying assumption that all GPUs in a server have the same hardware specifications (*e.g.,* compute cores and memory) inspired from the current generation multi-GPU servers [43, 46]. A key input is the memory requirement of each model R_m, which we determine using Aqua-profiler. The model's memory requirement is positive if it is a producer and negative if it is a consumer. For example, Audiogen in Figure 3a will have $R_m = 20GB$. The output of the encoding is an indicator variable $x_{m,s}$ that is 1 if model m is mapped to server s. To handle distributed serving, we assign indicator variables $x_{i,s}$ to each model shard i. P is a list of mappings, each storing a set of all the tensor parallel shards of a model.

Allocation of models to GPUs is subject to the constraints:

One model per GPU. Each model must map to one server:

$$\sum_{s=1}^{S} x_{m,s} = 1, \quad \forall m \in M \tag{1}$$

Tensor parallel shards are on the same server. Tensor parallel (TP) shards are typically placed on the same server and pipeline parallel shards on different servers [4, 63]. Accordingly, we map all TP shards of a model to a single server.

$$x_{p[j],s} == x_{p[j+1],s} \quad \forall j \in \{0 \ldots len(p) - 1\}, \forall p \in P \tag{2}$$

Models limited by number of GPUs per server. In Aqua, an ML model shard is placed on exactly one GPU. If one shard maps to a single GPU, the number of shards mapped to a server should not exceed the number of GPUs on the server:

$$\sum_{m=1}^{M} x_{m,s} \leq G \quad \forall s \in \{1 \ldots S\} \tag{3}$$

Balanced demand and supply of memory. We define mem_s as the sum of the memory requirements (R_m) of models mapped to a server. Recall that R_m can be positive (for a consumer) or negative (for a producer). Thus, mem_s captures the net memory available on the server after mapping models to it. mem_s will be positive if there is excess memory or negative if there is a deficit.

$$mem_s = \sum_{m=1}^{M} x_{m,s} \cdot R_m \quad \forall s \in \{1 \ldots S\} \tag{4}$$

Balanced number of consumer and producers. Despite keeping the memory usage on the server balanced using equation 4, the encoding can find solutions that map one producer and multiple consumers to the same server. We count the net number of models mapped to a server, where a consumer counts for −1 and a producer counts for +1 in Eq 5. We find solutions where eq_s is close to 0 to prevent producer bandwidth oversubscription.

$$eq_s = \sum_{m=1}^{M} x_{m,s} \cdot t \quad \forall s \in \{1 \ldots S\}, \quad t = \frac{|R_m|}{R_m} \tag{5}$$

Objective. Aqua-placer's goal is to ensure that models are mapped to servers such that there is no waste of GPU memory across servers (Eq 4) and each server has a balanced number of consumers and producers (Eq 5). We achieve this by picking the max of both memory utilization (mem_s) and number of models (eq_s) on a server and minimize their largest sum. To make the two parts of the objective have the same units, we multiply eq_s with the server's GPU memory:

$$\max_s(mem_s) + G_s \cdot \max_s(eq_s) \tag{6}$$

We encode the algorithm using a commercial optimization solver, Gurobi [31] and find that it reaches solutions within 2% of optimal in 5 seconds for clusters with up to 128 GPUs.

How frequently do we run Aqua-placer? When producer workloads change, Aqua-lib rapidly reclaims memory for the producer, limiting consumers' access to fast paging. While this prevents performance degradation for producers, the state of the cluster itself might not be optimal to maximize offloading opportunities. Infrastructure providers must thus trade off between maintaining suboptimal offloading or rerunning Aqua-placer to remap jobs with migration for better sharing opportunities. This trade-off is not explored in this paper and is left for future work.

6 Aqua-lib: elastic memory management

Aqua-lib implements a novel abstraction for *migratable* tensors in GPU memory, called Aqua Tensors. Aqua Tensors can be offloaded on to the memory of other GPUs using

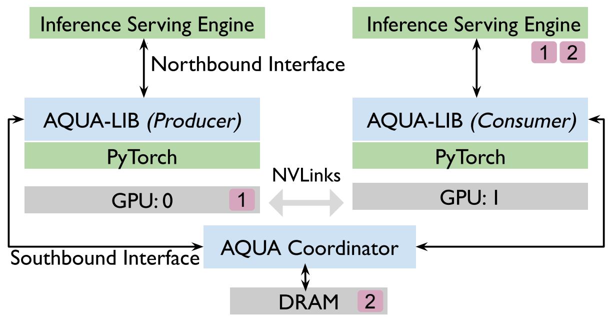


Figure 5. Design of Aqua. GPU 1 is hosting a consumer. Aqua-lib is aware that GPU 0 is a producer. Aqua-lib allows the model on GPU 1 to allocate Aqua Tensors that are offloaded to GPU 0's HBM (shown in a pink box with number 1). If GPU 0 only has enough memory to offload one tensor, Aqua-lib falls back to the host DRAM (shown using a pink box with number 2).

the fast inter-GPU network. Internally, Aqua-lib decides when Aqua Tensors should be migrated and to which GPU's memory, based on the memory utilization and inference workload on all other GPUs in the server. The ML model developer is oblivious to changes in the physical location of Aqua Tensors. We integrate Aqua with two popular ML serving engines, vLLM [35] and FlexGen [55], and the integration does not need big changes to their codebase (§8).

Aqua-lib's interfaces. An instance of Aqua-lib runs on each GPU of a multi-GPU server (Figure 5). Aqua-lib is initialized with the profile generated by Aqua-profiler. Aqua-lib uses this information to decide if the GPU has spare memory to be a producer. Aqua-lib implements a *northbound* and a *southbound* interface. Serving engines use northbound interface to interact with Aqua-lib. Using the northbound interface, serving systems also share their metrics like inference load (e.g., requests per second) and size of inference context with Aqua-lib at a tunable frequency. If the GPU is a producer and the serving engine reports an increase in inference load, Aqua-lib will re-claim the offloaded memory. Aqua-lib will offer the memory back to consumer when the load returns to the steady state value determined during profiling by Aqua-profiler. When Aqua offers memory from a producer or reclaims it back, it uses the northbound interface to inform the inference serving engine of how much memory it has after the event. The southbound interface enables Aqua-lib to communicate with a centralized thread-safe data store maintained by the Aqua *coordinator*. The coordinator tracks *requests* for memory offloads from consumer GPUs and memory *offers* from producer GPUs.

Central coordinator. The central coordinator keeps track of *consumers* and *producers* of HBM. Consumer GPUs need to offload tensors to the HBM memory of any potential producer GPUs. The coordinator program exposes a set of REST endpoints. Aqua-lib uses these REST endpoints to register requests for memory from consumer GPUs and offers of free memory from producer GPUs. The coordinator also exposes REST endpoints that allow producer GPUs to re-claim memory when they need it back. A GPU may start as a memory producer, but it may reclaim all the offered memory if the characteristics of its inference workload change.

Allocating Aqua Tensors. An ML model imports Aqua-lib and initializes it with the profile generated by Aqua-profiler. ML models use Aqua-lib to instantiate Aqua Tensors. Aqua-lib uses the southbound interface to register this allocation request with the coordinator via the appropriate REST endpoint (§8). The coordinator keeps track of GPUs that may have registered to be producers. Selecting which GPU will be the producer for a consumer GPU is explicitly done by the Aqua-placer (§5) before the model starts to execute on the consumer GPU. The coordinator returns a reference of the offloaded tensor location to the consumer's Aqua-lib. We note that if no producer GPUs exist in the system, Aqua-lib falls back to using the DRAM for offloading tensors, just like previous work [35, 55]. Freeing allocated tensors follows a similar logic where the consumer's Aqua-lib instance informs the coordinator which centrally manages bookkeeping of Aqua Tensors.

Reclaiming Aqua Tensors. On producer GPUs, Aqua-lib polls the northbound interface to find the current memory utilization and workload statistics of the GPU. Using this, Aqua-lib allows producers to reclaim their memory back from offloads when the inference load on the producer GPUs increases. This occurs when Aqua-lib finds that the workload reported by the inference serving engine has increased. Aqua-lib uses the REST API to let the coordinator know that the producer GPU would like the memory back.

Aqua-lib control loop. On a consumer GPU, Aqua-lib's control logic tests once every few inference iterations if any offloaded Aqua Tensors need to be freed up. Doing this check requires Aqua-lib calling the coordinator's REST API. This check ensures that if a producer has registered a request to reclaim its memory, consumers can respond to it in a timely manner. We note that the centralized coordinator's data store is thread-safe to prevent unexpected behavior when multiple GPUs make changes. In our design the overheads of Aqua-lib are very low since communication with central coordinator is infrequent – only once per a configurable number of inference iterations. We describe further implementation details of Aqua-lib in §8.

7 Fair scheduling prompts with Aqua

Default behavior of the scheduler in today's inference serving systems is to admit new requests only if there is enough GPU memory available for its inference context [4, 35]. Schedulers like VTC [54] provide fairness at a coarse granularity with admission control to avoid users with a high query rate consuming all the compute cycles. However, such coarse grained admission control can still lead to starvation when there is a surge in the number of legitimate users, beyond the serving capacity of the allocated hardware. If the GPU memory is fully occupied by prompts already being inferred, any additional requests need to *wait* or *starve* until enough space becomes available in the GPU memory. Even with auto-scaling, prompts can starve because provisioning new VMs incurs delays in the order of minutes [11].

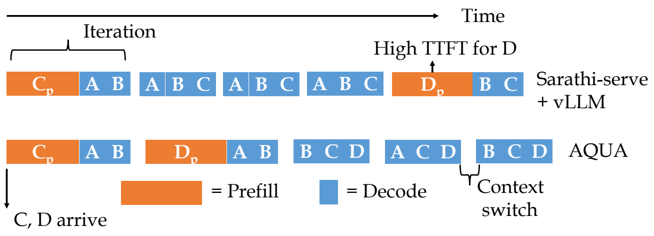


Figure 6. CFS enabled by AQUA during memory contention where only 3 out of the 5 prompts can fit on a GPU at a time.

Batching vs. timesharing for scheduling inference. Recall that state-of-the-art scheduling algorithms batch chunks of prefill computation with decoding computation of other prompts to increase compute efficiency (§ 4). The algorithm prioritizes decodes while batching prompts, which then leads to starvation of a subset of prompts when multiple prompts arrive simultaneously. In, Figure 6, C,D, arrive at the same time but D starves till A completes execution because the GPU's memory can accommodate only 3 prompts at a time. This contrasts with the behavior of operating systems that use context-switching to manage constrained resources under load with a completely fair scheduler (CFS) [60]. Context-switching has two steps: *preemption*, which saves the current process's state (registers, page table, *etc*) to DRAM, and *loading*, which loads the next process's state into the CPU.

Challenge. Naively applying time-slicing from OS scheduling, which prioritizes processes that have received the least compute time, undermines recent chunked-prefill optimizations [4]. These optimizations leverage the fact that prefill is compute-bound while decoding is memory-bandwidth-bound, allowing prefill and decoding tokens to be batched together, effectively providing "free" decoding. Time-sharing GPUs based on the least number of generated tokens would only batch prefill prompts, as they have zero generated tokens, negating this benefit.

AQUA's batch partitioning algorithm. In AQUA, we achieve fair scheduling while preserving the benefits of chunked-prefill. Given a batch size b, we partition it into p prefill tokens and d decode tokens. Our key insight to partition is to set d to its upper bound, corresponding to the maximum number of prompts that fit within the GPU memory. AQUA first fills p with prefill prompts having the least number of prefill tokens computed, d with decode prompts with the least number of tokens generated. The remaining slots in d are allocated to prompts in p. This ensures full utilization of GPU capacity when no prefills are present, while leveraging chunked-prefill optimization when they are. AQUA stops filling the batch if the GPU's memory is exhausted.

For simplicity, we measure time slices as the number of inference iterations, as each iteration of a given batch size takes the same time [66]. AQUA reschedules the batch every k iterations or when a request completes, paging out prompts that are not a part of the next batch and paging in prompts that were not on the GPU. Figure 6 shows how AQUA fairly allocates compute between prefill and decoding prompts, reducing TTFT for prompt D.

Efficient context switching. Simply using AQUA TENSORS as swap space is suboptimal for two reasons. First, vLLM stores the key-value tensors of all the prompts associated with a layer as one tensor. So, a given token's key and value tensors are scattered across tensors of different layers that are discontiguous and this leads to multiple small copies to the offloaded AQUA TENSORS. Second, NVLINKS bandwidth is poor for small data transfers, *e.g.*, between two A100 GPUs, the copy bandwidth of a 4MB buffer is 50 GB/s and 64 MB buffer is 200 GB/s. We solved this challenge by gathering smaller tensors into a temporary tensor on the GPU and copying that to the offloaded tensor. Similarly, AQUA-LIB copies offloaded data to a temporary tensor, and scatters it to respective smaller tensors.

Finally, the serving engine can query AQUA-LIB for the tensor location and fall back to FCFS from CFS when tensors are on DRAM to avoid high context switching overheads.

Other scheduling policies. Note that any policy that requires swapping can be implemented using AQUA for faster performance compared to naively swapping to DRAM. For example, priority fairness can be implemented with AQUA by preempting lower priority to swap space hosted with AQUA TENSORS. AQUA-LIB places these tensors on an interconnected GPU's memory whenever possible.

8 Implementation

We implement AQUA-LIB as a pip-installable Python library utilizing PyTorch [1] tensors with 500 lines of code (LOC). The library provides functions to convert standard CPU-based tensors to AQUA TENSORS and retrieve the corresponding GPU or DRAM pointer for compute access. Internally, AQUA-LIB communicates with the coordinator. We integrate the library's northbound interface with serving engines, allowing users to specify GPU memory offering and retention, managed by AQUA-PROFILER.

In vLLM v0.5.3, we implement CFS (§7) with 150 LOC, alongside a CUDA kernel to efficiently gather small buffers into larger ones for NVLINKS transfer with 200 LOC. We introduce a new prefill caching API with unique IDs, and both CFS and prefill caching share the same swap space for context preemption. AQUA-LIB is integrated into vLLM and FlexGen [55] to manage swap space using AQUA TENSORS instead of PyTorch CPU tensors in less than 50 LOC. A key challenge for Aqua Tensors is moving tensors from GPU to DRAM without causing data corruption. We solve this via the insight that inference engines have execution loop boundaries with no GPU compute, allowing safe transfers.

We implement Aqua-placer using Gurobi [31] in 700 LOC. Aqua coordinator is a socket server with REST APIs to offer, reclaim and allocate, deallocate memory as producers and consumers respectively, implemented in 250 LOC.

9 Evaluation

In this section we demonstrate that Aqua's fast preemption significantly improves responsiveness and throughput.

Experiment testbed. Our experiments utilize two separate testbeds. The primary testbed features a server with eight NVIDIA H100 GPUs, each with 80 GB of memory. These GPUs are interconnected via Nvlinks and NVSwitches, providing 450 GB/s bandwidth, and connected to DRAM via PCIe Gen5 with 50 GB/s bandwidth and the server has 1.5 TB of RAM. The secondary testbed consists of a server with two NVIDIA A100 GPUs, each with 80 GB of memory, directly connected via Nvlinks with 300 GB/s bandwidth. These GPUs connect to DRAM via PCIe Gen4 with 25 GB/s bandwidth. We use the H100 testbed for end-to-end tests and the A100 server for micro-benchmarks to minimize costs.

9.1 Models and workloads.

Datasets. We use the parti-prompts [28] dataset to evaluate vision generative models and the default prompts from Audiogen to evaluate audio generative models. Following recent research [4], we use the sharegpt [35] and arxiv summarization [14] datasets to evaluate LLMs.

Distributed serving. We adopt the default policy in vLLM of using tensor parallelism to distribute models across GPUs during inference [35, 56] as long as the model fits within a server and vary the number of GPUs a model is split across to represent diverse workloads. Specifically, Llama 70B can be deployed on 2 GPUs using tensor parallelism, but it can also be scaled to a larger number of GPUs at higher cost when traffic demands are consistently high. For example, when running on the ShareGPT dataset with H100 GPUs, increasing from 2 to 4 GPUs enables Llama 70B to handle ten times more requests, as 4 GPUs provide ample memory after loading the model weights. However, if an organization expects lower sustained traffic with occasional bursts, it is more cost-effective to use fewer GPUs and handle bursts as they arise. In our evaluation, we study both configurations and allow Aqua-profiler to dynamically determine when it should function as either a producer or consumer.

Producer workloads. For audio and vision generative models, we execute the model with a batch size that reaches the peak throughput first. For Llama 8B and 70B models, we use the ShareGPT dataset and run it with interactive workload SLA [4] and pick a request rate that maintains a steady queue length, *i.e.,* the request arrival rate is equal to the request completion rate. We use the same chunk size as recent efforts that developed chunked prefill [4] in our experiments.

Consumer workloads. We evaluate three different consumer workloads, a bursty workload that requires fair scheduling (CFS) introduced in § 7, a prefill caching workload and a memory-constrained long prompt workload.

Fair scheduling during bursts. We use two models to evaluate the bursty workload, Yi-34B on 1 GPU and Llama-70B on 2 GPUs distributed with tensor parallelism. We determine the stable request rate for both the models on the ShareGPT dataset and double the request rate for a duration of 1 minute to create a burst. Assuming an upper bound of 1 minute to provision new instances to handle the burst, we implement the state-of-the-art approach proposed in recent work [59] to handle load-balancing while auto-scaling. We track the request arrival rate every 5 seconds and whenever the request rate increases, we trigger an auto-scale notification. Our workload generator listens to the auto-scale notification and simulates the behavior of the load-balancer by first waiting for a minute for the VM to spin up and then redirecting the newly arriving requests to a dummy http server until the number of requests in the burst has been redirected. At this point, all the instances are back to serving requests at the stable rate. We use a dummy server instead of a VM to avoid allocating additional GPUs and save cost.

Prefill caching. We use Yi-34B-200K model served using vLLM on 2 GPUs to have enough GPU memory to accommodate a large prompt and evaluate prefill caching. We pick a large prompt with 100 thousand tokens to cache and interleave this prompt along with the ShareGPT dataset. Each time the prompt is issued, we only modify the last 256 tokens to reuse the cached prompt to model use cases such as question answering on a long context.

Long prompt. We use the OPT-30B model served using FlexGen [55] to evaluate long prompts that exceed the available memory on the GPU after loading the model weights. FlexGen offloads the inference context of a prompt to the DRAM by preempting the previous layer's context and loading the next layer's context. We determine the median prompt length from the arxiv summarization dataset to be over 7000 tokens, so we sample prompts of length 8192 to evaluate FlexGen since its evaluation script requires a single prompt length parameter for the duration of the experiment.

9.2 Baselines

We evaluate the CFS workload using two baselines: the default vLLM scheduler (FCFS) and vLLM with CFS. While non-preemptive admission control schedulers like VTC [54] prevent malicious users from monopolizing compute resources, they still suffer from queuing delays during a surge of legitimate users. To simulate this behavior, our vLLM baseline treats each request in a burst as a new unique user, capturing the unresponsiveness. Thus, the vLLM lines in our graphs also represent VTC performance.

Model	Workload	Engine	GPUs	Type
Llama3.1 70B	ShareGPT	vLLM + CFS	2	C
Yi-34B-200K	ShareGPT + Prefill Cache	vLLM	2	C
OPT-30B	Long-prompt	FlexGen	1	C
Llama3.1 8B	ShareGPT	vLLM	1	P
Llama3.1 70B	ShareGPT	vLLM	4	P
SD-3, SD-XL, Kandinsky	Parti prompts	Diffusers	1	P
MusicGen, AudioGen	Audio descriptions	PyTorch	1	P

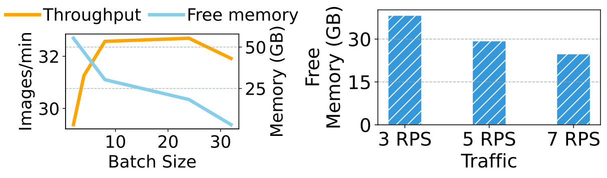
Table 1. Models, workloads, serving engines, and their type (P = Producer, C = Consumer). SD = Stable-Diffusion.

We use our implementation of vLLM with prefill caching that preempts to DRAM as the baseline for prefill caching. We use the vanilla FlexGen preempting to DRAM as the baseline to evaluate long prompt workload. All experiments use the default scheduler unless specifically evaluating CFS.

Evaluation metrics. We use Time To First Token (TTFT), Time Per Output Token (TPOT) and decoding throughput or generation throughput in tokens per second.

9.3 End to end evaluation.

We run an end-to-end evaluation on 8 GPU H100 servers and we set the cluster size to 16 to avoid high costs.

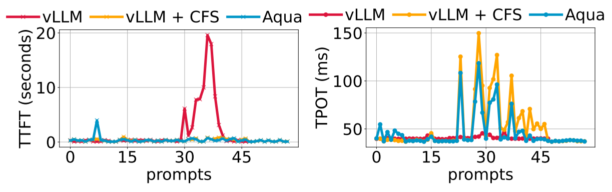


(a) Stable-diffusion-3 **(b)** Llama 3.1 8B

Figure 7. Figure 7a shows the profile generated by Aqua on Stable-diffusion-3 which reaches peak throughput around a batch size of 8 Images with close to 30 GB of unused memory on the GPU. Figure 7b shows that at a steady rate of 5 RPS which adheres to the SLA, there is more than 25 GB of free memory on Llama 8B.

How are models classified? Figure 7a presents the trace generated by Aqua-profiler for the Stable-diffusion-3 model on an H100 GPU. The profiler increases the batch size during inference until throughput saturates, recording the corresponding free GPU memory. Figure 7b shows the trace for the Llama 3.1 8B model. Following prior work [4, 35], we select an SLA with a TTFT under 1 second and an inter-token latency (ITL) below 100 milliseconds, targeting interactive workloads with a batch size of 512. Based on this SLA and ShareGPT input request distribution, our profiler tests various request rates and captures the available free GPU memory during serving. Our experiments found that the maximum sustainable traffic for the LLM within the SLA was 5 requests per second (5 RPS) shown in Figure 7. We similarly profile all other models in our experimental setup to classify producers and consumers.

Placing models on GPUs. After profiling, we sample 10% from audio models, 15% from vision models and the remainder from LLMs (to represent the higher popularity of LLMs) from the models in Table 1, till we exhaust the available GPUs. We execute Aqua-placer with the sampled models and their memory profiles from Aqua-profiler.



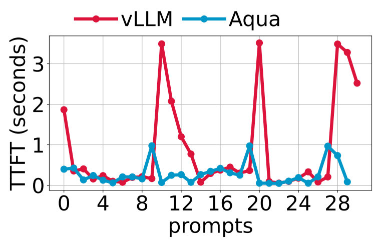
(a) TTFT **(b)** TPOT

Figure 8. Figure 8a shows the TTFT of the requests arriving in order to Llama 3.1 70B running on 2 GPUs using tensor parallelism. The requests are sampled from ShareGPT. Figure 8b shows the TPOT of the requests in order. Aqua minimizes the TTFT while incurring up to 1.5× lower TPOT compared to vLLM + CFS.

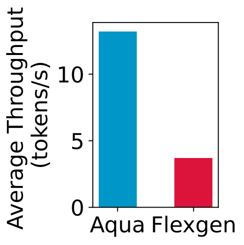
How much does CFS benefit from Aqua? Aqua-placer matched the distributed CFS workload running Llama 3.1 70B on 2 GPUs with two single-GPU models, Llama 8B and Audiogen. The CFS workload's two shards use Aqua-lib to allocate swap space, transparently placed on the GPUs running Llama 8B and Audiogen. We start the workload with stable request rate for CFS and after issuing 25 prompts at this rate, the workload generator doubles the request rate for a minute. This surge, handled by the baseline FCFS scheduler, results in high TTFT of 20 seconds, as shown in Figure 8a.

A CFS policy with vLLM reduces TTFT but incurs a high TPOT from preemption overheads to DRAM (Figure 8b). In contrast, Aqua not only reduces TTFT by 20× but also reduces TPOT by up to 1.5× compared to vLLM + CFS. Aqua shortens the burst impact as well: in Figure 8a, TPOT remains high between 25-48 requests with vLLM + CFS, but only between 25-40 with Aqua. This improvement stems from Aqua-lib's fast preemption, which transparently offloads data via Nvlinks to neighboring GPUs. Figure 8 also shows that the behavior of CFS is identical to FCFS when there is no memory contention, during the first 25 and after 50 requests.

Improvements on prefill caching. Aqua-placer matched the distributed Yi-34B-200K running on 2 GPUs with two single-GPU producers, Stable-Diffusion-3 (SD-3) and Kandinsky (Kand). We start with a stable request rate and issue a 100K token prompt, using an API to cache the inference context for all but the last 256 tokens. Then, 10% of requests use the cached tokens as a prefix with different 256-token suffixes, while the rest are sampled from ShareGPT. The serving engine computes the context initially, taking around 10 seconds, caches it in the swap space, and loads it into the



(a) TTFT on Yi-34B-200K



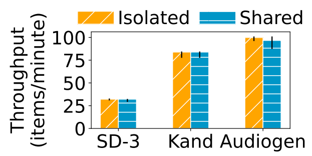
(b) OPT-30B

Figure 9. Figure 9a shows the TTFT of requests in order sent to Yi-34B-200K served on 2 GPUs with tensor parallelism, where spikes occur due to loading cached context from swap space. Figure 9b shows the decoding throughput of OPT-30B serving on 1 GPU. Aqua significantly improves performance of both workloads.

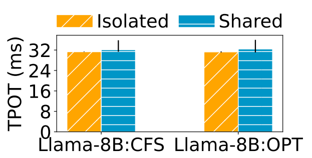
GPU when a request with the same prefix arrives. Figure 9a traces 30 steady-state requests after caching.

The baseline sees TTFT spikes up to 4 seconds on each cache hit, with subsequent requests also facing high TTFT as they are batched with the long prompt and wait for loading its context into the GPU. Aqua-lib minimizes the context loading time from swap space to GPU, reducing the long prompt's TTFT by 3.5×. This also lowers the TTFT for subsequent requests batched with the cached prompt.

Improvements on memory constrained inference. Aqua-placer matched OPT-30B running on a single GPU with Llama 3.1 8B. We run OPT-30B with FlexGen which is used for offline inference scenarios [48] with low resources and aims to improve the throughput of the system by efficiently swapping context to the DRAM. We measure the generation throughput of FlexGen on a sequence of prompts of length 8192 for 10 minutes and report the average throughput across prompts. The context generated by the prompt with 8K tokens is too large to fit on the GPU after loading the model weights and the system starts swapping to execute inference. Figure 9b demonstrates that Aqua consistently achieves a 4× improvement in generation throughput compared to baseline swapping to DRAM throughout the experiment.



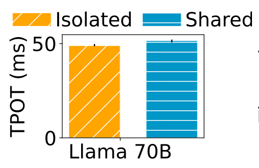
(a) Throughput



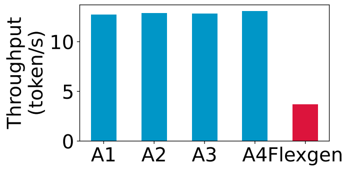
(b) TPOT

Figure 10. Figure 10a shows negligible impact on throughput when audio and image producers share memory with consumer workloads. The isolated bars reflect producer performance without consumers. Figure 10b shows that memory sharing increases the average TPOT of 2 Llama 8B instances by about 4.5 milliseconds.

What is the impact of sharing memory on producers? We executed all the producers that shared memory in the previous experiments without consumers to compare it with the performance when sharing memory with a consumer. Figure 10 shows that the impact of sharing memory is low for producers of all modalities. The difference in throughput is less than 5% for audiogen and non-existent for vision generative models. We use average TPOT to measure the performance of LLM producers since it captures the time taken to execute one inference iteration. Figure 10b shows that the impact of sharing memory is about 4.5 milliseconds on TPOT. A consumer accessing the producer's GPU memory could lead to memory bandwidth contention, as shown in recent work [2]. We posit that the reason for the reduction in performance of LLMs and audiogen models is because they rely on transformer layers that are known to be bottlenecked by memory bandwidth of the GPU [4] and any bandwidth contention will reduce the performance.



(a) Llama-70B TPOT



(b) OPT-30B throughput

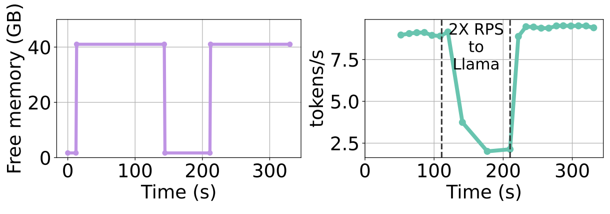
Figure 11. Figure 11a shows that the impact of sharing memory on the average TPOT over the duration of 10 minutes is 5% on Llama 70B served on 4 GPUs using tensor parallelism on the ShareGPT dataset. Figure 11b shows the throughput of the 4 OPT-30B instances during the same duration which is consistently better than the baseline by 4×. (A=Aqua)

What is the impact on distributed producers? Since Aqua-placer did not match any consumer with a distributed LLM producer, we study the impact of sharing memory over Nvlinks on a distributed LLM's performance in this experiment. Inference with tensor parallelism requires inter-GPU communication via Nvlinks, which are also used when consumers access memory on the producer's GPUs, potentially affecting performance. To assess this, we run the Llama 3.1 70B model across 4 GPUs with tensor parallelism and 4 instances of OPT-30B on another 4 GPUs. Each GPU serving Llama has 15 GB of free space and is a producer to one of the OPT-30B which needs 10 GB swap space to infer on an 8K tokens prompt. We pick OPT-30B for this experiment since it frequently utilizes the Nvlinks to swap inference context.

We ran the experiment for 10 minutes, with Llama handling steady requests and OPT-30B generating tokens from long prompts. Figure 11 shows that sharing memory results in a negligible TPOT overhead on Llama is 5%. We investigated the cause for the low overhead and determined two reasons: first, the communication size during tensor parallelism is only a few megabytes. Second, this communication occurs between layers, leaving the interconnect idle during computation. The small buffer sizes combined with fact that the interconnect is idle during computation, leads to the underutilization of NVLINKS. As a result, consumer's traffic barely result in any additional impact. We believe that this overhead can be eliminated if NVIDIA exposes an API to set priority on the packets which we can use to prioritize tensor parallelism packets over producer-consumer traffic.

9.4 AQUA's adaptability

In this subsection, we demonstrate AQUA's adaptability to varying producer workload demands and AQUA's extensibility to different datasets, using the A100 testbed.



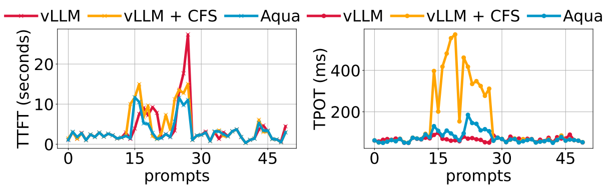
(a) Free memory (Llama 8B) **(b)** OPT-30B throughput.

Figure 12. AQUA's elasticity opportunistically improves the throughput of a memory constrained workload by 6×.

AQUA's elasticity. In this experiment, we host Llama 3.1 8B as producer and OPT-30B as consumer job with AQUA on a GPU server with 2 A100 GPUs. We vary the number of inference requests sent to the producer. Figure 12a shows the available free memory on the GPU serving Llama 8B. At the start, the serving engine reserves all the memory to accommodate the inference context of incoming requests. AQUA-LIB notices that the inference load is steady and using the profiled data, it offers memory on this GPU to consumers.

Near the 60-second mark in Figure 12a, we begin the long prompt inference, achieving a high throughput of 8 tokens per second. At the 120 seconds mark, we double the traffic to the producer. This increases the number of requests in the pending queue and AQUA-LIB notices this and rapidly reclaims memory to accommodate the load. Reclaiming memory reduces throughput of the consumer (Figure 12b). The throughput again increases when LLM is done serving all the requests. AQUA-LIB again detects the steady traffic and transparently moves the offloaded tensors of the consumer back to the producer's GPU. This shows that AQUA can dynamically respond to changes in producer workloads and opportunistically improve consumer performance.

AQUA's benefits extend to different datasets. In this experiment, we use the arxiv summarization dataset and Yi-34B to show how AQUA's benefits carry over to other datasets. The arxiv dataset has longer prompts, a median prompt is around 7000 tokens compared to the 2000 tokens in ShareGPT dataset. So, we start by using AQUA-PROFILER to profile the swap space requirement for Yi-34B executing on a single A100 GPU to handle bursts. Given the stable request rate, the profiler creates bursts that are multiples of the stable



(a) TTFT **(b)** TPOT

Figure 13. Figure 13a shows the TTFT of the requests arriving in order to Yi-34B running on 1 A100 GPU. The requests are sampled from Arxiv dataset. Figure 13b shows the TPOT of the requests.

request rate and determines that the model requires 6GB, 15GB and 40GB of swap space to handle bursts for a minute that are 2×, 3× and 5× the stable request rate.

We begin by hosting Llama 8B on one GPU and Yi-34B on another, issuing requests at a stable rate. To simulate a burst, we triple the request rate to Yi-34B for one minute. Figure 13 shows that AQUA efficiently handles the burst, reducing peak TTFT by 2× and lowering TPOT by 2× through fast preemption. TTFTs for prompts between 10-20 are higher across all systems because prompts are long and the prefill stage is mainly bottlenecked by compute.

10 Related work

GPU job schedulers. Usher [57] is the most recent scheduler that profiles compute and memory utilization on a GPU and multiplexes complementary models on the GPU to increase resource utilization. Alpa-serve [36] dynamically scales the GPUs allocated to the job by allocating underutilized GPUs of other workloads to jobs with high traffic rates. Similarly, Salus [68], clockwork [30], Orion [18] and PipeSwitch [19] multiplex models at the model, layer and kernel granularity to efficiently time share the compute on the GPU. All these systems tightly couple compute and memory allocation on a GPU while multiplexing models and miss the opportunity to utilize the free memory capacity when the compute is saturated. AQUA bridges this gap and maximizes the memory utilization on multi-GPU servers even when there is no opportunity to share compute.

Schedulers for online LLM inference. Recently, the community has proposed many algorithms for scheduling prompts arriving at an LLM [4, 54, 59, 65]. Sarathi-serve [4] proposed a scheduler to batch prefills and decodes. Llumnix [59] schedules prompts across LLM instances by load balancing, autoscaling and live-migrating prompts. VTC [54] does coarse grained fair scheduling for LLM serving engines via admission control to prevent bad actors from overwhelming the infrastructure. VTC also mentions that preemptive scheduling is orthogonal to their work and should be addressed in the future. CFS in AQUA complements these systems by handling bursts while auto-scaling kicks in and by fairly allocating resources to admitted prompts. Efforts like MLFQ [65] also build preemptive schedulers and Aqua-lib can improve their performance even more with fast preemption.

CUDA Unified virtual memory. CUDA unified memory provides a single virtual address space spanning GPUs and the host memory. The CUDA driver manages page faults and prefetching allowing GPUs to allocate and access memory seamlessly across all the GPUs and DRAM. However, its default prefetching strategy is insufficient for LLM inference, where inference contexts are scattered across non-contiguous memory regions. Moreover, the CUDA driver does not opportunistically utilize idle GPU memory across the NVLink and NVSwitch interconnect. Prior work, such as Memory Harvester [23], addresses this limitation by leveraging idle GPU memory as swap space at the driver level but remains application-agnostic, giving workloads no control over eviction and prefetching. This lack of control can lead to inefficient paging, such as evicting model weights instead of inference context, increasing overhead. In contrast, LLM serving engines manually manage prefetching and eviction at the application layer to optimize memory usage. Aqua improves upon CUDA unified memory by enabling explicit control over prefetch order and opportunistically migrating offloaded data to faster GPU memory, maximizing paging performance for LLM inference.

Other generative engines. Orca [67] is an LLM inference engine that introduced continuous batching, while vLLM [35] improved this by designing paged attention. Deepspeed-zero [12] was the baseline for FlexGen [55]. Given that Aqua can enhance FlexGen's performance, similar benefits can be extend to Deepspeed. HeteGen [69] builds upon FlexGen by utilizing both the CPU and GPU for computation. Since HeteGen also relies on PCIe for CPU to GPU I/O, Aqua is complementary to it and can further improve its performance. Unlike CacheGen [37], which trades accuracy for compression and stores many prompts in remote memory, Aqua caches contexts while maintaining accuracy which is suitable for scenarios that use critical data like healthcare, legal and finance. We note that there are simulators [3] facilitating fast profiling. Aqua-profiler can also interface with such simulators and profile even faster but we measure directly from the hardware in this work for accuracy. Dist-serve [70] and Splitwise [50] propose using different hardware for the prefill and decoding stage. As Aqua-profiler can classify disaggregated workloads as producers or consumers, Aqua is complementary to them.

Swapping DRAM pages on RDMA. Infiniswap [29] and Fastswap [10] accelerate paging in Operating Systems by harvesting memory and moving pages across DRAM of interconnected servers over RDMA. The memory allocation algorithms in these systems do not guarantee bandwidth isolation since the same producer can be shared across multiple consumers. Understanding the behavior of closed systems like Nvlinks and NVSwitch under contention is critical to provide paging bandwidth guarantees but is challenging since their congestion control algorithm is closed-source. Even though Aqua's design allows us to implement similar memory allocation algorithms [10, 29] to allocate memory across GPUs, we developed Aqua-placer because we want to provide bandwidth guarantees during paging. We leave the exploration of modeling paging bandwidth over Nvlinks during contention to future work.

11 Acknowledgements

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The authors of this work are also supported by NSF Award #2444537 and the Cornell Bowers CIS-LinkedIn Grant.

References

[1] 2024. PyTorch: An open source machine learning framework that accelerates the path from research prototyping to production deployment. https://pytorch.org. Accessed: 2024-05-03.

[2] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. 2023. Host Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 275–287. https://doi.org/10.1145/3603269.3604878

[3] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A Large-Scale Simulation Framework For LLM Inference. arXiv:2405.05465 [cs.LG] https://arxiv.org/abs/2405.05465

[4] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[5] Mistral AI. 2023. Mistral. https://huggingface.co/docs/transformers/main/model_doc/mistral.

[6] Mistral AI. 2023. Mixtral. https://huggingface.co/docs/transformers/model_doc/mixtral.

[7] Meta AI. 2024. CodeLLaMa. https://github.com/Meta-Llama/codellama.

[8] Meta AI. 2024. LLaMA 3.2. https://www.llama.com/.

[9] Together AI. 2024. Together: The fastest cloud platform for building and running generative AI. https://www.together.ai/. Accessed: 2024-03-16.

[10] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. https://doi.org/10.1145/3342195.3387522

[11] Amazon Web Services. 2024. *EC2 Auto Scaling Warm Pools*. Accessed: 2024-10-12.

[12] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 46, 15 pages.

[13] Anthropic. [n. d.]. Prompt Caching. https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching. Accessed: 2024-09-11.
[14] arXiv. 2023. Arxiv summarization dataset. https://huggingface.co/datasets/ccdv/arxiv-summarization. Accessed: 2024-10-12.
[15] Amazon Web Services (AWS). 2024. Generative AI-Use Cases and Resources. https://aws.amazon.com/generative-ai/use-cases/. Accessed on March 16, 2024.
[16] Microsoft Azure. [n. d.]. Azure AI Studio. Accessed on March 16, 2024. Available at: https://azure.microsoft.com/en-us/products/ai-studio/.
[17] Microsoft Azure. 2024. Azure LLM Inference Dataset 2024. https://github.com/Azure/AzurePublicDataset/blob/master/analysis/AzureLLMInferenceDataset2024.ipynb. Accessed: 2025-02-04.
[18] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. [n. d.]. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *EuroSys 2024*.
[19] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 499–514. https://www.usenix.org/conference/osdi20/presentation/bai
[20] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2000. A Neural Probabilistic Language Model. In *Advances in Neural Information Processing Systems*, T. Leen, T. Dietterich, and V. Tresp (Eds.), Vol. 13. MIT Press. https://proceedings.neurips.cc/paper_files/paper/2000/file/728f206c2a01bf572b5940d7d9a8fa4c-Paper.pdf
[21] Bob Briscoe, Anna Brunstrom, Andreas Petlund, David Hayes, David Ros, Ing-Jyh Tsang, Stein Gjessing, Gorry Fairhurst, Carsten Griwodz, and Michael Welzl. 2016. Reducing Internet Latency: A Survey of Techniques and Their Merits. *Commun. Surveys Tuts.* 18, 3 (July 2016), 2149–2196. https://doi.org/10.1109/COMST.2014.2375213
[22] Cerebras 2021. The future of AI is Wafer-Scale. https://www.cerebras.net/product-chip/.
[23] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. 2022. Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 625–638. https://www.usenix.org/conference/atc22/presentation/choi-sangjin
[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]
[25] Google AI for Developers. [n. d.]. Gemini API Pricing. https://ai.google.dev/pricing. Accessed: 2024-09-11.
[26] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall. *IEEE Micro* 44, 3 (2024), 33–39. https://doi.org/10.1109/MM.2024.3373763
[27] Google. 2024. *Google user abandonment stats.* Accessed: 2024-10-12.
[28] Google. 2024. Parti-prompts. https://huggingface.co/datasets/nateraw/parti-prompts. Accessed: 2024-05-02.
[29] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu
[30] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati
[31] Gurobi. 2024. Gurobi. https://www.gurobi.com/. Accessed: 2024-05-03.
[32] Intel Gaudi AI accelerator 2021. Intel Gaudi AI accelerator. https://habana.ai/products/gaudi/.
[33] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. arXiv:2304.01433 [cs.AR]
[34] Felix Kreuk, Gabriel Synnaeve, Adam Polyak, Uriel Singer, Alexandre Défossez, Jade Copet, Devi Parikh, Yaniv Taigman, and Yossi Adi. 2023. AudioGen: Textually Guided Audio Generation. arXiv:2209.15352 [cs.SD]
[35] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (<conf-loc>, <city>Koblenz</city>, <country>Germany</country>, </conf-loc>) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165
[36] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. [n. d.]. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*.
[37] 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. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 38–56. https://doi.org/10.1145/3651890.3672274
[38] Microsoft. 2024. AI Is Transforming Businesses. https://azure.microsoft.com/en-us/blog/azure-openai-service-10-ways-generative-ai-is-transforming-businesses/, accessed on March 16, 2024.
[39] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL]
[40] MosaicML. 2023. MPT. https://huggingface.co/docs/transformers/main/model_doc/mpt.
[41] NVIDIA. 2024. NVIDIA Blackwell Architecture. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/ Accessed: 2024-10-12.
[42] NVIDIA. 2024. NVLink & NVSwitch: Fastest HPC Data Center Platform. https://www.nvidia.com/en-us/data-center/nvlink/. Accessed: 2024-03-16.
[43] NVIDIA Corporation. 2024. NVIDIA B200. https://www.nvidia.com/en-us/data-center/dgx-b200/. Accessed: 2024-04-28.
[44] NVIDIA Corporation. 2024. NVIDIA DGX A100 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf. Accessed: 2024-04-21.
[45] NVIDIA Corporation. 2024. NVIDIA H100 Tensor Core GPU Datasheet. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet. Accessed: 2024-04-21.
[46] Nvidia DGX Systems 2021. Nvidia DGX Systems. https://www.nvidia.com/en-us/data-center/dgx-systems/.
[47] OpenAI. 2024. OpenAI API. Accessed on March 16, 2024. Available at: https://openai.com/blog/openai-api.
[48] OpenAI. 2024. OpenAI batch API. https://platform.openai.com/docs/guides/batch. Accessed: 2024-09-11.
[49] OpenAI. 2024. OpenAI Customer Stories. https://openai.com/customer-stories. Accessed on March 16, 2024.

[50] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative LLM inference using phase splitting. arXiv:2311.18677 [cs.AR] https://arxiv.org/abs/2311.18677

[51] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-Resolution Image Synthesis with Latent Diffusion Models. arXiv:2112.10752 [cs.CV]

[52] Amazon Web Services. [n. d.]. Amazon SageMaker. Accessed on March 16, 2024. Available at: https://aws.amazon.com/sagemaker/.

[53] Amazon Web Services. 2024. Mewtant Case Study. urlhttps://aws.amazon.com/solutions/case-studies/mewtant-case-study/, accessed on 2024-03-16.

[54] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2024. Fairness in Serving Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 965–988. https://www.usenix.org/conference/osdi24/presentation/sheng

[55] 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. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. arXiv:2303.06865 [cs.LG]

[56] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[57] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. [n. d.]. USHER: Holistic Interference Avoidance for Resource Optimized ML Inference. https://www.usenix.org/conference/osdi24/presentation/shubha. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*.

[58] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Sigcomm '15*.

[59] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 173–191. https://www.usenix.org/conference/osdi24/presentation/sun-biao

[60] The Linux Kernel Developers. [n. d.]. CFS Scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html. Accessed: 2024-03-17.

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[62] Abhishek Vijaya Kumar, Arjun Devraj, Darius Bunandar, and Rachee Singh. 2024. A case for server-scale photonic connectivity. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks* (Irvine, CA, USA) *(HotNets '24)*. Association for Computing Machinery, New York, NY, USA, 290–299. https://doi.org/10.1145/3696348.3696856

[63] vLLM. 2023. Distributed LLM serving. https://docs.vllm.ai/en/latest/serving/distributed_serving.html. Accessed: 2024-10-12.

[64] VLLM. 2024. LLama31: Enhancing LLM Serving Efficiency. https://blog.vllm.ai/2024/07/23/llama31.html. Accessed on September 11, 2024.

[65] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Distributed Inference Serving for Large Language Models. arXiv:2305.05920 [cs.LG] https://arxiv.org/abs/2305.05920

[66] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[67] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[68] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. arXiv:1902.04610 [cs.DC]

[69] Xuanlei Zhao, Bin Jia, Haotian Zhou, Ziming Liu, Shenggan Cheng, and Yang You. 2024. HeteGen: Heterogeneous Parallel Inference for Large Language Models on Resource-Constrained Devices. https://arxiv.org/abs/2403.01164. Accessed on September 11, 2024.

[70] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

A Artifact Appendix

A.1 Abstract

We are releasing the source code of AQUA at https://github.com/aquaml. AQUA needs a multi-GPU server with at least 2 NVIDIA GPUs interconnected with NVLINKS. AQUA's documentation is available at https://aquaml.github.io/.

A.2 Artifact check-list (meta-information)

Here is a list of the repositories that are available in the artifact.

- **AQUA-LIB which includes AQUA TENSORS.**
- **vLLM with AQUA's CFS algorithm**
- **AQUA-PLACER.**
- **PyTorch scripts with integrated AQUA-LIB to execute stable-diffusion.**
- **FlexGen with integrated AQUA-LIB.**
- **Audiogen with integrated AQUA-LIB.**
- **Publicly available?: Yes**